universität
wien

DIPLOMARBEIT

Black Box Optimization with Data Analysis

angestrebter akademischer Grad

Magister der Naturwissenschaften
(Mag. rer. nat.)

Verfasser:	Kevin Kofler
Matrikel-Nummer:	0100446
Studienrichtung:	Mathematik
Betreuer:	Ao. Univ.-Prof. Dipl.-Ing. Dr. Hermann Schichl

Wien, am 15. 11. 2007

Curriculum Vitae

Kevin Kofler
born July 1st, 1983 in Oberpullendorf (Austria)
Italian citizen

Education
High school diploma: French *baccalauréat général* and Austrian *AHS-Matura* at the Lycée Français de Vienne, Vienna, Austria, June 22nd resp. 29th, 2001
Studies: *Mathematics* and *Software and Information Engineering* at the University of Vienna since winter term 2001
Foreign semester: Fall term 2001 at the Bowling Green State University, Bowling Green, Ohio, USA

Awards
1st prize at the *Rallye mathématique d'Alsace* in 2000 and 2001
5th *accessit* at the French *Concours général de Mathématiques* in 2001

Jobs
Teaching assistant (Tutor) at the University of Vienna, March – July 2005
Teaching assistant (Tutor) at the University of Vienna, October 2005 – February 2006
Google Summer of Code, May – September 2006

Chapter 1

Introduction

This thesis presents an algorithm solving black box optimization problems using mainly methods from data analysis and an implementation of this algorithm. In this introduction, we will first present the purpose and goals of our algorithm, then the organization of this thesis and finally the thanks and acknowledgements.

1.1 Goals

The main goal of our thesis is to cross the barrier between the fields of optimization and data analysis by applying methods from data analysis to optimization problems. In particular, we apply data analysis techniques to obtain information about black box functions, i.e. functions for which we do not know an algebraic expression.

We will present both an algorithm and a reference implementation to solve optimization problems where:

- both the objective function and the constraints may be black box functions,
- we do not have any gradient or Hessian information for those black box functions,
- the functions are assumed to be expensive to compute, thus the number of function evaluations shall be kept as small as possible,

using methods from data analysis:

- covariance models,
- Gaussian mixture models (GMMs) and the Expectation-Maximization (EM) iteration,

- ratio-reject.

The implementation is licensed under the GNU General Public License, version 3 [45] or later, with special exceptions allowing to link with the third-party optimizers used.

1.2 Organization

In chapter 2, we will discuss the **mathematical background** for our algorithm. We will give an introduction to the subject of optimization and the current state of the art to give our work some context, and we will define some important concepts from data analysis and statistics which are central to our algorithm. In chapter 3, we will define the exact **model** we operate on and describe our **algorithm** in purely mathematical terms. In chapter 4, we will document our concrete **implementation**, detailing the format of user input, the actual implementation of the algorithm described in the previous chapter and the interfaces to third-party libraries. In chapter 5, we will summarize the **results** obtained with our implementation, both in terms of speed and quality. Finally, chapter 6 will **conclude** the thesis with an outlook on possible future improvements.

1.3 Acknowledgements

First of all, I would like to thank my advisor, Prof. Hermann Schichl, as well as the head of our working group, the Computational Mathematics Group at the Faculty of Mathematics of the University of Vienna, Prof. Arnold Neumaier, who have both been of invaluable help both through their lecture courses which introduced me into the subjects and through the hints they gave me. I would also like to thank Dr. Arthur Flexer of the Institute for Medical Cybernetics and Artificial Intelligence of the Medical University of Vienna, whose *AI Methods of Data Analysis* course gave me further insights into the subject of data analysis. Thanks also go to Stefan Vigerske, the author of the LaGO optimizer [41], whose function enclosure method is at the base of our method to handle implicit equality constraints, and to the authors of the third-party libraries DONLP2 [30], Ipopt [33], MUMPS [35] and `lp_solve` [39] used in our implementation (see section 2.5). I would also like to thank my student colleagues who regularly attended the seminar where we exchanged progress reports and the occasional hints, especially Petra Grell from whom I got the idea to write this thesis in English. Finally, I would like to thank my parents who supported me throughout my studies.

Chapter 2

Mathematical Background

In this chapter, we will give some background on the mathematical context of our algorithm and the methods it employs. In the first section, we give an overview of optimization, presenting the central definitions and a summary of the prior work done in the field. The next three sections give an introduction into the methods from data analysis and statistics which are used extensively in our algorithm, namely covariance models, outlier detection and cubic regression. The last section lists the third-party libraries we employ and describes briefly how they work.

2.1 Optimization Problems

This section aims at giving an overview about the context of this thesis. We will start by giving some motivation and possible applications for optimization. Next, we will provide the general definitions of a global and a local optimization problem. Finally, we will present the most important existing optimization algorithms, first for local optimization, then for the harder global case. The references for most of this section were Prof. Schichl's Optimization [1] and Global Optimization [2] lectures. For each subsection, we will provide consultable references where more details can be found.

2.1.1 Motivation

Optimization aims at finding the minimum or maximum of an *objective function* subject to certain *constraints*. Optimization problems come up in a wide range of applications. For example, wherever there is money to be made, there is an optimization problem: maximize the profit. Other domains where optimization problems come up include:

- prediction of chemical reactions,

- protein folding,
- logistics,
- road construction,
- scheduling problems, e.g. train schedules or class schedules,
- power station control,
- robotics,
- graph structure problems, e.g. the maximum clique problem,
- packing problems, e.g. the knapsack problem,
- nonlinear least-squares problems, e.g. optimal placement of microphones in an orchestra.

Constraint satisfaction problems can be interpreted as a special case of optimization problems where the objective function is constant.

2.1.2 Definitions

An *optimization problem* is a problem of the form

$$\begin{aligned} \min \quad & f(x) \\ \text{s.t.} \quad & x \in G, \end{aligned}$$

where f is a function from a superset U of G to $\mathbb{R}$ called the *objective function*. (Theoretically, any totally-ordered set can be considered for the range of the objective function, but this is rarely useful in practice and most algorithms only work on real-valued objective functions. Therefore, only the real-valued case will be considered.) The set G is called the *feasible set*, any point $x \in G$ is called a *feasible point*, any $x \notin G$ is called an *infeasible point*. We will assume $G \subseteq U \subseteq \mathbb{R}^n$. Further assumptions on G and U will be made depending on the algorithm being considered. Usually, G will be given by inequalities

$$F_l \leq F(x) \leq F_u$$

(where F_l, F_u, and $F(x)$ are vectors and the inequalities are componentwise vector inequalities), the *constraints*. Many algorithms also assume that $U = \mathbb{R}^n$, i.e. that the function is also defined (evaluable) at infeasible points.

Strictly speaking, the definition above only defines *minimization problems*, an optimization problem can also be a *maximization problem*

$$\begin{aligned} \max \quad & f(x) \\ \text{s.t.} \quad & x \in G. \end{aligned}$$

However, this can be trivially reformulated as

$$\begin{aligned} \min \quad & -f(x) \\ \text{s.t.} \quad & x \in G. \end{aligned}$$

A *global solution* of the optimization problem

$$\begin{aligned} \min \quad & f(x) \\ \text{s.t.} \quad & x \in G \end{aligned}$$

is a feasible point $\hat{x}$ such that $f(x) \geq f(\hat{x})\ \forall x \in G$. *Global optimization* means searching for such a global solution. This type of solution is easy to define, but unfortunately it is hard to find. Therefore, another type of solution is often considered: a *local solution* of the above optimization problem is a feasible point $\hat{x}$ such that $f(x) \geq f(\hat{x})$ for all x in a neighborhood of $\hat{x}$ in G.

Obviously, any global solution is also a local solution. Therefore, finding a local solution is easier than finding a global one. Thus, we will first consider algorithms for local optimization, and only then for the global case.

These and more definitions can be found in most texts about optimization, e.g. [3].

2.1.3 Local Optimization Algorithms

Local optimization algorithms mostly differ in the way they handle constraints. Therefore, we first give an overview of unconstrained optimization techniques, which are shared by most local optimization algorithms in practical use (as well as some global ones). Next, we describe the most common approaches used to deal with constraints:

- penalty and barrier approaches, which work by modifying the objective function to take the constraints into account,
- filter methods, which treat a constrained optimization problem as concurrent optimization of the objective and the constraint violation and solve that using techniques from Pareto optimization theory,
- sequential quadratic programming (SQP) techniques, which generate a sequence of quadratic approximations for the constraints.

As those approaches are independent of the unconstrained optimization algorithm in use, they can also be used unmodified for global optimization problems, which are solved with different classes of algorithms.

Unconstrained Optimization

Unconstrained local optimization algorithms work by constructing a sequence of iterates x_k which, given certain conditions, converges to a local optimum. A very common and effective way to construct such iterates is *line search*. In each step, a line search method starts from the iterate x_k and picks

- a search direction s_k (or a nonlinear search curve $s_k(\alpha)$ with $s_k(0) = 0$) and subsequently
- a step length $\alpha_k \in \mathbb{R}^+$ along the search direction.

The next iterate x_{k+1} is then given by $x_k + \alpha_k s_k$ (or $x_k + s_k(\alpha_k)$ in the general case). Concrete line search algorithms differ by the methods they employ for each of these steps.

Most practical methods used to find the search direction are based on the *Newton direction* $-H^{-1}g$ where g is the gradient of the objective function and H its Hessian at x_k, which minimizes the quadratic Taylor approximation of the function around x_k (assuming the Hessian is positive definite). *Newton methods* use the exact Hessian, *Quasi-Newton methods* replace H with an approximation B which satisfies the *Quasi-Newton equation* $g(x_1) - g(x_0) = B\,(x_1 - x_0)$.

The method used to find the step length is called the *line search technique*. The ideal line search is *exact line search*, which chooses α_k such that the objective at x_{k+1} is globally minimal along the search direction (or curve). Practical line search techniques are approximations for this, requiring much less effort.

Unconstrained local optimization is described in much higher detail in [4].

Penalty and Barrier Methods

Both penalty and barrier methods handle constraints by replacing the objective function f with a function f_σ taking the constraints into account through an added term. Concretely, $f_\sigma(x) = f(x) + p_\sigma(x)$ where $\sigma \in \mathbb{R}^+$ and the *penalty term* or *barrier term* $p_\sigma(x)$ is given by the constraints and the parameter σ.

Penalty methods use a penalty term $p_\sigma(x) = \sigma\varphi(\mathrm{cv}(x))$ where $\mathrm{cv}(x)$ is the *constraint violation* $\mathrm{cv}(x) = \sum_i \mathrm{cv}_i(x)$ where

$$\mathrm{cv}_i(x) = \begin{cases} F_i(x) - F_{iu}, F_i(x) > F_{iu} \\ F_{il} - F(x), F_i(x) < F_{il} \\ 0, \text{else} \end{cases}$$

and $\varphi(\lambda)$ is a monotone increasing function of λ with $\varphi(0) = 0$ (i.e. $p_\sigma(x) = 0$ for all feasible points x). In theory, the larger σ is taken, the better the approximation, but in practice this leads to stiff, ill-conditioned problems, so σ is usually increased adaptively. Penalty methods can be used for both equality and inequality constraints.

Barrier methods start from an *interior point*, i.e. a point satisfying the strict inequality $F_{il} < F_i(x) < F_{iu}$ for each constraint F_i, and force the iterates to stay inside the feasible domain by adding a barrier term $p_\sigma(x)$ which tends to infinity as $F_i(x)$ nears the borders F_{il} or F_{iu}. p_σ is defined such that it becomes steeper with higher σ, and again σ is usually increased adaptively. Equality constraints cannot be treated this way, they have to be considered separately, for example by variable substitution. The substitution can be done numerically by solving a nonlinear system of equations in each step; the resulting method is called a *reduced gradient method*.

A more detailed description of penalty and barrier methods, some common choices for penalty and barrier functions and their theoretical background can be found in [5].

Filter Methods

The main idea which leads to filter methods is the following: instead of always starting from the last iterate x_i to search for a new x_{i+1}, we want to start from the "best" x_j in $\{x_1, \ldots, x_i\}$. The problem is that it is not obvious what the "best" x_j actually is. In the unconstrained case, it is obvious: the best x_j is the one with the smallest objective value. Assuming monotone descent, this is actually always x_i. Once we introduce constraints, however, it is no longer obvious which x_j is the best, because we now have 2 parameters to consider: the objective function and the constraint violation.

An obvious way to generalize the choice of a best point to the constrained case is to use a penalty approach: the best x_j is the one minimizing the penalty function f_σ. The weakness of this approach, however, is the arbitrary parameter σ, which is very hard to choose in practice.

Filter methods [6, 7] thus eliminate this arbitrary parameter, instead considering the constrained optimization problem as the simultaneous optimization (*Pareto optimization*) problem of minimizing both the objective function and the constraint violation. This leads to a set of "best" points x_j, the ones which are *Pareto optimal*, i.e. those for which there is no x_k with $f(x_k) \leq f(x_j)$ and $\mathrm{cv}(x_k) < \mathrm{cv}(x_j)$ or $f(x_k) < f(x_j)$ and $\mathrm{cv}(x_k) \leq \mathrm{cv}(x_j)$. This x_j is usually not unique. The individual filter methods differ in the way a single x_j is picked out of the set of Pareto-optimal points.

Sequential Quadratic Programming (SQP)

Another way to deal with constraints is *sequential quadratic programming* (SQP). A *quadratic programming* (QP) problem is an optimization problem with a quadratic objective function and linear constraints. SQP methods replace the original problem by QP approximations which are recomputed at each iteration step.

Concretely, given the problem:

$$\begin{array}{ll} \min & f(x) \\ \text{s.t.} & F(x) \geq 0 \\ & F_i(x) = 0 \ \forall i \in E, \end{array}$$

in each step, one searches for the s which minimizes

$$\begin{array}{ll} \min & \frac{1}{2} s^T H(x,y) s + g(x)^T s \\ \text{s.t.} & J(x)s + F(x) = 0, \end{array}$$

where $g(x)$ is the gradient of the objective function $f(x)$, $J(x)$ is the Jacobian of the constraint vector $F(x)$ and $H(x,y)$ is the Hessian of the Lagrangian $L(x,y) = f(x) - \sum_{i \in E} y_i F_i(x)$ with respect to x. [8, 9]

These QPs are then solved using standard line search techniques.

2.1.4 Global Optimization Algorithms

Global optimization is a harder problem than local optimization: it is obvious to see that it cannot be easier because any global optimum is also a local optimum, and in practice, it turns out to be strictly harder except in some special cases, such as convex problems (for which a local optimum is automatically global). The general case of global optimization has been proven NP-hard by reinterpreting the maximum clique problem, a known NP-hard problem from graph theory, as a global optimization problem [10]. More precisely, that problem is nonconvex quadratic, so even global optimization of QPs is NP-hard in the general (nonconvex) case.

A multitude of algorithms has been developed to solve such problems efficiently. We will first give a classification of the different types of algorithms and then describe some of the most popular algorithms in use today, focusing on approaches which can be used for optimization of black box functions with no gradients available.

Classification

Global optimization algorithms are classified [3] based on what type of convergence, if any, can be proven:

- *Incomplete* algorithms are based on clever heuristics, but have no provable convergence properties, in particular, there is no guarantee of not getting stuck in a local optimum, some do not even have the property to always find a local optimum.

- *Asymptotically complete* algorithms converge to a global optimum with probability 1 when run infinitely long (at least when using exact arithmetic), however at no finite point in time, it is known how far from a global optimum it is.

- *Complete* algorithms converge to a global optimum with certainty when run infinitely long and given exact arithmetic. Moreover, the algorithm knows after finite time that a global optimum has been found up to a given tolerance. However, rounding errors are not accounted for.

- *Rigorous* algorithms find a global optimum up to a given tolerance with certainty and in finite time, even when subject to rounding errors, except in degenerate or almost degenerate cases, in which the tolerances cannot be met. They usually rely on interval methods, rounding mode settings and/or rounding error control.

Unfortunately, rigorous enclosures are essentially a lost cause when dealing with black box functions, therefore rigorous methods will not be considered in this thesis.

Simulated Annealing

Simulated annealing algorithms [11, 12, 13] employ a heuristic based on the observation of the natural process of cristallization: a cristalline structure is one which globally minimizes a certain potential. This global minimum can be obtained by heating the material and slowly cooling it down.

Simulating this process leads to a heuristic based on a temperature parameter T, which is slowly decreased as the algorithm runs. At each step:

- a new point x_{new} is computed using some heuristic,

- if $f(x_{\text{new}}) < f(x_{\text{curr}})$, x_{new} is accepted as the new x_{curr},

- otherwise, acceptance depends on $f(x_{\text{new}}) - f(x_{\text{curr}})$ and the temperature T.

In the original simulated annealing algorithm, new points were generated pseudo-randomly using uniformly distributed numbers, and the probability of accepting an increase in function value was proportional to the Boltzmann distribution $e^{-\frac{K}{T}}$.

This leads to an asymptotically complete, but very slow heuristic. Practical simulated annealing algorithms often throw away asymptotical completeness to get faster convergence on practical examples.

Unfortunately, empirical results show that simulated annealing has trouble dealing with constraints other than bound constraints, convergence is much slower in that case.

Genetic Algorithms

Genetic algorithms [14, 15, 16] are also inspired by natural processes: evolution and natural selection. They begin with a set of N starting points, the *population*. At each step:

- pN ($p \in]0,1[$) points are retained (*selected*), the others are killed. Points with smaller function value have a higher probability of survival.
- The population is filled up by the reproduction of the remaining points. In order to create a new point:
  - two points are combined through a problem-dependent *crossover* procedure and/or
  - a random *mutation* is applied.

Genetic algorithms work well in practice, but they have no provable completeness properties.

Ant Colony

Another class of algorithms inspired by Nature is ant colony algorithms. [17, 18] Those algorithms mimic the behavior of a colony of ants trying to find the shortest path around an obstacle. As they walk, ants deposit pheromones, leaving a scent trail behind. When given a choice between turning left or right, an ant will pick one of the options randomly, however it will pick a direction with a higher concentration of pheromones with a higher probability. As the obstacle is being crossed from both sides, the shorter path will get filled with pheromones faster (because ants from the other side will start reaching it sooner), so more ants will prefer the shorter path, starting a positive feedback loop. Ant colony algorithms mimic this process to solve optimization problems such as the traveling salesman problem.

DIRECT

Unlike the above Nature-inspired methods, the DIRECT algorithm [19] is based on mathematical observations. It is a complete algorithm using no global information.

Therefore, it has to explore the entire search space. More precisely, it has to produce a sequence of points which is dense in the feasible domain, as proven by Törn and Žilinskas in [20]. DIRECT starts from a box (i.e. bound constraints) and constructs such a sequence by repeatedly splitting the box, the iterates are taken as the midpoints of the produced boxes. Of course, dumb exploration of all areas at the same speed would be very inefficient, therefore a tradeoff has to be made between local search around good points and guaranteeing completeness. DIRECT handles this problem by labeling its boxes with both their volume v and the objective function value f at the midpoint and only splitting the boxes which aren't *dominated* by another box, where (v, f) dominates (v', f') if $v > v'$ and $f < f'$. In other words, it only splits a box if there isn't a larger box with a smaller objective function value to split first.

There exist improved algorithms based on this idea, such as *multilevel coordinate search* [21], which only splits boxes up to a certain level, based on the observation that it's pointless to search around a local optimum up to a very small box when the tolerances are already met by the currently-achieved level.

Limitations

The biggest limitation of all the above algorithms is that they get into trouble as soon as we have not only a black box objective function, but also black box constraints. Most of them cannot even handle analytic constraints other than bound constraints in an efficient way. And none of them even tries to handle black box equality constraints. This thesis develops an algorithm designed to tackle this problem.

2.2 Covariance Models

2.2.1 Motivation

Consider the problem where we have a set, or *cloud*, of data points in a given space (in this thesis, we will consider only the finite real vector spaces $\mathbb{R}^m$ as our data point spaces) and want to find *structure* in the data. *Data analysis* aims at finding such structure in data. Of course, the concept of structure is a vague, abstract concept. Therefore, there exists a multitude of data analysis methods, each considering a different form of structure.

In this section, the structure we want to find is:

- How far are the points apart from each other?
- What are the directions in which most of the variation happens?

Moreover, we are not after local information such as the pairwise distances between individual points, but after a global structure. The natural solution to this problem is given by the concept of covariance models. The references for this section were Prof. Neumaier's Mathematical Methods of Data Analysis lecture [22] for the first four subsections and Univ.-Ass. Flexer's AI Methods of Data Analysis lecture [23] for the last two ones. As for the previous section, in each subsection, consultable references providing more details will be given.

2.2.2 Covariance Ellipsoids

Covariance Matrix

Let $(x_i)_{i=1}^n$ be a sequence of vectors in $\mathbb{R}^m$. Then the *mean* $\bar{x}$ of x_i is defined by

$$\bar{x} = \frac{\sum_{i=1}^n x_i}{n}$$

and the *covariance matrix* C_x of x_i by

$$C_x = \frac{\sum_{i=1}^n (x_i - \bar{x})(x_i - \bar{x})^T}{n}.$$

The covariance matrix will be denoted by just C if it is obvious what sequence the covariances are being taken of. C is obviously symmetric as a tensor product scaled by a constant. It is also positive semidefinite because

$$\forall y \in \mathbb{R}^m : y^T C y = \frac{1}{n}\sum_{i=1}^n y^T (x_i - \bar{x})(x_i - \bar{x})^T y = \frac{1}{n}\sum_{i=1}^n \left(y^T (x_i - \bar{x})\right)^2 \geq 0.$$

Weighted Covariance Matrix

Let $(x_i)_{i=1}^n$ be a sequence of vectors in $\mathbb{R}^m$ and $(w_i)_{i=1}^n$ be a sequence in $\mathbb{R}^+$. Then the *weighted mean* $\bar{x}$ of x_i with *weights* w_i is defined by

$$\bar{x} = \frac{\sum_{i=1}^n w_i x_i}{\sum_{i=1}^n w_i}$$

and the *weighted covariance matrix* C_x of x_i with *weights* w_i by

$$C_x = \frac{\sum_{i=1}^n w_i (x_i - \bar{x})(x_i - \bar{x})^T}{\sum_{i=1}^n w_i}.$$

This matrix has the same properties as the unweighted covariance matrix, in particular, it is also symmetric and positive semidefinite. Therefore, it can be used wherever an unweighted covariance matrix can. Setting all w_i to 1 yields the unweighted mean and covariance matrix.

Covariance Ellipsoids

Let C_x be an invertible covariance matrix. (This implies C_x is positive definite, because it is always positive semidefinite.) Then all inequalities of the form

$$(x - \bar{x})^T C_x^{-1} (x - \bar{x}) \leq \epsilon, \quad x \in \mathbb{R}^m$$

for $\epsilon \in \mathbb{R}^+$ form ellipsoids. These ellipsoids are called *covariance ellipsoids* (or *error ellipsoids*) for x_i and the error ϵ.

The covariance ellipsoid is centered around $\bar{x}$, and it is more elongated in those directions in which x_i varies the most. For example, assume C is diagonal, i.e. the covariances $C_{ij}, i \neq j$ all vanish. Then the ellipsoid's axes are parallel to the coordinate axes, and their lengths are the scaled variances $C_{ii}\sqrt{\epsilon}$, thus the ellipsoid is more elongated for those i where the variance C_{ii} is high. In the general case, the axes of the ellipsoid are the eigenvalues of C (which are also its singular values because C is always positive semidefinite), which correspond to the directions spanned by the linear combinations of x_i which vary the most.

A detailed description of covariance matrices and error ellipsoids can be found in [24].

2.2.3 Nonlinear Covariance Models

In the algorithm, the covariance models will be used to approximate functions locally. Unfortunately, ellipsoids are bad approximations for functions, even locally, because the high symmetry of an ellipsoid makes it impossible for it to follow a curve without also following symmetrical curves with opposite curvature. Thus, better approximations are needed. The solution to this problem is to introduce nonlinear covariance models.

Let $(x_i)_{i=1}^n$ and $(y_i)_{i=1}^n$ be sequences of vectors in $\mathbb{R}^m$ and $\mathbb{R}^p$, respectively. Then let z_i be the vector formed by the columns of the upper half of the symmetric matrix $x_i x_i^T$:

$$z_i = (x_{i11}\ x_{i12}\ x_{i22}\ x_{i13}\ x_{i23}\ x_{i33}\ \ldots\ x_{i1m} \ldots x_{imm})^T,$$

where $x_{ijk} = x_{ij}x_{ik}$, i.e. the product of the j^{th} and k^{th} component of x_i, and let

$$X_i := \begin{pmatrix} x_i \\ z_i \\ y_i \end{pmatrix}.$$

The nonlinear covariance model

$$\left(X - \bar{X}\right)^T C_X^{-1} \left(X - \bar{X}\right) \leq \epsilon, \quad X \in \mathbb{R}^{m+\frac{m(m+1)}{2}+p}$$

then produces better local approximations for a function $y = f(x)$ than the ellipsoids produced by the linear model, because f can be locally approximated by a quadratic, and the nonlinear models defined above can follow the curvature of a quadratic, due to the quadratic terms in x_i introduced by the model.

Proof: Consider the quadratic function $y = Az+Bx+c$ where $x \in \mathbb{R}^m$, $y \in \mathbb{R}^p$ and z formed from xx^T as above are variables and $A \in \mathbb{R}^{p\times\frac{m(m+1)}{2}}$, $B \in \mathbb{R}^{p\times m}$ and $c \in \mathbb{R}^p$ are constant. Let I be the identity matrix. The equation can be rewritten as

$$\begin{aligned}
& Az + Bx - (y - c) = 0 \\
\Leftrightarrow\ & (Az + Bx - (y - c))^T (Az + Bx - (y - c)) \leq 0 \\
\Leftrightarrow\ & z^T A^T Az + x^T B^T Bx + (y - c)^T (y - c) + z^T A^T Bx + x^T B^T Az \\
& -z^T A^T (y - c) - (y - c)^T Az - x^T B^T (y - c) - (y - c)^T Bx \leq 0 \\
\Leftrightarrow\ & \left(\begin{pmatrix} x \\ z \\ y \end{pmatrix} - \begin{pmatrix} 0 \\ 0 \\ c \end{pmatrix}\right)^T \begin{pmatrix} B^T B & B^T A & -B^T \\ A^T B & A^T A & -A^T \\ -B & -A & I \end{pmatrix} \left(\begin{pmatrix} x \\ z \\ y \end{pmatrix} - \begin{pmatrix} 0 \\ 0 \\ c \end{pmatrix}\right) \leq 0
\end{aligned}$$

which is a degenerate covariance ellipsoid for the above nonlinear covariance model. The ellipsoid is degenerate because the matrix is singular and can thus not be the inverse of a covariance matrix, and the right hand side ϵ is 0. However, it can be approximated as closely as wanted by a non-degenerate covariance model because the set of regular matrices of a given dimension is dense in the set of square matrices of the same dimension.

2.2.4 Gaussian Mixture Models

Gaussian mixture models [25] are approximations of a probability density by a sum of Gaussian bell curves. They are defined by the *mixture distribution*

$$p(x) = \sum_{j=1}^{M} P(j)p(x|j), \quad x \in \mathbb{R}^N,$$

where the *mixing priors* $0 \leq P(j) \leq 1$ for each j are fixed and sum up to 1, and the *class-conditional densities* $p(x|j)$ are given by the *multi-dimensional Gaussian distribution*

$$p(x|j) = \frac{e^{-\frac{(x-\mu)^T C^{-1}(x-\mu)}{2}}}{(2\pi)^{\frac{N}{2}}\sqrt{det(C)}},$$

where $\mu \in \mathbb{R}^N$ is called the *mean* of the Gaussian distribution and $C \in \mathbb{R}^{N\times N}$ its *covariance matrix*. If C is symmetric positive definite, then $\sqrt{det(C)}$ is the determinant of the Cholesky factor of C.

The gradient of the multi-dimensional Gaussian distribution is given by

$$g(x|j) = -\frac{e^{-\frac{(x-\mu)^T C^{-1}(x-\mu)}{2}} C^{-1}(x-\mu)}{(2\pi)^{\frac{N}{2}}\sqrt{det(C)}} = -p(x|j)C^{-1}(x-\mu)$$

and the Hessian by

$$\begin{aligned} H(x|j) &= -p(x|j)C^{-1} - g(x|j)\left(C^{-1}(x-\mu)\right)^T \\ &= -p(x|j)C^{-1} + p(x|j)C^{-1}(x-\mu)\left(C^{-1}(x-\mu)\right)^T \\ &= p(x|j)\left(C^{-1}(x-\mu)\left(C^{-1}(x-\mu)\right)^T - C^{-1}\right). \end{aligned}$$

In addition to approximating a density, Gaussian mixture models also contain clustering information: every class-conditional density represents a cluster in the data. If the densities show high overlap, the data points don't form clearly-separated clusters.

2.2.5 Expectation-Maximization Iteration

For Gaussian mixture models to be useful, a method to compute the parameters $P(j)$, μ_j and C_j such that the resulting Gaussian mixture model approximates the density of the points $(x_i)_{i=1}^n$ is needed. In this section, an iterative maximum-likelihood algorithm will be described.

The likelihood L of the mixture model is defined as the sum of the probabilities given by the model to each of the points:

$$L = \sum_{i=1}^{n} p(x_i).$$

According to Bayes's theorem, the *posteriors* $P(j|x)$ are given by

$$P(j|x) = \frac{p(x|j)P(j)}{p(x)}.$$

The parameters at the maximum of L satisfy the following optimality conditions:

$$\begin{aligned} \mu_j &= \frac{\sum_{i=1}^n P(j|x_i)x_i}{\sum_{i=1}^n P(j|x_i)} \\ C_j &= \frac{\sum_{i=1}^n P(j|x_i)(x_i-\mu_j)(x_i-\mu_j)^T}{\sum_{i=1}^n P(j|x_i)} \\ P(j) &= \frac{\sum_{i=1}^n P(j|x_i)}{n}. \end{aligned}$$

This assertion is proven for a more general case in [26].

Unfortunately, these equations have unknowns on both sides. However, they can be interpreted as fixed points of the following iteration:

$$\mu_j^{\text{new}} = \frac{\sum_{i=1}^n P^{\text{old}}(j|x_i)x_i}{\sum_{i=1}^n P^{\text{old}}(j|x_i)}$$
$$C_j^{\text{new}} = \frac{\sum_{i=1}^n P^{\text{old}}(j|x_i)\left(x_i-\mu_j^{\text{new}}\right)\left(x_i-\mu_j^{\text{new}}\right)^T}{\sum_{i=1}^n P^{\text{old}}(j|x_i)}$$
$$P^{\text{new}}(j) = \frac{\sum_{i=1}^n P^{\text{old}}(j|x_i)}{n}.$$

This leads to the following iterative algorithm:

1. Start with a simple guess for the distribution $p(x)$.
2. Repeat a sufficient number of times:
   2.1. *E Step*: Compute (*estimate*) the posteriors $P^{\text{old}}(j|x_i)$ from the parameters μ_j^{old}, C_j^{old} and $P^{\text{old}}(j)$.
   2.2. *M Step*: Compute the new parameters μ_j^{new}, C_j^{new} and $P^{\text{new}}(j)$ (*maximize* the likelihood) from the posteriors $P^{\text{old}}(j|x_i)$.

If this iteration converges, the result is a fixed point of the function above, i.e. a solution to the equation system above and thus a critical point of the likelihood L of the Gaussian mixture model.

This algorithm is called the *Expectation-Maximization Iteration* [26].

2.3 Ratio-Reject

Another method from data analysis used in our algorithm is *ratio-reject* [27] (the term *ratio-reject* was coined in [28]), which performs *outlier rejection*, i.e. detects points in a data set which are too far from the cluster the data set is clustered around.

