UNIVERSITÉ DE MONTRÉAL

OPTIMIZATION OF ALGORITHMS WITH THE OPAL FRAMEWORK

CONG KIEN DANG
DÉPARTEMENT DE MATHÉMATIQUES ET DE GÉNIE INDUSTRIEL
ÉCOLE POLYTECHNIQUE DE MONTRÉAL

THÈSE PRÉSENTÉE EN VUE DE L'OBTENTION
DU DIPLÔME DE PHILOSOPHIÆ DOCTOR
(MATHÉMATIQUES DE L'INGÉNIEUR)
JUIN 2012

UNIVERSITÉ DE MONTRÉAL

ÉCOLE POLYTECHNIQUE DE MONTRÉAL

Cette thèse intitulée :

OPTIMIZATION OF ALGORITHMS WITH THE OPAL FRAMEWORK

présentée par : DANG Cong Kien
en vue de l'obtention du diplôme de : Philosophiæ Doctor
a été dûment acceptée par le jury d'examen constitué de :

M. LE DIGABEL Sébastien, Ph.D., président
M. AUDET Charles, Ph.D., membre et directeur de recherche
M. ORBAN Dominique, Doct.Sc., membre et codirecteur de recherche
M. BASTIN Fabian, Doct.Sc., membre
M. WILD Stefan, Ph.D., membre

Acknowledgements

I am deeply grateful to my supervisors Charles Audet and Dominique Orban. Their academic knowledge, creativity, availability and particularly great efforts to explain things clearly and simply guided me through my exploration and helped shape my work in many important ways.

I would like to show my gratitude to my friends, Thu Lan and Phuong Anh, who helped me greatly during the writing process of my thesis. Not only did I take advantage of their detailed corrections but I also learned a lot to improve my writing skills.

Many thanks to everyone in GERAD, in the department of Applied Mathematics and Industrial Engineering, as well as in Center of Consultation in Mathematics who have helped me out in one way or another. A new life, new working environments, new culture gave me a lot of difficulties, without help and understanding, I could not have this final happy ending.

Finally, I would like to thank my parents, my wife and my sister of their continuous, infinite supports.

Résumé

La question d'identifier de bons paramètres a été étudiée depuis longtemps et on peut compter un grand nombre de recherches qui se concentrent sur ce sujet. Certaines de ces recherches manquent de généralité et surtout de re-utilisabilité. Une première raison est que ces projets visent des systèmes spécifiques. En plus, la plupart de ces projets ne se concentrent pas sur les questions fondamentales de l'identification de bons paramètres. Et enfin, il n'y avait pas un outil puissant capable de surmonter des difficulté dans ce domaine. En conséquence, malgré un grand nombre de projets, les utilisateurs n'ont pas trop de possibilité à appliquer les résultats antérieurs à leurs problèmes.

Cette thèse propose le cadre OPAL pour identifier de bons paramètres algorithmiques avec des éléments essentiels, indispensables. Les étapes de l'élaboration du cadre de travail ainsi que les résultats principaux sont présentés dans trois articles correspondant aux trois chapitres 4, 5 et 6 de la thèse.

Le premier article introduit le cadre par l'intermédiaire d'exemples fondamentaux. En outre, dans ce cadre, la question d'identifier de bons paramètres est modélisée comme un problème d'optimisation non-lisse qui est ensuite résolu par un algorithme de recherche directe sur treillis adaptatifs. Cela réduit l'effort des utilisateurs pour accomplir la tâche d'identifier de bons paramètres.

Le deuxième article décrit une extension visant à améliorer la performance du cadre OPAL. L'utilisation efficace de ressources informatiques dans ce cadre se fait par l'étude de plusieurs stratégies d'utilisation du parallélisme et par l'intermédiaire d'une fonctionnalité particulière appelée l'interruption des tâches inutiles.

Le troisième article est une description complète du cadre et de son implémentation en Python. En plus de rappeler les caractéristiques principales présentées dans des travaux antérieurs, l'intégration est présentée comme une nouvelle fonctionnalité par une démonstration de la coopération avec un outil de classification. Plus précisément, le travail illustre une coopération de OPAL et un outil de classification pour résoudre un problème d'optimisation des paramètres dont l'ensemble de problèmes tests est trop grand et une seule évaluation peut prendre une journée.

Abstract

The task of parameter tuning question has been around for a long time, spread over most domains and there have been many attempts to address it. Research on this question often lacks in generality and re-utilisability. A first reason is that these projects aim at specific systems. Moreover, some approaches do not concentrate on the fundamental questions of parameter tuning. And finally, there was not a powerful tool that is able to take over the difficulties in this domain. As a result, the number of projects continues to grow, while users are not able to apply the previous achievements to their own problem.

The present work systematically approaches parameter tuning by figuring out the fundamental issues and identifying the basic elements for a general system. This provides the base for developing a general and flexible framework called OPAL, which stands for OPtimization of ALgorithms. The milestones in developing the framework as well as the main achievements are presented through three papers corresponding to the three chapters 4, 5 and 6 of this thesis.

The first paper introduces the framework by describing the crucial basic elements through some very simple examples. To this end, the paper considers three questions in constructing an automated parameter tuning framework. By answering these questions, we propose OPAL, consisting of indispensable components of a parameter tuning framework. OPAL models the parameter tuning task as a blackbox optimization problem. This reduces the effort of users in launching a tuning session.

The second paper shows one of the opportunities to extend the framework. To take advantage of the situations where multiple processors are available, we study various ways of embedding parallelism and develop a feature called ”interruption of unnecessary tasks” in order to improve performance of the framework.

The third paper is a full description of the framework and a release of its Python implementation. In addition to the confirmations on the methodology and the main features presented in previous works, the integrability is introduced as a new feature of this release through an example of the cooperation with a classification tool. More specifically, the work illustrates a cooperation of OPAL and a classification tool to solve a parameter optimization problem of which the test problem set is too large and

an assessment can take a day.

Table of Contents

List of Tables

List of Figures

List of Appendices

Acronyms and abbreviations

AEOS	Automated Empirical Optimization of Software
ATLAS	Automatically Tuned Linear Algebra Software
BLAS	Basic Linear Algebra Subprograms
CUTEr	Constrained and Unconstrained Testing Environment, revisited
DFO	Derivative-Free Optimization
DIRECT	DIviding RECTangles
EGO	Efficient Global Optimization
FFTW	Fastest Fourier Transform in the West
GRASP	Greedy Randomized Adaptive Search Procedure
GPS	Generalized Pattern Search
ILS	Iterated Local Search
IPC	Inter-Process Communication
LAPACK	Linear Algebra PACKage
LSF	Load Sharing Facility
MPI	Message Passing Interface
MADS	Mesh Adaptive Direct Search
MDO	Multi-disciplinary Design Optimization
MFN	Minimum Frobenius Norm
MNH	Minimal Norm Hessian
MMAS	Max-Min Ant System
MILP	Mixed Integer Linear Programming
NOMAD	Nonlinear Optimization with MADS
OPAL	OPtimization of ALgorithms
ORBIT	Optimization by Radial Basis function Interpolation in Trust region
PHiPAC	Portable High Performance Ansi C
RBF	Radial Basis Function
SKO	Sequential Kriging Optimization
SOM	Self-Organizing Map
SPO	Sequential Parameter Optimization
SPOT	Sequential Parameter Optimization Toolbox

STOP	Selection Tool for Optimization Parameters
TRUNK	Trust Region method for UNKonstrained optimization
VNS	Variable Neighborhood Search