Let X be a finite set of data points in $\mathbb{R}^n$. Let d be a distance in $\mathbb{R}^n$ and, for a finite subset Y of $\mathbb{R}^n$, let $NN^Y(x)$ be the nearest neighbor to x in Y, i.e. the point in $Y \setminus x$ which minimizes $d(x, y)$. Let

$$\rho(x) = \frac{d(x, NN^X(x))}{d(NN^X(x), NN^{X\setminus x}(NN^X(x)))}, \quad x \in \mathbb{R}^n.$$

Then, for a given threshold $s \in \mathbb{R}$, ratio-reject considers as *outliers* in X all points $x \in X$ which satisfy

$$\rho(x) > \bar{\rho}(X) + s\ \sigma_\rho(X),$$

where $\bar{\rho}(X)$ is the mean and $\sigma_\rho(X)$ the standard deviation of all $\rho(x)$ for $x \in X$. [23]

2.4 Cubic Regression

Let $(x_i, y_i)_{i=1}^n$ be a sequence in $\mathbb{R}^2$. The *cubic regression* for this sequence is defined as the cubic least-squares polynomial

$$f(x) = ax^3 + bx^2 + cx + d,$$

where the parameters a, b, c and d minimize the error

$$E = \sum_{i=1}^{n} \left(y_i - f(x_i)\right)^2.$$

By inserting the definition of $f(x)$ and expanding, we get

$$\begin{aligned}
E &= \textstyle\sum_{i=1}^n \left(y_i - (ax_i^3 + bx_i^2 + cx_i + d)\right)^2 \\
&= \textstyle\sum_{i=1}^n \left(y_i^2 - y_i\left(ax_i^3 + bx_i^2 + cx_i + d\right) + \left(ax_i^3 + bx_i^2 + cx_i + d\right)^2\right) \\
&= \textstyle\sum_{i=1}^n \left(y_i^2 - 2x_i^3 y_i a - 2x_i^2 y_i b - 2x_i y_i c - 2y_i d + x_i^6 a^2 + 2x_i^5 ab\right. \\
&\quad \left. +2x_i^4 ac + 2x_i^3 ad + x_i^4 b^2 + 2x_i^3 bc + 2x_i^2 bd + x_i^2 c^2 + 2x_i cd + d^2\right) \\
&= \textstyle\sum_{i=1}^n \left(y_i^2\right) - 2\sum_{i=1}^n \left(x_i^3 y_i\right) a - 2\sum_{i=1}^n \left(x_i^2 y_i\right) b - 2\sum_{i=1}^n \left(x_i y_i\right) c \\
&\quad -2\textstyle\sum_{i=1}^n \left(y_i\right) d + \sum_{i=1}^n \left(x_i^6\right) a^2 + 2\sum_{i=1}^n \left(x_i^5\right) ab + 2\sum_{i=1}^n \left(x_i^4\right) ac \\
&\quad +2\textstyle\sum_{i=1}^n \left(x_i^3\right) ad + \sum_{i=1}^n \left(x_i^4\right) b^2 + 2\sum_{i=1}^n \left(x_i^3\right) bc + 2\sum_{i=1}^n \left(x_i^2\right) bd \\
&\quad + \textstyle\sum_{i=1}^n \left(x_i^2\right) c^2 + 2\sum_{i=1}^n \left(x_i\right) cd + nd^2.
\end{aligned}$$

The first-order optimality conditions $grad(E) = 0$ for E are thus (after simplifying away the common factor 2 and bringing the constant terms to the other side):

$$\begin{aligned}
&\textstyle\sum_{i=1}^n \left(x_i^6\right) a + \sum_{i=1}^n \left(x_i^5\right) b + \sum_{i=1}^n \left(x_i^4\right) c + \sum_{i=1}^n \left(x_i^3\right) d = \sum_{i=1}^n \left(x_i^3 y_i\right), \\
&\textstyle\sum_{i=1}^n \left(x_i^5\right) a + \sum_{i=1}^n \left(x_i^4\right) b + \sum_{i=1}^n \left(x_i^3\right) c + \sum_{i=1}^n \left(x_i^2\right) d = \sum_{i=1}^n \left(x_i^2 y_i\right), \\
&\textstyle\sum_{i=1}^n \left(x_i^4\right) a + \sum_{i=1}^n \left(x_i^3\right) b + \sum_{i=1}^n \left(x_i^2\right) c + \sum_{i=1}^n \left(x_i\right) d = \sum_{i=1}^n \left(x_i y_i\right), \\
&\textstyle\sum_{i=1}^n \left(x_i^3\right) a + \sum_{i=1}^n \left(x_i^2\right) b + \sum_{i=1}^n \left(x_i\right) c + nd = \sum_{i=1}^n \left(y_i\right),
\end{aligned}$$

[29]. For $f(0) = d$, Cramer's rule yields the expression

$$d = \frac{\begin{vmatrix} \sum_{i=1}^n x_i^6 & \sum_{i=1}^n x_i^5 & \sum_{i=1}^n x_i^4 & \sum_{i=1}^n x_i^3 y_i \\ \sum_{i=1}^n x_i^5 & \sum_{i=1}^n x_i^4 & \sum_{i=1}^n x_i^3 & \sum_{i=1}^n x_i^2 y_i \\ \sum_{i=1}^n x_i^4 & \sum_{i=1}^n x_i^3 & \sum_{i=1}^n x_i^2 & \sum_{i=1}^n x_i y_i \\ \sum_{i=1}^n x_i^3 & \sum_{i=1}^n x_i^2 & \sum_{i=1}^n x_i & \sum_{i=1}^n y_i \end{vmatrix}}{\begin{vmatrix} \sum_{i=1}^n x_i^6 & \sum_{i=1}^n x_i^5 & \sum_{i=1}^n x_i^4 & \sum_{i=1}^n x_i^3 \\ \sum_{i=1}^n x_i^5 & \sum_{i=1}^n x_i^4 & \sum_{i=1}^n x_i^3 & \sum_{i=1}^n x_i^2 \\ \sum_{i=1}^n x_i^4 & \sum_{i=1}^n x_i^3 & \sum_{i=1}^n x_i^2 & \sum_{i=1}^n x_i \\ \sum_{i=1}^n x_i^3 & \sum_{i=1}^n x_i^2 & \sum_{i=1}^n x_i & n \end{vmatrix}},$$

which can be written as

$$d = \frac{d_1 \sum_{i=1}^{n} y_i - d_2 \sum_{i=1}^{n} x_i y_i + d_3 \sum_{i=1}^{n} x_i^2 y_i - d_4 \sum_{i=1}^{n} x_i^3 y_i}{d_1 n - d_2 \sum_{i=1}^{n} x_i + d_3 \sum_{i=1}^{n} x_i^2 - d_4 \sum_{i=1}^{n} x_i^3},$$

where

$$d_1 = \begin{vmatrix} \sum_{i=1}^{n} x_i^6 & \sum_{i=1}^{n} x_i^5 & \sum_{i=1}^{n} x_i^4 \\ \sum_{i=1}^{n} x_i^5 & \sum_{i=1}^{n} x_i^4 & \sum_{i=1}^{n} x_i^3 \\ \sum_{i=1}^{n} x_i^4 & \sum_{i=1}^{n} x_i^3 & \sum_{i=1}^{n} x_i^2 \end{vmatrix},$$

$$d_2 = \begin{vmatrix} \sum_{i=1}^{n} x_i^6 & \sum_{i=1}^{n} x_i^5 & \sum_{i=1}^{n} x_i^4 \\ \sum_{i=1}^{n} x_i^5 & \sum_{i=1}^{n} x_i^4 & \sum_{i=1}^{n} x_i^3 \\ \sum_{i=1}^{n} x_i^3 & \sum_{i=1}^{n} x_i^2 & \sum_{i=1}^{n} x_i \end{vmatrix},$$

$$d_3 = \begin{vmatrix} \sum_{i=1}^{n} x_i^6 & \sum_{i=1}^{n} x_i^5 & \sum_{i=1}^{n} x_i^4 \\ \sum_{i=1}^{n} x_i^4 & \sum_{i=1}^{n} x_i^3 & \sum_{i=1}^{n} x_i^2 \\ \sum_{i=1}^{n} x_i^3 & \sum_{i=1}^{n} x_i^2 & \sum_{i=1}^{n} x_i \end{vmatrix},$$

$$d_4 = \begin{vmatrix} \sum_{i=1}^{n} x_i^5 & \sum_{i=1}^{n} x_i^4 & \sum_{i=1}^{n} x_i^3 \\ \sum_{i=1}^{n} x_i^4 & \sum_{i=1}^{n} x_i^3 & \sum_{i=1}^{n} x_i^2 \\ \sum_{i=1}^{n} x_i^3 & \sum_{i=1}^{n} x_i^2 & \sum_{i=1}^{n} x_i \end{vmatrix}.$$

In this expression, the denominator and the subdeterminants d_1, d_2, d_3 and d_4 do not depend on the y_i. This can be used to extrapolate a vector-valued function $F : \mathbb{R} \to \mathbb{R}^k$ towards zero componentwise. Neither the denominator nor the subdeterminants have to be recomputed for each component.

2.5 Third-party Libraries

2.5.1 Local Nonlinear Optimization

Each step of our algorithm computes a model which needs to be locally optimized. We use a third-party optimization library for this purpose. One of two optimization packages can be used.

DONLP2

The first supported local optimizer is Peter Spellucci's DONLP2 [30, 31, 32]. We use the C version with dynamic memory allocation (`donlp2_intv_dyn`). In our experience, this is the faster solver for small problems.

DONLP2 solves problems of the form

$$\begin{aligned} \min \quad & f(x) \\ \text{s.t.} \quad & x \in \mathbb{R}^n \\ & x_l \leq x \leq x_u \\ & b_l \leq Ax \leq b_u \\ & c_l \leq c(x) \leq c_u \end{aligned}$$

using an SQP method: it constructs a sequence of equality-constrained quadratic approximations, which are optimized using quadratic programming techniques. DONLP2 uses an active set method, a purely local error criterion for the Kuhn-Tucker conditions is used to estimate the active set. It requires function values and gradients as input, the Hessian of the Lagrangian is estimated using a slightly modified version of the Pantoja-Mayne update. It uses variable dual scaling and an improved Armijo-type stepsize algorithm. Bounds on the variables are treated in a gradient-projection like fashion. More details can be found in the included documentation and in Spellucci's papers [31] and [32].

Ipopt and MUMPS

Alternatively, the Ipopt [33, 34] optimizer from the COIN project can be used. Ipopt needs a solver for sparse symmetric linear systems of equations. In our tests, we used MUMPS [35, 36, 37, 38] in sequential mode.

Ipopt (Interior Point OPTimizer) solves problems of the form

$$\begin{aligned} \min \quad & f(x) \\ \text{s.t.} \quad & x \in \mathbb{R}^n \\ & x_l \leq x \leq x_u \\ & c_l \leq c(x) \leq c_u \end{aligned}$$

using an interior point line search filter method. Ipopt uses a sparse matrix representation for Jacobians and Hessians. Linear constraints are not handled specially, linearity is only considered in the case where it leads to zeros in the sparsity structure of the Hessian of the Lagrangian. A primal-dual barrier approach is used. More details can be found in [34].

MUMPS (MUltifrontal Massively Parallel Solver) solves linear equations of the form $Ax = b$ where A is a square sparse matrix. A can be either unsymmetric, symmetric positive definite or general symmetric. The systems which Ipopt solves are general symmetric. MUMPS uses a multifrontal direct method to factor the matrix into LU or LDL^T form (symmetric matrices use the latter). The algorithm is designed to be highly parallelizable, and MUMPS can do its computations sequentially or in parallel. In our tests, the sequential version was used. The details can be found in the papers [36, 37, 38].

2.5.2 Linear Programming: lp_solve

To compute approximations of equality constraints, linear programming is used. We used the `lp_solve` [39] library, which solves linear programs using the revised simplex method [40]. `lp_solve` also supports mixed-integer linear programming, but we do not use this feature.

Chapter 3

Model and Algorithm

This chapter gives a detailed mathematical description of our algorithm. We start by describing the model formulation we accept as input. In a second section, we then present the algorithm we use to solve the model.

3.1 Model

Our algorithm solves optimization problems of the form

$$\begin{aligned} \min \quad & c^T \begin{pmatrix} x \\ y \end{pmatrix} \\ \text{s.t.} \quad & y = F_1(x) \\ & F_2(x) = 0 \\ & x_l \leq x \leq x_u \\ & F_l \leq y \leq F_u \end{aligned}$$

where x and y are variable vectors, c, x_l, x_u, F_l, and F_u are constant vectors (the bounds can be infinite), inequalities are component-wise, and

$$F(x) = \begin{pmatrix} F_1(x) \\ F_2(x) \end{pmatrix}$$

is assumed to be an **expensive**, **black box** function. By this we mean that no closed-form algebraic expression for the function is known, no gradients are available, and the bulk of the runtime of the algorithm on a real-world problem is expected to be given by the function evaluations. These assumptions are central to the design of the algorithm. We perform an **incomplete global** optimization on this model. As explained in chapter 2 (*Mathematical Background*), this means we attempt to find a global solution for our model, but are unable to guarantee globality. In fact, we cannot even guarantee always finding a local optimum, due

to the lack of gradients and any sort of global information. Despite this lack of guarantees, the algorithm performs well in practice. The entire chapter 5 will be dedicated to practical results.

The above form was chosen very carefully. For one, all black box optimization problems can be represented this way. For example, consider a problem with a black box objective function F and bound constraints:

$$\begin{aligned} \min \quad & F(x) \\ \text{s.t.} \quad & x_l \leq x \leq x_u \end{aligned}$$

This can be trivially rewritten as:

$$\begin{aligned} \min \quad & \begin{pmatrix} 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} \\ \text{s.t.} \quad & y = F(x) \\ & x_l \leq x \leq x_u \\ & -\infty \leq y \leq \infty \end{aligned}$$

which is of the required form. And secondly, simpler formulations we considered lose too much information which is essential for the performance of the algorithm.

The formulation we first considered was

$$\begin{aligned} \min \quad & c^T x \\ \text{s.t.} \quad & y = F(x) \\ & x_l \leq x \leq x_u \\ & F_l \leq y \leq F_u \end{aligned}$$

which in theory can also represent all black box optimization problems. However, we found two flaws with this formulation in practice. The first flaw is that this formulation needlessly blows up the search space when black box (or even nonlinear algebraic) objective functions are encountered. Again, consider the problem with black box objective function F and bound constraints. The proposed formulation would turn that problem into

$$\begin{aligned} \min \quad & \begin{pmatrix} 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} \\ \text{s.t.} \quad & y = F(x_1) - x_2 \\ & x_l \leq x_1 \leq x_u \\ & F_l \leq x_2 \leq F_u \\ & 0 \leq y \leq 0. \end{aligned}$$

which is highly inefficient because we now have an extra dimension x_2 to search. Luckily, we found that it required almost no extra effort to handle the more general

formulation

$$\begin{aligned} \min \quad & c^T \begin{pmatrix} x \\ y \end{pmatrix} \\ \text{s.t.} \quad & y = F(x) \\ & x_l \leq x \leq x_u \\ & F_l \leq y \leq F_u \end{aligned}$$

(compared to the original formulation, we only added the y term to our linear objective function) which eliminates the spurious variable.

Unfortunately, these considerations made a second flaw apparent: even with the revised formulation, only **explicit** equality constraints were accounted for, **implicit** equality constraints could still not be handled efficiently. For example, consider the problem

$$\begin{aligned} \min \quad & c^T x \\ \text{s.t.} \quad & F(x) = 0 \\ & x_l \leq x \leq x_u. \end{aligned}$$

Even with our revised formulation, the best obtainable rewrite is

$$\begin{aligned} \min \quad & c^T x \\ \text{s.t.} \quad & y = F(x) \\ & x_l \leq x \leq x_u \\ & 0 \leq y \leq 0. \end{aligned}$$

which is still suboptimal. In fact, while this might not be immediately obvious, there is still an excess x dimension here: assuming a well-behaved F, only a lower-dimensional submanifold of the search space of x actually contains feasible points. This is a hard problem to deal with, and most (if not all) existing black box optimization algorithms choke on it. One theoretically possible solution would be to lower the dimension of the search space by numerically solving the implicit constraint for the excess variables, however this would mean solving a nonlinear system of equations for each new point to consider, requiring prohibitively many function evaluations. Thus, this is not an option in practice. Instead, we keep implicit equality constraints $F_2(x) = 0$ separate in our problem formulation and handle them in a special way aiming at reducing the search space. Section 3.2.5 will be dedicated to this problem. However, even with this method, implicit equality constraints cannot be handled as efficiently as explicit ones, therefore it is still useful to take advantage of explicit formulations where they exist.

3.2 Algorithm

This section proposes an algorithm to solve the above model. First, we give a rough overview of the algorithm, then we detail each step. We start by describing

our method for starting point generation, then the local search technique, then our global search method which looks for unexplored regions, and finally we present a way to deal with implicit equality constraints in the global search. We will go into depth over the mathematical description of the algorithm, we will however leave the details of the concrete implementation for the next chapter. We will also motivate our decisions by mentioning some alternative approaches we tried without success.

3.2.1 Overview

The outline of our algorithm is the following:

1. If we do not have enough starting points, generate some.
2. While the maximum number of function evaluations is not reached,
   - for an even point (local search):
     (a) pick a "best" point using a Pareto filter method (see section 2.1.3),
     (b) compute a regularized weighted nonlinear covariance model (see section 2.2.2) around the point,
     (c) optimize the model using a third-party local optimizer;
   - for an odd point (global search):
     (a) compute a Gaussian mixture model (see section 2.2.4) approximating the point density,
     (b) optimize the model (minimize the density, in order to search in unexplored regions) using a third-party local optimizer.
3. If we have implicit equality constraints, try extrapolating a feasible point by:
   (a) picking the points which are Pareto-optimal (within the set of computed points) for the simultaneous minimization of the objective f and the constraint violation cv,
   (b) throwing out those which are either already feasible to the desired tolerance (cv too small) or too far from feasibility (cv too large),
   (c) throwing out possible outliers using ratio-reject (see section 2.3),
   (d) using cubic regression (see section 2.4) to extrapolate cv to 0 from the remaining points (which are assumed to be a good approximation for the Pareto front, i.e. the set of theoretically Pareto-optimal points),
   (e) computing the actual constraint violation to verify actual feasibility.

The global search ignores inequality constraints, i.e. the bounds F_l and F_u for the explicit equality constraints, by design, because we only have local information for the constraints, so we can't reliably tell which points are feasible and which aren't, and in addition evaluating at infeasible points can give us information important to find further feasible regions. (The explicit equality constraints themselves are irrelevant because the global search only searches in the x coordinates.) For implicit equality constraints however, it has proven impractical to completely ignore them and blindly search everywhere, therefore we approximate them with quadratics, and actually optimize the GMM twice: first without the equality constraints, then, using the result from this optimization as a starting point, again with the approximations for the constraints. (Optimizing the model directly with the constraints from a generic starting point has proven too hard for the local optimizers, thus the two-stage approach.) Only the result with the constraints is retained.

3.2.2 Starting Point Generation

To generate starting points, we use a heuristic suggested by Prof. Neumaier. Let N be the total number of starting points to generate. If we already have at least N user-provided starting points, we can skip this step. Otherwise, let m be the dimension of the points we want to generate, i.e. we want to construct $x_1, \ldots, x_N \in \mathbb{R}^m$. We construct a grid of $(2N)^m$ points $x[1, \ldots, 1], \ldots, x[2N, \ldots, 2N]$, equidistant along each dimension, such that $x[i_1, \ldots, i_{j-1}, 1, i_{j+1}, \ldots, i_m]_j = x_{lj}$ and $x[i_1, \ldots, i_{j-1}, 2N, i_{j+1}, \ldots, i_m]_j = x_{uj}$. (Note that this construction is purely theoretical, in practice we will not compute the coordinates of these exponentially many points!) We then proceed to fill this grid semi-randomly, but taking care not to generate more than one point with a given x_j coordinate, and preferring points farther away from existing ones to closer points. This heuristic is designed to prevent the formation of random clusters of starting points, with other areas remaining unexplored.

The way we obtain such a filling is:

1. If we are given starting points by the user, we round these starting points to the closest points on the grid (just for the purpose of the starting point generation step; later in the algorithm, the actual user-provided points will be used), and proceed as if these points had been generated by the automated heuristic.

2. For each dimension $j = 1, \ldots, m$, we pick an index i_j which has not been used yet. Given that there are $2N$ total possibilities for i_j and only N starting points are to be generated, there are always at least $N + 1$ such possibilities. Each unused i_j is given the same probability. The resulting point is the point $x[i_1, \ldots, i_m]$ on the grid.

3. We repeat the above procedure for a total of 10 points, then retain the one the farthest away from the existing points, i.e. with the largest euclidean distance to the closest existing point. (If there are several points with the same distance to existing points, any of them can be picked.) The other 9 points generated in this step are discarded.

4. We repeat the above two steps until we have N retained points.

It shall be noted that the rounded points from step 1 might not have been valid for automated generation, in particular there might be several points with the same j^{th} index i_j. However, this doesn't impact the algorithm in any way, and therefore we do not attempt to "correct" user-provided starting points.

3.2.3 Local Search

Our local search starts at a previously-found point, constructs a local surrogate model approximating the black box optimization problem around the point, then optimizes that problem using a third-party local optimizer.

Choice of Best Point

The point to start the local search at is picked using a Pareto filter method as described in section 2.1.3. Unfortunately, in the presence of constraints, there is (in general) no one best point, as there is both the objective function value and the constraint violation to take into account. We first tried to define the best point (at which the local search is to be started) using a penalty approach: we defined the best point as the point which minimizes

$$p(x) = c^T \begin{pmatrix} x \\ y \end{pmatrix} + N\mathrm{cv}_1(x) + \kappa(N)\|F_2(x)\|_1$$

where

$$\mathrm{cv}_1(x) = \sum_{x_i<x_{li}} (x_{li} - x_i) + \sum_{x_i>x_{ui}} (x_i - x_{ui}) + \sum_{y_i<F_{li}} (F_{li} - y_i) + \sum_{y_i>F_{ui}} (y_i - F_{ui}),$$

$$\kappa(N) = \begin{cases} \sqrt{\frac{N}{18}}, & N < 18 \\ \frac{N}{18}, & 18 \le N < 77 \\ \frac{N\sqrt{N}}{158}, & N \ge 77 \end{cases}$$

and N is the number of already-evaluated points (so the penalty increases adaptively as the algorithms proceeds). Unfortunately, we found that this penalty function was very sensitive to the parameters. In particular, the coefficient $\kappa(N)$

for the last term (the violation of the implicit equality constraints) required so much tweaking to get one test case to converge and was so sensitive to very small parameter changes that we found it impossible to generalize this approach in any meaningful way. Therefore, a filter approach was picked instead.

As in the general Pareto filter method, our best point is a point from the set P of x_j for which there is no x_k with $c^T\binom{x_k}{y_k} \leq c^T\binom{x_j}{y_j}$ and $e(x_k) < e(x_j)$ or $c^T\binom{x_k}{y_k} < c^T\binom{x_j}{y_j}$ and $e(x_k) \leq e(x_j)$. For $e(x)$, we did not pick the raw constraint violation $\mathrm{cv}(x) = \mathrm{cv}_1(x) + \|F_2(x)\|_1$, but the weighted term $e(x) = N\mathrm{cv}_1(x) + \kappa(N)\|F_2(x)\|_1$, though we do not expect this to make a lot of difference in practice. We then pick a "best" point from P using the following recipe:

- If the most recent point x_r in P has never been used as the starting point for a local search, we pick it with a probability of $1 - \frac{1}{|P|}$.
- Otherwise, i.e. with a probability of $\frac{1}{|P|}$ if x_r was never used and always if it was, we pick a random point out of P (which can also be x_r) with equal probabilities (i.e. each point is picked with probability $\frac{1}{|P|}$).

Note that the set P is never empty by construction, therefore it is always possible to pick such a point. The local search is then started from this point.

Surrogate Model

Let x_{best} be the point constructed above. We compute a nonlinear covariance model as in section 2.2.3 around this point: we consider the points

$$X_i = \begin{pmatrix} x_i \\ z_i \\ y_i \end{pmatrix}$$

where x_i are our iterates, $y_i = \binom{y_{1i}}{y_{2i}} = F(x_i) = \binom{F_1(x_i)}{F_2(x_i)}$ and z_i is the vector formed by the columns of the upper half of the symmetric matrix $x_i x_i^T$:

$$z_i = (x_{i11}\ x_{i12}\ x_{i22}\ x_{i13}\ x_{i23}\ x_{i33}\ \ldots\ x_{i1m} \ldots x_{imm})^T,$$

where $x_{ijk} = x_{ij}x_{ik}$, i.e. the product of the j^{th} and k^{th} component of x_i, and the weights

$$w_i = \frac{1}{\|x_i - x_{\mathrm{best}}\|_2^6 \sqrt{p(x_i) - p_{\min} + \frac{1}{10}}}$$

with the $p(x)$ from above and $p_{\min} = \min_i p(x_i)$, i.e. the closer to x_{best}, the higher the weight (guaranteeing locality) and the smaller $p(x)$ (i.e. the better the point),

the higher the weight (guaranteeing a better fit in the area most likely to contain the optimum), but priority is given to locality (while not discarding global information completely). The best point itself (which would have infinite weight) is ignored, it is used to center the covariance model instead. The precise formula is the result of empirical experimentation.

We also tried another approach where we did not treat the best point specially, instead computing both a mean vector and a covariance matrix with the weights

$$\tilde{w}_i = \frac{1}{\left(\|x_i - x_{\text{best}}\|_2^2 + \frac{1}{2}\right)^4 \sqrt{p(x_i) - p_{\min} + \frac{1}{10}}},$$

however the resulting models performed badly on the Rosenbrock function, causing the algorithm to fail to converge. Forcing the covariance models to be centered around the best point ensures a more accurate model around that point; in particular, this trick gets the algorithm to search in a sensible direction for the Rosenbrock function.

In the presence of implicit equality constraints, there is an additional heuristic: if we have enough points (we used a hardcoded lower limit of 28 points in our implementation, but it may make sense to account for the dimension of the points), we consider only the half with the lowest equality constraint violation $\|F_2(x)\|_1$ and discard the other half. We force the mean of the covariance model to X_{best} (i.e. the X_i with $x_i = x_{best}$), compute the weighted covariance matrix C_X of the finite sequence $(X_i)_i$ with weights w_i as in section 2.2.2 and build a model

$$k_{\text{low}} \leq \left(X - \bar{X}\right)^T C_X^{-1} \left(X - \bar{X}\right) \leq k_{\text{up}}.$$

The inversion C_X^{-1} is implemented in practice by computing a Cholesky factorization $C_X = LL^T$ (remember that C_X is always positive semidefinite), which gives us an easy way to handle singular or near-singular C_X by regularizing the Cholesky factorization: we replace zeros or near-zeros in the diagonal of the Cholesky factor, i.e. $L_{ii} \approx 0$, by εC_{Xii} with a small constant ε if $C_{Xii} \neq 0$, and 1 otherwise. More precisely, we regularize the diagonal elements with $L_{ii} \leq \varepsilon C_{Xii}$ with the above ε, the non-strict inequality ensures the case $L_{ii} = 0$ is always taken into account too. In higher dimensions, we add an additional εC_{Xii} to all L_{ii} independently of the value of L_{ii} because this proved beneficial in our tests, the models in higher dimensions tended to be too close to degeneracy without this tweak; we do not do this in lower dimensions because our tests showed the stronger regularization to be counterproductive in low dimensions. Let L be the regularized Cholesky factor and $M := L^{-T}L^{-1}$. Then the model we consider is $k_{\text{low}} \leq k(X) \leq k_{\text{up}}$ with

$$k(X) = \left(X - \bar{X}\right)^T M \left(X - \bar{X}\right).$$

What is left to do to put the directional covariance information into practical use is to pick sensible values for the bounds k_{low} and k_{up}. Let m be the dimension of the iterates, i.e. $x_i \in \mathbb{R}^m$. Let N be the set formed by the $2m+1$ iterates X_i (out of those used to build the covariance model, so points discarded because of an excessive equality constraint violation are not considered) closest to x_{best}. If we don't have $2m+1$ such iterates, let N be the set of all applicable iterates. In case of points with equal distance, all points with the same distance from x_{best} as the $(2m+1)^{\text{st}}$ closest point are added to N. The distance considered is the Euclidean distance in the x component $\|x_i - x_{\text{best}}\|_2$. We define

$$k_{\text{low}} = \min_{X \in N} k(X), \quad k_{\text{up}} = 2^{m-1} \max_{X \in N} k(X).$$

Finally, we obtain the surrogate problem

$$\begin{aligned}
\min \quad & c^T \begin{pmatrix} x \\ y_1 \end{pmatrix} \\
\text{s.t.} \quad & k_{\text{low}} \leq k(X) \leq k_{\text{up}} \\
& y_2 = 0 \\
& z = (x_{11}\ x_{12}\ x_{22}\ x_{13}\ x_{23}\ x_{33}\ \ldots\ x_{1m} \ldots x_{mm})^T \\
& x_l \leq x \leq x_u \\
& F_l \leq y \leq F_u,
\end{aligned}$$

eliminate the redundant variables y_2 and add bounds z_l and z_u for z which are the result of interval multiplications (without rounding control) on the matching components of x_l and x_u:

$$\begin{aligned}
\min \quad & c^T \begin{pmatrix} x \\ y_1 \end{pmatrix} \\
\text{s.t.} \quad & k_{\text{low}} \leq k \left(\begin{pmatrix} x \\ z \\ y_1 \\ 0 \end{pmatrix} \right) \leq k_{\text{up}} \\
& z = (x_{11}\ x_{12}\ x_{22}\ x_{13}\ x_{23}\ x_{33}\ \ldots\ x_{1m} \ldots x_{mm})^T \\
& x_l \leq x \leq x_u \\
& F_l \leq y \leq F_u \\
& z_l \leq z \leq z_u
\end{aligned}$$

and optimize the resulting problem using a local optimizer, with

$$X_{\text{best}} = \begin{pmatrix} x_{\text{best}} \\ z_{\text{best}} \\ y_{\text{best}} \end{pmatrix}$$

as the starting point. We accept the optimum $\hat{x}$ as our next iterate.

3.2.4 Global Search

The goal of our global search is to "fill the gaps" in the search space. Mathematically, this means we want to find points in areas where the density of already existing points is low. Therefore, our global search works by minimizing a density estimator. The motivation for this is given by the observation that neglecting certain areas can lead to missing the global optimum (whenever it happens to be located in the neglected area), an empirical observation which has been formalized and proven by Törn and Žilinskas in [20]. Unfortunately, their theoretical result is not of immediate use to us: they proved that a global optimization algorithm using only local information at the iterates can be complete only if the iterates lie dense in the search space, but both denseness of the iterates and completeness of the algorithm are asymptotical concepts, they only apply if we allow an infinite number of iterates. In practice however, the number of iterates is finite, and in our case, function evaluations (of the black box constraints) are assumed to be expensive, and each new iterate implies a new function evaluation, which forces us to stop after a relatively small number of iterates. Therefore, asymptotical results are only of limited use, and thus we do not attempt to construct a sequence which lies dense in the search space when an infinite number of points are constructed, because this will not be useful in practice anyway. Instead, we take a more heuristic approach, using methods from data analysis to define a concept of density for a finite number of points, which can then be optimized. In this section we will assume that there are no implicit equality constraints, the next section will present a way to deal with those.

As the density estimator we want to minimize, we use a Gaussian mixture model (GMM, see section 2.2.4), over the x coordinates only, as those are the only ones we can control. Let N be the number of points we have already computed, then we consider a sum of $\lfloor \frac{N}{4} \rfloor$ Gaussians. As for the local search, we regularize the covariance matrices of the Gaussians during the Cholesky factorization. In this case, if the diagonal element L_{ii} of the cholesky factor is smaller than a small ε, we simply set $L_{ii} := \varepsilon$. In our implementation, ε was taken to be the square root of the machine epsilon `DBL_EPSILON`, but in principle any $\varepsilon > 0$ will do. (In practice, however, a too small ε will cause numerical difficulties, a too large ε will lose too much information.)

We compute the GMMs using the Expectation Maximization (EM) iteration as described in section 2.2.5. As our starting points for the Gaussians, we take every 4^{th} iterate. We rotate through our iterates, so the same point is used only once every 4 iterations. The Gaussians are implicitly regularized during the factorization at each step, ensuring that the algorithm doesn't crash in a singular configuration. We stop the iteration after a fixed 10 steps. This stopping criterion is arbitrary, but it is hard to give a better one, as the EM iteration tends to

become numerically unstable when running too many iterations (at least this was our observation on our test cases). In particular, in the case where the iteration converges to a singular model, more and more useful information is lost to regularization with each iteration step. In addition, stopping after a fixed number of steps also ensures reasonably bounded computation time.

As for the local search, we minimize this density estimator using a third-party local optimizer. In this case, we optimize over the x coordinates only. The optimization is unconstrained in the case without implicit equality constraints, because inequality constraints are ignored by design, both due to the inability to determine the feasible region reliably in the absence of global information and because evaluating the constraint in infeasible areas can give us useful information to approximate it within the feasible region, especially at the boundary. As our starting point, we pick the center of the box. We use the derivative information we explicitly computed in section 2.2.4. As for the local search, we accept the optimum $\hat{x}$ as our next iterate.

We also experimented with approaches based on cluster analysis and subsequent density estimation within the clusters (by simply computing the mean and covariance within the cluster and using a Gaussian distribution with these two parameters, or rather the sum of these Gaussians over all clusters, as the estimator). However, with both attempts at cluster analysis we tried:

- a simple approach based on the minimum distance μ between two points, where two points were defined as close to each other if their (Euclidean) distance did not exceed $\alpha\mu$ for some tunable constant α, and the transitive hull of this closeness relation formed the equivalence relation whose equivalence classes were defined as our clusters,

- a hierarchical approach suggested by Prof. Neumaier, based on a potential function $V(C) = \sum_{x \in C} \|x - \bar{x}\|$ and a cost function $c(C_1 \cup C_2) = V(C_1 \cup C_2) - V(C_1) - V(C_2)$ where two clusters C_1 and C_2 were merged if the cost $c(C_1 \cup C_2)$ of the merger did not exceed a threshold (fixed again to $\alpha\mu$ with some α), the starting configuration being of one point per cluster.

We found that the parameter α was impossible to tune to produce satisfying clusters in a sufficiently general way (and using a more complex criterion would just have lead to even more magic parameters). Therefore, we decided to go with the EM procedure which gives us both the clusters and the density estimator at the same time and for which the resulting Gaussian mixture model satisfies a provable optimality criterion (maximum expectation) at the fixed point, if any (which also means the clustering is optimal in the sense of this criterion).

3.2.5 Implicit Equality Constraints

A very hard problem in black box optimization is how to handle implicit equality constraints, i.e. our $F_2(x) = 0$, in an efficient way, the (ideal) goal being not to blow up the search space more than if the equality constraint was explicit. For our local search, this goal was easily reached by substituting $y_2 = 0$ in the resulting surrogate problem, which eliminates the variables y_2 created by the implicit constraints completely. When doing global search, however, the implicit equality constraints are a much more serious problem, and we can no longer reach our goal by a simple substitution. In fact, we can't reach it fully at all, we can only approximate it. However, we have seen in our experiments that it is still essential to do this, as without this treatment, all the global search was wasted on infeasible areas, handling implicit equality constraints specially helped a lot.

During global search, we do not want to have to search in variables which are determined by constraints. This is because the goal of global search is to fill the gaps in the search space we can control. Explicit constraints make this easy: we just search in the space spanned by the independent variables and evaluate at the resulting point to get the values of the dependent variables. With implicit constraints, however, we cannot simply do this, because a relation like $F(x, y) = 0$ doesn't give us **any** information on what y values, if any, are valid for a given x, especially in our black box algorithm where we do not have any idea about what F looks like. The only assumption we can reasonably make is that each implicit equality constraint corresponds to a hypersurface (i.e. a submanifold of dimension one less than the dimension of the containing space) or a union of hypersurfaces in our space of independent variables x. While there are counterexamples even for that (just consider the obvious case $F \equiv 0$ which results in the trivial constraint $0 = 0$, or the case $F \equiv 1$ which results in the always infeasible $1 = 0$), most practical implicit equality constraints form such hypersurfaces, therefore our heuristics are tuned for the common case. (This is in contrast to implicit inequality constraints which do not reduce the dimension of the search space except in degenerate cases.) In this case, the ideal goal would be to eliminate one dimension from the search space for each equality constraint. Given that this is not possible for the reasons discussed above, our algorithm aims instead at making the search space as narrow as possible in the excess dimension. This is done by introducing quadratic constraints in the global search which attempt to enclose the correct equality constraints from both directions. Since we cannot guarantee that the enclosures will always be rigorous (due to the lack of global information), we rectify our enclosures over time, such that all already computed points are always within the enclosures for the constraints (ignoring rounding errors as everywhere else in the algorithm). At the same time, we generate additional enclosures with every new point we retain, whether it results from a local or a global search (but not for

the starting points).