Chapter 1

INTRODUCTION

Despite progresses in computational technology, the need to improve numerical routines remains. We need to improve performance in terms of computational resource consumption and computing time; we wish to achieve better results in precision; or we simply want to extend the applicability of routines in terms of solvable problem classes. In practice, for a numerical routine, it is possible to improve on any of the three phases of its lifetime: design, implementation and operation. In the design and implementation phases, performance is determined by algorithm complexity, local convergence or evaluation complexity (Cartis *et al.*, 2012), while quality is assessed by global convergence or numerical stability (Higham, 2002). In the operation phase, quality and performance are reflected in measurable and less abstract notions such as computing time, memory consumption and accuracy (in terms of significant digits, etc). No matter what forms they take, performance and quality are usually sensitive notions influenced by a large number of factors. In order to control the quality and performance of a routine, we try to capture as many influencing factors as possible, model them as parameters and assign suitable values. Any modification in the first phase can lead to modifications in the next two phases and normally results in a new algorithm or routine. Modifications in the implementation phase that aim at better performance or quality are referred to as *source code adaptation.* Choosing a suitable setting for parameters at run time is called *parameter tuning.* Parameter tuning can be done manually using trial and error or automatically by a finite procedure. It can also be done analytically by exact computations or empirically based on a finite set of input data called *test problems.* Among these possibilities, our work concentrates on developing a framework for *empirical automated parameter tuning.*

An empirical method for automated parameter tuning is an iterative method where each iteration executes at least three steps: propose *parameter values*, evaluate the algorithm with these values and finally make a decision on stopping or continuing to the next iteration. The initial suggestion for parameter values is normally provided by users or simply the default values. In subsequent iterations, the tuning method suggests other settings using some strategies and information obtained in previous iterations. The strategy is different for each method and becomes one of the characteristics that distinguish tuning methods. The evaluation is performed by launching the *target routine* over a set of preselected test problems. Issues for a quality assessment strategy include test problem selection, analyzing the results and quantifying the tuning goals. In the last step, the main task is to compare the obtained quality assessment of the current parameter setting with the tuning goal in order to decide whether to go on to the next iteration or not. There is no standard response to this question and its answer usually depends on users goals. Thus, in combining possibilities for each step, there are many ways to build an empirical automated tuning procedure. Parameter tuning is still a research question, which means that there is currently no unique satisfactory method for all users. The difficulty comes from many sources. First of all, it is not easy to identify parameters that impact the tuning objective. The hidden relation between parameters and performance adds uncertainty to any automated tuning strategy. Secondly, the numerous parameters and their distribution create an intricate search space that prevents manual tuning. For example, the routine IPOPT (Wächter and Biegler, 2006), an interior point solver, has nearly 50 parameters, most of which can be real number. Thus, enumeration of all possibilities is impossible and the trial and error strategy is usually unsatisfactory. Finally, an effective tuning strategy usually requires expert knowledge of the algorithm and a thorough understanding of the effects of the parameters. As a consequence, it is not easy to create a general framework that works well on all algorithms.

Recent achievements in optimization, particularly in *blackbox optimization*, provide another approach to the tuning problem. The tuning question can easily be modeled as an optimization problem, in which variables are the tuning parameters and performance or quality are expressed as objective functions and constraints. As a result, several methods of optimization can be applied here. However, classical optimization methods depend strongly on the structure of the problem. With a parameter optimization problem, we do not have much information about the structure except the function value at some given *parameter points* (each parameter point corresponds to a set of parameter values). The situation becomes even worse when the function value is estimated using uncertain empirical outputs such as computing time. An optimization method that depends less on problem structure and handles the lack of information on problem structure such as *direct-search methods* holds promising prospects for solving the *parameter optimization problem*. We chose MADS (Audet and Dennis, Jr., 2006)(Mesh Adaptive Direct Search) as our fundamental algorithm to solve the parameter optimization problems.

By studying the parameter tuning problem both from the perspective of an iterative method and as an optimization problem, we design a *framework of algorithmic parameter optimization* called OPAL (OPtimization of ALgorithms). OPAL is general and flexible enough to apply to the question of improving the performance of any routine. We concentrate on algorithmic parameters, and thus sometimes we refer to the problem of algorithmic parameter optimization as *algorithm optimization*. Like many empirical methods, each iteration of our method goes through three steps involving the NOMAD (Le Digabel, 2011), an implementation of the MADS method: *(i)* NOMAD proposes a parameter settings, *(ii)* the *target algorithm* with these settings is evaluated on a set of test problems, *(iii)* the evaluation result provides information to NOMAD to launch the next iteration. As a result, we require minimum effort from users to define a parameter optimization problem with the following main elements: parameter description, a set of test problems, evaluation measurements, and tuning goals expressed as an objective function and constraints. After defining a parameter optimization problem, all the remaining work is performed by the NOMAD solver.