The general approach to obtain such enclosures was suggested by Stefan Vigerske, who is using it successfully in his LaGO optimizer [41, 42]. However, we adapted his approach for our problem. At each new point, we generate two quadratic estimates for each implicit equality constraint, one from above and one from below. (Unlike LaGO, we don't sample new points for the sole purpose of improving the constraint estimates, but instead compute the optimal enclosures from the points we get during the main algorithm. The worst which can happen from lacking constraint information is to land at an infeasible point, which will naturally help improving our enclosures. Function evaluations being expensive, we cannot afford doing additional sampling at the expense of local search.) Consider the implicit equality constraint $F_{2l}(x) = 0$. We start by computing enclosures

$$\underline{F}(x) = \sum_{i,j} \underline{a}_{ij} x_i x_j + \sum_i \underline{b}_i x_i + \underline{c}$$

and

$$\overline{F}(x) = \sum_{i,j} \overline{a}_{ij} x_i x_j + \sum_i \overline{b}_i x_i + \overline{c}$$

which are the best in the sense of the linear programs

$$\begin{array}{rl} \max & \sum_k \underline{F}(x_k) \\ \text{s.t.} & \forall k : \underline{F}(x_k) \le F_{2l}(x_k) \\ & \underline{F}(x_{\text{new}}) = F_{2l}(x_{\text{new}}) \end{array}$$

resp.

$$\begin{array}{rl} \min & \sum_k \overline{F}(x_k) \\ \text{s.t.} & \forall k : \overline{F}(x_k) \ge F_{2l}(x_k) \\ & \overline{F}(x_{\text{new}}) = F_{2l}(x_{\text{new}}) \end{array}$$

or in coordinates:

$$\begin{array}{rl} \max & \sum_{i,j} \left(\sum_k x_{ki} x_{kj}\right) \underline{a}_{ij} + \sum_i \left(\sum_k x_{ki}\right) \underline{b}_i + N\underline{c} \\ \text{s.t.} & \forall k : \sum_{i,j} x_{ki} x_{kj} \underline{a}_{ij} + \sum_i x_{ki} \underline{b}_i + \underline{c} \le F_{2l}(x_k) \\ & \sum_{i,j} x_{\text{new}i} x_{\text{new}j} \underline{a}_{ij} + \sum_i x_{\text{new}i} \underline{b}_i + \underline{c} = F_{2l}(x_{\text{new}}) \end{array}$$

resp.

$$\begin{array}{rl} \min & \sum_{i,j} \left(\sum_k x_{ki} x_{kj}\right) \overline{a}_{ij} + \sum_i \left(\sum_k x_{ki}\right) \overline{b}_i + N\overline{c} \\ \text{s.t.} & \forall k : \sum_{i,j} x_{ki} x_{kj} \overline{a}_{ij} + \sum_i x_{ki} \overline{b}_i + \overline{c} \ge F_{2l}(x_k) \\ & \sum_{i,j} x_{\text{new}i} x_{\text{new}j} \overline{a}_{ij} + \sum_i x_{\text{new}i} \overline{b}_i + \overline{c} = F_{2l}(x_{\text{new}}) \end{array}$$

where N is the number of points, i.e. $N = \sum_k 1$. For the first enclosure (using the data from the starting points), the last constraint, which requires an exact fit at the new point, is omitted. The linear programs are solved using the `lp_solve`

library. Unlike LaGO, which proceeds to computing convex enclosures out of these, we work with the (in general nonconvex) quadratic enclosures directly.

We then apply the following relaxation: let's assume the enclosure $\underline{F}(x) \leq F(x) \leq \overline{F}(x)$ holds for all x. Then the pair of constraints $\overline{F}(x) \geq 0$ and $\underline{F}(x) \leq 0$ forms a relaxation for the equality constraint $F(x) = 0$. We apply this to our constraint $F_{2l}(x) = 0$, replacing it with the two constraints $\overline{F}(x) \geq -\tau$ and $\underline{F}(x) \leq \tau$, where $\tau \geq 0$ is a small heuristic tolerance compensating for the fact that our enclosures may be off due to lack of information. We add these two constraints to our global search minimization problem, which is then solved in two steps (as doing it in one step didn't achieve satisfying convergence with the two supported local optimizers): we first run the global search with only bound constraints, starting at the center of the box, then we use this point as the starting point for the fully constrained global search.

We never throw away enclosures, instead we accumulate more and more of them as the algorithm proceeds. We do, however, correct them if we find them to be wrong, i.e. if we find a new point x' with $F_{2l}(x') < \underline{F}(x')$ or $F_{2l}(x') > \overline{F}(x')$. This is simply done by adjusting the constant term $\underline{c}$ resp. $\overline{c}$ by the amount needed to make the new point fit, i.e. to obtain $F_{2l}(x') = \underline{F}(x')$ resp. $F_{2l}(x') = \overline{F}(x')$.

Chapter 4

Implementation

This chapter details our implementation of the algorithm we have just described. It also contains excerpts from the source code, which can be obtained at `http://www.tigen.org/kevin.kofler/bbowda/`. We will start by documenting the files describing the input model, which have to be filled in by the user. In a second section, we describe our implementation of the algorithm itself. In a third section, we describe the abstraction used to interface our algorithm with third-party local NLP optimizers.

The implementation is implemented in the ISO C99 language [43, 44] and licensed under the GNU General Public License, version 3 [45] or later, with special exceptions allowing to link with the third-party optimizers used.

4.1 Copyright and License Notice

The exact copyright notice and license terms for the implementation are reproduced hereforth:

```
/* bbowda - Black Box Optimization With Data Analysis
   Copyright (C) 2006-2007 Kevin Kofler <Kevin@tigcc.ticalc.org>

   This program is free software: you can redistribute it and/or modify
   it under the terms of the GNU General Public License as published by
   the Free Software Foundation, either version 3 of the License, or
   (at your option) any later version. A copy of the GNU General Public
   License version 3 can be found in the file gpl-3.0.txt.

   Linking bbowda statically or dynamically (directly or indirectly) with
   other modules is making a combined work based on bbowda. Thus, the terms
   and conditions of the GNU General Public License cover the whole
   combination.
```

```
In addition, as a special exception, the copyright holder of bbowda gives
you permission to combine the bbowda program:
* with free software programs or libraries that are released under the
  GNU Library or Lesser General Public License (LGPL), either version 2
  of the License, or (at your option) any later version,
* with free software programs or libraries that are released under the
  IBM Common Public License (CPL), either version 1.0 of the License, or
  (at your option) any later version,
* with code included in the standard release of MUMPS under the MUMPS
  Conditions of Use as reproduced in licenses.txt (or modified versions
  of such code, with unchanged license) and
* if you qualify for a free of charge license of DONLP2, with code
  included in the standard release of DONLP2 under the DONLP2 Conditions
  of Use as reproduced in licenses.txt (or modified versions of such code,
  with unchanged license).

You may copy and distribute such a system following the terms of the GNU
GPL for bbowda and the licenses of the other code concerned, provided that
you include the source code of that other code when and as the GNU GPL
requires distribution of source code.

Note that people who make modified versions of bbowda are not obligated
to grant this special exception for their modified versions; it is their
choice whether to do so. The GNU General Public License gives permission
to release a modified version without this exception; this exception also
makes it possible to release a modified version which carries forward
this exception.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program.  If not, see <http://www.gnu.org/licenses/>. */
```

4.2 Required User Input

User input has to be provided in two C files, one header and one source file, and compiled with the project. Problem dimensions have to be specified in a header file `probldim.h` because many array sizes in the source code depend on these. The actual implementation of the function must be provided as C source code in `problem.c`. Of course, this does not mean the full implementation has to be in C. It is also possible to call external routines or read input from an external source, such as the console, a file or a network, from within `problem.c`. The program must then be recompiled with the user input (using `make` to build against DONLP2 or

`make -f Makefile.ipopt` to build against Ipopt) and run: `./exe`.

We will explain the required input on the following example, which is the *dispatch* example from *GLOBALLib* [46], with some hand-added bounds:

$$
\begin{aligned}
&\min y_1 \\
\text{s.t.}\quad y_1 &= .00533x_1^2 + 11.669x_1 + .00889x_2^2 + 10.333x_2 + .00741x_3^2 \\
&\quad +10.833x_3 \\
y_2 &= x_1 + x_2 + x_3 - x_4 \\
0 &= .000766x_1 + .0000342x_2 - .000189x_3 + x_4 - .000676x_1^2 \\
&\quad -.000521x_2^2 - .000294x_3^2 - .0000953x_1x_2 + .0000507x_1x_3 \\
&\quad -.0000953x_2x_1 - .0000901x_2x_3 + .0000507x_3x_1 \\
&\quad -.0000901x_3x_2 - .040357 \\
x_1 &\in [50, 200] \\
x_2 &\in [37.5, 150] \\
x_3 &\in [45, 180] \\
x_4 &\in [-200, 320] \\
y_1 &\in [1000, 7000] \\
y_2 &\in [210, 730]
\end{aligned}
$$

4.2.1 Problem Dimensions (probldim.h)

```
/* USER INPUT: Problem dimensions */
#define DIMX 4
#define DIMY 2
#define DIMY_EQ 1
```

The first required input is the dimensions of the problem. `DIMX` is the number of x (independent) variables. `DIMY` is the number of y variables, i.e. the number of variables given by explicit equality conditions. `DIMY_EQ` is the number of implicit equality conditions.

```
/* USER INPUT: Number of starting points */
#define NUMINITPTS 0
```

It is possible to provide some starting points, e.g. if a good starting point is known from the application. Here, we set `NUMINITPTS` to 0, which means all starting points will be automatically generated through the procedure described in section 3.2.2.

```
/* USER INPUT: Maximum points to evaluate */
#define MAXPTS 100
```

This important parameter tells the algorithm how many function evaluations can be used. Each new point found by local or global search requires one function evaluation. Note that extrapolation may use one additional function evaluation, so expect a maximum of MAXPTS + 1 function evaluations if DIMY_EQ $\neq 0$.

```
/* USER INPUT: Tolerance for optimum feasibility */
#define OPTIMUM_TOL .001
```

In black box optimization, it is usually not possible to guarantee exact feasibility, especially in the presence of implicit equality constraints. This parameter tells the algorithm by how much each bound or implicit equality constraint can be violated before a point has to be declared infeasible. Infeasible points will never be reported as a solution. This tolerance is also used during the extrapolation step. Due to the construction of the algorithm, the bounds for the x variables should never be violated, but the y bounds and the implicit equality constraints may be.

```
/* USER INPUT: If defined, ignore equality constraints for global search */
#undef GLOBAL_SEARCH_IGNORES_EQ_CONSTRAINTS
```

This option may be used to force the global search to search the entire x space even in the presence of implicit equality constraints. Use of this option is not recommended.

```
/* USER INPUT: Tolerance for the constraints estimating the equality constraints
               during global search */
#define ESTIMATE_CONSTRAINT_TOL .01
```

This tolerance is used when handling implicit equality constraints to compensate for approximation errors made when computing the enclosures. It is the parameter τ from section 3.2.5. This parameter is set here in order to allow it to be tuned for a specific problem, we recommend using .01 as in this example.

4.2.2 Problem Implementation (problem.c)

```
/* min cT (x, F(x)) */
DOUBLE c[DIMX+DIMY]={0.,0.,0.,0.,1,0.,};
```

These are the coefficients of the variables in the objective function, first for x, then for y. The objective function is required to be linear. If your objective function is nonlinear (or if the coefficients are not known), substitute a variable y_i for it (or its nonlinear part) and introduce an explicit equality constraint for y_i. In this example, the objective function is simply y_1 (the result of the preceding transformation).

```
/* s.t. Flow <= F(x) <= Fup */
DOUBLE Flow[DIMY]={1000,210,};
DOUBLE Fup[DIMY]={7000,730,};
```

These are the bounds for the y variables. They are required to be finite, but they may be almost arbitrarily large (everything below $2 \cdot 10^{19}$ should be safe).

```
/*      xlow <= x <= xup */
DOUBLE xlow[DIMX]={50,37.5,45,-200,};
DOUBLE xup[DIMX]={200,150,180,320,};
```

And these are the bounds for the x variables. These are also required to be finite, and it is important for them to be as close together as possible because these are the bounds defining the search space for the global search. Thus, if you know there cannot be any feasible points with x_i smaller or larger than a given value, always set the bounds accordingly to ensure maximum efficiency.

```
/* starting points */
DOUBLE initpts[NUMINITPTS][DIMX]={};
```

Starting points (x coordinates only), if any, may be provided here. The number of starting points given here must match the number `NUMINITPTS` given in `probldim.h`. Starting points should be located within the bounds for the x variables, but are otherwise not required to be feasible.

```
/* evaluate F(x) */
void evaluate_F(const DOUBLE *x, DOUBLE *F)
{
  *(F++) = 11.669 * x[0] + 10.333 * x[1] + 10.833 * x[2]
           + .00533 * pow(x[0], 2) + .00889 * pow(x[1], 2)
           + .00741 * pow(x[2], 2);
  *(F++) = x[0] + x[1] + x[2] - x[3];
  *(F++) = .000766 * x[0] + 3.42e-05 * x[1] - .000189 * x[2] + x[3]
           - .000676 * pow(x[0], 2) - .000521 * pow(x[1], 2)
           - .000294 * pow(x[2], 2) - .0001906 * x[0] * x[1]
           + 5.07e-05 * x[0] * x[2] + 5.07e-05 * x[0] * x[2]
           - .0001802 * x[1] * x[2] - .040357;
}
```

This is the callback which evaluates the constraints. It takes a vector `x` (containing the independent variables x) as input and must produce an output vector `F`. The input has the dimension `DIMX`, the output must have the dimension `DIMY+DIMY_EQ` and be correctly sorted (explicit constraints first, implicit equality constraints second). **Please keep in mind that the first index in a C array is 0, not 1!**

In this example, the output vector is simply computed through explicit formulas. In general, however, it may be any arbitrary C function, which implies external subroutines in any language which can be linked to C may be used, I/O may be performed etc. This is the **only** callback through which user-provided functions are evaluated, in particular, **no** gradient information is required or used.

4.3 Implementation of the Algorithm

Our implementation is divided into one source file for each logical module: `main.c` contains the main function, `covar.c` contains functions to compute local covariance models for the local search, `eqconst.c` encloses implicit equality constraints, `gmmem.c` contains functions to compute global GMMs for the global search using EM iteration and `eval.c` contains functions evaluating different properties of the model, such as the current best point. The implementation also calls into the user-provided model as described in the previous section and the interface to third-party NLP optimizers as described in the next section.

4.3.1 Main Function (main.c)

Our main function implements the overall structure of the algorithm as described in section 3.2.1.

```
static const int enable_global_search=TRUE;
```

This variable allows disabling the global search. It is retained mainly for experimental purposes. All our results are obtained with the global search enabled.

```
int main(void) {
  size_t i;
  int ret;

  /* ensure reproducible results */
  srand(31337);
```

We initialize the pseudorandom number generator to a known constant seed in order to ensure we will get the same results over several runs of the algorithm, as true randomness would make debugging very difficult and bring no gains in practice.

```
  /* evaluate constraints at starting points */
  init_points();
```

Next, we evaluate the constraints at the user-provided starting point. The actual implementation of this function is located in `eval.c`.

```
/* if we don't have enough starting points, generate more */
#if DIMX<5
  #define N_START (DIMX+DIMY+DIMY_EQ+6)
#elif DIMX<10
  #define N_START (DIMX+DIMX+DIMY+DIMY_EQ+2)
#else
  #define N_START (4*DIMX+DIMY+DIMY_EQ-16)
#endif
if (numcurrpts < N_START) {
  /* consider a grid of (N_START*2)*(N_START*2) points
     don't generate 2 points on the same xk */
  unsigned char used_x[DIMX][N_START*2]={};
  /* mark all the already used xk */
  size_t i, k, n_used_x[DIMX]={};
  DOUBLE delta_x[DIMX];
  for (k=0; k<(size_t)DIMX; k++) {
    delta_x[k]=(xup[k]-xlow[k])/(DOUBLE)(N_START*2-1);
  }
  for (i=0; i<numcurrpts; i++) {
    /* round the given starting points to the closest x and y on the grid */
    DOUBLE *x=currpts[i];
    for (k=0; k<(size_t)DIMX; k++) {
      if (delta_x[k]!=0.) { /* filter out equality bounds */
        long j=lround((x[k]-xlow[k])/delta_x[k]);
        if (j<0) { /* sanity check */
          printf("WARNING: Starting value %lf out of bounds [%lf,%lf].\n",
                 x[k],xlow[k],xup[k]);
          j=0;
        }
        if (j>=2*N_START) { /* sanity check */
          printf("WARNING: Starting value %lf out of bounds [%lf,%lf].\n",
                 x[k],xlow[k],xup[k]);
          j=2*N_START-1;
        }
        if (!used_x[k][j]) n_used_x[k]++;
        used_x[k][j]=TRUE;
      }
    }
  }
  /* now loop until we have enough starting points */
  while (numcurrpts < N_START) {
    /* generate a starting point */
    /* try 10 points, pick the one the farthest away */
    DOUBLE new_x[DIMX], dist=-INFINITY;
    size_t t, j[DIMX];
    for (t=0; t<10; t++) {
      DOUBLE new_x_t[DIMX], dist_t=INFINITY;
      size_t j_t[DIMX], i;
      /* generate point */
```

```
      for (k=0; k<(size_t)DIMX; k++) {
        /* equiprobability among the not yet used xk */
        size_t j,jtk_index=((unsigned long long)rand()
                            *(unsigned long
                              long)((size_t)(N_START*2)-n_used_x[k]))
                           /((unsigned long long)(RAND_MAX)+1ull);
        for (j=0; j<(size_t)(N_START*2); j++) {
          if (!used_x[k][j]) {
            if (!jtk_index--) {
              new_x_t[k]=xlow[k]+(DOUBLE)j*delta_x[k];
              j_t[k]=j;
              break;
            }
          }
        }
        /* compute squared distance to the closest point */
        for (i=0; i<numcurrpts; i++) {
          DOUBLE *x=currpts[i], dist_t_i=0.;
          for (k=0; k<(size_t)DIMX; k++) {
            dist_t_i+=(new_x_t[k]-x[k])*(new_x_t[k]-x[k]);
          }
          if (dist_t_i<dist_t) dist_t=dist_t_i;
        }
        /* keep the point if it's better */
        if (dist_t>dist) {
          dist=dist_t;
          memcpy(j,j_t,(size_t)DIMX*sizeof(size_t));
          memcpy(new_x,new_x_t,(size_t)DIMX*sizeof(DOUBLE));
        }
      }
      /* mark the xk[j] as used */
      for (k=0; k<(size_t)DIMX; k++) {
        used_x[k][j[k]]=TRUE;
        n_used_x[k]++;
      }
      /* evaluate constraints at the new starting point */
      printf("Generated starting point x=[");
      if (DIMX) {
        printf("%lf",new_x[0]);
      }
      for (i=1; i<(size_t)DIMX; i++) {
        printf(",%lf",new_x[i]);
      }
      printf("]\n");
      new_point(new_x);
    }
  }
```

This section implements the algorithm for starting point generation described in section 3.2.2. We first create our grid, but as already discussed there, we do not actually compute all the exponentially many points on the grid. Instead, we only keep one vector by dimension to keep track of the already used indices. First, we fill in the indices already used by the user-provided starting points (when rounding to the nearest points on the grid as discussed in section 3.2.2), then we generate the actual points. We also warn about user-provided starting points which aren't within the bounds for the x variables if we detect any during the rounding procedure.

```
/* compute first global over-/underestimates of equality constraints */
compute_1st_global_eq_cst_estimates();
```

Next, we call this function from `eqconst.c` to compute a first enclosure for the implicit equality constraints from the startup points (both the user-provided ones and the automatically computed ones). The function is a noop if there are no implicit equality constraints.

```
/* optimize until we have reached the maximum number of evaluations */
while (numcurrpts < MAXPTS) {
```

Now that everything is set up, we can start the main loop of our implementation.

```
  if (enable_global_search && (numcurrpts & 1)) { /* global (gapfilling)
   search */
    /* compute a density function (gmm) */
    build_density_gmm();
    /* minimize the density function under the constraints */
    ignore_constraints=TRUE;
    if (num_estimate_constraints) {
      ret=solve_nlp();
      printf("Global presearch found x=[");
      if (DIMX) {
        printf("%lf",optimum_x[0]);
      }
      for (i=1; i<(size_t)DIMX; i++) {
        printf(",%lf",optimum_x[i]);
      }
      printf("] (reason: %d)\n",ret);
      ignore_constraints=FALSE;
    }
    ret=solve_nlp();
    /* free the density function parameters */
    free_gmm();
    printf("Global search found x=[");
```

This section implements the global search as described in section 3.2.4. The functions called are implemented in `gmmem.c`, except for `solve_nlp`, which is part of the interface to the NLP libraries. See the documentation of `gmmem.c` for details.

```
    } else { /* local search */
      /* choose best point */
      get_best_point();
      /* build regularized covariance model around the point */
      build_local_regcovar_model();
      /* optimize the covariance model */
      ret=solve_nlp();
      printf("Local search found x=[");
    }
```

This section implements the local search as described in section 3.2.3. The `get_best_point` function is implemented in `eval.c`, `build_local_regcovar_model` in `covar.c` and `solve_nlp` is part of the interface to the NLP libraries. Again, details can be found in the documentation of the respective functions.

```
    if (DIMX) {
      printf("%lf",optimum_x[0]);
    }
    for (i=1; i<(size_t)DIMX; i++) {
      printf(",%lf",optimum_x[i]);
    }
    printf("] (reason: %d)\n",ret);
    /* evaluate constraints at optimum */
    new_point(optimum_x);
  }
```

This section is common to the local and global searches. It outputs the point found by the search, then calls the `new_point` function from `eval.c` to retain it. `new_point` also takes care of evaluating the constraints at the new point and, in the presence of implicit equality constraints, updating the enclosures for those.

```
#if DIMY_EQ
  extrapolate_point();
#endif
```

If we have implicit equality constraints, we try extrapolating a feasible point from near-feasible ones with good objective function values. See the documentation of `extrapolate_point` in `eval.c` for details.

```
  printf("Optimum: %lf at x=[",get_optimum_x());
  if (DIMX) {
```

```
    printf("%lf",optimum_x[0]);
  }
  for (i=1; i<(size_t)DIMX; i++) {
    printf(",%lf",optimum_x[i]);
  }
  printf("]\n");

  return 0;
}
```

Finally, we output the optimal point found and its objective function value and return success. Only points feasible up to the user-provided tolerance `OPTIMUM_TOL` are considered. `get_optimum_x` is implemented in `eval.c`.

4.3.2 Local Covariance Models (covar.c)

This file creates the surrogate models for the local search as described in section 3.2.3. The actual optimization is then done by the NLP optimizer. We assume a "best point" has already been picked, this is the task of `get_best_point` in `eval.c`. ("Best point" is in quotes because a Pareto filter approach is used, meaning there is no one best point.)

```
/* Cholesky factor of the covariance matrix */
static DOUBLE L[DIMX+DIMZ+DIMY+DIMY_EQ][DIMX+DIMZ+DIMY+DIMY_EQ];
/* Transpose of L, reversed, to avoid cache misses in compute_MX */
static DOUBLE LTrev[DIMX+DIMZ+DIMY+DIMY_EQ][DIMX+DIMZ+DIMY+DIMY_EQ];
/* Copy of currpts sorted by the norm of the equality constraint violation */
static DOUBLE (*usedpts)[DIMX+DIMY+DIMY_EQ]=NULL;
/* Full number of points in usedpts */
static size_t usedpts_size;
/* Number of points in usedpts which should actually be used */
static size_t numusedpts;
```

These variables are used internally in the functions below. `L` is the Cholesky factor of the covariance matrix, it should only be used through the public function `compute_MX`. `LTrev` is used for performance reasons: while profiling, we found the function `compute_MX` to be called very often, and solving the upper triangular system $L^T u = v$ was found to cause many cache misses when done using `L`. Thus, it pays to rearrange the matrix `L` for more efficient cache use. `numusedpts` is the number of points actually considered for the covariance model, after eliminating those rejected due to an excessive implicit equality constraint violation as described in section 3.2.3. The `usedpts` array is a sorted version of the global `currpts` array, so that we only have to loop through the first `numusedpts` elements of `usedpts` to enumerate the elements actially used, `usedpts_size` keeps track of its number of

elements so new points can be inserted using insertion sort rather than having to resort the entire array each time.

```
/* Global variables */
DOUBLE Xbar[DIMX+DIMZ+DIMY+DIMY_EQ];
DOUBLE klow;
DOUBLE kup;
```

These global variables are part of the public interface. They represent $\bar{X}$, k_{low} and k_{up} in the covariance model, respectively.

```
static void for_each_used(DOUBLE *acc, DOUBLE *aux, void (*update)(const DOUBLE
 *, DOUBLE *, DOUBLE *))
{
  size_t j;
  for (j=0; j<numusedpts; j++) {
    update(usedpts[j],acc,aux);
  }
}
```

This is an internal utility function which calls a function through the function pointer `update` for each actually used point, passing two pointers to `DOUBLE` passed as arguments through so statistical measures such as sums can be accumulated.

```
/* MX = C^-1 X = (L LT)^-1 X = L^-T L^-1 X */
void compute_MX(const DOUBLE *X, DOUBLE *MX)
{
  size_t i,j,k;
  /* MX := L^-1 X */
  for (i=0; i<(size_t)(DIMX+DIMZ+DIMY+DIMY_EQ); i++) {
    MX[i]=X[i];
    for (j=0; j<i; j++) {
      MX[i]-=L[i][j]*MX[j];
    }
    MX[i]/=L[i][i];
  }
  /* MX := L^-T X */
  /* try hard to avoid cache misses */
  for (k=0; k<(size_t)(DIMX+DIMZ+DIMY+DIMY_EQ); k++) {
    j=i;
    DOUBLE d=LTrev[k][--i];
    for (; j<(size_t)(DIMX+DIMZ+DIMY+DIMY_EQ); j++) {
      MX[i]-=LTrev[k][j]*MX[j];
    }
    MX[i]/=d;
  }
}
```

This public function computes the matrix-vector product $MX = C^{-1}X = L^{-T}L^{-1}X$ needed in the covariance model by solving two triangular systems of linear equations. As discussed above, we use the rearranged matrix `LTrev` in the second loop to reduce the number of cache misses.

```
/* empirical formula for the weight:
   * our point weights decrease with the 6th power of the distance
   * the worse the point, the less it will be weighted
   weight = (||x-xbest||^2)^(-6)/(1/10+(goodness-best_goodness))^(1/2) */
static DOUBLE weight_of_point(const DOUBLE *x)
{
  size_t i;
  DOUBLE goodness=get_point_goodness(x), weight=0.;
  for (i=0; i<(size_t)DIMX; i++) {
    DOUBLE d=best_point[i]-x[i];
    weight += d*d;
  }
  if (weight == 0.) return 0.;
  weight = pow(weight,-3.);
  weight /= sqrt(.1+(goodness-best_goodness));
  return weight;
}
```

This internal function computes the weight of a point in the weighted covariance model using the empirical formula indicated in the comment. The best point has weight zero, it is used to center the covariance model instead.

```
static void compute_covar(const DOUBLE *x, DOUBLE *Xbar, DOUBLE *count)
{
  size_t i,j,k,l,a,b;
  DOUBLE weight=weight_of_point(x);
  for (i=0; i<(size_t)DIMX; i++) {
    for (k=0; k<=i; k++) {
      L[i][k] += weight*(x[i]-Xbar[i])*(x[k]-Xbar[k]);
    }
  }
  for (i=0,a=(size_t)DIMX; i<(size_t)DIMX; i++) {
    for (j=0; j<=i; j++,a++) {
      for (k=0; k<(size_t)DIMX; k++) {
        L[a][k] += weight*(x[i]*x[j]-Xbar[a])*(x[k]-Xbar[k]);
      }
      for (k=0,b=(size_t)DIMX; b<=a; k++) {
        for (l=0; l<=k && b<=a; l++,b++) {
          L[a][b] += weight*(x[i]*x[j]-Xbar[a])*(x[k]*x[l]-Xbar[b]);
        }
      }
    }
  }
```

```
  for (i=0; i<(size_t)(DIMY+DIMY_EQ); i++) {
    for (k=0; k<(size_t)DIMX; k++) {
      L[i+DIMX+DIMZ][k] += weight*(x[i+DIMX]-Xbar[i+DIMX+DIMZ])*(x[k]-Xbar[k]);
    }
    for (k=0,b=(size_t)DIMX; k<(size_t)DIMX; k++) {
      for (l=0; l<=k; l++,b++) {
        L[i+DIMX+DIMZ][b] +=
         weight*(x[i+DIMX]-Xbar[i+DIMX+DIMZ])*(x[k]*x[l]-Xbar[b]);
      }
    }
    for (k=0; k<=i; k++) {
      L[i+DIMX+DIMZ][k+DIMX+DIMZ] +=
       weight*(x[i+DIMX]-Xbar[i+DIMX+DIMZ])*(x[k+DIMX]-Xbar[k+DIMX+DIMZ]);
    }
  }
  (*count)+=weight;
}
```

This internal function is used together with `for_each_used` to compute the weighted covariance matrix and the weighted count, i.e. the sum of the weights, of the used points, with the given mean `Xbar`. The covariance matrix is stored in `L` because it will be factored in place.

```
static void compute_kup_klow(const DOUBLE *x, DOUBLE *dmax, DOUBLE *aux
 ATTR_UNUSED)
{
  /* compute k = XT M X */
  DOUBLE X[DIMX+DIMZ+DIMY+DIMY_EQ], MX[DIMX+DIMZ+DIMY+DIMY_EQ], *p=X, k=0.,
   d=0.;
  size_t i,j;
  for (i=0; i<(size_t)DIMX; i++) {
    d += (x[i]-best_point[i])*(x[i]-best_point[i]);
  }
  if (d>*dmax) return;
  for (i=0; i<(size_t)DIMX; i++) {
    *(p++) = x[i];
  }
  for (i=0; i<(size_t)DIMX; i++) {
    for (j=0; j<=i; j++) {
      *(p++) = x[i]*x[j];
    }
  }
  for (i=0; i<(size_t)(DIMY+DIMY_EQ); i++) {
    *(p++) = x[DIMX+i];
  }
  for (i=0; i<(size_t)(DIMX+DIMZ+DIMY+DIMY_EQ); i++) {
    X[i] -= Xbar[i];
  }
```

```
  compute_MX(X,MX);
  for (i=0; i<(size_t)(DIMX+DIMZ+DIMY+DIMY_EQ); i++) {
    k += X[i]*MX[i];
  }

  /* update kup, klow */
  if (k>kup) kup=k;
  if (k<klow) klow=k;
}
```

This internal function is used together with `for_each_used` to compute the lower bound `klow` and the upper bound `kup` for the covariance model. We ignore points which are more than `dmax` away from the best point, evaluate the covariance model for the remaining ones, and retain the smallest result as `klow` and the largest one as `kup`.

```
static int compare_DOUBLE(const void *p, const void *q)
{
  return (*(DOUBLE *)p>*(DOUBLE *)q)-(*(DOUBLE *)p<*(DOUBLE *)q);
}
```

This is the `qsort` callback used to sort distances.

```
static int compare_yeq(const void *p, const void *q)
{
  size_t i;
  DOUBLE yeq1=0., yeq2=0.;
  // compute norm(p[yeq])^2 and norm(q[yeq])^2
  for (i=(size_t)(DIMX+DIMY); i<(size_t)(DIMX+DIMY+DIMY_EQ); i++) {
    yeq1+=i[(DOUBLE*)p]*i[(DOUBLE*)p];
    yeq2+=i[(DOUBLE*)q]*i[(DOUBLE*)q];
  }
  // compare them
  return (yeq1>yeq2)-(yeq1<yeq2);
}
```

This is the `qsort` callback used to sort implicit equality constraint violations.

```
void build_local_regcovar_model(void)
{
```

This is the main function of `covar.c`.

```
  if (DIMY_EQ) {
    if (usedpts) {
      /* make room for the new points in usedpts */
      usedpts=xrealloc(usedpts,
```

```
      numcurrpts*((size_t)(DIMX+DIMY+DIMY_EQ))*sizeof(DOUBLE));
    /* sort the new points by the norm of the equality constraint violation
       using insertion sort */
    size_t i,j;
    for (i=usedpts_size; i<numcurrpts; i++) {
      /* This could be done more efficiently using binary search, but the
         memmove is O(n) anyway. We can't use bsearch as it only returns exact
         matches. */
      for (j=0; j<i; j++) {
        if (compare_yeq(currpts[i],usedpts[j])>=0) break;
      }
      memmove(usedpts+j+1,usedpts+j,
       (i-j)*((size_t)(DIMX+DIMY+DIMY_EQ))*sizeof(DOUBLE));
      memcpy(usedpts[j],currpts[i],
       ((size_t)(DIMX+DIMY+DIMY_EQ))*sizeof(DOUBLE));
    }
  } else {
    /* allocate a copy of currpts */
    usedpts=xmalloc(numcurrpts*((size_t)(DIMX+DIMY+DIMY_EQ))*sizeof(DOUBLE));
    memcpy(usedpts,currpts,
     numcurrpts*((size_t)(DIMX+DIMY+DIMY_EQ))*sizeof(DOUBLE));
    /* sort the points by the norm of the equality constraint violation */
    qsort(usedpts,numcurrpts,(size_t)(DIMX+DIMY+DIMY_EQ)*sizeof(DOUBLE),
     compare_yeq);
  }
  usedpts_size=numcurrpts;
  /* use only the first half unless we don't have enough points */
  numusedpts=(numcurrpts>=28)?(numcurrpts>>1):numcurrpts;
```

As described in section 3.2.3, if we have implicit equality constraints, we do not use all the points, but only the half with the smallest implicit equality constraint violations, except at the beginning where we have less than 28 points (an arbitrary threshold). Therefore, we maintain `usedpts` as a sorted copy of `currpts`, which allows simply looping through the first `numusedpts` points of `usedpts` to enumerate the points which shall be used. The passage above takes care of this sorting.

```
} else {
  numusedpts=numcurrpts;
  usedpts=currpts;
}
```

If we do not have implicit equality constraints, we can simply use the original `currpts` array directly.

```
/* compute the covariance matrix */
size_t i,j,k;
DOUBLE n=0., Cdiag[DIMX+DIMZ+DIMY+DIMY_EQ], d[numusedpts], dmax;
```

```
for (k=0; k<(size_t)DIMX; k++) {
  Xbar[k]=best_point[k];
}
for (i=0; i<(size_t)DIMX; i++) {
  for (j=0; j<=i; j++) {
    Xbar[k++]=best_point[i]*best_point[j];
  }
}
for (k=0; k<(size_t)(DIMY+DIMY_EQ); k++) {
  Xbar[k+DIMX+DIMZ]=best_point[k+DIMX];
}
for (i=0; i<(size_t)(DIMX+DIMZ+DIMY+DIMY_EQ); i++) {
  for (j=0; j<=i; j++) {
    L[i][j]=0.;
  }
}
for_each_used(Xbar,&n,compute_covar);
for (i=0; i<(size_t)(DIMX+DIMZ+DIMY+DIMY_EQ); i++) {
  for (j=0; j<=i; j++) {
    L[i][j]/=n;
  }
}
```

Next, we compute the actual covariance matrix using the `for_each_used` function with the `compute_covar` callback. The algorithm used is simply the definition of the covariance matrix, where the mean is forced to the best point.

```
/* save the diagonal of the matrix */
for (j=0; j<(size_t)(DIMX+DIMZ+DIMY+DIMY_EQ); j++) {
  Cdiag[j]=L[j][j];
}
```

We save the diagonal of the unfactored covariance matrix in a local array because we need it for regularization.

```
/* compute the regularized Cholesky factorization of the matrix */
for (j=0; j<(size_t)(DIMX+DIMZ+DIMY+DIMY_EQ); j++) {
  DOUBLE Ljj;
  /* regularize */
  if (L[j][j]<=sqrt(DBL_EPSILON)*Cdiag[j]) {
    L[j][j]=(Cdiag[j]==0.)?1.:(sqrt(DBL_EPSILON)*Cdiag[j]);
  }
  /* in higher dimensions, regularize harder */
  if (DIMX > 6)
    L[j][j]+=sqrt(DBL_EPSILON)*Cdiag[j];

  Ljj=L[j][j];
  for (i=j+1; i<(size_t)(DIMX+DIMZ+DIMY+DIMY_EQ); i++) {
```

```
      for (k=i; k<(size_t)(DIMX+DIMZ+DIMY+DIMY_EQ); k++) {
        L[k][i]-=L[k][j]*L[i][j]/Ljj;
      }
    }
    for (k=j; k<(size_t)(DIMX+DIMZ+DIMY+DIMY_EQ); k++) {
      L[k][j]/=sqrt(Ljj);
    }
  }
```

Next, we compute the Cholesky factorization of the covariance matrix. The factorization is computed in place. If we detect a diagonal element which is too small, we regularize the factorization. This process uses the diagonal elements C_{jj} (`Cdiag[j]`) of the unfactored C saved above: We check if $L_{jj} \leq \sqrt{\texttt{DBL\_EPSILON}}\, C_{jj}$. If it is, we replace it with $\sqrt{\texttt{DBL\_EPSILON}}\, C_{jj}$, or with the arbitrary 1 if $C_{jj} = 0$. This corresponds to adding $\sqrt{\texttt{DBL\_EPSILON}}\, C_{jj} - L_{jj}$ or 1 to the original diagonal element C_{jj}. In higher dimensions, we add an additional $\sqrt{\texttt{DBL\_EPSILON}}\, C_{jj}$ to the diagonal.

```
  /* compute the reversed transpose */
  for (i=0; i<(size_t)(DIMX+DIMZ+DIMY+DIMY_EQ); i++) {
    for (j=0; j<=i; j++) {
      LTrev[(size_t)(DIMX+DIMZ+DIMY+DIMY_EQ-1)-j][i]=L[i][j];
    }
  }
```

We also compute the rearranged version `LTrev` of L.

```
  if (numusedpts<=(DIMX<<1)) {
    dmax=INFINITY;
  } else {
    /* compute distances from best point */
    for (j=0; j<numusedpts; j++) {
      d[j]=0.;
      for (i=0; i<(size_t)DIMX; i++) {
        d[j]+=(usedpts[j][i]-best_point[i])*(usedpts[j][i]-best_point[i]);
      }
    }
    /* sort them */
    qsort(d,numusedpts,sizeof(DOUBLE),compare_DOUBLE);
    /* take the 2*DIMX+1st entry as dmax, i.e. take at least 2*DIMX+1 points
     into
       account */
    dmax=d[DIMX<<1];
  }
```

We then compute the maximum distance up to which points are considered for `klow` and `kup`, in order to obtain a sufficiently local covariance model. As described

in section 3.2.3, we consider the 2 DIMX + 1 points closest to the best point (where in case of a tie we accept all points with the same distance dmax), or all used points if we don't have 2 DIMX + 1 of them.

```
  /* compute kup, klow */
  kup=-INFINITY;
  klow=INFINITY;
  for_each_used(&dmax,NULL,compute_kup_klow);
```

Using this information, we compute klow and kup.

```
  /* fudge kup for higher dimensions so we get reasonably-sized trust regions,
     otherwise the local search gets stuck in a non-optimal point */
  kup *= pow(2.,(DOUBLE)(DIMX-1));
```

This trick works around a problem we encountered during testing: Our covariance models not only approximate the functions, but also implicitly define a trust region to which the local search is restricted. Unfortunately, in higher dimensions, these trust regions turn out way too small, causing the algorithm to get stuck in a non-stationary point and fail to converge to the actual minimum. (We observed this phenomenon with a simple convex quadratic function.) To compensate for this effect, we multiply the upper bound for our error ellipsoid by a correction factor of $2^{\texttt{DIMX}-1}$.

```
  /* set the NLP solver to optimize the local covariance model */
  optimization_problem=0;
}
```

Finally, we tell our NLP solver interface that the next problem to solve is a local covariance model.