In simple situations, users do not need to know and provide much information to launch a parameter tuning session. However, to make the framework more flexible and sophisticated in situations where the users know more about their algorithms, they can provide more information to accelerate the search or guide it toward promising regions. For example, users can refine the parameter space by defining more *parameter constraints* to prevent unnecessary target algorithm runnings. Users can also define *surrogate models* for their problems to help NOMAD to propose promising parameter values. In the meantime, from the computing point of view, the tuning process is composed of fairly independent steps, for example, the observation of the algorithm over a set of test problems; thus, there are possibilities to exploit parallelism in the framework.

In this introduction, we have presented a brief overview of the typical research involving algorithmic parameter optimization. In chapter two, we discuss relevant literature with a focus on the three main questions of an empirical method and issues relating to parameter optimization problem. The third chapter gives an outline of the remainder of the thesis, with chapter 4, 5 and 6 reserved for three papers on this problem. Finally, the last chapter shows some conclusions and perspectives.

Chapter 2

EMPIRICAL OPTIMIZATION OF ALGORITHMS: STATE OF THE ART

As presented in the previous chapter, we can improve the performance of a numerical routine in the development stage (also known as source code adaptation) and in the operaton stage (also known as parameter tuning). Since source code adaptation can also be considered as parameter tuning of a particular source code generator, hereafter, we use the term parameter tuning to indicate both source code adaptation and parameter tuning. In this section, we first review some typical projects involving parameter tuning to show that this is a domain of active research. Next, by identifying common elements of these approaches, we examine how these projects answer three basic questions of an automated parameter tuning procedure: *(i)* what are the parameters, *(ii)* how to assess the effect of parameter settings and *(iii)* how to explore the parameter setting space. In the final section we describe related issues where parameter tuning is examined as a blackbox optimization problem, especially in the context of direct search methods.

2.1 Automatic parameter tuning is an active research area

Better performance can be achieved at the development or operation stage. In the development stage, the binary code generation is optimized in terms of the performance that depends on the compiler and the platform where the software is built. There are two approaches for optimizing code generation. The first approach creating an important branch of research, called compiler, is based on analytical studies that are therefore outside of the scope of this thesis. The second one is concerned with adapting code to the running platform and is based mainly on the combination of empirical studies and search strategies. The search is performed on the set of code transformations allowed at the programming language level and is based on performance evaluation through empirical output.

PHiPAC[1](Portable High Performance Ansi C) (Bilmes *et al.*, 1998) is one of the earliest automatic tuning projects that aims to create high-performance linear algebra libraries in ANSI C. It is referred to as a methodology that contains a set of guidelines for producing high-performance ANSI code. It also includes a parameterized code generator based on the guidelines and scripts that automatically tune code for a particular system by varying the generators' parameters based on empirical results. PHiPAC is used to generate a matrix-matrix multiplier that can get around 90% of peak (on systems such as Sparcstation-20/61, IBM RS/6000-590, HP 712/80i) and on IBM, HP, SGI R4k, Sun Ultra-170, it can even produced a matrix multiplier that faster than the ones of vendor-optimized BLAS[2] (Basic Linear Algebra Subprograms) (Lawson *et al.*, 1979).

ATLAS[3](Automatically Tuned Linear Algebra Software) (Whaley *et al.*, 2001), a more recent project on numerical linear algebra routines, is an implementation of the Automated Empirical Optimization of Software paradigm, abbreviated as AEOS. The initial goal of ATLAS was to provide an efficient implementation of the BLAS library. However, ATLAS was recently extended to include higher level routines from the LAPACK (Linear Algebra PACKage) library. ATLAS supports automated tuning in all three levels of BLAS. For level 1 BLAS, which contains routines doing vector-vector operations, ATLAS provides a set of pre-defined codes contributed from many sources (with varying floating point unit usage and loop unrolling) from which a best code is selected based on evaluations of this set. The set of pre-defined code is enriched over time. In fact, tuning by ATLAS at this level does not achieve significant improvements; speedup typically ranges from 0% to 15%. The observed efficiency in level 2 BLAS is much better; the speedup can reach up to 300% in some cases. The reason is that level 2 includes vector-matrix routines that are much more complex than the level 1 routines in terms of both data transfer and loop structure; consequent to these facts, there are more possibilities to be optimized. ATLAS also initiates the

1. `http://www.icsi.berkeley.edu/~bilmes/phipac/`
2. `http://www.netlib.org/blas/`
3. `http://math-atlas.sourceforge.net/`

idea of optimizing BLAS by replacing the global search engine with a model-driven optimization engine based on a robust framework of micro-benchmarking called X-Ray (Yotov, 2006).

The work of Yotov (2006) is not about an empirical method but the contribution to the field is remarkable. It initially starts out to study the differences in the performance of BLAS tuned by ATLAS and that supported by compiler restructuring. They firstly study whether there is a compiler restructuring that produces the same code generated by ATLAS. Furthermore, by recognizing the fact that ATLAS uses a fairly simple search procedure to get optimal parameters of the source code generator, the author proposes a model to get these values instead of an iterative method. The model computes optimal values from a set of hardware specifications (such as CPU frequency, instruction latency, instruction throughput, etc) that are gathered by a micro-benchmark system. The experimental results state that a micro-benchmark system of high accuracy with a good model can give as good parameters as those found by ATLAS. This implies that the optimality found by the ATLAS algorithm is proved at certain levels. In order to improve the result, a local search heuristic is applied to a neighborhood of the parameter values computed by the model.

Sparsity[4] (Im and Yelick, 1998) and OSKI[5] (Optimized Sparse Kernel Interface, Vuduc *et al.* 2005) focus on a narrower direction in tuning linear algebra libraries - sparse matrix manipulation. Sparsity addresses the issue of poor performance of general sparse matrix-vector multipliers due to spatial locality. Recognizing that performance is also highly dependent on methods of sparse matrix representation and on hardware platforms, Sparsity allows users to automatically build sparse matrix kernels that are tuned to their target matrices and machines. OSKI, inspired by Sparsity and PHiPAC, is a collection of low-level C primitives for use in a solver or in an application. In OSKI, "tuning" refers to the process of selecting the data structure and code transformations to get the fastest implementation of a kernel in the context of matrix and target machine. The selection is essentially the output of a decision making system whose input are benchmark data of a code transformation, matrix characteristics, workload from program monitoring, history and heuristic models.

In addition to linear algebra libraries, signal processing is a promising ground for

4. `http://www.cs.berkeley.edu/~yelick/sparsity/`
5. `http://bebop.cs.berkeley.edu/oski/`

empirical source code adaptation. Among many projects, SPIRAL[6] (Püschel *et al.*, 2005, 2011) is the best example despite its restricted consideration on linear signal transforms. SPIRAL optimizes code by exploiting not only hardware factors but also mathematical factors of transforms (Milder, 2010); it optimizes at both the algorithmic and the implementation levels. More specifically, a transform can be represented as formulas based on different mathematical factors. Hence, there are usually many choices of representing a single transform. These formulas are next implemented by considering appropriate target programming languages, compiler options, as well as target hardware characteristics. To search for the best combination of representation and implementation, SPIRAL takes advantage of both search and learning techniques. For example, the current version of SPIRAL deploys two search methods: dynamic programming and evolutionary search. The learning is accomplished by reformulating the problem of parameter tuning in the form of a Markov decision process and reinforcement learning. SPIRAL shows very interesting experimental results (Püschel *et al.*, 2005), including performance spread with respect to runtime within the formula space for a given transform, comparison against the best available libraries, benchmark of generated code for DCT (Discrete Cosine Transformation) and WHT (Walsh-Hadamard Transform) transforms. In summary, the idea behind SPIRAL is to choose the best implementation when we have multiple implementations of multiple formulas of a transform; this is similar to the PetaBricks[7] (Ansel *et al.*, 2011) project that targets scientific softwares.