4.3.3 Equality Constraint Approximation (eqconst.c)

This file contains functions to enclose implicit equality constraints as described in section 3.2.5. The term **estimate constraints** will be used in this section to refer to the computed estimated enclosures for the original implicit equality constraints which are used during the global search. As described in section 3.2.5, the estimate constraints are quadratic. In the actual practical implementation, it may happen that solving the linear programs for the estimate constraints runs into numerical difficulties, in which case we simply skip the offending enclosure, as the worst, this can lead to, is a larger search space as necessary for the global search.

```
/* number of over-/underestimate constraints */
size_t num_estimate_constraints=0;
/* coefficients of over-/underestimate constraints */
DOUBLE (*estimate_constraint_coeffs)[1+DIMX+DIMZ]=NULL;
```

These public globals keep track of the number and the coefficients of the estimate constraints. As the constraints are quadratic, we need to save $1 + \texttt{DIMX} + \texttt{DIMZ}$ coefficients to fully represent one such constraint, where `DIMX` is the dimension of the x variables and `DIMZ` the number of quadratic and bilinear products of x variables.

```
static size_t *n_estimated_constraint=NULL;
static unsigned char *is_overestimator=NULL;
```

These internal arrays keep track of which original constraint is being approximated by the estimate constraint and whether it is an underestimator or an overestimator. This information is used when rectifying enclosures revealed incorrect by new computed points. It is necessary to maintain this data because estimate constraints may not be retained in the case of numerical difficulties, and thus the number of the estimate constraint is not sufficient to deduce this information.

```
static void eqconst_optimize_lp(const DOUBLE *X, const int over, const size_t
 constraint_no)
{
```

This internal function generates the linear program for one estimate constraint and calls the `lp_solve` library to solve it. The required input is the number of the constraint to approximate, whether it is an underestimator or an overestimator and the point, if any, at which the fit should be exact. The parameter `X` may be `NULL` to indicate that no exact fit is needed. This feature is used when computing the initial estimates from the starting points.

```
  lprec *lp;
  REAL row[DIMX+DIMZ+2]={}; /* must be 1 more then number of columns ! */
  /* (Sigh. I can understand DONLP2 using 1-based indexing, being ported from
     Fortran, but this makes just no sense. Index 0 is not used at all.) */
```

The linear program to be passed to `lp_solve` is passed in these variables. A feature of the `lp_solve` interface which may be unexpected to C programmers is that it uses index 1 as the base of its arrays even in the C interface, thus the comment warning about this unusual interface.

```
  /* create a new LP model */
  lp=make_lp(0,1+DIMX+DIMZ);
  if (!lp) {
    fprintf(stderr, "make_lp failed\n");
    exit(1);
  }

  /* For the underestimate, we want to maximize to get the best (closest) one.
```

```
    For the overestimate, we want to minimize to get the best (closest) one. */
  if (!over) set_maxim(lp);

  /* bounds: upper defaults to +Inf, which is OK, lower defaults to 0, we need
   -Inf */
  size_t i;
  REAL infinite=get_infinite(lp); /* lp_solve uses a fake infinity */
  for (i=1; i<=(size_t)(1+DIMX+DIMZ); i++) {
    set_lowbo(lp,i,-infinite);
  }

  /* objective function
     coefficient of qi: sum phi_i(xj) for all j */
  size_t j;
  row[1]=numcurrpts; /* constant term */
  for (j=0; j<numcurrpts; j++) {
    REAL *p=row+2;
    DOUBLE *x=currpts[j];
    size_t k,l;
    for (k=0; k<(size_t)DIMX; k++) { /* linear terms */
      *(p++)+=x[k];
    }
    for (k=0; k<(size_t)DIMX; k++) { /* quadratic/bilinear terms */
      for (l=0; l<=k; l++) {
        *(p++)+=(x[k]*x[l]);
      }
    }
  }
  if (!set_obj_fn(lp,row)) {
    fprintf(stderr, "set_obj_fn failed\n");
    exit(1);
  }

  /* constraints
     coefficient of qi in constraint j: phi_i(xj) */
  for (j=0; j<numcurrpts; j++) {
    REAL *p=row+1;
    DOUBLE *x=currpts[j];
    size_t k,l;
    *(p++)=1.; /* constant term */
    for (k=0; k<(size_t)DIMX; k++) { /* linear terms */
      *(p++)=x[k];
    }
    for (k=0; k<(size_t)DIMX; k++) { /* quadratic/bilinear terms */
      for (l=0; l<=k; l++) {
        *(p++)=(x[k]*x[l]);
      }
    }
    if (!add_constraint(lp,row,(X &&
```

```
    x==X)?EQ:(over?GE:LE),x[DIMX+DIMY+constraint_no])) {
     fprintf(stderr, "add_constraint failed\n");
     exit(1);
   }
 }
```

Next, we set up the problem using `lp_solve`'s interfaces. As said in section 3.2.5, the problem formulation is

$$\begin{array}{rl} \max & \sum_{i,j}\left(\sum_k x_{ki}x_{kj}\right)\underline{a}_{ij} + \sum_i\left(\sum_k x_{ki}\right)\underline{b}_i + N\underline{c} \\ \text{s.t.} & \forall k : \sum_{i,j} x_{ki}x_{kj}\underline{a}_{ij} + \sum_i x_{ki}\underline{b}_i + \underline{c} \le F_{2l}(x_k) \\ & \sum_{i,j} x_{\text{new}i}x_{\text{new}j}\underline{a}_{ij} + \sum_i x_{\text{new}i}\underline{b}_i + \underline{c} = F_{2l}(x_{\text{new}}) \end{array}$$

for underestimators and

$$\begin{array}{rl} \min & \sum_{i,j}\left(\sum_k x_{ki}x_{kj}\right)\overline{a}_{ij} + \sum_i\left(\sum_k x_{ki}\right)\overline{b}_i + N\overline{c} \\ \text{s.t.} & \forall k : \sum_{i,j} x_{ki}x_{kj}\overline{a}_{ij} + \sum_i x_{ki}\overline{b}_i + \overline{c} \ge F_{2l}(x_k) \\ & \sum_{i,j} x_{\text{new}i}x_{\text{new}j}\overline{a}_{ij} + \sum_i x_{\text{new}i}\overline{b}_i + \overline{c} = F_{2l}(x_{\text{new}}) \end{array}$$

for overestimators, where N is the number of points, i.e. $N = \sum_k 1$, or `numcurrpts` in the code. x_{new} corresponds to the function parameter `X`.

```
/* solve the problem */
set_verbose(lp,IMPORTANT);
int ret=solve(lp);
if (ret<0) {
  fprintf(stderr, "lp_solve failed\n");
  exit(1);
}
if (ret)
  printf("lp_solve returned non-zero retval %d\n",ret);
```

We then solve the problem and check the return value from `lp_solve`. According to `lp_solve`'s documentation, a negative return value indicates an out-of-memory condition, a positive return value for a continuous (not mixed-integer) linear program indicates an infeasible, unbounded or degenerate problem or a failure to reach the optimum solution due to numerical difficulties. We treat the out-of-memory error as a hard error, and the positive return values by skipping the unusable estimate constraint.

```
else {
  /* save the result */
  get_variables(lp,row);
  size_t j=num_estimate_constraints++,i;
  estimate_constraint_coeffs=xrealloc(estimate_constraint_coeffs,
    num_estimate_constraints*((size_t)(1+DIMX+DIMZ))*sizeof(DOUBLE));
```

```
    n_estimated_constraint=xrealloc(n_estimated_constraint,
                                    num_estimate_constraints*sizeof(size_t));
    n_estimated_constraint[j]=constraint_no;
    is_overestimator=xrealloc(is_overestimator,num_estimate_constraints);
    is_overestimator[j]=over;
    REAL *p=row;
    DOUBLE *q=estimate_constraint_coeffs[j];
    /* transform constraints to >=0 form */
    if (over) { /* overestimator >= 0 */
      for (i=0; i<(size_t)(1+DIMX+DIMZ); i++) *(q++)=*(p++);
    } else { /* underestimator <= 0 */
      for (i=0; i<(size_t)(1+DIMX+DIMZ); i++) *(q++)=-*(p++);
    }
  }
```

If `lp_solve` was successful, we save the result in the global array of estimate constraints. In order to have to maintain only one array of constraints, we flip the sign of the underestimators so we have only ≥ 0 constraints.

```
  /* clean up */
  delete_lp(lp);
}
```

Finally, we clean up by freeing the problem we allocated.

```
static void eqconst_optimize_lps(const DOUBLE *X, const int over)
{
  size_t i;
  for (i=0; i<(size_t)DIMY_EQ; i++) {
    eqconst_optimize_lp(X,over,i);
  }
}
```

This internal function generates all underestimate or overestimate constraints for a new point `X`, or the initial ones if `X` is `NULL`, by calling `eqconst_optimize_lp` repeatedly, once for each implicit equality constraint. The parameters `X` and `over` are simply passed through to `eqconst_optimize_lp`.

```
/* compute global over-/underestimates of equality constraints around a new
   point, or the first estimate if X is NULL */
void compute_global_eq_cst_estimates_around(const DOUBLE *X)
{
#ifndef GLOBAL_SEARCH_IGNORES_EQ_CONSTRAINTS
  if (X) {
    /* fix previous estimates for the new point */
    size_t j,k,l;
    for (j=0; j<num_estimate_constraints; j++) {
```

```
      /* evaluate quadratic polynomial */
      DOUBLE *p=estimate_constraint_coeffs[j], *q=p;
      DOUBLE z=*(p++); /* constant term */
      for (k=0; k<DIMX; k++) { /* linear terms */
        z+=*(p++) * X[k];
      }
      for (k=0; k<DIMX; k++) { /* quadratic/bilinear terms */
        for (l=0; l<=k; l++) {
          z+=*(p++) * (X[k]*X[l]);
        }
      }
      DOUBLE zactual=X[DIMX+DIMY+n_estimated_constraint[j]];
      /* check if the constraint is violated */
      if (is_overestimator[j]) {
        /* should be an overestimator */
        if (z<zactual) { /* oops */
          *q+=zactual-z; /* fix it */
        }
      } else {
        /* -z should be an underestimator */
        if (-z>zactual) { /* oops */
          *q-=zactual+z; /* fix it */
        }
      }
    }
  }

  eqconst_optimize_lps(X,0); /* compute underestimate */
  eqconst_optimize_lps(X,1); /* compute overestimate */
#endif
}
```

This is the main function of `eqconst.c`. If the option `GLOBAL_SEARCH_IGNORES_EQ_CONSTRAINTS` is set, it simply does nothing. Otherwise, it will first rectify the estimate constraints which are violated by the new point `X` (except when computing the first estimates), then compute the new underestimates and overestimates by calling `eqconst_optimize_lps` twice. It shall be noted that this also ends up doing nothing if there are no implicit equality constraints, as the loop in `eqconst_optimize_lps` will be empty, thus no estimate constraints will be generated at all, and therefore the rectifying loop will also be empty.

As described in section 3.2.5, incorrect enclosures are rectified by changing the constant term by the amount needed to make them fit. This is verified by evaluating the quadratic at the new point, then looking up the original constraint and the direction of approximation which were used to generate it, looking up the actual value of the implicit equality constraint at the point (which was evaluated

during the function evaluation done as the point was retained) and comparing.

A convenience macro

```
#define compute_1st_global_eq_cst_estimates() \
  compute_global_eq_cst_estimates_around(NULL)
```

is defined in the corresponding header file `eqconst.h` for the case of computing the initial estimates. The main reason for this macro is to let the function call be self-documenting instead of having to pass a magic `NULL`.

4.3.4 Global GMMs and EM Iteration (gmmem.c)

This file creates the density models for the global search as described in section 3.2.4. The actual optimization is then done by the NLP optimizer. Implicit equality constraints are not handled in this file, they are taken care of by `eqconst.c`, which is documented in the previous section.

```
static const size_t n_steps=10;
```

This arbitrary constant sets the number of steps in the EM iteration.

```
size_t num_gaussians;
static DOUBLE *GMM_P;
static DOUBLE (*GMM_L)[DIMX][DIMX];
static DOUBLE (*GMM_Xbar)[DIMX];
```

The above variables are the parameters of the Gaussian mixture model

$$p(x) = \sum_{j=1}^{M} P(j) \frac{e^{-\frac{\left(x-\bar{X}_j\right)^T C_j^{-1}\left(x-\bar{X}_j\right)}{2}}}{(2\pi)^{\frac{N}{2}}\sqrt{det(C_j)}}.$$

They represent M and the arrays of $P(j)$, the Cholesky factors L_j of C_j and $\bar{X}_j$, respectively.

```
/* MX = C^-1 X = (L LT)^-1 X = L^-T L^-1 X */
static void compute_MiX(const DOUBLE *X, DOUBLE *MX, size_t idx)
{
  size_t i,j;
  /* MX := L^-1 X */
  for (i=0; i<(size_t)DIMX; i++) {
    MX[i]=X[i];
    for (j=0; j<i; j++) {
      MX[i]-=GMM_L[idx][i][j]*MX[j];
    }
    MX[i]/=GMM_L[idx][i][i];
```

```
  }
  /* MX := L^-T X */
  /* i is unsigned and will wrap to max size_t after reaching 0 */
  for (i=(size_t)DIMX-1; i<(size_t)DIMX; i--) {
    for (j=i+1; j<(size_t)DIMX; j++) {
      MX[i]-=GMM_L[idx][j][i]*MX[j];
    }
    MX[i]/=GMM_L[idx][i][i];
  }
}
```

As for the local search, this function computes the matrix-vector product $C_i^{-1}X$ needed in each term of the GMM by solving two triangular systems of linear equations. Unlike its equivalent for the local search, this function is not part of the public interface, it is intended to be called only indirectly through the functions `gmm_prob`, `gmm_grad` and `gmm_hess` below, which evaluate the GMM's value, gradient and Hessian, respectively, at a given point.

```
/* p = exp(-1/2 * (X-Xbar)T M (X-Xbar)) / ((2*pi)^(N/2) * det(L)) */
static DOUBLE gmm_prob_i(const DOUBLE *X, size_t idx)
{
  size_t i;
  DOUBLE x[DIMX],Mx[DIMX],p=0.;

  /* (X-Xbar)T M (X-Xbar) */
  for (i=0; i<(size_t)DIMX; i++) {
    x[i]=X[i]-GMM_Xbar[idx][i];
  }
  compute_MiX(x,Mx,idx);
  for (i=0; i<(size_t)DIMX; i++) {
    p+=x[i]*Mx[i];
  }

  /* exp(-1/2 * (X-Xbar)T M (X-Xbar)) / ((2*pi)^(N/2)) */
  p=exp((-.5)*p)/pow(2.*M_PI,(DOUBLE)DIMX*.5);

  /* divide by det(L) */
  for (i=0; i<(size_t)DIMX; i++) {
    p/=GMM_L[idx][i][i];
  }

  /* if p is infinite, use our fake infinity instead */
  if (isinf(p)) p=INF;

  return p;
}
```

This internal function computes each term of the GMM by its definition, given above. There is one possible source of numerical failure in this function: if $p_0 =$

$x^T Mx$ is a huge negative number, then $e^{-\frac{p_0}{2}}$ will overflow. We catch this and just set p to a huge finite number (the same one used to represent infinite bounds in the NLP optimizers) in that case. This avoids NaNs which would get propagated throughout the entire GMM in the next EM iteration step, rendering it unusable.

```
DOUBLE gmm_prob(const DOUBLE *X)
{
  DOUBLE p=0.;
  size_t idx;
  for (idx=0; idx<num_gaussians; idx++) {
    p+=gmm_prob_i(X,idx)*GMM_P[idx];
  }
  return p;
}
```

This function computes the probability of a point according to the GMM, i.e. the GMM's value at that point, by its definition, given above. We simply loop through all terms, call `gmm_prob_i` for each and sum them all up, with the weights given by `GMM_P`.

```
/* g = - (exp(-1/2 * (X-Xbar)T M (X-Xbar)) * M (X-Xbar)) / ((2*pi)^(N/2) *
 det(L))
     = -p * M (X-Xbar)
   We have to sum this over all Gaussians. */
void gmm_grad(const DOUBLE *X, DOUBLE *g)
{
  size_t idx,i;
  for (i=0; i<(size_t)DIMX; i++) {
    g[i]=0.;
  }

  for (idx=0; idx<num_gaussians; idx++) {
    DOUBLE x[DIMX],Mx[DIMX],p;

    /* add -p * M (X-Xbar) to g */
    for (i=0; i<(size_t)DIMX; i++) {
      x[i]=X[i]-GMM_Xbar[idx][i];
    }
    compute_MiX(x,Mx,idx);
    p=gmm_prob_i(X,idx)*GMM_P[idx];
    for (i=0; i<(size_t)DIMX; i++) {
      g[i]-=p*Mx[i];
    }
  }
}
```

This function computes the gradient of the GMM at the given point through the formula derived in section 2.2.4. The gradients are needed for the NLP optimizers.

```
#define DIMH ((DIMX*(DIMX+1))>>1)

/* H = -p * M - g (M (X-Xbar))T
     = -p * M + p * (M (X-Xbar)) * (M (X-Xbar))T
     = p * (M (X-Xbar) (M (X-Xbar))T - M)
   We have to sum this over all Gaussians. */
void gmm_hess(const DOUBLE *X, DOUBLE *H)
{
  size_t idx,i,j;
  for (i=0; i<(size_t)DIMH; i++) {
    H[i]=0.;
  }

  for (idx=0; idx<num_gaussians; idx++) {
    DOUBLE x[DIMX],Mx[DIMX],Mi[DIMX],p;
    DOUBLE *ptr=H;

    /* compute M (X-Xbar) */
    for (i=0; i<(size_t)DIMX; i++) {
      x[i]=X[i]-GMM_Xbar[idx][i];
    }
    compute_MiX(x,Mx,idx);
    p=gmm_prob_i(X,idx)*GMM_P[idx];
    /* compute Hidx */
    for (i=0; i<DIMX; i++) {
      for (j=0; j<DIMX; j++) {
        x[j]=(j==i)?1.:0.;
      }
      /* Midx is symmetric, so we'll get away with computing columns instead of
       rows */
      compute_MiX(x,Mi,idx);
      for (j=0; j<=i; j++) {
        /* Hidx[i][j] = p * ((Midx (X-Xbar))[i] * (Midx (X-Xbar))[j] -
         Midx[i][j]) */
        *(ptr++)+=p*(Mx[i]*Mx[j]-Mi[j]);
      }
    }
  }
}
```

This function computes the Hessian of the GMM at the given point through the formula derived in section 2.2.4. As the Hessian is symmetric, only half of the matrix is actually computed. The Hessians can be used for the NLP optimization if Ipopt is used.

```
void build_density_gmm(void)
{
  size_t n,curridx;
```

```
DOUBLE (*weights)[numcurrpts];

/* Initialize GMM */
num_gaussians=numcurrpts>>2;
GMM_P=xmalloc(num_gaussians*sizeof(DOUBLE));
GMM_L=xmalloc(num_gaussians*(size_t)DIMX*(size_t)DIMX*sizeof(DOUBLE));
GMM_Xbar=xmalloc(num_gaussians*(size_t)DIMX*sizeof(DOUBLE));
/* Take every 4th point as starting point for a cluster, rotating points.
   (The same point is used only every 4 iterations.) */
for (curridx=0, n=((numcurrpts-1)&3); curridx<num_gaussians; curridx++, n+=4)
 {
  size_t i,j;
  /* Default GMM means to starting points. */
  memcpy(GMM_Xbar[curridx],currpts[n],(size_t)DIMX*sizeof(DOUBLE));
  /* Default GMM priors to 1/num_gaussians. */
  GMM_P[curridx]=1./(DOUBLE)num_gaussians;
  /* Default GMM covariances to the identity matrix. */
  for (i=0; i<(size_t)DIMX; i++) {
    for (j=0; j<i; j++) {
      GMM_L[curridx][i][j]=0.;
    }
    GMM_L[curridx][i][i]=1.;
  }
}

/* Allocate weight matrix */
weights=xmalloc(num_gaussians*numcurrpts*sizeof(DOUBLE));

/* EM iteration */
for (n=0; n<n_steps; n++) {
  size_t p;
  /* E Step: compute weights */
  for (p=0; p<numcurrpts; p++) {
    /* P(j|x)=p(x|j)*P(j)/p(x), p(x)=sum(j) p(x|j)*P(j) */
    DOUBLE px=0.;
    for (curridx=0; curridx<num_gaussians; curridx++) {
      weights[curridx][p]=gmm_prob_i(currpts[p],curridx)*GMM_P[curridx];
      px+=weights[curridx][p];
    }
    if (px<DBL_EPSILON) { /* avoid division by zero, default to
     equiprobability */
      for (curridx=0; curridx<num_gaussians; curridx++) {
        weights[curridx][p]=1./(DOUBLE)num_gaussians;
      }
    } else {
      for (curridx=0; curridx<num_gaussians; curridx++) {
        weights[curridx][p]/=px;
      }
    }
```

```
}

/* M Step: compute new covariances */
for (curridx=0; curridx<num_gaussians; curridx++) {
  /* compute the covariance matrix */
  size_t i,j,k;
  DOUBLE n=0.,real_n;
  for (i=0; i<(size_t)DIMX; i++) {
    GMM_Xbar[curridx][i]=0.;
  }
  for (p=0; p<numcurrpts; p++) {
    n+=weights[curridx][p];
  }
  real_n=n;
  if (!n) {
    n=(DOUBLE)numcurrpts;
    for (p=0; p<numcurrpts; p++) {
      weights[curridx][p]=1.;
    }
  }
  for (p=0; p<numcurrpts; p++) {
    size_t l;
    for (l=0; l<(size_t)DIMX; l++) {
      GMM_Xbar[curridx][l] += weights[curridx][p]*currpts[p][l];
    }
  }
  for (i=0; i<(size_t)DIMX; i++) {
    GMM_Xbar[curridx][i]/=n;
  }
  for (i=0; i<(size_t)DIMX; i++) {
    for (j=0; j<=i; j++) {
      GMM_L[curridx][i][j]=0.;
    }
  }
  for (p=0; p<numcurrpts; p++) {
    size_t l,m;
    for (l=0; l<(size_t)DIMX; l++) {
      for (m=0; m<=l; m++) {
        GMM_L[curridx][l][m] +=
         weights[curridx][p]*(currpts[p][l]-GMM_Xbar[curridx][l])
                                                  *(currpts[p][m]-GMM_Xbar[
                                                   curridx][m]);
      }
    }
  }
  for (i=0; i<(size_t)DIMX; i++) {
    for (j=0; j<=i; j++) {
      GMM_L[curridx][i][j]/=n;
    }
```

```
      }

      /* compute the regularized Cholesky factorization of the matrix */
      for (j=0; j<(size_t)DIMX; j++) {
        DOUBLE Ljj;
        /* regularize */
        if (GMM_L[curridx][j][j]<sqrt(DBL_EPSILON)) {
          GMM_L[curridx][j][j]=sqrt(DBL_EPSILON);
        }

        Ljj=GMM_L[curridx][j][j];
        for (i=j+1; i<(size_t)DIMX; i++) {
          for (k=i; k<(size_t)DIMX; k++) {
            GMM_L[curridx][k][i]-=GMM_L[curridx][k][j]*GMM_L[curridx][i][j]/Ljj;
          }
        }
        for (k=j; k<(size_t)DIMX; k++) {
          GMM_L[curridx][k][j]/=sqrt(Ljj);
        }
      }

      /* compute the mixing prior */
      GMM_P[curridx]=real_n/(DOUBLE)numcurrpts;
    }
  }

  /* free weight matrix */
  free(weights);

  /* set the NLP solver to optimize the GMM */
  optimization_problem=1;
}
```

This function creates the GMM. As described in section 3.2.4 and in a comment, we take every 4^{th} iterate as our starting points, rotating through our iterates, so the same point is used only once every 4 iterations. We then run a standard EM iteration as described in section 2.2.5, the number of steps being given by the constant `n_steps` at the beginning of the file. The only deviation from the theoretical EM iteration procedure is that some regularization is done to avoid dividing by zero, both when computing probabilities and when computing the Cholesky factorization. For the probabilities, the divisions would be of $\frac{0}{0}$ type, so we simply fall back to equiprobability if we would divide by zero. For the Cholesky factorization, we have to regularize the covariance matrix by adding to the diagonal. We check if $L_{jj} < \sqrt{\texttt{DBL\_EPSILON}}$ and set it to $\sqrt{\texttt{DBL\_EPSILON}}$ in this case. Finally, we tell our NLP solver interface that the next problem to solve is a global GMM.

```
void free_gmm(void)
{
  free(GMM_P);
  free(GMM_L);
  free(GMM_Xbar);
}
```

This function frees the memory allocated for the computed GMM once it is no longer needed.

4.3.5 Evaluation of the Model (eval.c)

This file contains miscellaneous routines which do not fit into any of the above categories. Most of them are related to evaluating function values or other properties of the model, thus the name.

```
size_t numcurrpts;
DOUBLE (*currpts)[DIMX+DIMY+DIMY_EQ];
```

These global variables keep track of the number of points we evaluated the black box constraints at and the coordinates and function values themselves. They are used all over our implementation.

```
DOUBLE best_goodness;
```

This global variable contains the best "goodness" value encountered, which is used for weighting the local covariance models.

```
DOUBLE *best_point;
```

This global keeps a pointer to the "best point" picked by the Pareto filter, at which the local search is to be started. Note that this is not necessarily the point with the best goodness. Our first approach was to pick that point as the best point, which would have amounted to a penalty approach, however that approach turned out to be too sensitive to the weight given to the constraint violation.

```
void init_points(void)
{
  size_t i;
  numcurrpts=NUMINITPTS;
  currpts=xmalloc((size_t)NUMINITPTS*((size_t)(DIMX+DIMY+DIMY_EQ))*sizeof(DOUBLE
   ));
  for (i=0; i<(size_t)NUMINITPTS; i++) {
    memcpy(currpts[i],initpts[i],DIMX*sizeof(DOUBLE));
    evaluate_F(initpts[i],currpts[i]+DIMX);
  }
}
```

This initialization function is the first routine called as the algorithm starts (after initializing the pseudorandom number generator). It allocates the `currpts` array, stores the user-provided starting points into `currpts` and evaluates the constraints at each of them.

Note that the regular `evaluate_F` callback function is called for the starting points (as for any other point). There is currently no straightforward API to provide the function values at the starting points beforehand if they are known, but the function evaluation callback can be an arbitrary C function and thus it is possible to provide cached or precomputed function values for the starting points in the user-provided `evaluate_F` callback.

```
void new_point(const DOUBLE *x)
{
  size_t i=numcurrpts++;
  currpts=xrealloc(currpts,
   numcurrpts*((size_t)(DIMX+DIMY+DIMY_EQ))*sizeof(DOUBLE));
  memcpy(currpts[i],x,DIMX*sizeof(DOUBLE));
  evaluate_F(x,currpts[i]+DIMX);
  /* compute global over-/underestimates of equality constraints around the new
   point */
  compute_global_eq_cst_estimates_around(currpts[i]);
}
```

This function does the bookkeeping necessary whenever a new point is found. The point is entered into the `currpts` array, the constraint functions are evaluated at the point, and `compute_global_eq_cst_estimates_around` from `eqconst.c` is called to compute the new enclosures for the implicit equality constraints at the new point and update existing enclosures if they are incorrect at the new point.

```
DOUBLE get_optimum_x(void)
{
  size_t j;
  DOUBLE optimum=INFINITY;
#ifdef OPTIMUM_TOL
  const DOUBLE tol=OPTIMUM_TOL;
#else
  const DOUBLE tol=sqrt(DBL_EPSILON);
#endif
  for (j=0; j<numcurrpts; j++) {
    DOUBLE cTx=0.;
    DOUBLE *x=currpts[j];
    size_t i;
    for (i=0; i<(size_t)DIMX; i++) {
      if (x[i]<xlow[i] || x[i]>xup[i]) {
        goto infeasible; /* infeasible point */
      }
```

```
    }
    for (i=0; i<(size_t)DIMY; i++) {
      if (x[DIMX+i]<Flow[i]-tol || x[DIMX+i]>Fup[i]+tol) {
        goto infeasible; /* infeasible point */
      }
    }
    for (i=0; i<(size_t)DIMY_EQ; i++) {
      if (fabs(x[DIMX+DIMY+i])>=tol) {
        goto infeasible; /* infeasible point */
      }
    }
    for (i=0; i<(size_t)(DIMX+DIMY); i++) {
      cTx+=c[i]*x[i];
    }
    if (cTx<optimum) {
      memcpy(optimum_x,x,(size_t)DIMX*sizeof(DOUBLE));
      optimum=cTx;
    }
    infeasible:;
  }
  return optimum;
}
```

This function is called at the end of the algorithm to determine the optimum x out of the computed points. The optimum point is defined as the point among the ones feasible up to the user-provided `OPTIMUM_TOL` with the smallest objective function value. The return value is the function value at the optimal point, the actual coordinates are stored in the global variable `optimum_x`.

```
/* The weights are determined using a penalty function approach, with a penalty
   term increasing over time:
   min cT x + numcurrpts * constraint_violation */
static DOUBLE get_point_penalty(const DOUBLE *x)
{
  DOUBLE penalty=0.;
  size_t i;
  for (i=0; i<(size_t)DIMX; i++) {
    if (x[i]<xlow[i]) {
      penalty+=numcurrpts*(xlow[i]-x[i]);
    } else if (x[i]>xup[i]) {
      penalty+=numcurrpts*(x[i]-xup[i]);
    }
  }
  for (i=0; i<(size_t)DIMY; i++) {
    if (x[DIMX+i]<Flow[i]) {
      penalty+=numcurrpts*(Flow[i]-x[DIMX+i]);
    } else if (x[DIMX+i]>Fup[i]) {
      penalty+=numcurrpts*(x[DIMX+i]-Fup[i]);
```

```
    }
  }
  for (i=0; i<(size_t)DIMY_EQ; i++) {
    penalty+=(numcurrpts<18?sqrt(numcurrpts/18.):
                  numcurrpts<77?numcurrpts/18.:
                  numcurrpts*sqrt(numcurrpts)/158.)*fabs(x[DIMX+DIMY+i]);
  }
  return penalty;
}
```

This internal utility function is used when computing the goodness of a point (used to weight the local covariance models) and as the constraint violation term for the Pareto filter in `get_best_point`. It computes the penalty term $e(x) = N\mathrm{cv}_1(x) + \kappa(N)\|F_2(x)\|_1$, where N is the number of computed points `numcurrpts` and cv_1, κ and F_2 are as defined in section 3.2.3.

```
static DOUBLE get_point_constraint_violation(const DOUBLE *x)
{
  DOUBLE penalty=0.;
  size_t i;
  for (i=0; i<(size_t)DIMX; i++) {
    if (x[i]<xlow[i]) {
      penalty+=(xlow[i]-x[i]);
    } else if (x[i]>xup[i]) {
      penalty+=(x[i]-xup[i]);
    }
  }
  for (i=0; i<(size_t)DIMY; i++) {
    if (x[DIMX+i]<Flow[i]) {
      penalty+=(Flow[i]-x[DIMX+i]);
    } else if (x[DIMX+i]>Fup[i]) {
      penalty+=(x[DIMX+i]-Fup[i]);
    }
  }
  for (i=0; i<(size_t)DIMY_EQ; i++) {
    penalty+=fabs(x[DIMX+DIMY+i]);
  }
  return penalty;
}
```

This internal utility function is similar to the above, however it computes the unweighted constraint violation $\mathrm{cv}(x) = \mathrm{cv}_1(x) + \|F_2(x)\|_1$. It is used for extrapolation.

```
static DOUBLE get_point_cTx(const DOUBLE *x)
{
  DOUBLE cTx=0.;
```

```
  size_t i;
  for (i=0; i<(size_t)(DIMX+DIMY); i++) {
    cTx+=c[i]*x[i];
  }
  return cTx;
}
```

This internal utility function evaluates the objective function $c^T\binom{x}{y}$ (where $y = F_1(x)$) at a point x.

```
DOUBLE get_point_goodness(const DOUBLE *x)
{
  return get_point_cTx(x)+get_point_penalty(x);
}
```

This function computes the goodness of a point, i.e. the function $p(x)$ defined in section 3.2.3. It is simply the sum of the objective function and the penalty term.

```
void get_best_point(void)
{
  static unsigned char *used=NULL;
  static size_t used_size=0;
  /* use the penalty to compute the optimum goodness, it's used for weighting */
  DOUBLE cTx[numcurrpts], penalty[numcurrpts];
  size_t i,j;
  DOUBLE optimum=INFINITY;
  best_point=NULL;
  /* new points are not used */
  used=xrealloc(used,numcurrpts);
  memset(used+used_size,0,numcurrpts-used_size);
  used_size=numcurrpts;
  for (j=0; j<numcurrpts; j++) {
    DOUBLE cTx_j=get_point_cTx(currpts[j]);
    DOUBLE penalty_j=get_point_penalty(currpts[j]);
    DOUBLE cTx_penalty=cTx_j+penalty_j;
    cTx[j]=cTx_j;
    penalty[j]=penalty_j;
    if (cTx_penalty<optimum) {
      optimum=cTx_penalty;
    }
  }
  best_goodness=optimum;
  /* now pick a "best point" using the filter approach */
  unsigned char pareto_optimal[numcurrpts];
  size_t num_pareto_points=0;
  memset(pareto_optimal,1,numcurrpts);
  for (i=0; i<numcurrpts; i++) {
    for (j=0; j<numcurrpts; j++) {
```

```
      /* if j is strictly better than i, i is not Pareto-optimal */
      if (j!=i && pareto_optimal[j] /* avoid unnecessary comparisons */ &&
          cTx[j] <= cTx[i] && penalty[j] <= penalty[i] &&
          (cTx[j] < cTx[i] || penalty[j] < penalty[i])) {
        pareto_optimal[i]=0;
        goto dont_count;
      }
    }
    num_pareto_points++;
    dont_count:;
  }
  /* pick a pseudorandom Pareto-optimal point as the best point
     pick a later one with a higher probability */
  /* find the most recent Pareto-optimal point */
  /* i is unsigned and will overflow to ULONG_MAX */
  for (i=numcurrpts-1; i<numcurrpts; i--) {
    if (pareto_optimal[i]) break;
  }
  /* if it is not used, pick it with a probability of 1-1/num_pareto_points */
  if (used[i] || rand()<(RAND_MAX/(int)num_pareto_points)) {
    /* otherwise use equiprobability */
    size_t pareto_point_index=((unsigned long long)rand()
                               *(unsigned long long)num_pareto_points)
                              /((unsigned long long)(RAND_MAX)+1ull);
    for (i=0; i<numcurrpts; i++) {
      if (pareto_optimal[i]) {
        if (!pareto_point_index--) break;
      }
    }
  }
  best_point=currpts[i];
  used[i]=1;
  printf("Starting local search at x=[");
  if (DIMX) {
    printf("%lf",best_point[0]);
  }
  for (i=1; i<(size_t)DIMX; i++) {
    printf(",%lf",best_point[i]);
  }
  printf("]\n");
  return;
}
```

This function implements the Pareto filter to pick a "best point" described in section 3.2.3: we consider the points which are Pareto-optimal for the objective function $f(x) = c^T \binom{x}{F_1(x)}$ and the weighted constraint violation $e(x) = N\mathrm{cv}_1(x) + \kappa(N)\|F_2(x)\|_1$ (i.e. the penalty term computed by `get_point_penalty`) over the set of computed points x_k and pick one of these points with the probabilities

indicated in the comments and in section 3.2.3. The static array `used` is used to keep track of which points we have already started a local search at, so we don't give the most recent Pareto-optimal point a higher probability if it has already been used.

```
void extrapolate_point(void)
{
  DOUBLE cTx[numcurrpts], penalty[numcurrpts];
  size_t i,j;
  for (j=0; j<numcurrpts; j++) {
    cTx[j]=get_point_cTx(currpts[j]);
    penalty[j]=get_point_constraint_violation(currpts[j]);
  }
  /* determine the Pareto-optimal points */
  unsigned char pareto_optimal[numcurrpts];
  size_t num_pareto_points=0;
  memset(pareto_optimal,1,numcurrpts);
  for (i=0; i<numcurrpts; i++) {
    for (j=0; j<numcurrpts; j++) {
      /* if j is strictly better than i, i is not Pareto-optimal */
      if (j!=i && pareto_optimal[j] /* avoid unnecessary comparisons */ &&
          cTx[j] <= cTx[i] && penalty[j] <= penalty[i] &&
          (cTx[j] < cTx[i] || penalty[j] < penalty[i])) {
        pareto_optimal[i]=0;
        goto dont_count;
      }
    }
    num_pareto_points++;
    dont_count:;
  }
  /* throw away the points with excess constraint violation
     also throw away the points which are already feasible up to the tolerance,
     it's no use getting any better there */
  for (i=0; i<numcurrpts; i++) {
    if (pareto_optimal[i]
#ifdef OPTIMUM_TOL
        && (penalty[i]>OPTIMUM_TOL*256. || penalty[i]<OPTIMUM_TOL)
#else
        && (penalty[i]>sqrt(DBL_EPSILON)*256. || penalty[i]<sqrt(DBL_EPSILON))
#endif
       ) {
      pareto_optimal[i]=0;
      num_pareto_points--;
    }
  }
  /* do a ratio reject to get rid of outliers */
  if (num_pareto_points>=3) {
    DOUBLE rho[numcurrpts],rho_mean=0.,rho_var=0.;
    for (i=0; i<numcurrpts; i++) {
```

```
  if (pareto_optimal[i]) {
    /* compute rho[i] */
    DOUBLE d_i_NNi=INFINITY, d_NNi_NNNNi=INFINITY;
    size_t NNi;
    /* find nearest neighbor of i and distance */
    for (j=0; j<numcurrpts; j++) {
      if (j!=i && pareto_optimal[j]) {
        DOUBLE d_i_j=0;
        size_t k;
        for (k=0; k<(size_t)DIMX; k++) {
          d_i_j+=(currpts[j][k]-currpts[i][k])*(currpts[j][k]-currpts[i][k])
           ;
        }
        if (d_i_j<d_i_NNi) {
          d_i_NNi=d_i_j;
          NNi=j;
        }
      }
    }
    /* find nearest neighbor of NNi and distance */
    for (j=0; j<numcurrpts; j++) {
      if (j!=i && j!=NNi && pareto_optimal[j]) {
        DOUBLE d_NNi_j=0;
        size_t k;
        for (k=0; k<(size_t)DIMX; k++) {
          d_NNi_j+=(currpts[j][k]-currpts[NNi][k])*(currpts[j][k]-currpts[NN
           i][k]);
        }
        if (d_NNi_j<d_NNi_NNNNi) {
          d_NNi_NNNNi=d_NNi_j;
        }
      }
    }
    /* rho[i] is the ratio */
    rho[i]=d_i_NNi/d_NNi_NNNNi;
    rho_mean+=rho[i];
  }
}
/* compute mean and variance */
rho_mean/=num_pareto_points;
for (i=0; i<numcurrpts; i++) {
  if (pareto_optimal[i]) {
    rho_var+=(rho[i]-rho_mean)*(rho[i]-rho_mean);
  }
}
rho_var/=num_pareto_points;
/* ratio-reject cutoff = mean + 3*stddev */
DOUBLE cutoff=rho_mean+3*sqrt(rho_var);
/* now reject the outliers */
```

```
  for (i=0; i<numcurrpts; i++) {
    if (pareto_optimal[i] && rho[i]>cutoff) {
      printf("Extrapolation: outlier cv=%lf, x=[",penalty[i]);
      if (DIMX) {
        printf("%lf",currpts[i][0]);
      }
      for (j=1; j<(size_t)DIMX; j++) {
        printf(",%lf",currpts[i][j]);
      }
      printf("] rejected (rho=%lf>%lf)\n",rho[i],cutoff);
      pareto_optimal[i]=0;
      num_pareto_points--;
    }
  }
}
/* compute the cubic regression and extrapolate to 0 using Cramer's rule */
DOUBLE S0=0.,S1=0.,S2=0.,S3=0.,S4=0.,S5=0.,S6=0.;
DOUBLE Sx0[DIMX]={},Sx1[DIMX]={},Sx2[DIMX]={},Sx3[DIMX]={};
for (i=0; i<numcurrpts; i++) {
  if (pareto_optimal[i]) {
    DOUBLE cv=penalty[i];
    DOUBLE cvn=cv;
    DOUBLE *x=currpts[i];
    DOUBLE cvxn[DIMX];
    printf("Extrapolation considering cv=%lf, x=[",cv);
    if (DIMX) {
      printf("%lf",x[0]);
    }
    for (j=1; j<(size_t)DIMX; j++) {
      printf(",%lf",x[j]);
    }
    printf("]\n");
    S0+=1.;
    for (j=0; j<DIMX; j++) Sx0[j]+=x[j];
    S1+=cv;
    for (j=0; j<DIMX; j++) {
      cvxn[j]=cv*x[j];
      Sx1[j]+=cvxn[j];
    }
    cvn*=cv;
    S2+=cvn;
    for (j=0; j<DIMX; j++) {
      cvxn[j]*=cv;
      Sx2[j]+=cvxn[j];
    }
    cvn*=cv;
    S3+=cvn;
    for (j=0; j<DIMX; j++) {
      cvxn[j]*=cv;
```

```
        Sx3[j]+=cvxn[j];
      }
      cvn*=cv;
      S4+=cvn;
      cvn*=cv;
      S5+=cvn;
      cvn*=cv;
      S6+=cvn;
    }
  }
  DOUBLE d1=S6*S4*S2+S5*S3*S4+S4*S5*S3-S4*S4*S4-S3*S3*S6-S2*S5*S5;
  DOUBLE d2=S6*S4*S1+S5*S3*S3+S4*S5*S2-S3*S4*S4-S2*S3*S6-S1*S5*S5;
  DOUBLE d3=S6*S3*S1+S5*S2*S3+S4*S4*S2-S3*S3*S4-S2*S2*S6-S1*S4*S5;
  DOUBLE d4=S5*S3*S1+S4*S2*S3+S3*S4*S2-S3*S3*S3-S2*S2*S5-S1*S4*S4;
  DOUBLE D=S0*d1-S1*d2+S2*d3-S3*d4;
  if (D<DBL_EPSILON*128. && D>-DBL_EPSILON*128.) {
    printf("Extrapolation matrix too ill-conditioned (D=%lg)\n",D);
    return;
  }
  for (j=0; j<DIMX; j++) {
    optimum_x[j]=(Sx0[j]*d1-Sx1[j]*d2+Sx2[j]*d3-Sx3[j]*d4)/D;
  }
  printf("Extrapolation found x=[");
  if (DIMX) {
    printf("%lf",optimum_x[0]);
  }
  for (i=1; i<(size_t)DIMX; i++) {
    printf(",%lf",optimum_x[i]);
  }
  printf("]\n");
  new_point(optimum_x);
}
```

This function implements the extrapolation step described in section 3.2.1. We consider only the points which are Pareto-optimal for the objective function $f(x)$ and the constraint violation $\mathtt{cv}(x)$ over the set of computed points. Next, we throw away the points which are already feasible up to `OPTIMUM_TOL`, which need no further extrapolation towards smaller constraint violations, as well as the points with $\mathtt{cv}(x) > 256$ `OPTIMUM_TOL`, which are too far from feasibility to be relevant. (The factor 256 is arbitrary and the result of a few experiments, it might be beneficial to tune it further.) We then apply the ratio-reject procedure from data analysis (see section 2.3) to get rid of outliers. The goal of this procedure is to obtain a set of points which should approximate the theoretical Pareto curve, i.e. the set of Pareto-optimal points over all $x \in [x_l, x_u]$, as closely as possible. Finally, we lay a cubic regression curve through the remaining points using the formulas derived in section 2.4 and use this to extrapolate $\mathtt{cv}(x)$ towards zero. We check

if the determinant is reasonably large to avoid numerical difficulties or outright division by zero. If the extrapolation was successful, i.e. if we have had enough points and a nonsingular extrapolation system, we evaluate the constraints at the extrapolated point to verify actual feasibility.

4.4 Interface to Third-Party NLP Libraries

Our interface to the third-party NLP optimizer libraries serves two purposes:

- abstracting the actual optimizer used, in order to support plugging in either DONLP2 or Ipopt and
- implementing the callbacks required by those optimizers.

The optimizer abstraction is contained in the header file `nlpopt.h` and the wrapper for the respective optimizer (`donlp2.h` or `ipopt.h`). The callbacks are contained in `userfu.c` for DONLP2 and in `ipopt.c` for Ipopt.

Adding support for another NLP optimizer requires both implementing the abstraction and porting or wrapping the callbacks.

4.4.1 Optimizer Abstraction (nlpopt.h)

This file is a thin wrapper around the optimizers, which makes sure other files need not know which NLP optimizer is in use. Its main use is to include the wrapper for the respective optimizer:

```
#ifdef USE_IPOPT
#include "ipopt.h"
#else // default to donlp2
#include "donlp2.h"
#endif
```

It also declares global variables common to both sets of optimizer callbacks:

```
#include "probldim.h"
extern DOUBLE optimum_x[DIMX];
extern int optimization_problem;
extern int ignore_constraints;
```

4.4.2 DONLP2 Wrapper and Callbacks (donlp2.h, userfu.c)

The DONLP2 wrapper `donlp2.h` contains the DONLP2-specific parts of the optimizer abstraction.

```
#include "o8para.h"
```

We start by including the header file provided by DONLP2 itself, which declares the types used by DONLP2, such as `DOUBLE` and `REAL`.

```
#define INF 1.e20
```

This macro corresponds to the large number used to represent infinity within DONLP2.

```
void donlp2(void);
extern REAL optite;
#define solve_nlp() (donlp2(), (int)optite+11)
```

The `solve_nlp` function is the primary abstraction for the optimizer which is being used. For DONLP2, the implementation is a simple macro which calls `donlp2` and translates the return code, which DONLP2 stores into the global variable `optite`, into a C-style return value. The remainder of the work, such as the choice of problem to compute function values and gradients for, is done in the callbacks, because DONLP2 calls the callback functions by name, and can therefore only work with a single set of callbacks.

The file `userfu.c` contains the implementation of the callback functions for DONLP2, which represent the two types of optimization models used during our algorithm: local surrogate models and global density minimization. The interfaces these callbacks are implementing can be found in more detail in the documentation included with DONLP2.

```
DOUBLE optimum_x[DIMX];
int optimization_problem;
int ignore_constraints;
```

The global variable `optimization_problem` controls which type of problem is to be solved next. It is 0 for the local surrogate model

$$
\begin{aligned}
\min \quad & c^T \begin{pmatrix} x \\ y_1 \end{pmatrix} \\
\text{s.t.} \quad & k_{\text{low}} \leq k \left(\begin{pmatrix} x \\ z \\ y_1 \\ 0 \end{pmatrix} \right) \leq k_{\text{up}} \\
& z = (x_{11}\ x_{12}\ x_{22}\ x_{13}\ x_{23}\ x_{33}\ \ldots\ x_{1m} \ldots x_{mm})^T \\
& x_l \leq x \leq x_u \\
& F_l \leq y \leq F_u \\
& z_l \leq z \leq z_u
\end{aligned}
$$

(see section 3.2.3) and 1 for the global density model

$$\begin{array}{ll} \min & \sum_{j=1}^{M} P(j) \frac{e^{-\frac{\left(x-\bar{x}_j\right)^T C_j^{-1}\left(x-\bar{x}_j\right)}{2}}}{(2\pi)^{\frac{N}{2}}\sqrt{det(C_j)}} \\ \text{s.t.} & x_l \leq x \leq x_u \\ & \overline{F}_2(x) \geq -\tau \\ & \underline{F}_2(x) \leq \tau \end{array}$$

(see sections 3.2.4, 3.2.5 and 2.2.4).
The global variable `ignore_constraints` disables the constraints containing $\overline{F}_2(x)$ or $\underline{F}_2(x)$ in the global search, leaving only the bound constraints. This is used to find a suitable starting point for the fully constrained global search. The result is stored in the variable `optimum_x`. Only the x coordinates are retained. The approximation for y_1 obtained from the local search is discarded (as is the information that the approximation for y_2 is always forced to zero by the substitution performed in section 3.2.3), instead the actual $F(x)$ is evaluated at the new point.

```
/* ************************************************************************
 */
/*                              donlp2-intv size initialization
 */
/* ************************************************************************
 */
void user_init_size(void) {
    #define  X extern
    #include "o8comm.h"
    #include "o8fint.h"
    #include "o8cons.h"
    #undef   X


    /* problem dimension n = dim(x), nlin=number of linear constraints
       nonlin = number of nonlinear constraints  */

    switch (optimization_problem) {
      case 0:
        n      = DIMX+DIMZ+DIMY;
        nlin   = 0;
        nonlin = 1+DIMZ;
        break;
      case 1:
        n      = DIMX;
        nlin   = 0;
        nonlin = ignore_constraints ? 0 : num_estimate_constraints;
        break;
    }
```

```
    iterma = 4000;
    nstep = 20;
}
```

This callback tells DONLP2 the dimensions of the problems we want to solve: the number of variables `n`, the number of linear constraints (other than bound constraints) `nlin` and the number of nonlinear constraints `nonlin`, as well as some additional parameters, for which we used the numbers used in the DONLP2 examples: the maximum number of iterations to perform `iterma` and the number of tries in the backtracking allowed `nstep`.

```
/* ************************************************************************
 */
/*                              donlp2-intv standard setup
 */
/* ************************************************************************
 */
void user_init(void) {
    #define  X extern
    #include "o8comm.h"
    #include "o8cons.h"
    #undef   X

    INTEGER  i,j,k,l;
    DOUBLE optimum=INFINITY;

    /* name is ident of the example/user and can be set at users will       */
    /* the first static character must be alphabetic. 40 characters maximum */

    strcpy(name,"surrogate");

    /* x is initial guess and also holds the current solution */
    /* problem dimension n = dim(x), nlin=number of linear constraints
       nonlin = number of nonlinear constraints  */

    analyt = TRUE;
    epsdif = 1.e-16;   /* gradients exact to machine precision */
    /* if you want numerical differentiation being done by donlp2 then:*/
    /* epsfcn   = 1.e-16; *//* function values exact to machine precision */
    /* taubnd   = 5.e-6; */
    /* bounds may be violated at most by taubnd in finite differencing */
    /*  bloc    = TRUE; */
    /* if one wants to evaluate all functions  in an independent process */
    /* difftype = 3; *//* the most accurate and most expensive choice */

    nreset = n;
```

```
del0 = 0.2e0;
tau0 = 1.e0;
tau  = 0.1e0;
big  = INF;
switch (optimization_problem) {
  case 0:
    /* starting value: best point */
    for (i = 1 ; i <= DIMX ; i++) {
        x[i] = best_point[i-1];
    }
    for (j=1; j<=DIMX; j++) {
        for (k=1; k<=j; k++) {
            x[i++] = x[j]*x[k];
        }
    }
    for (; i <= DIMX+DIMZ+DIMY ; i++) {
        x[i] = best_point[i-(DIMZ+1)];
    }

    /*  set lower and upper bounds */
    /* x */
    for (i = 1 ; i <= DIMX ; i++) {
        low[i] = xlow[i-1];
        up[i] = xup[i-1];
    }
    /* z */
    for (j=0; j<DIMX; j++) {
        for (k=0; k<=j; k++) {
            /* interval multiplication [xlow[j],xup[j]]*[xlow[k],xup[k]] */
            DOUBLE
             bounds[4]={xlow[j]*xlow[k],xlow[j]*xup[k],xup[j]*xlow[k],
             xup[j]*xup[k]};
            DOUBLE zlow=*bounds, zup=*bounds;
            for (l=1; l<4; l++) {
                if (bounds[l]<zlow) zlow=bounds[l];
                if (bounds[l]>zup) zup=bounds[l];
            }
            low[i] = zlow;
            up[i++] = zup;
        }
    }
    /* y */
    for (; i <= DIMX+DIMZ+DIMY ; i++) {
        low[i] = Flow[i-(DIMX+DIMZ+1)];
        up[i] = Fup[i-(DIMX+DIMZ+1)];
    }
    /* covariance model interval constraint */
    low[i] = klow;
    up[i++] = kup;
```

```
        /* equality constraints */
        for (j=0; j<DIMZ; j++) {
            low[i] = 0.;
            up[i++] = 0.;
        }
        break;
      case 1:
        if (ignore_constraints) {
            /* starting value: center of box */
            for (i = 1 ; i <= DIMX ; i++) {
                x[i] = (xlow[i-1] + xup[i-1]) * .5;
            }
        } else {
            /* starting value: unconstrained minimum */
            for (i = 1 ; i <= DIMX ; i++) {
                x[i] = optimum_x[i-1];
            }
        }

        /*  set lower and upper bounds */
        for (i = 1 ; i <= DIMX ; i++) {
            low[i] = xlow[i-1];
            up[i] = xup[i-1];
        }
        if (!ignore_constraints) {
            /* estimate constraints */
            for (j=0; j<(INTEGER)num_estimate_constraints; j++) {
                low[i] = -ESTIMATE_CONSTRAINT_TOL;
                up[i++] = INF;
            }
        }
        break;
    }

    silent = TRUE;

    return;
}
```

This is the DONLP2 initialization callback. We set the starting values, bounds and some parameters. In particular, we indicate that we have analytically computed gradients and we set DONLP2 to silent mode because otherwise the many DONLP2 invocations lead to way too much debugging output.

```
/* ***************************************************************************
 */
/*                                  special setup
 */
```

```
/* ****************************************************************************
 */
void setup(void) {
    #define  X extern
    #include "o8comm.h"
    #undef   X
    te0=TRUE;
    /* enforce valid delmin */
    if (!optimization_problem && delmin + delmin >= kup - klow)
      delmin = (kup - klow) * .499;
    return;
}
```

This callback allows overriding some DONLP2 default settings. DONLP2 first calls `user_init`, then sets some defaults based on that, then allows `setup` to override them. We set `te0` to `TRUE`, which means to log a line for each iteration step, however this line is not output to `stdout` due to silent mode being used, it is only logged to a file in the case of a failure, in order to allow debugging. For the local search, we also enforce a valid setting of the `delmin` variable: for a constraint with distinct lower and upper bounds k_l and k_u, there must not be a point k satisfying both $k - k_l \leq$ `delmin` and $k_u - k \leq$ `delmin`. It follows that `delmin` $< \frac{k_u - k_l}{2}$ must hold. The default setting does not always satisfy this for our covariance constraint, so we adjust `delmin` if the property does not hold.

```
/* ****************************************************************************
 */
/*  the user may add additional computations using the computed solution here
 */
/* ****************************************************************************
 */
void solchk(void) {
    #define  X extern
    #include "o8comm.h"
    #undef   X
    #include "o8cons.h"

    INTEGER i;
    for (i=0; i<DIMX; i++)
        optimum_x[i]=x[i+1];

    return;
}
```

This callback is called by DONLP2 once a solution has been found. We save the result to the global `optimum_x`.

```
/* ****************************************************************************
 */
```

```
/*                                  objective function
 */
/* ************************************************************************
 */
void ef(DOUBLE x[],DOUBLE *fx) {
    #define  X extern
    #include "o8fuco.h"
    #undef   X


    INTEGER i;
    DOUBLE f=0.;
    switch (optimization_problem) {
      case 0:
        for (i=0; i<DIMX; i++) {
            f += c[i]*x[i+1];
        }
        for (i=0; i<DIMY; i++) {
            f += c[i+DIMX]*x[i+(DIMX+DIMZ+1)];
        }
        *fx = f;
        break;
      case 1:
        *fx = gmm_prob(x+1);
        break;
    }

    return;
}
```

This callback implements the objective function. One caveat with DONLP2 is that it has been ported from FORTRAN, which uses 1-based arrays, and thus it uses 1 as the first array index, not 0 as would be expected in C. Otherwise, the implementation is straightforward.

```
/* ************************************************************************
 */
/*                            gradient of objective function
 */
/* ************************************************************************
 */
void egradf(DOUBLE x[],DOUBLE gradf[]) {
    #define  X extern
    #include "o8fuco.h"
    #undef   X

    INTEGER  j;
```

```
    switch (optimization_problem) {
      case 0:
        for (j=0; j<DIMX; j++) {
            gradf[j+1] = c[j];
        }
        for (; j<DIMX+DIMZ; j++) {
            gradf[j+1] = 0.;
        }
        for (j=0; j<DIMY; j++) {
            gradf[j+(DIMX+DIMZ+1)] = c[j+DIMX];
        }
        break;
      case 1:
        gmm_grad(x+1,gradf+1);
        break;
    }

    return;
}
```

Likewise, this callback implements the gradient of the objective function.

```
/* ************************************************************************
 */
/*                compute the i-th equality constaint, value is hxi
 */
/* ************************************************************************
 */
void econ(INTEGER type ,INTEGER liste[], DOUBLE x[],DOUBLE con[],
              LOGICAL err[]) {
    #define  X extern
    #include "o8fuco.h"
    #undef   X
    INTEGER i,j,k,l;
    DOUBLE z;
    INTEGER liste_loc_size;
    switch (optimization_problem) {
      case 0:
        liste_loc_size = DIMZ+1 ;
        break;
      case 1:
        liste_loc_size = ignore_constraints ? 0 : (INTEGER)
         num_estimate_constraints ;
        break;
    }
    INTEGER liste_loc[liste_loc_size+1];
/* if type != 1 only a selection is evaluated the indices being taken from    */
/* liste. since we have no evaluation errors here err is never touched        */
```

```
if ( type == 1 )
{
   liste_loc[0] = liste_loc_size ;
   for ( i = 1 ; i<=liste_loc[0] ; i++ ) { liste_loc[i] = i ; }
}
else
{
   liste_loc[0] = liste[0] ;
   for ( i = 1 ; i<=liste[0] ; i++ ) { liste_loc[i] = liste[i];}
}
for ( j = 1 ; j <= liste_loc[0] ; j++ )
{
  i = liste_loc[j] ;

  switch (optimization_problem) {
    case 0:
      if (i==1) {
          /* compute (X-Xbar)T M (X-Xbar) */
          DOUBLE X[DIMX+DIMZ+DIMY+DIMY_EQ], MX[DIMX+DIMZ+DIMY+DIMY_EQ];
          z=0.;
          for (k=0; k<DIMX+DIMZ+DIMY; k++) {
              X[k]=x[k+1]-Xbar[k];
          }
          for (; k<DIMX+DIMZ+DIMY+DIMY_EQ; k++) {
              X[k]=-Xbar[k];
          }
          compute_MX(X,MX);
          for (k=0; k<DIMX+DIMZ+DIMY+DIMY_EQ; k++) {
              z+=X[k]*MX[k];
          }
          con[1]=z;
      } else {
          /* compute zi-xk*xl */
          k=0;
          l=i-2;
          while (l>k) {
              k++;
              l-=k;
          }
          con[i]=x[i+(DIMX-1)]-x[k+1]*x[l+1];
      }
      break;
    case 1:
      {
        /* evaluate quadratic polynomial */
        DOUBLE *p=estimate_constraint_coeffs[i-1];
        z=*(p++); /* constant term */
        for (k=1; k<=DIMX; k++) { /* linear terms */
```

```
            z+=*(p++) * x[k];
          }
          for (k=1; k<=DIMX; k++) { /* quadratic/bilinear terms */
            for (l=1; l<=k; l++) {
              z+=*(p++) * (x[k]*x[l]);
            }
          }
          con[i]=z;
        }
        break;
      }

    }
    return;
}
```

This callback implements the constraints. As indicated in a comment, DONLP2 can request all (`type` $= 1$) or only some (`type` $\neq 1$) of the constraints. In the latter case, `liste` contains the list of needed constraints. We construct a full list in `liste_loc` if we have to evaluate all constraints, otherwise we copy `liste` to `liste_loc`. We then evaluate the constraints contained in `liste_loc`.

```
/* ****************************************************************************
 */
/*              compute the gradient of the i-th equality constraint
 */
/* ****************************************************************************
 */
void econgrad(INTEGER liste[] ,INTEGER shift ,  DOUBLE x[],
              DOUBLE **grad) {
    #define  X extern
    #include "o8fuco.h"
    #undef   X

    INTEGER  i,j,k,l;
    DOUBLE   z;

    INTEGER liste_loc_size;
    switch (optimization_problem) {
      case 0:
        liste_loc_size = DIMZ+1 ;
        break;
      case 1:
        liste_loc_size = ignore_constraints ? 0 : (INTEGER)
         num_estimate_constraints ;
        break;
    }
    INTEGER liste_loc[liste_loc_size+1];
```

```
liste_loc[0] = liste[0] ;
for ( i = 1 ; i<=liste_loc[0] ; i++ ) { liste_loc[i] = liste[i];}
for ( j = 1 ; j <= liste_loc[0] ; j++ )
{
  i = liste_loc[j] ;
  switch (optimization_problem) {
    case 0:
      if (i==1) {
          /* compute ((X-Xbar)T M (X-Xbar))' = 2 M (X-Xbar) */
          DOUBLE X[DIMX+DIMZ+DIMY+DIMY_EQ], MX[DIMX+DIMZ+DIMY+DIMY_EQ];
          for (k=0; k<DIMX+DIMZ+DIMY; k++) {
              X[k]=x[k+1]-Xbar[k];
          }
          for (; k<DIMX+DIMZ+DIMY+DIMY_EQ; k++) {
              X[k]=-Xbar[k];
          }
          compute_MX(X,MX);
          for (k=0; k<DIMX+DIMZ+DIMY; k++) {
              z=MX[k];
              grad[k+1][i+shift] = 2.*z;
          }
      } else {
          for (k=1; k<=DIMX+DIMZ+DIMY; k++) {
              grad[k][i+shift] = 0.e0;
          }
          /* compute (zi-xk*xl)' = ezi - xl exk - xk exl */
          k=0;
          l=i-2;
          while (l>k) {
              k++;
              l-=k;
          }
          grad[i+(DIMX-1)][i+shift] = 1.;
          grad[k+1][i+shift] -= x[l+1];
          grad[l+1][i+shift] -= x[k+1];
      }
      break;
    case 1:
      {
        /* evaluate gradient of the quadratic polynomial */
        DOUBLE *p=estimate_constraint_coeffs[i-1]+1; /* skip constant term
         */
        for (k=1; k<=DIMX; k++) { /* linear terms (constant derivatives) */
          grad[k][i+shift]=*(p++);
        }
        for (k=1; k<=DIMX; k++) { /* quadratic/bilinear terms (linear
         derivatives) */
          for (l=1; l<=k; l++) {
```

```
                grad[k][i+shift]+=*p * x[l];
                grad[l][i+shift]+=*(p++) * x[k];
                /* If k=l, then this is grad[k][i+shift]+=2 * *(p++) * x[k],
                   which is correct, since d(xk^2)/dxk=2 xk.
                   For the bilinear terms, d(xk*xl)/dxk=xl, d(xk*xl)/dxl=xk. */
              }
            }
          }
          break;
      }

    }
    return;
}
```

Likewise, this callback evaluates the Jacobian of the constraints, or the gradients listed in the parameter `liste`.

```
/* ************************************************************************
 */
/*                        user functions (if bloc == TRUE)
 */
/* ************************************************************************
 */
void eval_extern(INTEGER mode) {
    #define  X extern
    #include "o8comm.h"
    #include "o8fint.h"
    #undef   X
    #include "o8cons.h"

    return;
}
```

This callback does nothing because it is not actually used, as we don't use the `bloc` mode.