FFTW[8] (Fastest Fourier Transform in the West) (Frigo and Johnson, 2005) is a C subroutine library for computing the discrete Fourier transform (DFT) in one or more dimensions, of a real or complex input of arbitrary size. The library is able to adapt to a new situation not only in terms of input data size but also performance. The optimization in FFTW is interpreted in the sense that FFTW does not use a fixed algorithm for computing the transform, but instead adapts the DFT algorithm by choosing different plans that work well on underlying hardware in order to increase performance. The performance adaptation can be performed automatically by a FFTW component called a *planner* or manually by advanced users who can customize FFTW. Hence, FFTW is a parameter tuning tool rather than source code

6. `http://spiral.net/index.html`
7. `http://projects.csail.mit.edu/petabricks/`
8. `http://www.fftw.org/`

adaptation like SPIRAL.

PetaBricks (Ansel *et al.*, 2011) and Orio[9] (Hartono *et al.*, 2009) both target the source code adaptation problem for a program or a segment code in any domain. The generalization is obtained by proposing particular programming directives or even a programming language to specify the possibilities of tuning in the target segment code. PetaBricks allows to tune a target routine in two levels by having multiple implementations of multiple algorithms for a target routine. For example, in order to sort an integer array, we can from several sorting algorithms; and corresponding to the selected algorithm, several implementations are considered. Orio only focuses on the implementation level by proposing an annotation language that is actually the programming directives. A high-level segment code, enclosed by these directives, will be implemented in different ways corresponding to variations of the architecture specifications such as the blocking size, cache size, etc. A good implementation is selected based on the performance of running the generated code.

The work of Balaprakash *et al.* (2011b) can be considered as a source code adaptation project in the sense that it formulates with the help of the Orio annotations the tuning questions of a set of basic kernels used broadly and intensively in scientific applications. These problems are solved for each hardware architecture to obtain the most suitable implementation for each kernel. The contributions to the automated tuning field are the formulas of kernel optimization problems plus a particular algorithm to solve effectively these problems.

In practice, the question of parameter tuning has been studied by many researchers. We can list here some examples: optimization of control parameters for genetic algorithms (Grefenstette, 1986), automatic tuning of inlining heuristics (Cavazos and O'Boyle, 2005), tuning performance of the MMAS (Max-Min Ant System) heuristic (Ridge and Kudenko, 2007), automatic tuning of a CPLEX solver for MILP (Mixed Integer Linear Programming) (Baz *et al.*, 2009), automatic tuning of GRASP (Greedy Randomized Adaptive Search Procedure) with path re-linking (Festa *et al.*, 2010), using entropy for parameter analysis of evolutionary algorithms (Smit and Eiben, 2010), modern continuous optimization algorithms for tuning real and integer algorithm parameters (Yuan *et al.*, 2010). However, all these projects target specific algorithms, maximally take advantage of particular expert knowledge to get the best possible results and avoid the complexity of a general automated tuning framework.

9. `http://trac.mcs.anl.gov/projects/performance/wiki/Orio`

The number of tuning projects continues to increase, indicating that the concern still exists. Thereby, it begs the question of a general framework where the basic questions of automated tuning are imposed and answered more clearly. The recent projects presented in the following paragraphs pay more attention to these questions.

STOP[10](Selection Tool for Optimization Parameters, Baz *et al.* 2007) is a tuning tool based on software testing and machine learning. More specifically, it uses an intelligent sampling of parameter points in the search space assuming that each parameter has a small discrete set of values. This assumption is acceptable for the intended target problem of tuning MILP branch-and-cut algorithms. At the time of release, STOP set the parameter values in order to minimize the total solving time over a set of test problems. No statistical technique is used because the authors assume that good settings on the test problems will be good for other, similar problems.

ParamILS[11] is a versatile tool for parameter optimization and can be applied to an arbitrary algorithm regardless of the tuning scenario and objective and with no limitation on the number of parameters. It is derived from efforts of designing effective algorithms for hard problems (Hutter *et al.*, 2007). Essentially, it is based on the ILS (Iterated Local Search) (Lourenço *et al.*, 2010) meta-heuristic. ParamILS is supported by a verification technique that helps to avoid over-confidence and over-tuning. However, due to the characteristics of the employed local search algorithm, it works only with discrete parameters; continuous parameters need to first be discretized. Moreover, local search performance depends strongly on neighborhood definition that is drawn from knowledge on the parameters of the target algorithm. But ParamILS has not a way to customize the neighborhood definition for a variable.

The above projects are based on heuristics and focus only on specific target algorithms or particular parameter types. Recently, the question of parameter tuning was approached more systematically where the connection between the parameter tuning and stochastic optimization is established. These projects are based on a framework called *Sequential Parameter Optimization* (SPO) (Bartz-Beielstein *et al.*, 2010b; Preuss and Bartz-Beielstein, 2007), which is a combination of classical experiment design and stochastic blackbox optimization. The main idea of this approach is to use a stochastic model called a *response surface model* to predict relations between three principal elements of the automated tuning question: parameters, performance

10. `http://www.rosemaryroad.org/brady/software/`
11. `http://www.cs.ubc.ca/labs/beta/Projects/ParamILS/`

and test problems. Response surface models can be useful in order to study a parameter tuning problem in some different aspects: interpolate empirical performance from evaluated parameter settings, extrapolate to previously-unseen regions of the parameter space, and qualify the impacts of parameter settings as well as test problems. SPOT[12] (Bartz-Beielstein, 2010) is a R package implementing the general idea of SPO that are applied in the context of parameter tuning. Extensions such as SPO$^+$, can be found in the works of Hutter *et al.* (2010a, 2009).

SPO and our framework OPAL have one thing in common: they both model parameter tuning as a blackbox optimization problem. However, the blackbox model in SPO is assumed to be a stochastic blackbox model while we do not impose any assumption over blackbox model.

2.2 The basic questions of an automated tuning method

We can identify three common basic elements of all the projects presented in the above section: parameterization, performance evaluation and parameter search strategy. The first is the question of identifying variable factors that influence performance and describing these factors in terms of parameters. For algorithmic parameter optimization, the variables are defined explicitly; thus identifying parameters is usually not difficult. In contrast, identifying variables for source code adaptation is often difficult since source code performance is influenced by many hidden factors. In addition to the parameter identification, it is necessary to define the parameter space by specifying the domain for each parameter, simple relations between parameters.

The second question is about assessing the quality of parameter settings. In other words, it is the problem of comparing the empirical performance of the target algorithm with various parameter settings. Obviously, this question depends on the tuning objective. In general, comparison criteria are established based on observations of running the algorithm over the set of test problems, such as computing time, consumed memory, and/or algorithm output. In the simplest case, we can choose an observation as a criterion of comparison, but criteria can be expressed in more sophisticated ways such as a function and even as an output of a computation process

12. `http://cran.r-project.org/web/packages/SPOT/index.html`

or a simulation process whose inputs are the observations from running the target algorithm. The main concern for an assessment method is the inaccuracy that comes from two sources: uncertainty of some observations and empirical noise. However, it is not possible to totally eliminate the inaccuracy; we often must balance the cost of inaccuracy reduction and the sophistication of a search method.

The last question concerns the search strategy in parameter space. The complexity of a strategy depends on the two previous questions. The larger the parameter space is, the more sophisticated the strategy we need. The less accurate the performance evaluation is, the more flexible the search strategy we need to come up with. The more specific the application is, the more particular the knowledge is to be integrated into the strategy. The fact of the matter is we have no guide to build a search strategy.

2.2.1 Parameterizing the target algorithm

The process of algorithm parameterization is composed of two tasks: identifying parameters and describing them. The former will imply which parameters are involved. Parameters of an algorithm (or a routine) are generally all the factors that can be changed by an user of the algorithm that have an impact on the implementation's performance. For the code adaptation, these factors vary from one language to another, and from one architecture to another. However, the spectrum of code generators' parameters is usually not very broad. As a consequence, parameter spaces can be described well once the parameters are identified. In contrast, identifying parameters in algorithmic parameter tuning is less difficult but describing the parameter space or selecting a subset of parameters which are significant is a big issue.