4.4.3 Ipopt Wrapper and Callbacks (ipopt.h, ipopt.c)

The Ipopt wrapper `ipopt.h` contains the Ipopt-specific parts of the optimizer abstraction.

```
#include "IpStdCInterface.h"
```

We start by including the header file provided by Ipopt itself, which declares the types and functions used by Ipopt, such as `Number`.

```
#define DOUBLE Number
```

This macro abstracts the fact that Ipopt uses `Number` where DONLP2 uses `DOUBLE`. (Internally, both default to the builtin `double` type, but we try not to make use of this fact for higher portability.)

```
#define INF 2.e19
```

This macro corresponds to the large number used to represent infinity within Ipopt.

```
int solve_nlp(void);
```

The `solve_nlp` function is the primary abstraction for the optimizer which is being used. For Ipopt, the implementation is an actual function which sets up the problem structure and passes the correct function pointers to Ipopt depending on which optimization problem is to be solved. Being a C++ program, the current version of Ipopt uses a much more object-oriented approach (even in the C interface) than DONLP2, which is ported from FORTRAN: a problem is represented as a structure, and callbacks are called through function pointers, not by name as in DONLP2.

The file `ipopt.c` contains the implementation of the callback functions for Ipopt, which represent the two types of optimization models used during our algorithm: local surrogate models and global density minimization. They work very similarly to their counterparts for DONLP2, but implement different interfaces, which can be found more in detail in the documentation included with Ipopt.

```
#undef USE_BFGS /* instead of symbolic Hessian */
```

This option serves mainly for testing purposes. Ipopt can use analytically computed Hessians or it can approximate them using limited-memory BFGS. Defining `USE_BFGS` turns off the analytical Hessians and tells Ipopt to use the limited-memory BFGS approach instead. Use of this option is not recommended.

```
int ignore_constraints;
DOUBLE optimum_x[DIMX];
int optimization_problem;
```

These global variables work exactly the same as their equivalents in `userfu.c` for DONLP2.

```
/* The number of nonzero entries in the lower triangular part of a full
   symmetric matrix. */
#define FULL_SYMM_MAT(n) (((n)*((n)+1))>>1)
```

This utility macro is used to compute the sizes of the analytical Hessians.

```
static Bool eval_f_0(Index n, Number* x, Bool new_x,
                     Number* obj_value, UserDataPtr user_data)
{
  Index i;
  Number f=0.;
  for (i=0; i<DIMX; i++) {
    f+=c[i]*x[i];
  }
  for (i=0; i<DIMY; i++) {
    f+=c[i+DIMX]*x[i+(DIMX+DIMZ)];
  }
  *obj_value=f;
  return TRUE;
}
```

This callback evaluates the objective function for the local search. As Ipopt calls callbacks by pointer and not by name, we can use separate callbacks for the different problems. We make use of this possibility.

```
static Bool eval_grad_f_0(Index n, Number* x, Bool new_x,
                          Number* grad_f, UserDataPtr user_data)
{
  Index j;
  for (j=0; j<DIMX; j++) {
    grad_f[j]=c[j];
  }
  for (; j<DIMX+DIMZ; j++) {
    grad_f[j]=0.;
  }
  for (j=0; j<DIMY; j++) {
    grad_f[j+(DIMX+DIMZ)]=c[j+DIMX];
  }
  return TRUE;
}
```

This callback evaluates the gradient of the objective function for the local search.

```
static Bool eval_g_0(Index n, Number* x, Bool new_x,
                     Index m, Number* g, UserDataPtr user_data)
{
  Index i,k,l;

  /* compute (X-Xbar)T M (X-Xbar) */
  Number X[DIMX+DIMZ+DIMY+DIMY_EQ], MX[DIMX+DIMZ+DIMY+DIMY_EQ];
  Number z=0.;
  for (k=0; k<DIMX+DIMZ+DIMY; k++) {
```

```
    X[k]=x[k]-Xbar[k];
  }
  for (; k<DIMX+DIMZ+DIMY+DIMY_EQ; k++) {
    X[k]=-Xbar[k];
  }
  compute_MX(X,MX);
  for (k=0; k<DIMX+DIMZ+DIMY+DIMY_EQ; k++) {
    z+=X[k]*MX[k];
  }
  *g=z;

  for (i=1; i<=DIMZ; i++) {
    /* compute zi-xk*xl */
    k=0;
    l=i-1;
    while (l>k) {
      k++;
      l-=k;
    }
    g[i]=x[i+(DIMX-1)]-x[k]*x[l];
  }
  return TRUE;
}
```

This callback evaluates the constraints for the local search.

```
static Bool eval_jac_g_0(Index n, Number *x, Bool new_x,
                         Index m, Index nele_jac,
                         Index *iRow, Index *jCol, Number *values,
                         UserDataPtr user_data)
{
  if (!values) { /* sparsity structure */
    Index i=1,j,k,l;
    /* first row full */
    for (j=0; j<(DIMX+DIMZ+DIMY); j++) {
      iRow[j]=0; jCol[j]=j;
    }
    /* equality constraints */
    for (k=0; k<DIMX; k++) {
      for (l=0; l<k; l++) { /* bilinear terms in z */
        iRow[j]=i; jCol[j++]=l;
        iRow[j]=i; jCol[j++]=k;
        iRow[j]=i; jCol[j++]=(i++)+(DIMX-1);
      }
      /* quadratic terms in z */
      iRow[j]=i; jCol[j++]=k;
      iRow[j]=i; jCol[j++]=(i++)+(DIMX-1);
    }
  } else { /* values */
```

```
    Index j=0,k,l;
    /* compute ((X-Xbar)T M (X-Xbar))' = 2 M (X-Xbar) */
    Number X[DIMX+DIMZ+DIMY+DIMY_EQ], MX[DIMX+DIMZ+DIMY+DIMY_EQ];
    for (k=0; k<DIMX+DIMZ+DIMY; k++) {
      X[k]=x[k]-Xbar[k];
    }
    for (; k<DIMX+DIMZ+DIMY+DIMY_EQ; k++) {
      X[k]=-Xbar[k];
    }
    compute_MX(X,MX);
    for (k=0; k<DIMX+DIMZ+DIMY; k++) {
      Number z=MX[k];
      values[j++]=2.*z; /* Jac[0][j] */
    }
    /* equality constraints */
    for (k=0; k<DIMX; k++) {
      for (l=0; l<k; l++) { /* bilinear terms in z */
        values[j++]=-x[k]; /* Jac[i][k] */
        values[j++]=-x[l]; /* Jac[i][l] */
        values[j++]=1.; /* Jac[i][i+(DIMX-1)] */
      }
      /* quadratic terms in z */
      values[j++]=-2.*x[k]; /* Jac[i][k] */
      values[j++]=1.; /* Jac[i][i+(DIMX-1)] */
    }
  }
  return TRUE;
}
```

This callback evaluates the Jacobian of the constraints for the local search. Unlike DONLP2, Ipopt uses sparse matrix arithmetic, thus the Jacobian is represented as a sparse matrix. Ipopt requests the sparsity structure at the beginning, then only the values contained in the sparsity structure shall be returned at each request.

```
static Bool eval_h_0(Index n, Number *x, Bool new_x, Number obj_factor,
                     Index m, Number *lambda, Bool new_lambda,
                     Index nele_hess, Index *iRow, Index *jCol,
                     Number *values, UserDataPtr user_data)
{
#ifdef USE_BFGS
  return FALSE;
#else
  if (!values) { /* sparsity structure */
    /* full Hessian */
    Index k=0,i,j;
    for (i=0; i<DIMX+DIMZ+DIMY; i++) {
      for (j=0; j<=i; j++) {
        iRow[k]=i; jCol[k++]=j;
```

```
      }
    }
  } else { /* values */
    /* The objective function is linear, skip it. */
    /* first constraint: Hessian = 2 M */
    Index k=0,i,j,l;
    Number X[DIMX+DIMZ+DIMY+DIMY_EQ], MX[DIMX+DIMZ+DIMY+DIMY_EQ];
    for (i=0; i<DIMX+DIMZ+DIMY; i++) {
      for (l=0; l<DIMX+DIMZ+DIMY+DIMY_EQ; l++) {
        X[l]=(l==i)?(2.*(*lambda)):0.;
      }
      /* M is symmetric, so we'll get away with computing columns instead of
       rows */
      compute_MX(X,MX);
      for (j=0; j<=i; j++) {
        values[k++]=MX[j];
      }
    }
    /* equality constraints */
    j=0;
    for (k=0; k<DIMX; k++) {
      for (l=0; l<k; l++) { /* bilinear terms in z */
        values[j]-=lambda[j+1];
        j++;
      }
      /* quadratic terms in z */
      values[j]-=2.*lambda[j+1];
      j++;
    }
  }
  return TRUE;
#endif
}
```

This callback evaluates the Hessian of the Lagrangian for the local search. More precisely, Ipopt uses a generalized Lagrangian where the objective function is also weighted with a coefficient `obj_factor`, which allows a Hessian of the constraints only to be requested when needed. Like the Jacobians, the Hessians are also represented as sparse matrices, however we cannot make use of this fact, our sparsity structure is always full.

```
static Bool eval_f_1(Index n, Number* x, Bool new_x,
                     Number* obj_value, UserDataPtr user_data)
{
  *obj_value=gmm_prob(x);
  return TRUE;
}
```

This callback evaluates the objective function for the global search.

```
static Bool eval_grad_f_1(Index n, Number* x, Bool new_x,
                          Number* grad_f, UserDataPtr user_data)
{
  gmm_grad(x,grad_f);
  return TRUE;
}
```

This callback evaluates the gradient of the objective function for the global search.

```
static Bool eval_g_1(Index n, Number* x, Bool new_x,
                     Index m, Number* g, UserDataPtr user_data)
{
  if (!ignore_constraints) {
    Index i,k,l;
    for (i=0; i<(Index)num_estimate_constraints; i++) {
      /* evaluate quadratic polynomial */
      Number *p=estimate_constraint_coeffs[i];
      Number z=*(p++); /* constant term */
      for (k=0; k<DIMX; k++) { /* linear terms */
        z+=*(p++) * x[k];
      }
      for (k=0; k<DIMX; k++) { /* quadratic/bilinear terms */
        for (l=0; l<=k; l++) {
          z+=*(p++) * (x[k]*x[l]);
        }
      }
      g[i]=z;
    }
  }
  return TRUE;
}
```

This callback evaluates the constraints for the global search. Obviously, in the case where `ignore_constraints` is set, i.e. where we ignore all non-bound constraints, there is nothing to do.

```
static Bool eval_jac_g_1(Index n, Number *x, Bool new_x,
                         Index m, Index nele_jac,
                         Index *iRow, Index *jCol, Number *values,
                         UserDataPtr user_data)
{
  if (!ignore_constraints) {
    if (!values) { /* sparsity structure */
      /* full Jacobian */
      Index k=0,i,j;
      for (i=0; i<(Index)num_estimate_constraints; i++) {
        for (j=0; j<DIMX; j++) {
          iRow[k]=i; jCol[k++]=j;
```

```
        }
      }
    } else { /* values */
      Index i,k,l;
      for (i=0; i<(Index)num_estimate_constraints; i++) {
        /* evaluate gradient of the quadratic polynomial */
        Number *p=estimate_constraint_coeffs[i]+1; /* skip constant term */
        Number *grad=values+i*DIMX;
        for (k=0; k<DIMX; k++) { /* linear terms (constant derivatives) */
          grad[k]=*(p++);
        }
        for (k=0; k<DIMX; k++) { /* quadratic/bilinear terms (linear
         derivatives) */
          for (l=0; l<=k; l++) {
            grad[k]+=*p * x[l];
            grad[l]+=*(p++) * x[k];
            /* If k==l, then this is grad[k][i+shift]+=2 * *(p++) * x[k],
               which is correct, since d(xk^2)/dxk=2 xk.
               For the bilinear terms, d(xk*xl)/dxk=xl, d(xk*xl)/dxl=xk. */
          }
        }
      }
    }
  }
  return TRUE;
}
```

This callback evaluates the Jacobian of the constraints for the global search. In this case, the Jacobian is always full. Again, there is obviously nothing to do when `ignore_constraints` is set.

```
static Bool eval_h_1(Index n, Number *x, Bool new_x, Number obj_factor,
                     Index m, Number *lambda, Bool new_lambda,
                     Index nele_hess, Index *iRow, Index *jCol,
                     Number *values, UserDataPtr user_data)
{
#ifdef USE_BFGS
  return FALSE;
#else
  if (!values) { /* sparsity structure */
    /* full Hessian */
    Index k=0,i,j;
    for (i=0; i<DIMX; i++) {
      for (j=0; j<=i; j++) {
        iRow[k]=i; jCol[k++]=j;
      }
    }
  } else { /* values */
    Index i,k,l;
```

```
    /* Hessian of the objective function */
    gmm_hess(x,values);
    for (i=0; i<FULL_SYMM_MAT(DIMX); i++) {
      values[i]*=obj_factor;
    }
    if (!ignore_constraints) {
      for (i=0; i<(Index)num_estimate_constraints; i++) {
        /* Hessian of the quadratic polynomial */
        Number *p=estimate_constraint_coeffs[i]+(1+DIMX); /* skip constant and
         linear terms */
        Number *q=values;
        for (k=0; k<DIMX; k++) { /* quadratic/bilinear terms (linear
         derivatives) */
          for (l=0; l<=k; l++) {
            *(q++)+=lambda[i]*((k==l)?2.:1.)*(*(p++));
          }
        }
      }
    }
  }
  return TRUE;
#endif
}
```

This callback evaluates the Hessian of the Lagrangian for the global search. Again, we cannot make use of the sparsity, our Hessian is always full.

```
int solve_nlp(void)
{
  Number x[DIMX+DIMZ+DIMY];
  IpoptProblem problem=NULL;
  Index i,j,k,l;

  switch (optimization_problem) {
    case 0:
      {
        /* lower and upper bounds */
        Number x_L[DIMX+DIMZ+DIMY];
        Number x_U[DIMX+DIMZ+DIMY];
        Number g_L[1+DIMZ];
        Number g_U[1+DIMZ];
        /* x */
        for (i=0; i<DIMX; i++) {
          x_L[i]=xlow[i];
          x_U[i]=xup[i];
        }
        /* z */
        for (j=0; j<DIMX; j++) {
          for (k=0; k<=j; k++) {
```

```
      /* interval multiplication [xlow[j],xup[j]]*[xlow[k],xup[k]] */
      Number
       bounds[4]={xlow[j]*xlow[k],xlow[j]*xup[k],xup[j]*xlow[k],
       xup[j]*xup[k]};
      Number zlow=*bounds, zup=*bounds;
      for (l=1; l<4; l++) {
        if (bounds[l]<zlow) zlow=bounds[l];
        if (bounds[l]>zup) zup=bounds[l];
      }
      x_L[i]=zlow;
      x_U[i++]=zup;
    }
  }
  /* y */
  for (; i<DIMX+DIMZ+DIMY; i++) {
    x_L[i]=Flow[i-(DIMX+DIMZ)];
    x_U[i]=Fup[i-(DIMX+DIMZ)];
  }
  /* covariance model interval constraint */
  *g_L=klow;
  *g_U=kup;
  /* equality constraints */
  for (i=1; i<=DIMZ; i++) {
    g_L[i]=0.;
    g_U[i]=0.;
  }
  /* nele_jac: first row full: DIMX+DIMZ+DIMY
               + rows for quadratic terms in z: 2 nonzero entries * DIMX
               + rows for bilinear terms in z: 3 nonzero entries *
                (DIMZ-DIMX)
               = DIMX+DIMZ+DIMY + 2*DIMX + 3*(DIMZ-DIMX)
               = DIMX+DIMZ+DIMY + 2*DIMX + 3*DIMZ - 3*DIMX
               = DIMY + 4*DIMZ */
  problem=CreateIpoptProblem(/*n=*/DIMX+DIMZ+DIMY,x_L,x_U,
                             /*m=*/1+DIMZ,g_L,g_U,
                             /*nele_jac=*/DIMY+(DIMZ<<2),
                             /*nele_hess=*/FULL_SYMM_MAT(DIMX+DIMZ+DIMY)
                              /*full Hessian*/,
                             /*index_style=*/0,
                             eval_f_0,eval_g_0,eval_grad_f_0,eval_jac_g_0,
                             eval_h_0);
  /* starting value: best point */
  for (i=0; i<DIMX; i++) {
    x[i]=best_point[i];
  }
  for (j=0; j<DIMX; j++) {
    for (k=0; k<=j; k++) {
      x[i++] = x[j]*x[k];
    }
```

```
      }
      for (; i<DIMX+DIMZ+DIMY; i++) {
        x[i]=best_point[i-DIMZ];
      }
    }
    break;
  case 1:
    {
      Index
       num_constraints=ignore_constraints?0:(Index)num_estimate_constraints;
      /* estimate constraints */
      Number g_L[num_constraints];
      Number g_U[num_constraints];
      for (j=0; j<(Index)num_constraints; j++) {
        g_L[j]=-ESTIMATE_CONSTRAINT_TOL;
        g_U[j]=INF;
      }
      problem=CreateIpoptProblem(/*n=*/DIMX,/*x_L=*/xlow,/*x_U=*/xup,
                                 /*m=*/num_constraints,g_L,g_U,
                                 /*nele_jac=*/DIMX*num_constraints /* full
                                  Jacobian */,
                                 /*nele_hess=*/FULL_SYMM_MAT(DIMX) /* full
                                  Hessian */,
                                 /*index_style=*/0,
                                 eval_f_1,eval_g_1,eval_grad_f_1,eval_jac_g_1,
                                 eval_h_1);
      if (ignore_constraints) {
        /* starting value: center of box */
        for (i=0; i<DIMX; i++) {
          x[i]=(xlow[i]+xup[i])*.5;
        }
      } else {
        /* starting value: unconstrained minimum */
        for (i=0; i<DIMX; i++) {
          x[i]=optimum_x[i];
        }
      }
    }
    break;
}

if (!problem) {
  fprintf(stderr, "CreateIpoptProblem failed\n");
  exit(1);
}
/* Scale objective for better performance */
if (optimization_problem) {
  if (!AddIpoptNumOption(problem,(char *)"obj_scaling_factor",16384.)) {
    fprintf(stderr, "AddIpoptNumOption failed\n");
```

```
      exit(1);
    }
  }
#ifdef USE_BFGS
  /* These are of course const char * (the underlying C++ functions take const
     std::string &), but the prototype is wrong, so silence -Wwrite-strings
     warnings. */
  if (!AddIpoptStrOption(problem,(char *)"hessian_approximation",(char
   *)"limited-memory")) {
    fprintf(stderr, "AddIpoptStrOption failed\n");
    exit(1);
  }
#endif
  if (!AddIpoptIntOption(problem,(char *)"print_level",2 /*J_WARNING*/)) {
    fprintf(stderr, "AddIpoptIntOption failed\n");
    exit(1);
  }
  int ret=IpoptSolve(problem,x,NULL,NULL,NULL,NULL,NULL,NULL);
  FreeIpoptProblem(problem);
  /* remember optimum x */
  for (i=0; i<DIMX; i++) optimum_x[i]=x[i];
  return ret;
}
```

This function implements the main optimizer abstraction for Ipopt: We create temporary arrays for the bounds, then allocate the problem structure with `CreateIpoptProblem` (which copies the bounds internally, so the arrays going out of scope is not a problem), set the starting point and some options and run the actual optimization function (`IpoptSolve`). Finally, we free the problem structure with `FreeIpoptProblem` and save the result, which is contained in the local variable `x`, into the global variable `optimum_x`. The `obj_scaling_factor` is required for the global search because our objective function for that problem has a rather small scale and Ipopt is very sensitive to scaling. The value of 16384 is the result of some experiments, other nearby powers of two also work. (We use a power of two so scaling does not introduce rounding errors.) The `print_level` option reduces the amount of debugging output to an acceptable level for the many invocations of Ipopt (warnings and errors only).

Chapter 5

Results

In this chapter, we give some metrics about the performance of the algorithm, both in terms of speed and quality. The first section describes the approach used to obtain the results, in order to allow them to be reproduced. The second section presents the results themselves.

5.1 Testing Methods

We tested our implementation on a few test cases from the libraries 1 and 2 of the COCONUT Benchmark [47, 48]. These libraries contain common test problems converted to COCONUT's internal DAG representation, which can be converted to C code using APIs from the COCONUT Environment [49, 3]. Library 1 contains problems taken from GLOBALLib [46] and the Handbook of Test Problems in Local and Global Optimization [50]. Library 2 corresponds to Vanderbei's CUTE Test Collection [51].

It is important to note that these test cases involve cheap, analytical functions, not expensive black box functions as in the real-world problems our algorithm is targeted at. This implies that all time measurements essentially only measure the time spent within the algorithm. Therefore, the number of function evaluations is an important metric which must be taken into account when estimating the time spent on real-world problems.

We found that the DAG to C converter included in COCONUT, while close to what was needed, was not entirely suitable for our needs. Therefore, we rewrote the main C writer function `c_write::c_print` in `c_write.cc` to output the problems in the format we required. We were able to reuse the existing APIs within the function, so only `c_write::c_print` had to be modified. The modified file is reproduced below. It is licensed under the GNU Library or Lesser General Public License, version 2 [52] or later.

```
// C writer implementation -*- C++ -*-

// $Id: c_write.cc 2 2006-04-14 11:01:54Z herman $
// Copyright (C) 2001-2003 Hermann Schichl
// Copyright (C) 2007 Kevin Kofler
//
// This file is part of the COCONUT API.  This library
// is free software; you can redistribute it and/or modify it under the
// terms of the Library GNU General Public License as published by the
// Free Software Foundation; either version 2, or (at your option)
// any later version.

// This library is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
// Library GNU General Public License for more details.

// As a special exception, you may use this file as part of a free software
// library without restriction.  Specifically, if other files instantiate
// templates or use macros or inline functions from this file, or you compile
// this file and link it with other files to produce an executable, this
// file does not by itself cause the resulting executable to be covered by
// the Library GNU General Public License.  This exception does not however
// invalidate any other reasons why the executable file might be covered by
// the Library GNU General Public License.

/** @file c_write.cc */

#include <c_write.h>

namespace coco_rm {
using namespace coco;

static const char *HL_names[] = {
 "G",
 "C",
 "V",

 "+",
 "+",
 "*",
 "max",
 "min",
 "monome",
 "scprod",
 "norm",

 "/",
 "pow",
```

```
"sqrt",
"fabs",
"pow",
"exp",
"log",
"sin",
"cos",
"exp(pow",
"poly",

"pow",
"/",
"atan2",

"LINALG_lin",
"LINALG_quad",

"COMPLEX_re",
"COMPLEX_im",
"COMPLEX_arg",
"COMPLEX_conj",

"TABLE_lookup",
"TABLE_pwlin",
"TABLE_spline",
"TABLE_pwcl",
"TABLE_pwcr",

"LOGIC_in",
"LOGIC_if",
"LOGIC_and",
"LOGIC_or",
"LOGIC_not",
"LOGIC_implies",

"CARD_count",
"CARD_alldiff",
"CARD_histogram",
"CARD_level",

"INT_neighbor",
"INT_nogood",

"STOCH_expectation",
"STOCH_integral",

"MATRIX_det",
"MATRIX_cond",
"MATRIX_psd",
```

```
 "MATRIX_multiply",
 "MATRIX_fem",
 "MATRIX_const_multiply",
 "MATRIX_const_fem",

 NULL
};

#define HL_CLASS_NAME c_write

#define _HL_WRITE_WITH_GHOST     1

#define HL_PRECEDENCE_FUNC       1
#define HL_PRECEDENCE_SUM        10
#define HL_PRECEDENCE_PROD       50
#define HL_PRECEDENCE_TERMINAL   1000

} // namespace coco_rm

#include <templates/hl_visitor.h>

// This constant is used instead of infinity to ensure finite bounds.
// (Kevin Kofler)
#define INF_BOUND 1000.

namespace coco_rm {

void c_write::c_print(const model& DAG, std::ostream& o,
                      const control_data& __c) const
{
  std::string headers;
  std::string objective_preamble;
  std::string objective_postamble;
  std::string ofunction_end;
  std::string constraints_preamble;
  std::string constraints_postamble;
  std::string x_v, y_v, k_v;
  std::string switch_preamble;
  std::string switch_postamble;
  std::string cfunction_end;
  std::string footers;
  std::string vtype;
  std::string ktype;
  bool print_y, with_comment;

  __c.assign("header", headers, _headers);
  __c.assign("objective preamble", objective_preamble, _objective_preamble);
  __c.assign("objective postamble", objective_postamble, _objective_postamble);
  __c.assign("objective function end", ofunction_end, _ofunction_end);
```

```
  __c.assign("constraints preamble", constraints_preamble,
             _constraints_preamble);
  __c.assign("constraints postamble", constraints_postamble,
             _constraints_postamble);
  __c.assign("x", x_v, _x_v);
  __c.assign("y", y_v, _y_v);
  __c.assign("kappa", k_v, _k_v);
  __c.assign("switch preamble", switch_preamble, _switch_preamble);
  __c.assign("switch postamble", switch_postamble, _switch_postamble);
  __c.assign("constraint functions end", cfunction_end, _cfunction_end);
  __c.assign("footer", footers, _footers);
  __c.assign("type", vtype, _vtype);
  __c.assign("kappa type", ktype, _ktype);

  __c.assign("print multipliers", print_y, true);
  __c.assign("with comments", with_comment, true);

  hlwrite_visitor p(o, x_v, y_v, k_v, print_y, with_comment);
  int i;
  unsigned int n;
  char *name(NULL);
  double d;
  int precsave = o.precision(22);

// Mihaly, 12.12.2003
// set the filib interval precision as well
  int intprecsave = interval::precision(22);

// This section rewritten by Kevin Kofler to output the problem.c file in the
// format needed by bbowda.
// Dimension of the x vector (number of independent variables)
  int dimx = DAG.number_of_variables();
// Number of implicit inequality resp. equality constraints
// (All constraints are assumed implicit. As always in our formulation, implicit
// inequality constraints are represented as explicit equality constraints with
// bounds for the variable on the left hand side.)
  int dimconstr = 0, dimeqconstr = 0;
// 1 if the objective function is nonlinear, which means it has to be
// reformulated as an explicit equality constraint. Currently, we cannot detect
// linear objective functions, so we always do this substitution unless the
// objective function is constant zero (constraint satisfaction problem). This
// probably makes the results more realistic anyway as we do not have the
// coefficients (or possibly do not even know if the objective function is
// linear) in a real black box problem.
  int nonlin_obj = 0;
// Output the header common to all problems
  o << "#include \"problem.h\"\n#include <math.h>\n\n/* USER INPUT: */\n/* min "
   "cT (x, F(x)) */\nDOUBLE c[DIMX+DIMY]={";
```

```
// Count equality and inequality constraints
  for(std::vector<expression_walker>::const_iterator b =
   DAG.constraints.begin();
      b != DAG.constraints.end(); ++b)
  {
    if((*b)->is(ex_linear) || (*b)->is(ex_nonlin)) {
      if ((*b)->f_bounds.isPoint())
        dimeqconstr++;
      else
        dimconstr++;
    }
  }

// Detect if we need to do the variable substitution for the objective function,
// then output the coefficients c
  expression_walker obj=DAG.objective;
  if (!DAG.ocoeff) {
    for (int i=0; i<dimx+dimconstr; i++) o << "0.,";
// linear objective not implemented
//  } else if (!obj->is(ex_nonlin)) {
//    for (int i=0; i<dimx; i++) o << ???;
//    for (int i=0; i<dimconstr; i++) o << "0.,";
  } else {
    nonlin_obj = 1;
    for (int i=0; i<dimx; i++) o << "0.,";
    o << DAG.obj_mult() * DAG.ocoeff << ",";
    for (int i=0; i<dimconstr; i++) o << "0.,";
  }
  o << "};\n";

// We ignore (comment out) DAG.obj_adj() as it does not change the solution, and
// bbowda cannot handle it.
  if ((d = DAG.obj_adj() * DAG.ocoeff)) {
    o << "/* ";
    if(d > 0)
      o << "+ " << d;
    else if(d < 0)
      o << "- " << fabs(d);
    o << " */\n";
  }

// Collect the bounds for the variables and constraints, except for the implicit
// equality constraints for which we handle them later. For variables, we do not
// know if we have bounds, so we initialize them to infinite bounds (actually
// the finite replacement), constraints always have a bound interval, which may
// of course be infinite in one direction (or even both, though that makes the
// constraint redundant). The objective function also always carries a bound,
// though it is usually just [-inf,inf].
  double lbounds[dimx];
```

```
  double ubounds[dimx];
  for (int i=0; i<dimx; i++) {
    lbounds[i] = -INF_BOUND;
    ubounds[i] = INF_BOUND;
  }
  double clbounds[dimconstr];
  double cubounds[dimconstr];
  double *pclbounds = clbounds, *pcubounds = cubounds;

  for(std::vector<expression_walker>::const_iterator b =
   DAG.constraints.begin();
      b != DAG.constraints.end(); ++b)
  {
    interval bounds = (*b)->f_bounds;
    bool iscon = ((*b)->is(ex_linear) || (*b)->is(ex_nonlin));
    if (iscon && bounds.isPoint()) continue;
    double *plbound = iscon ? (pclbounds++) : lbounds + (*b)->params.nn();
    double *pubound = iscon ? (pcubounds++) : ubounds + (*b)->params.nn();
    double lbound = bounds.inf();
    *plbound = isinf(lbound) ? -INF_BOUND : lbound;
    double ubound = bounds.sup();
    *pubound = isinf(ubound) ? INF_BOUND : ubound;
  }

// Now output the bounds we collected
  o << "/* s.t. Flow <= F(x) <= Fup */\nDOUBLE Flow[DIMY]={";
  if (nonlin_obj) {
    double objlbound = DAG.objective->f_bounds.inf();
    o << (isinf(objlbound) ? -INF_BOUND : objlbound) << ",";
  }
  for (int i=0; i<dimconstr; i++) o << clbounds[i] << ",";
  o << "};\nDOUBLE Fup[DIMY]={";
  if (nonlin_obj) {
    double objubound = DAG.objective->f_bounds.sup();
    o << (isinf(objubound) ? INF_BOUND : objubound) << ",";
  }
  for (int i=0; i<dimconstr; i++) o << cubounds[i] << ",";
  o << "};\n/*      xlow <= x <= xup */\nDOUBLE xlow[DIMX]={";
  for (int i=0; i<dimx; i++) o << lbounds[i] << ",";
  o << "};\nDOUBLE xup[DIMX]={";
  for (int i=0; i<dimx; i++) o << ubounds[i] << ",";

// Generate the function evaluation function
  o << "};\n\n/* starting points */\nDOUBLE initpts[NUMINITPTS][DIMX]={};\n\n/*"
   " evaluate F(x) */\nvoid evaluate_F(const DOUBLE *x, DOUBLE *F)\n{\n";
// If we had to make the variable substitution, the first component is the
// objective function.
  if (nonlin_obj) {
    o << "  *(F++) = ";
```

```
    recursive_walk(DAG.objective, p);
    o << ";\n";
  }
// The next components are the inequality constraints.
  for(std::vector<expression_walker>::const_iterator b =
   DAG.constraints.begin();
      b != DAG.constraints.end(); ++b)
  {
    if(((*b)->is(ex_linear) || (*b)->is(ex_nonlin)) &&
     !(*b)->f_bounds.isPoint()) {
      o << "  *(F++) = ";
      recursive_walk(*b, p);
      o << ";\n";
    }
  }
// The last components are the equality constraints.
// We require them in the F2(x)=0 form, so the point interval for the bounds
// (i.e. the right hand side) is subtracted.
  for(std::vector<expression_walker>::const_iterator b =
   DAG.constraints.begin();
      b != DAG.constraints.end(); ++b)
  {
    if(((*b)->is(ex_linear) || (*b)->is(ex_nonlin)) && (*b)->f_bounds.isPoint())
     {
      o << "  *(F++) = ";
      recursive_walk(*b, p);
      d = -(*b)->f_bounds.inf();
      if(d > 0)
        o << " + " << d;
      else if(d < 0)
        o << " - " << fabs(d);
      o << ";\n";
    }
  }
  o << "}\n";

// Finally, output the probldim.h file (i.e. the problem dimensions) to stderr
// so both problem.c and probldim.h can be generated in one go and separately
// redirected.
  std::cerr << "#pragma once\n\n/* USER INPUT: Problem dimensions */\n#define "
   "DIMX "
            << dimx << "\n#define DIMY " << (dimconstr + nonlin_obj) <<
             "\n#define DIMY_EQ "
            << dimeqconstr << "\n\n/* USER INPUT: Number of starting points "
             "*/\n#define NUMINITPTS 0\n\n"
            "/* USER INPUT: Maximum points to evaluate */\n#define MAXPTS "
             "100\n\n"
            "/* USER INPUT: Tolerance for optimum feasibility */\n#define "
             "OPTIMUM_TOL .001\n\n"
```

```
            "/* USER INPUT: If defined, ignore equality constraints for global "
             "search */\n"
            "#undef GLOBAL_SEARCH_IGNORES_EQ_CONSTRAINTS\n\n"
            "/* USER INPUT: Tolerance for the constraints estimating the "
             "equality constraints\n"
            "               during global search */\n#define "
             "ESTIMATE_CONSTRAINT_TOL .01\n";
// End of the section rewritten by Kevin Kofler

// Mihaly, 12.12.2003
// restore default interval output precision
  interval::precision(intprecsave);

}

} // namespace coco_rm
```

One problem we encountered is that many variables in the test problems lack one or both bounds. (In some cases, the variables are truly unbounded, in others, bounds easily follow from the constraints.) Our algorithm, however, can only work with finite bounds. Moreover, it is important for the bounds for the x variables to be as close together as possible, as our algorithm performs better when more of the x within the bounds are also within the bounds for $y = F_1(x)$. This is mainly due to the fact that by design, our global search only handles implicit equality constraints specially, not inequality constraints (i.e. the bounds on the variables given by explicit equality constraints). The reasons for this tradeoff are explained in sections 3.2.1 and 3.2.5. Therefore, artificially large bounds like $[-1000, 1000]$ are not usable for most problems, at least for x. (The bounds for y are less sensitive because they are only used in the local surrogate models where locality is ensured by the covariance ellipsoid, so those bounds being larger than necessary is not a big deal.) Thus, we set reasonable bounds for the variables by hand where they are missing. Any added bounds will be presented together with the results below. The number of function values needed for convergence was determined by trial and error, future work can be put into termination criteria (see section 6.2).

In addition, we ran some simple scalability tests. In the first one, we used a trivial quadratic objective and used the usual variable substitution to turn it into the explicit equality constraint $y = F_1(x) = \sum_i \left(x_i - \left(\sqrt{2} - 1\right)\right)^2$ and the linear objective function y. We chose the bound constraints $[-1, 1]$ on x_i and $[-d, 2d]$, where d is the dimension of the vector x, on y. (The bounds on y are not a true constraint because $x_i \in [-1, 1]$ implies $y \in [0, 2d]$). We also tried the bounds $[-d, d]$ for y. We used the following `probldim.h`:

```
/* USER INPUT: Problem dimensions */
#define DIMX 1 /* scale this to scale the problem */
#define DIMY 1
```

```
#define DIMY_EQ 0

/* USER INPUT: Number of starting points */
#define NUMINITPTS 0

/* USER INPUT: Maximum points to evaluate */
#define MAXPTS 50

/* USER INPUT: Tolerance for optimum feasibility */
#define OPTIMUM_TOL .001

/* USER INPUT: If defined, ignore equality constraints for global search */
#undef GLOBAL_SEARCH_IGNORES_EQ_CONSTRAINTS

/* USER INPUT: Tolerance for the constraints estimating the equality constraints
               during global search */
#define ESTIMATE_CONSTRAINT_TOL .01
```

and the following `problem.c`:

```
/* USER INPUT: */
/* min cT (x, F(x)) */
DOUBLE c[DIMX+DIMY]={[0 ... DIMX-1]=0., [DIMX]=1.};
/* s.t. Flow <= F(x) <= Fup */
DOUBLE Flow[DIMY]={-(DOUBLE)DIMX};
DOUBLE Fup[DIMY]={(DOUBLE)(2*DIMX)};
/*      xlow <= x <= xup */
DOUBLE xlow[DIMX]={[0 ... DIMX-1]=-1.};
DOUBLE xup[DIMX]={[0 ... DIMX-1]=1.};

/* starting points */
DOUBLE initpts[NUMINITPTS][DIMX]={};

/* evaluate F(x) */
void evaluate_F(const DOUBLE *x, DOUBLE *F)
{
  size_t i;
  *F = 0.;
  for (i=0; i<DIMX; i++)
    *F += (x[i] - .4142135623731) * (x[i] - .4142135623731);
}
```

and `DIMX` was varied to scale the problem, `MAXPTS` was increased in increments of 50 until convergence was obtained. We measured the impact on the run time of the algorithm. In a second test, we represented the objective as an implicit rather than an explicit equality constraint:

$$\sum_{i=1}^{d-1} \left(x_i - \left(\sqrt{2} - 1 \right) \right)^2 - x_d = 0.$$

We used the following `probldim.h`:

```
#pragma once

/* USER INPUT: Problem dimensions */
#define DIMX 2 /* scale this to scale the problem */
#define DIMY 0
#define DIMY_EQ 1

/* USER INPUT: Number of starting points */
#define NUMINITPTS 0

/* USER INPUT: Maximum points to evaluate */
#define MAXPTS 200

/* USER INPUT: Tolerance for optimum feasibility */
#define OPTIMUM_TOL .001

/* USER INPUT: If defined, ignore equality constraints for global search */
#undef GLOBAL_SEARCH_IGNORES_EQ_CONSTRAINTS

/* USER INPUT: Tolerance for the constraints estimating the equality constraints
               during global search */
#define ESTIMATE_CONSTRAINT_TOL .01
```

and the following `problem.c`:

```
/* USER INPUT: */
/* min cT (x, F(x)) */
DOUBLE c[DIMX+DIMY]={[0 ... DIMX-2]=0., [DIMX-1]=1.};
/* s.t. Flow <= F(x) <= Fup */
DOUBLE Flow[DIMY]={};
DOUBLE Fup[DIMY]={};
/*      xlow <= x <= xup */
DOUBLE xlow[DIMX]={[0 ... DIMX-2]=-1., [DIMX-1]=-(DOUBLE)(DIMX-1)};
DOUBLE xup[DIMX]={[0 ... DIMX-2]=1., [DIMX-1]=(DOUBLE)(DIMX-1)*2.};

/* starting points */
DOUBLE initpts[NUMINITPTS][DIMX]={};

/* evaluate F(x) */
void evaluate_F(const DOUBLE *x, DOUBLE *F)
{
  size_t i;
  *F = 0.;
  for (i=0; i<DIMX-1; i++)
    *F += (x[i] - .4142135623731) * (x[i] - .4142135623731);
  *F -= x[DIMX-1];
}
```

As a third test, we used the multidimensional Rosenbrock function

$$y = \sum_{i=1}^{d-1} \left((1 - x_i)^2 + 100 \left(x_{i+1} - x_i^2 \right)^2 \right)$$

with the bound constraints $[-1, 2]$ on x_i and $[-1000, 2700]$ on y. We used the following `probldim.h`:

```
/* USER INPUT: Problem dimensions */
#define DIMX 2 /* scale this to scale the problem */
#define DIMY 1
#define DIMY_EQ 0

/* USER INPUT: Number of starting points */
#define NUMINITPTS 0

/* USER INPUT: Maximum points to evaluate */
#define MAXPTS 200

/* USER INPUT: Tolerance for optimum feasibility */
#define OPTIMUM_TOL .001

/* USER INPUT: If defined, ignore equality constraints for global search */
#undef GLOBAL_SEARCH_IGNORES_EQ_CONSTRAINTS

/* USER INPUT: Tolerance for the constraints estimating the equality constraints
               during global search */
#define ESTIMATE_CONSTRAINT_TOL .01
```

and the following `problem.c`:

```
/* USER INPUT: */
/* min cT (x, F(x)) */
DOUBLE c[DIMX+DIMY]={[0 ... DIMX-1]=0., [DIMX]=1.};
/* s.t. Flow <= F(x) <= Fup */
DOUBLE Flow[DIMY]={-1000.};
DOUBLE Fup[DIMY]={2700.};
/*      xlow <= x <= xup */
DOUBLE xlow[DIMX]={[0 ... DIMX-1]=-1.};
DOUBLE xup[DIMX]={[0 ... DIMX-1]=2.};

/* starting points */
DOUBLE initpts[NUMINITPTS][DIMX]={};

/* evaluate F(x) */
void evaluate_F(const DOUBLE *x, DOUBLE *F)
{
  size_t i;
```

```
  *F = 0.;
  for (i=0; i<DIMX - 1; i++)
    *F += (1. - x[i]) * (1. - x[i]) + 100. * (x[i+1] - x[i]*x[i]) * (x[i+1] -
     x[i]*x[i]);
}
```

The last, and hardest, scalability test, was the multidimensional Rosenbrock function written as an implicit equality constraint

$$\sum_{i=1}^{d-2} \left((1 - x_i)^2 + 100 \left(x_{i+1} - x_i^2 \right)^2 \right) - x_d = 0$$

with the bound constraints $[-1, 2]$ on $x_1, \ldots, x_{d-1}$ and $[-1, 1]$ on x_d. We used the following `probldim.h`:

```
/* USER INPUT: Problem dimensions */
#define DIMX 3 /* scale this to scale the problem */
#define DIMY 0
#define DIMY_EQ 1

/* USER INPUT: Number of starting points */
#define NUMINITPTS 0

/* USER INPUT: Maximum points to evaluate */
#define MAXPTS 300

/* USER INPUT: Tolerance for optimum feasibility */
#define OPTIMUM_TOL .001

/* USER INPUT: If defined, ignore equality constraints for global search */
#undef GLOBAL_SEARCH_IGNORES_EQ_CONSTRAINTS

/* USER INPUT: Tolerance for the constraints estimating the equality constraints
               during global search */
#define ESTIMATE_CONSTRAINT_TOL .01
```

and the following `problem.c`:

```
/* USER INPUT: */
/* min cT (x, F(x)) */
DOUBLE c[DIMX+DIMY]={[0 ... DIMX-2]=0., [DIMX-1]=1.};
/* s.t. Flow <= F(x) <= Fup */
DOUBLE Flow[DIMY]={};
DOUBLE Fup[DIMY]={};
/*      xlow <= x <= xup */
DOUBLE xlow[DIMX]={[0 ... DIMX-2]=-1., [DIMX-1]=-1.};
DOUBLE xup[DIMX]={[0 ... DIMX-2]=2., [DIMX-1]=1.};
```

```
/* starting points */
DOUBLE initpts[NUMINITPTS][DIMX]={};

/* evaluate F(x) */
void evaluate_F(const DOUBLE *x, DOUBLE *F)
{
  size_t i;
  *F = 0.;
  for (i=0; i<DIMX-2; i++)
    *F += (1. - x[i]) * (1. - x[i]) + 100. * (x[i+1] - x[i]*x[i]) * (x[i+1] -
     x[i]*x[i]);
  *F -= x[DIMX-1];
}
```

In this example, our primary measurement target was not so much speed, but whether convergence can be obtained at all.

We report results obtained both with DONLP2 and with Ipopt (using MUMPS as the linear solver) so the results with the different NLP optimizers can be compared, and also to reduce the influence of the performance of the third-party optimizers on our overall results.

The following versions of the third-party libraries were used to obtain the results below:

- `lp_solve` 5.5.0.10
- `donlp2_intv_dyn` dated 2006-12-12
- `Ipopt` 3.3.2
- `MUMPS` 4.7.3
- `ATLAS` 3.6.0-11.fc6 as the BLAS and LAPACK implementation

The tests were run on a Pentium 4 Northwood 2.6 GHz running a fully-updated Fedora 7. Except for `ATLAS`, for which we used the official Fedora package, the libraries were built from source.

We patched `DONLP2` to use `float.h` instead of hardcoded values for `DBL_EPSILON` and `DBL_MIN` which are only approximations because this improved the performance a lot in practice. An older version of `DONLP2` detected the values at runtime, which came up with the exact values, but caused other issues, thus the change to hardcoded values and the resulting regression. We discussed this with Prof. Spellucci, the author of `DONLP2`, but we were unable to track down why using these approximations instead of the exact values makes a difference at all. Therefore, we switched to using `float.h` as a quick fix to get rid of the regression, though this solution is not applicable upstream because it is not portable.

```
--- o8para.h.orig        2005-06-16 12:35:49.000000000 +0200
+++ o8para.h        2006-12-12 21:24:00.000000000 +0100
@@ -15,8 +15,12 @@

 /*  should be in float.h */

+#if 1
+#include <float.h>
+#else
 #define DBL_EPSILON 2.2e-16
 #define DBL_MIN 1.0e-308
+#endif


 /* Types */
```

We also patched the `lp_solve` compilation script `ccc` to enable debugging information (by changing `opts='-O3'` to `opts='-O3 -g'`) and the `Ipopt` configury and makefile to fix compilation issues (`ipopt.so` failing to link because it was not linked against `libc_nonshared.a`, which we fixed by just linking it against all the libraries it uses, which is how shared libraries are intended to be linked on Fedora:

```
--- Ipopt/src/Interfaces/Makefile.am.orig        2006-07-07 05:07:08.000000000 +0200
+++ Ipopt/src/Interfaces/Makefile.am        2006-11-18 23:39:30.000000000 +0100
@@ -37,7 +37,7 @@
         IpTNLP.hpp \
         IpTNLPAdapter.cpp IpTNLPAdapter.hpp

-libipopt_la_LIBADD = $(IPALLLIBS)
+libipopt_la_LIBADD = $(IPALLLIBS) @ADDLIBS@

 libipopt_la_DEPENDENCIES = $(IPALLLIBS)

--- Ipopt/src/Interfaces/Makefile.in.orig        2006-07-10 20:48:06.000000000 +0200
+++ Ipopt/src/Interfaces/Makefile.in        2006-11-18 23:39:34.000000000 +0100
@@ -306,7 +306,7 @@
         IpTNLP.hpp \
         IpTNLPAdapter.cpp IpTNLPAdapter.hpp

-libipopt_la_LIBADD = $(IPALLLIBS)
+libipopt_la_LIBADD = $(IPALLLIBS) @ADDLIBS@
 libipopt_la_DEPENDENCIES = $(IPALLLIBS)
 libipopt_la_LDFLAGS = $(LT_LDFLAGS)
 ADDLIBS_FILES = \
```

and `-with-mumps-dir` not working due to bad paths:

```
--- Ipopt/configure.orig        2007-06-21 17:54:36.000000000 +0200
+++ Ipopt/configure        2007-06-28 03:37:52.000000000 +0200
@@ -29314,7 +29314,11 @@
   # and we need the Fortran runtime libraries if we want to link with C/C++
   coin_need_flibs=yes
```

```
+if test "$use_mumps" = BUILD; then
   MUMPS_INCFLAGS="-I\`\$(CYGPATH_W) $coin_mumpssrcdir/MUMPS/libseq\` -I\`\$(CYGPATH_W) ...
+else
+  MUMPS_INCFLAGS="-I\`\$(CYGPATH_W) $mumps_dir/libseq\` -I\`\$(CYGPATH_W) $mumps_dir/include\`"
+fi

 fi
```

We used the following `Makefile.inc` for `MUMPS`:

```
# This file is part of MUMPS VERSION 4.6.3
# This Version was built on Thu Jun 22 13:22:44 2006
#
#
#Begin orderings

# NOTE that PORD is distributed within MUMPS by default. If you would like to
# use other orderings, you need to obtain the corresponding package and modify
# the variables below accordingly.
# For example, to have Metis available within MUMPS:
#          1/ download Metis and compile it
#          2/ uncomment (suppress # in first column) lines
#             starting with LMETISDIR,  LMETIS
#          3/ add -Dmetis in line ORDERINGSF
#             ORDERINGSF  = -Dpord -Dmetis
#          4/ Compile and install MUMPS
#             make clean; make   (to clean up previous installation)
#
#          Metis is now available as an internal ordering for MUMPS.
#

#LSCOTCHDIR = $(HOME)/JY/emilio/bin/generic
#LSCOTCH   = -L$(LSCOTCHDIR) -lesmumps  -lfax -lorder -lscotch -lsymbol -ldof -lgraph -lcommon -lm

LPORDDIR = ../PORD/lib/
IPORD    = -I../PORD/include/
LPORD    = -L$(LPORDDIR) -lpord

#LMETISDIR = /local/metis/
#IMETIS    = # Metis doesn't need include files (Fortran interface avail.)
#LMETIS    = -L$(LMETISDIR) -lmetis

# The following variables will be used in the compilation process.
#ORDERINGSF = -Dscotch -Dmetis -Dpord
ORDERINGSF  = -Dpord
ORDERINGSC  = $(ORDERINGSF)
LORDERINGS = $(LMETIS) $(LPORD) $(LSCOTCH)
IORDERINGS = $(IMETIS) $(IPORD) $(ISCOTCH)

#End orderings
########################################################################
########################################################################################

RM      = /bin/rm -f
CC      = gcc
FC = gfortran
FL = gfortran
AR = ar vr
RANLIB  = ranlib
```

```
INCSEQ = -I../libseq
LIBSEQ  =  -L../libseq -lmpiseq

LIBBLAS = -lblas
LIBOTHERS = -lpthread

#Preprocessor defs for calling Fortran from C (-DAdd_ or -DAdd__ or -DUPPER)
CDEFS   = -DAdd_

#Begin Optimization options
OPTF    = -O3 -g -fPIC -DPIC
OPTL    = -O3 -g -fPIC -DPIC
OPTC    = -O3 -g -fPIC -DPIC
#End Optimization options



INC = $(INCSEQ)
LIB = $(LIBSEQ)
LIBSEQNEEDED = libseqneeded
```

We also patched MUMPS to fix an uninitialized variable which caused segmentation faults on simple test cases:

```
--- MUMPS_4.7.3/src/dmumps_part9.F.orig         2007-05-04 15:57:44.000000000 +0200
+++ MUMPS_4.7.3/src/dmumps_part9.F        2007-10-01 01:29:25.000000000 +0200
@@ -790,6 +790,7 @@
       RINF3 = U(2)
       LORD = (JPERM(1).EQ.6)
       NUM = 0
+      I0 = 0
       DO 10 K = 1,N
         JPERM(K) = 0
         PR(K) = IP(K)
```

The applied patches can also be found next to the full source code of the implementation at `http://www.tigen.org/kevin.kofler/bbowda/`.

5.2 Results

In this section, we present the results obtained from the tests described above. In all our tests, we did not provide starting values by hand, relying instead on our automated starting point generation algorithm described in section 3.2.2.

In some cases, objective function values below the true minimum are returned. This can be explained by the fact that our algorithm can return points which are only feasible up to a given tolerance. It would be unrealistic to expect complete feasibility out of a black box algorithm, especially in the presence of implicit equality constraints.

5.2.1 COCONUT Benchmark Library 1

We ran our optimizer on two examples from library 1 of the COCONUT Benchmark: `circle` and `dispatch`.

circle

The best known result for the `circle` (*Circle Enclosing Points*) problem, found by `MINOS`, is:
`4.5742477881 at [5.3880763381, 6.3990975587, 4.5742477881]`

As there were no usable bounds for the three x variables, we provided the bounds $[0, 10]$ for all three. With `DONLP2` as the local optimizer, our algorithm found the solution:
`4.626279 at x=[5.069064,6.109449,4.626279]`
after 150 function evaluations and 17.430 seconds. (Increasing the number of allowed function evaluations did not lead to a better solution.) With `Ipopt` as the local optimizer, our algorithm found the solution:
`4.574240 at x=[5.388075,6.399094,4.574240]`
after 50 function evaluations and 52.046 seconds.

dispatch

The `dispatch` (*Economic Load Dispatch Including Transmission Losses*) problem originates from power generation. The best known solution, found by `MINOS`, is:
`3155.2879268581 at [50.0000000000, 75.4858804799, 93.2622541695, 8.7481346494]`

The model does not provide bounds for x_4 and the objective function nor an upper bound for the inequality constraint. Therefore we computed the bounds from those for the first three x variables and rounded to obtain the following bounds: $x_4 \in [-200, 320]$, the objective $y_1 \in [-1000, 7000]$ and the inequality constraint $y_2 \in [210, 730]$ ($y_2 \geq 210$ was the original inequality constraint). The objective function includes a constant term 653.1000000000000227374 which cannot be represented by our implementation, this term has to be added to the optimum function value provided by our optimizer. With `DONLP2` as the local optimizer, our algorithm found the solution:
`2502.184619 at x=[50.000000,76.060592,92.712255,8.773665]`
(i.e. an actual optimum of 3155.284619) after 250 function evaluations and 51.357 seconds. With `Ipopt` as the local optimizer, our algorithm found the solution:
`2502.173543 at x=[50.000000,75.527606,93.221193,8.749709]`
(i.e. an actual optimum of 3155.273543) after 251 function evaluations (250 plus one extrapolated point) and 1 minute 10.768 seconds.

5.2.2 COCONUT Benchmark Library 2

We ran our optimizer on three examples from library 2 of the COCONUT Benchmark: `aljazzaf`, `twobars` and `maratos`.

aljazzaf

The best known solution for the Aljazzaf example problem, found by `OQNLP`, is:
`75.0049000369 at [0.0000000000, 0.9999999940, 0.9999990004]`

We used the bounds $[-1, 1]$ for the x variables and $[0, 500]$ for y. With `DONLP2` as the local optimizer, our algorithm found the solution:
`74.962735 at x=[0.000209,0.999946,0.999368]`
after 201 function evaluations (200 plus one extrapolated point) and 2 minutes 1.482 seconds. With `Ipopt` as the local optimizer, our algorithm failed to find a point satisfying the feasibility tolerance. After 250 function evaluations and 19 minutes 32.282 seconds, extrapolation produced the 251^{st} point `[-0.000108,0.999998,1.000075]` which still fails to satisfy the tolerance.

twobars

The `twobars` (*Structural analysis of the simplest two bar scheme*) problem originates from mechanics. The best known solution, found by `DONLP2`, is:
`1.50865 at [1.41163, 0.377072]`

This model already provides usable bounds for the x variables, for y we kept the default $[-1000, 1000]$ from our converter. With `DONLP2` as the local optimizer, our algorithm found the solution:
`1.557142 at x=[1.497032,0.286213]`
after 150 function evaluations and 8.603 seconds. (Increasing the number of allowed function evaluations did not lead to a better solution.) With `Ipopt` as the local optimizer, our algorithm found the solution:
`1.508620 at x=[1.411632,0.377006]`
after 100 function evaluations and 55.188 seconds.

maratos

The `maratos` problem is hard to solve correctly because the variables in the optimal solution have very different scales. The best known solution, found by `DONLP2`, is:
`-0.999999 at [1, -1.72277e-06]`

For this problem, we used the bounds $[-2, 2]$ for all x and y variables. We also decreased our tolerance `OPTIMUM_TOL` to 10^{-5}. (We tried decreasing it further to 10^{-6}, but our optimizer failed to find a feasible point with that low a tolerance.) In addition, we modified the final output in `main.c` to print the second x coordinate

for the retained optimum in exponential (`%lg`) format. With `DONLP2` as the local optimizer, we obtained:
`-1.000000 at x=[1.000001,-5.2591e-05]`
after 100 function evaluations and 20.791 seconds. With `Ipopt` as the local optimizer, we obtained:
`-1.000000 at x=[1.000001,-8.36745e-05]`
after 100 function evaluations and 2 minutes 13.876 seconds.

5.2.3 Scalability Tests

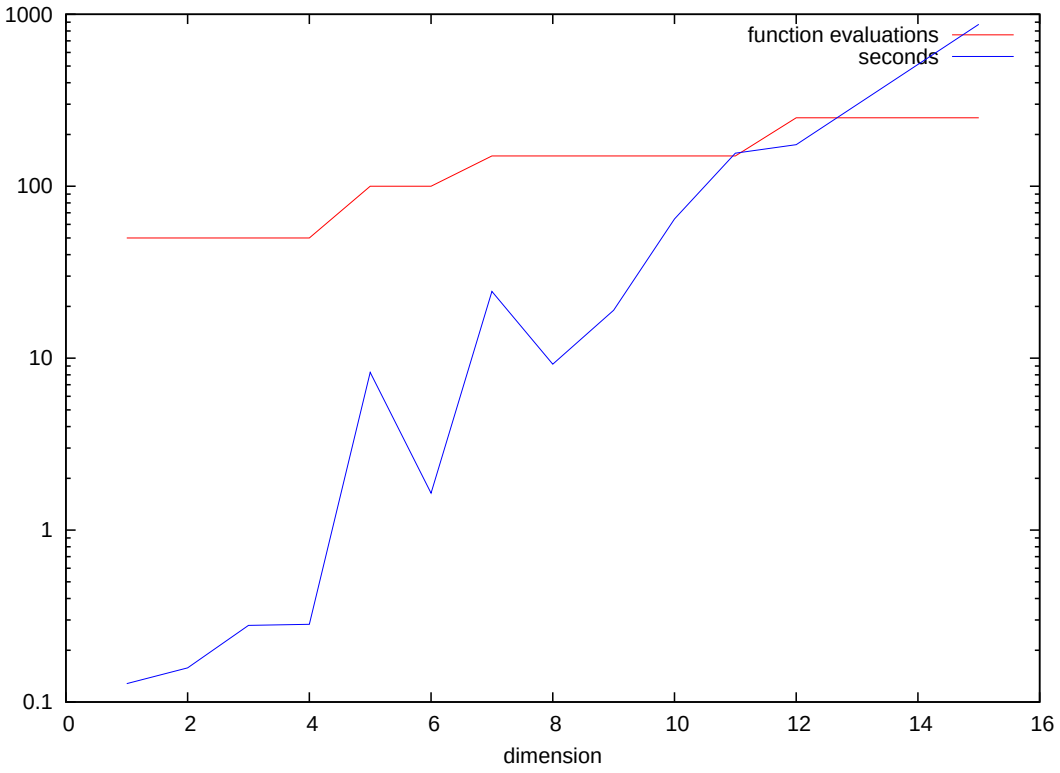
The results of the scalability tests will be presented in tabular form: d is the dimension `DIMX`, N is the number of function evaluations `MAXPTS`, which we increased in steps of 50 until convergence was obtained. For every d, we present N, the obtained result and the computation time.

Quadratic Objective as Explicit Equality Constraint

With the bounds $[-d, 2d]$ and `DONLP2`, we obtained the following results:

d	N	result	timing
1	50	0.000000 at x=[0.414214]	0m0.128s
2	50	0.000000 at x=[0.414213, 0.414213]	0m0.158s
3	50	0.000000 at x=[0.414212, 0.414213, 0.414214]	0m0.279s
4	50	0.000000 at x=[0.414206, 0.414213, 0.414211, 0.414213]	0m0.283s
5	100	0.000000 at x=[0.414174, 0.414214, 0.414213, 0.414191, 0.414285]	0m8.271s
6	100	0.000000 at x=[0.414233, 0.414241, 0.414197, 0.414209, 0.414223, 0.414203]	0m1.639s
7	150	0.000006 at x=[0.415589, 0.413880, 0.414325, 0.414730, 0.415262, 0.414554, 0.415660]	0m24.437s
8	150	0.000004 at x=[0.414313, 0.415250, 0.414286, 0.412777, 0.414215, 0.413806, 0.414707, 0.414004]	0m9.224s
9	150	0.000024 at x=[0.416683, 0.414457, 0.415900, 0.413471, 0.414682, 0.415776, 0.412320, 0.414312, 0.411350]	0m19.006s
10	150	0.002135 at x=[0.410454, 0.424900, 0.418432, 0.425663, 0.376545, 0.420193, 0.400081, 0.405694, 0.406608, 0.422729]	1m4.449s
11	150	0.005927 at x=[0.423523, 0.394638, 0.449685, 0.450797, 0.413286, 0.370070, 0.409869, 0.430780, 0.416167, 0.438829, 0.417056]	2m35.631s

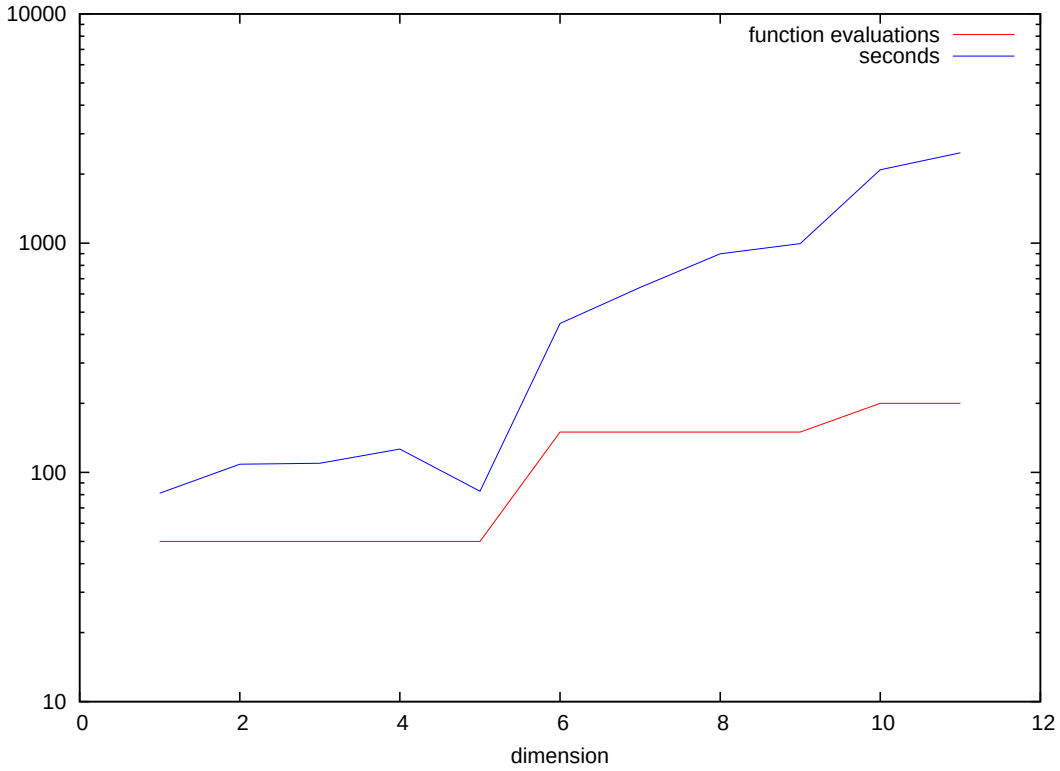
12	250	0.005325 at x=[0.421083, 0.416517, 0.396125, 0.439030, 0.403158, 0.406701, 0.399735, 0.450793, 0.446549, 0.416258, 0.375864, 0.405128]	2m54.620s
15	250	0.010697 at x=[0.379226, 0.435817, 0.416116, 0.451497, 0.426227, 0.431306, 0.397100, 0.380168, 0.368259, 0.403399, 0.454033, 0.440594, 0.446904, 0.413738, 0.402165]	14m33.505s



With the bounds $[-d, 2d]$ and `Ipopt`, we obtained the following results:

d	N	result	timing
1	50	0.000000 at x=[0.414214]	1m21.161s
2	50	0.000000 at x=[0.414214, 0.414214]	1m48.575s
3	50	0.000000 at x=[0.414212, 0.414213, 0.414214]	1m49.518s
4	50	0.000000 at x=[0.414214, 0.414206, 0.414217, 0.414218]	2m6.401s
5	50	0.000000 at x=[0.414212, 0.414209, 0.414217, 0.414217, 0.414215]	1m22.742s
6	150	0.001864 at x=[0.424792, 0.395376, 0.405232, 0.446523, 0.405346, 0.400285]	7m26.544s
7	150	0.000082 at x=[0.415469, 0.413441, 0.412501, 0.421436, 0.409857, 0.411803, 0.414716]	10m41.075s

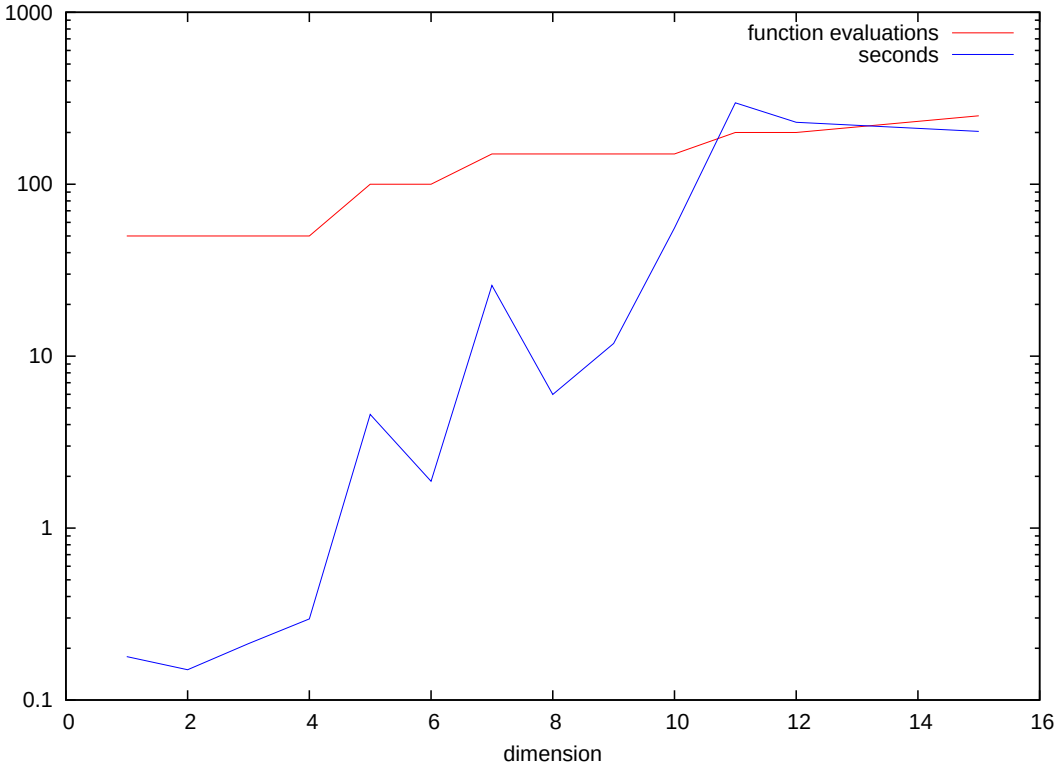
8	150	0.090652 at x=[0.476044, 0.377929, 0.469670, 0.595526, 0.415317, 0.223353, 0.466328, 0.312146]	14m58.880s
9	150	0.005280 at x=[0.463490, 0.391940, 0.424878, 0.410557, 0.432676, 0.396016, 0.426962, 0.376995, 0.411197]	16m35.947s
10	200	0.008180 at x=[0.408675, 0.364499, 0.439656, 0.373635, 0.427705, 0.446480, 0.408794, 0.433978, 0.388117, 0.446762]	34m49.248s
11	200	0.027408 at x=[0.413696, 0.422270, 0.459571, 0.423165, 0.386822, 0.341910, 0.471872, 0.444237, 0.306103, 0.451472, 0.370334]	41m19.946s



With the bounds $[-d, d]$ and `DONLP2`, we obtained the following results:

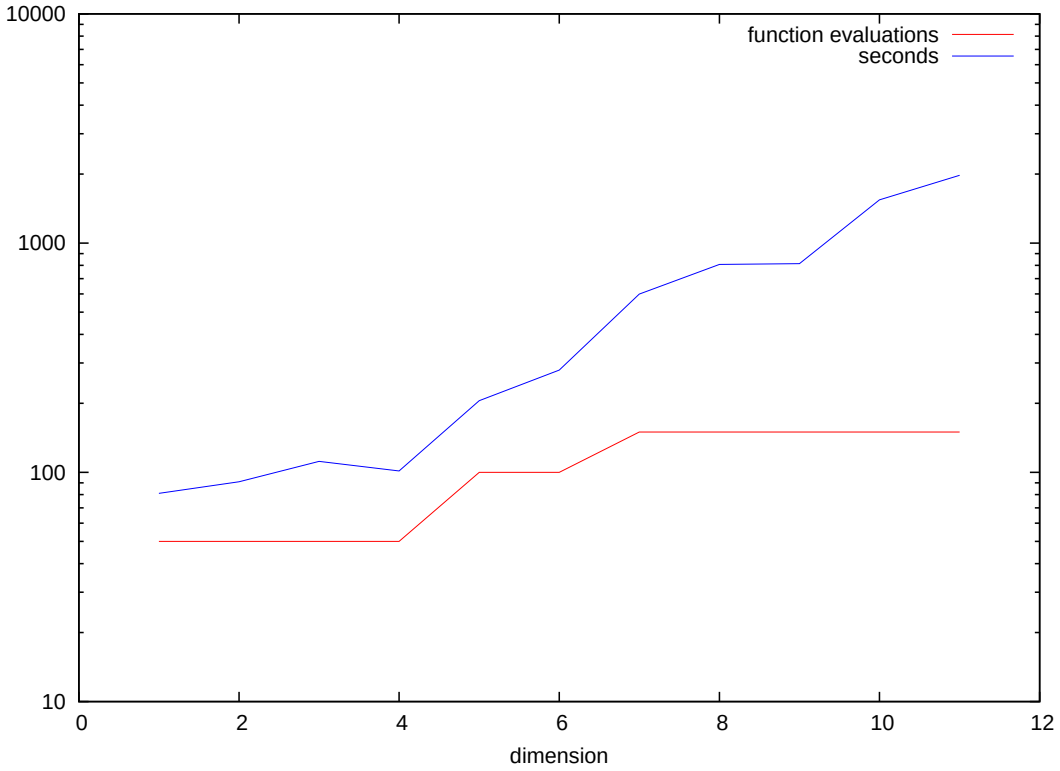
d	N	result	timing
1	50	0.000000 at x=[0.414214]	0m0.179s
2	50	0.000000 at x=[0.414213, 0.414213]	0m0.150s
3	50	0.000000 at x=[0.414209, 0.414212, 0.414214]	0m0.213s
4	50	0.000000 at x=[0.414210, 0.414209, 0.414209, 0.414209]	0m0.296s
5	100	0.000000 at x=[0.414181, 0.414182, 0.414248, 0.414230, 0.414236]	0m4.580s

6	100	0.000000 at x=[0.414258, 0.414215, 0.414191, 0.414175, 0.414201, 0.414188]	0m1.873s
7	150	0.000003 at x=[0.414214, 0.414372, 0.413694, 0.414045, 0.415357, 0.413341, 0.414615]	0m25.771s
8	150	0.000005 at x=[0.413441, 0.414298, 0.414133, 0.413661, 0.415208, 0.412796, 0.413929, 0.415292]	0m5.986s
9	150	0.000033 at x=[0.416754, 0.413384, 0.416301, 0.411805, 0.416508, 0.412610, 0.415267, 0.412961, 0.416480]	0m11.858s
10	150	0.000014 at x=[0.415348, 0.414646, 0.412863, 0.414172, 0.414913, 0.414468, 0.411721, 0.413994, 0.416010, 0.414990]	0m55.638s
11	200	0.000120 at x=[0.416546, 0.412628, 0.412814, 0.416710, 0.415514, 0.413084, 0.409532, 0.422484, 0.417369, 0.414557, 0.414801]	4m57.741s
12	200	0.001256 at x=[0.416139, 0.416125, 0.422931, 0.405190, 0.418267, 0.431585, 0.431566, 0.400974, 0.405323, 0.407736, 0.424156, 0.405428]	3m49.261s
15	250	0.000030 at x=[0.412119, 0.416130, 0.415487, 0.415483, 0.412698, 0.415777, 0.410633, 0.414821, 0.413724, 0.414758, 0.414624, 0.413491]	3m22.779s



With the bounds $[-d, d]$ and `Ipopt`, we obtained the following results:

d	N	result	timing
1	50	0.000000 at x=[0.414214]	1m20.927s
2	50	0.000000 at x=[0.414214, 0.414214]	1m30.977s
3	50	0.000000 at x=[0.414209, 0.414212, 0.414214]	1m51.573s
4	50	0.000000 at x=[0.414209, 0.414172, 0.414223, 0.414242]	1m41.482s
5	100	0.000101 at x=[0.415401, 0.409436, 0.408749, 0.415721, 0.407515]	3m25.474s
6	100	0.039120 at x=[0.435564, 0.247794, 0.465127, 0.371406, 0.484499, 0.374159]	4m39.175s
7	150	0.008043 at x=[0.436211, 0.431775, 0.376922, 0.347588, 0.394180, 0.382644, 0.418994]	9m58.997s
8	150	0.002461 at x=[0.434466, 0.396184, 0.448527, 0.401424, 0.395998, 0.413056, 0.421308, 0.415187]	13m27.480s
9	150	0.000392 at x=[0.409553, 0.419314, 0.424432, 0.417257, 0.401716, 0.410869, 0.411915, 0.421718, 0.413079]	13m33.714s
10	150	0.004097 at x=[0.406398, 0.409782, 0.399265, 0.397437, 0.418756, 0.416686, 0.463113, 0.430380, 0.402358, 0.387912]	25m44.301s
11	150	0.034805 at x=[0.447537, 0.452834, 0.327736, 0.399374, 0.440426, 0.335542, 0.409214, 0.365755, 0.408743, 0.417146, 0.290858]	32m55.416s



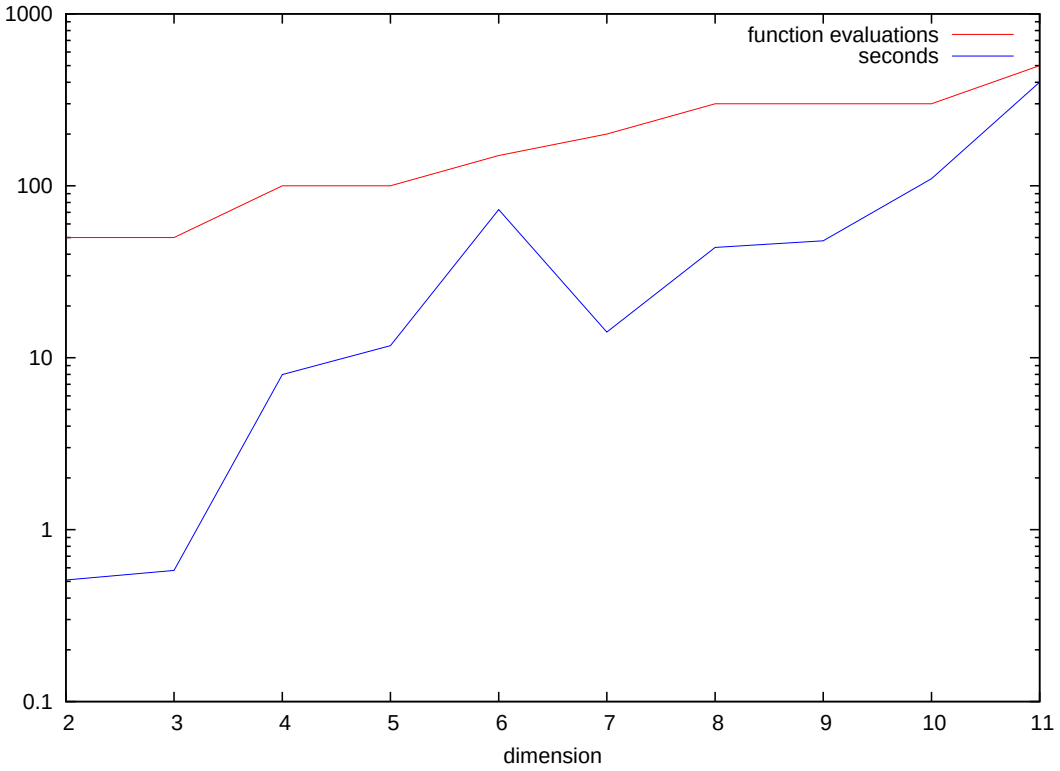
For $d = 6$, setting $N = 150$ did not improve the result either. Forcing our stronger regularization for higher dimensions to start at $d = 6$ instead of $d = 7$ improved the result for $d = 6$ and $N = 150$ to `0.000003 at x=[0.413465,0.414506,0.415319,0.414102,0.415094,0.413376] in 7m11.201s`, however doing so would break the convergence for other problems, notably the 6-dimensional Rosenbrock function with `DONLP2` as the local optimizer, so we opted against that change.

Quadratic Objective as Implicit Equality Constraint

To get these examples to converge in a timely manner for dimensions 7 and higher, a few tweaks were needed. First of all, we had to set the `GLOBAL_SEARCH_IGNORES_EQ_CONSTRAINTS` option for dimension 7 and higher, because the LPs needed to approximate the implicit equality constraint for the global search took too long to compute: without this option, solving $d = 7$ with `DONLP2` appeared to converge, but took over 3 hours! `lp_solve` also ran into numerical problems in higher dimensions, which means most of that time was wasted without finding actual enclosures for our constraint. And secondly, we had to increase the tolerance `OPTIMUM_TOL` (abbreviated as "tol" in the tables) from the default .001 in higher dimensions because the tolerance could not be reached, possibly partly due to the global search not being able to take the

constraint into account. With `DONLP2`, we obtained the following results:

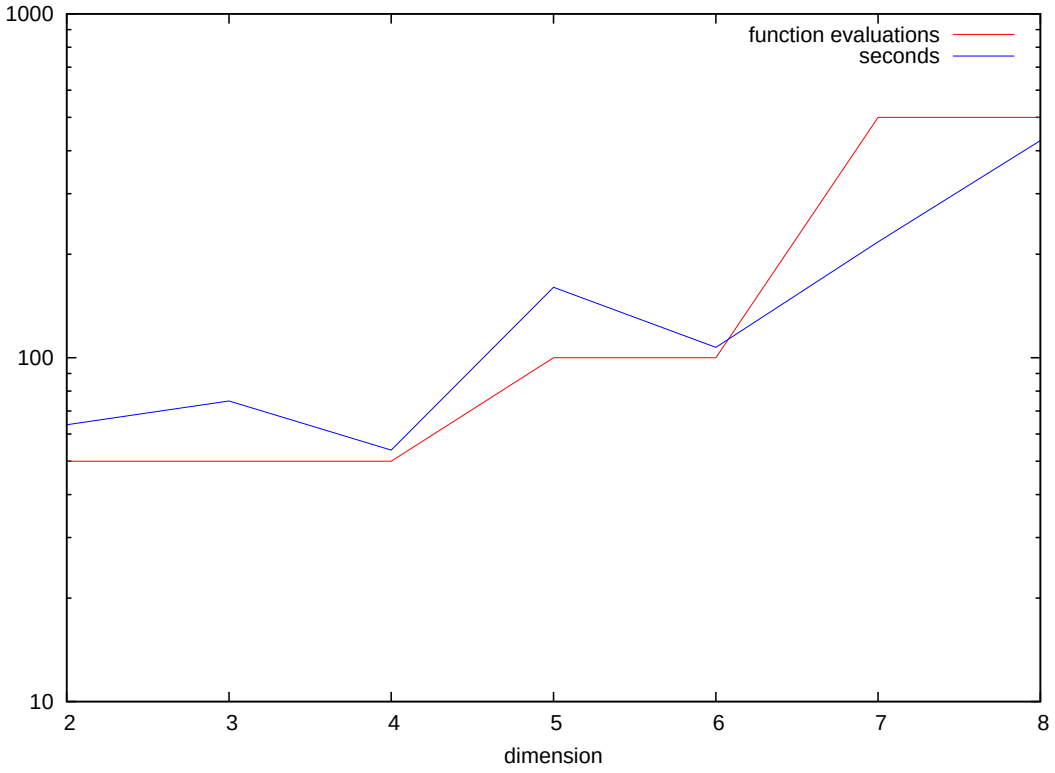
d	N	tol	result	timing
2	50	.001	-0.000608 at x=[0.414239, -0.000608]	0m0.509s
3	50	.001	-0.000804 at x=[0.414217, 0.414164, -0.000804]	0m0.579s
4	100	.001	0.000008 at x=[0.414357, 0.414898, 0.412877, 0.000008]	0m7.986s
5	100	.001	-0.000984 at x=[0.414299, 0.415849, 0.414732, 0.413624, -0.000984]	0m11.750s
6	150	.001	0.007643 at x=[0.396069, 0.455927, 0.455240, 0.420797, 0.345321, 0.007643]	1m12.683s
7	200	.01	0.000680 at x=[0.416074, 0.410845, 0.415781, 0.405332, 0.410764, 0.400547, 0.000680]	0m14.112s
8	300	.01	-0.001772 at x=[0.413609, 0.411803, 0.414844, 0.412973, 0.413767, 0.411765, 0.411749, -0.001772]	0m43.814s
9	300	.1	-0.034728 at x=[0.413816, 0.401053, 0.410315, 0.397650, 0.415350, 0.411022, 0.466831, 0.447111, -0.034728]	0m47.906s
10	300	.1	-0.039629 at x=[0.421163, 0.445016, 0.396696, 0.428404, 0.475153, 0.401993, 0.465980, 0.487537, 0.447921, -0.039629]	1m50.080s
11	500	.5	-0.046004 at x=[0.580370, 0.614280, 0.345878, 0.299603, 0.434615, 0.388781, 0.437947, 0.351515, 0.504935, 0.373158, -0.046004]	6m40.942s



The reduction in computation time between dimensions 6 and 7 is due to the fact that we disabled the LPs for the enclosure of the implicit equality constraint in the global search for dimension 7 and higher, otherwise the computation time would have increased significantly instead.

With `Ipopt`, we obtained the following results:

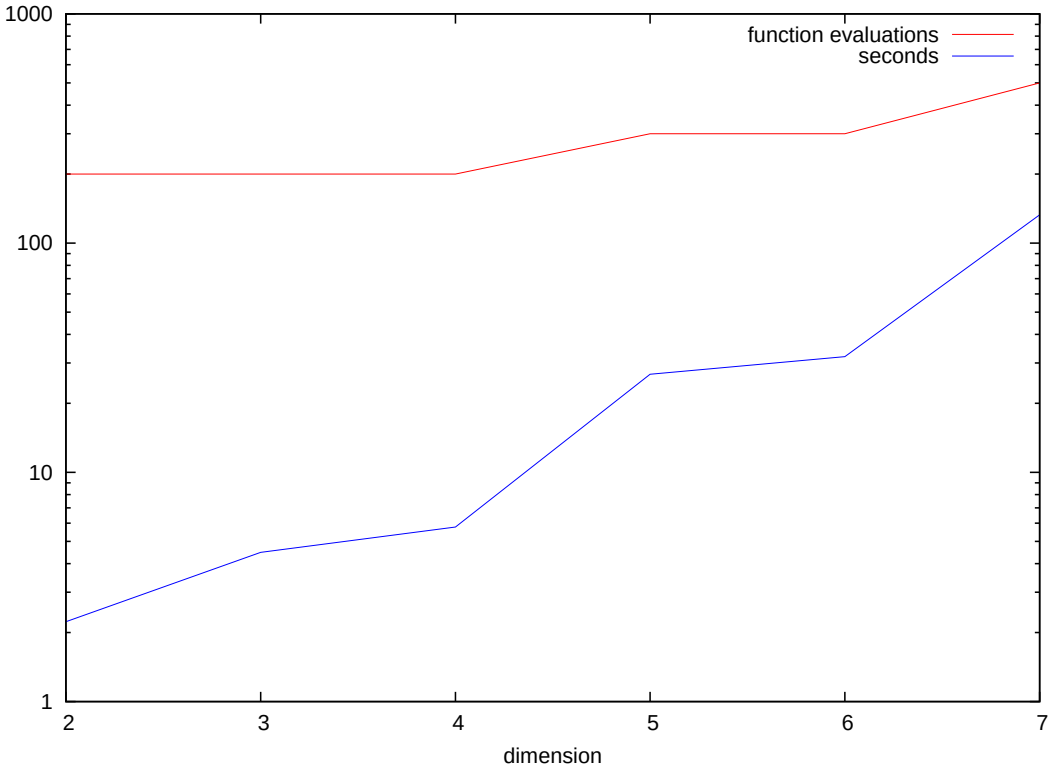
d	N	tol	result	timing
2	50	.001	-0.000629 at x=[0.413773, -0.000629]	1m3.856s
3	50	.001	-0.000781 at x=[0.414208, 0.414219, -0.000781]	1m14.854s
4	50	.001	-0.000599 at x=[0.414420, 0.414073, 0.413959, -0.000599]	0m53.872s
5	100	.001	0.000139 at x=[0.411850, 0.413529, 0.419301, 0.400990, 0.000139]	2m40.308s
6	100	.001	-0.000747 at x=[0.414288, 0.413886, 0.415680, 0.413546, 0.413768, -0.000747]	1m47.122s
7	500	.1	-0.098857 at x=[0.424909, 0.418121, 0.427544, 0.435220, 0.414645, 0.394952, -0.098857	3m37.305s
8	500	.5	-0.050436 at x=[0.405903, 0.372246, 0.260290, 0.504925, 0.207737, 0.740475, 0.222677, -0.050436]	7m8.389s



Rosenbrock Function as Explicit Equality Constraint

With `DONLP2`, we obtained the following results:

d	N	result	timing
2	200	0.001435 at x=[0.962191, 0.925574]	0m2.230s
3	200	0.011359 at x=[0.955130, 0.909057, 0.827020]	0m4.474s
4	200	0.008234 at x=[0.980435, 0.960568, 0.922109, 0.851505]	0m5.777s
5	300	0.046058 at x=[0.975658, 0.952061, 0.909707, 0.833079, 0.699526]	0m26.803s
6	300	0.020165 at x=[0.994074, 0.986399, 0.970718, 0.943735, 0.886766, 0.788724]	0m31.959s
7	500	0.053847 at x=[0.994816, 0.987393, 0.973595, 0.947083, 0.896174, 0.803800, 0.646551]	2m12.756s



With `Ipopt`, we could not obtain convergence even for $d = 2$:

d	N	result	timing
2	200	0.135674 at x=[0.631745, 0.399890]	6m7.647s

Higher values for N didn't improve the solution either.

Rosenbrock Function as Implicit Equality Constraint

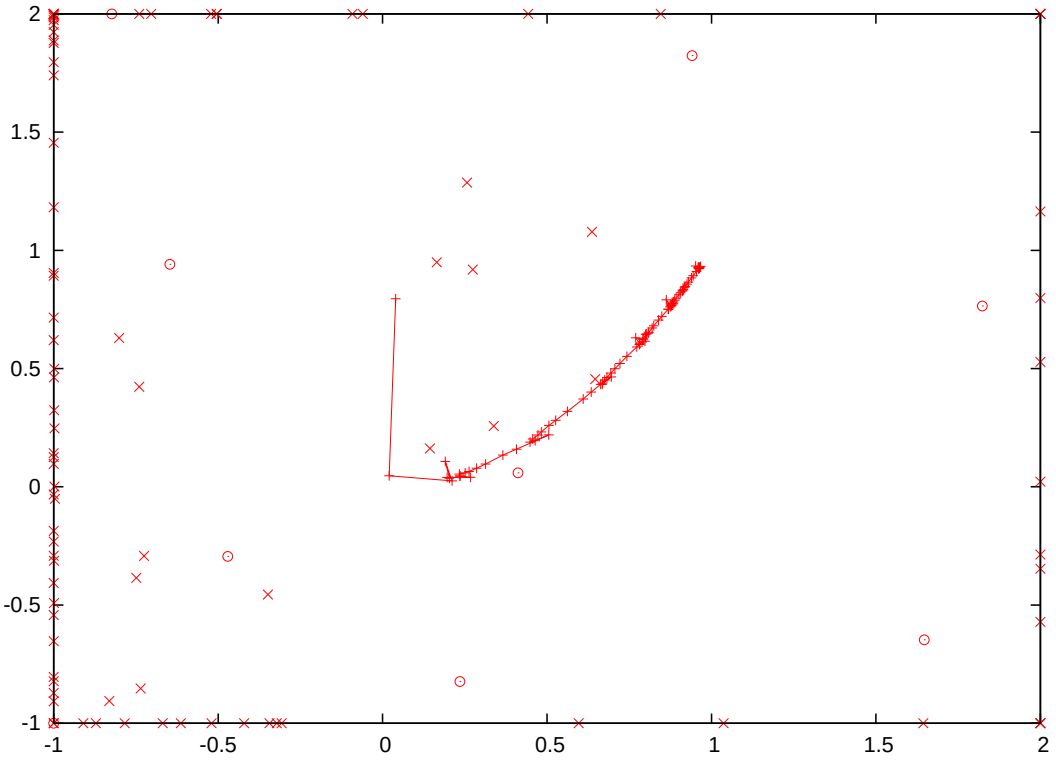
With `DONLP2`, we obtained the following results:

d	N	result	timing
3	100	0.018488 at x=[1.016196,1.019243,0.018488]	0m4.896s
3	200	0.000040 at x=[0.995731,0.991338,0.000040]	0m18.634s
4	does not converge		

With `Ipopt`, we could not obtain convergence even for $d = 3$.

5.2.4 Graphical Representation

The following figure shows the points evaluated for the original 2-dimensional Rosenbrock function expressed as an explicit equality constraint using `DONLP2` as the local optimizer. The circles represent the starting points, the X-es the points found by the global search and the pluses connected by a line the points found by the local search.



Chapter 6

Conclusion

In this chapter, we will wrap up our results and give an outlook about possible future improvements to the algorithm.

6.1 Summary

We presented an algorithm applying methods from data analysis to the solution of black box optimization problems. Using quadratic covariance models for heuristic local function enclosure and Gaussian mixture models for global density estimation, we constructed an optimization algorithm which can handle both objective functions and constraints without any gradient or Hessian information. As we need not compute any difference quotients, the number of required function evaluations is low, so the algorithm lends itself particularly well to functions which are expensive to evaluate. Our algorithm is an incomplete global optimization algorithm: completeness cannot be achieved due to the lack of global information and any asymptotical properties are of limited use because we expect our optimizer to be used with expensive functions and thus a low cap on the number of function evaluations. Thus, we did not pursue asymptotical completeness, but instead focused on density estimation aiming at filling the gaps in the search space in an optimal way given a finite number of points, and taking the points found by the local search into account.

We also presented a way to handle black box implicit equality constraints: our local covariance models are formulated in a way naturally accounting for them; for our global density models, we solve linear programs to find heuristic quadratic over- and underestimators for each implicit equality constraint, which provide enclosures for the feasible set. As the enclosures cannot be made rigorous due to the lack of global information, we adjust them for each newly computed point, ensuring that all known points are always within the current enclosures. As black box implicit

equality constraints are still an active field of research, this is a significant result.

We documented a working implementation of the above concepts, written in the ISO C99 language [43, 44] and licensed under the GNU General Public License, version 3 [45] or later, with special exceptions allowing to link with the third-party optimizers used. The required third-party libraries are `lp_solve` for linear programs and either `DONLP2` or `Ipopt`, which requires `MUMPS` and implementations of `BLAS` and `LAPACK` (we recommend `ATLAS`, which not only performs better than the reference implementations, but also gave more accurate results in our tests), for the non-linear surrogate and density models.

Finally, we tested our algorithm on a few low-dimensional example programs from the COCONUT benchmark and performed some scalability tests. We obtained satisfying results on the problems we tested: on the five low-dimensional problems from the COCONUT benchmark we tested, convergence was obtained in all cases, though we were unable to solve the Aljazzaf and Maratos examples to full precision. The algorithm scales up to a dimension of around 15 for the simple quadratic objective. It is also able to optimize the multidimensional Rosenbrock function up to dimension 7. The problems with implicit equality constraints did not scale as well, however we were able to optimize the 2-dimensional Rosenbrock function as a 3-dimensional implicit equation to a very high precision. We observed a strong dependence of the results on the local optimizer used: for some test cases, the best result was obtained with `DONLP2`, for others with `Ipopt`; the result with the other optimizer was often significantly worse.

6.2 Outlook

While we were successful at proving the concept and obtained some very promising results, there are several potential improvements which can be the subject of future research.

As can be seen in the graphical representation, our global search produces many points on the border of the box being searched and few points in the interior. Fine-tuning the global search to produce more interior points is likely to improve global convergence.

Another possible improvement may be obtained for the stopping criterion: currently, our algorithm always evaluates exactly the maximum number of points, it is unable to verify whether convergence has already occurred or not. It is possible to obtain gradient estimates from the covariance matrices we compute: As our covariance models are centered around the best point, the covariance $C_{x_i y_j}$ represents the expected value of $(x_i - x_{\text{best}\,i})\left(y_j - y_{\text{best}\,j}\right)$, the variance $S_{x_i} = C_{x_i x_i}$ the expected value of $(x_i - x_{\text{best}\,i})^2$, which allows us to approximate the directional derivative $\frac{\partial y_j}{\partial x_i}$ at x_{best} by the quotient $\frac{C_{x_i y_j}}{S_{x_i}}$. We can also generalize this idea to

the multi-dimensional case and use the solution g of the linear system of equations $C_{xx}g = C_{y_jx}$ as an estimate for the gradient of y_j, which is more expensive to compute, but more accurate, because it takes the correlation within the x variables into account. (Gradient estimates can also be obtained in other ways, e.g. from the eigenspaces of the covariance matrix.) These estimates may be used to verify the Kuhn-Tucker conditions, and thus provide a stopping criterion.

Another possibility for the stopping criterion problem would be to allow for interactively stopping the algorithm and still outputting the current optimum and also to make it easy to restart the algorithm after an interruption. This might be useful even in the presence of an improved automated stopping criterion, because the user usually knows best when he or she is satisfied with the result. We did not implement this feature in our current implementation because our main goal was to experiment with the mathematical concepts, not to implement a production optimizer.

Implicit equality constraints in higher dimensions are another place where there is definitely room for improvement. Most, if not all, algorithms currently on the market fail to handle black box implicit equality constraints, so our algorithm is pioneering this domain. In the lowest dimensions, our algorithm handles such constraints very well, but in medium to high dimensions (starting at around 5 to 7), we ran into both numerical and speed-related difficulties with our linear programs. One approach might be finding a way to get the size of the linear programs under control even in higher dimensions. Another approach worth trying would be solving the linear programs with an interior point method rather than the simplex method.

The extrapolation technique used to obtain feasible points with good objective function values in the presence of implicit equality constraints could also be a target for improvement: when it works, it usually produces very good points, however sometimes the extrapolation matrix is too ill-conditioned and sometimes bad input points are missed by the outlier detection and force bad extrapolated points.

Given the high sensitivity of the results to the local optimizer used for the surrogate and density models, it would be worthwhile to try other local optimizers, such as `KNITRO`. It might even be worth a try to run a global optimizer on the global density models.

Finally, the current implementation is a prototype and does not always use the most efficient algorithms for its computations. To get it to scale to higher dimensions, optimizing the implementation for speed may be worthwhile.

Bibliography

[1] H. Schichl: Optimierung. Lecture notes. Universität Wien, 2004.

[2] H. Schichl: Globale Optimierung. Lecture notes. Universität Wien, 2005.

[3] A. Neumaier: Complete Search in Continuous Global Optimization and Constraint Satisfaction. Acta Numerica 2004 (A. Iserles, ed.). Cambridge University Press 2004, pp. 271–369.

[4] J. Nocedal: Theory of algorithms for unconstrained optimization. Acta Numerica 1992 (A. Iserles, ed.), pp. 199–242. Cambridge Univ. Press, Cambridge, 1992.

[5] A. Fiacco and G.P. McCormick: Sequential Unconstrained Minimization Techniques. Classics in Applied Mathematics 4. SIAM, Philadelphia, 1990.

[6] R. Fletcher and S. Leyffer: Nonlinear programming without a penalty function. Mathematical Programming, 91:239–270, 2002.

[7] R. Fletcher, S. Leyffer and Ph.L. Toint: A Brief History of Filter Methods. SIAG/Optimization Views-and-News, 18(1):2–12, 2007.

[8] A.D. Padula: Introduction to SQP Methods. Online slides. `http://www.caam.rice.edu/~adpadu/`

[9] A.R. Conn, N.I.M. Gould, and Ph.L. Toint: Trust Region Methods (Section 15.1–15.3). MPS/SIAM Series on Optimization. SIAM, Philadelphia 2000.

[10] T.S. Motzkin and E.G. Straus: Maxima for graphs and a new proof of a theorem of turan. Canad. J. Math. 17(4):533–540, 1965.

[11] S. Kirkpatrick, C.D. Geddat, Jr., and M.P. Vecchi: Optimization by simulated annealing. Science 220 (1983), pp. 671–680.

[12] L. Ingber: Simulated annealing: Practice versus theory. Math. Comput. Modelling 18 (1993), pp. 29–57.

[13] P.J.M. Van Laarhoven and E.H.L. Aarts: Simulated Annealing: Theory and Applications. Kluwer, Dordrecht, 1987.

[14] J. Holland: Genetic algorithms and the optimal allocation of trials. SIAM J. Computing 2 (1973), pp. 88–105.

[15] S. Forrest: Genetic algorithms: principles of natural selection applied to computation. Science 261 (1993), pp. 872–878.

[16] Z. Michalewicz: Genetic Algorithm + Data Structures = Evolution Programs, 3rd ed. Springer, New York, 1996.

[17] M. Dorigo, V. Maniezzo and A. Colorni: The ant system: optimization by a colony of cooperating agents. IEEE Trans. Systems, Man, Cyber. Part B, 26 (1991), pp. 29–41.

[18] M. Dorigo, P. Balaprakash and M.A. Montes de Oca: Ant Colony Optimization. Web page. `http://www.aco-metaheuristic.org/`

[19] D.R. Jones, C.D. Perttunen and B.E. Stuckman: Lipschitzian optimization without the Lipschitz constant. J. Optimization Th. Appl. 79 (1993), pp. 157–181.

[20] A. Törn and A. Žilinskas: Global Optimization. Lecture Notes in Computer Science, Vol. 350. Springer-Verlag, Berlin, 1989.

[21] W. Huyer and A. Neumaier: Global optimization by multilevel coordinate search. J. Global Optimization 14 (1999), pp. 331–355.

[22] A. Neumaier: Mathematische Methoden der Datenanalyse. Lecture notes. Universität Wien, 2004.

[23] A. Flexer: AI Methoden der Datenanalyse. Lecture notes. Medizinische Universität Wien, 2006.

[24] P.R. Wolf and C.D. Ghilani: Adjustment Computations. John Wiley and Sons, Inc., New York, 1997.

[25] J.H. Wolfe: Pattern clustering by multivariate mixture analysis. Multivariate Behavioral Research, 5 (1970), pp. 329–350.

[26] A. Dempster, N. Laird and D. Rubin: Maximum likelihood from incomplete data via the EM algorithm. Journal of the Royal Statistical Society, Series B, 39(1):1–38, 1977.

[27] D.M.J. Tax, R.P.W. Duin: Outlier detection using classifier instability. Amin A. et al. (eds.), Advances in Pattern Recognition, Proc. Jont IAPR Int. Workshop SSPR'98 and SPR'98. Lecture Notes in Computer Science, Springer, pp. 593–601, 1998.

[28] A. Flexer, E. Pampalk and G. Widmer: Novelty detection based on spectral similarity of songs. Proceedings of 6$^{\text{th}}$ International Conference on Music Information Retrieval, pp. 260–263, September 2005.

[29] W.H. Jones: Derivation: Cubic Regression. NASA, 2000. `http://www.grc.nasa.gov/WWW/price000/lap/htm/derivation_cubicregression.html`

[30] P. Spellucci: DONLP2. Software package downloadable from `http://www.mathematik.tu-darmstadt.de:8080/ags/ag8/Mitglieder/spellucci_en.html`

[31] P. Spellucci: An SQP method for general nonlinear programs using only equality constrained subproblems. Math. Prog. 82 (1998), pp. 413–448.

[32] P. Spellucci: A new technique for inconsistent problems in the SQP method. Math. Meth. of Oper. Res. 47, (1998), 355–400. Physica Verlag, Heidelberg, Germany.

[33] A. Wächter et al.: Ipopt. Software package downloadable from `https://projects.coin-or.org/Ipopt`

[34] A. Wächter and L. T. Biegler: On the Implementation of a Primal-Dual Interior Point Filter Line Search Algorithm for Large-Scale Nonlinear Programming. Mathematical Programming 106(1), pp. 25–57, 2006.

[35] P. Amestoy, I. Duff, A. Guermouche, J. Koster, J.-Y. L'Excellent, S. Pralet et al.: MUMPS. Software package downloadable from `http://graal.ens-lyon.fr/MUMPS/`

[36] P.R. Amestoy, I.S. Duff and J.-Y. L'Excellent: Multifrontal parallel distributed symmetric and unsymmetric solvers. Comput. Methods in Appl. Mech. Eng., 184, pp. 501–520, 2000.

[37] P.R. Amestoy, I.S. Duff, J. Koster and J.-Y. L'Excellent: A fully asynchronous multifrontal solver using distributed dynamic scheduling. SIAM Journal of Matrix Analysis and Applications, Vol 23, No 1, pp. 15–41, 2001.

[38] P.R. Amestoy, A. Guermouche, J.-Y. L'Excellent and S. Pralet: Hybrid scheduling for the parallel solution of linear systems. Parallel Computing Vol 32 (2), pp. 136–156, 2006.

[39] M. Berkelaar, P. Notebaert, K. Eikland et al.: lp_solve. Software package downloadable from http://groups.yahoo.com/group/lp_solve/

[40] G.B. Dantzig: Computational Algorithm of the Revised Simplex Method. Rand Corporation, 1953.

[41] I. Nowak and S. Vigerske: LaGO. Software package downloadable from https://projects.coin-or.org/LaGO

[42] I. Nowak and S. Vigerske: LaGO - a (heuristic) Branch and Cut algorithm for nonconvex MINLPs. Preprint, Institut für Mathematik, Humboldt-Universität zu Berlin (ISSN 0863-0976), 12. http://www.mathematik.hu-berlin.de/publ/pre/2006/P-06-24.ps

[43] ISO C Working Group JTC1/SC22/WG14: ISO/IEC 9899:1999 – Programming languages – C. International Organization for Standardization, 1999.

[44] ISO C Working Group JTC1/SC22/WG14: WG14/N1124 – ISO/IEC 9899:TC2. Committee draft, 2005. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf

[45] Free Software Foundation: GNU General Public License, Version 3. Free Software Foundation, 2007. http://www.gnu.org/licenses/gpl-3.0.html

[46] GamsWorld group et al.: GLOBAL Library. GAMS World. http://www.gamsworld.org/global/globallib.htm

[47] O. Shcherbina et al.: COCONUT Benchmark. Package downloadable from http://www.mat.univie.ac.at/~neum/glopt/coconut/Benchmark/Benchmark.html

[48] O. Shcherbina, A. Neumaier, D. Sam-Haroud, Xuan-Ha Vu and Tuan-Viet Nguyen, Benchmarking global optimization and constraint satisfaction codes. Ch. Bliek, Ch. Jermann and A. Neumaier (eds.), Global Optimization and Constraint Satisfaction, pp. 211–222. Springer, Berlin 2003.

[49] H. Schichl et al.: COCONUT Environment. Software package downloadable from http://www.mat.univie.ac.at/coconut-environment/

[50] C.A. Floudas, P.M. Pardalos, C.S. Adjiman, W.R. Esposito, Z.H. Gümüs, S.T. Harding, J.L. Klepeis, C.A. Meyer and C.A. Schweiger: Handbook of Test Problems in Local and Global Optimization. Kluwer, Dordrecht 1999.

[51] B. Vanderbei, H.Y. Benson et al.: Cute Models. Package downloadable from `http://www.sor.princeton.edu/~rvdb/ampl/nlmodels/cute/index.html`

[52] Free Software Foundation: GNU Library General Public License, Version 2. Free Software Foundation, 1991. `http://www.gnu.org/licenses/old-licenses/library.html`