ATLAS optimizes the BLAS library using both techniques, parameter tuning and source code adaptation. In the former, ATLAS focuses on parameters of the library kernels such as the *blocking factor* and the *cache level*. These parameters control the cache utilization and as a consequence, directly influence speedup. The source code adaptation is performed by choosing the best code from a pre-fixed set of codes provided by many contributors or by automatically generating codes from templates. In the first approach, we have a single parameter whose possible values correspond to the set of contributions of source code. In the second approach, the set of parameters of a template includes *L1 data cache tile size*, the *L1 data cache tile size for non-copying version*, *register tile size*, *unroll factor*, *latency for computation scheduling*, *choices of combined or separate multiply and add instructions* and *load scheduling*. Corresponding to each combination of parameter settings, a source code of the routine is generated from the template. For other projects concentrating on linear algebra routines (except for Sparsity and OSKI which exploit particularities of sparse matrices), the set of involved parameters is typically selected from the set proposed by ATLAS.

Most project have parameters expressing the selections called the selecting parameters. The domain of this parameter type is sometimes simple as an unordered set of values. ATLAS uses a categorical parameter to indicate a the set of predefined code. Sometime, the domain is a set of a particular order such as the case of the plan selecting parameter in FFTW or the component selecting in ParamILS. The domain can be so complicated to be represented by a set, for example, each selection of the algorithm selecting parameter in PetaBrick is represented as a tree.

For some tuning projects that includes a selecting components, the parameter space can be changed corresponding to each selection. For example, SPIRAL optimizes a routine by finding the most suitable implementation of a transform. In order to get a possible implementation for a transform, SPIRAL translates the transform to formulas, and codes these formulas in a target programming language. Parameters of the formula generation stage include *the atomic size of formula*, *formula characteristics* such as parallelizable construct or vectorizable construct. Hence, in the second stage, in addition to the parameters of a code generator, the particular parameters of the selected formulas will be considered.

2.2.2 Empirical quality evaluation methods

Empirical quality evaluation involves the process of comparing performance of two instances of parameters through a set of experiments, and it is sometimes referred to as *experimental comparison*. More concretely, within the context of an empirical automated tuning project, the assessment is performed through a sequence of tasks, running the target algorithm over a set of test problems, collecting all concerned observations (referred to as *atomic measures* in OPAL terminology) on each test problem and building comparison criteria (referred to as *composite measures* in OPAL terminology) from the observations. Examples of observations are wall-clock running time and outputs of the algorithm (such as the solution, number of significant digits of solution, norm of the gradient, etc). There are two main issues related to the empirical comparison. The first is that atomic measures are often noisy. A typical example is the wall-clock time: running the target algorithm on a machine can result in different running times in different launches due to operating system, influence. The second is that, because observations are obtained from running algorithm over a set of test problems, they are concrete instances of a performance measure. A better performance based on these concrete instances does not guarantee a better performance in general or on other instances. The different techniques of quality evaluation deal with the two issues in different ways.

The simplest way to deal with the noise of some elementary measures is to choose an alternative one that is less noisy. For example, to benchmark an optimization solver, the number of function evaluations may be more suitable than the computing time. CUTEr (Constrained and Unconstrained Testing Environment, revisited) (Gould *et al.*, 2003a) is a project that provides systematic measures for assessing an optimization solver. To provide a general description of CUTEr[13],

> it is a versatile testing environment for optimization and linear algebra solvers. The package contains a collection of test problems, along with Fortran 77, Fortran 90/95 and Matlab tools intended to help developers design, compare and improve new and existing solvers.

CUTEr provides a mechanism to extract the number of metrics (such as objective functions and constraint functions, the final value of objective function, of gradient norm, etc). Such a performance assessment of an optimization solver may better reflect the practical performance of a solver, independent of the platform.

Other common ways to address the noise are drawn from statistics. For example, the noise of an observation can be reduced by repeatedly launching the target algorithm over the set of test problems and taking the sampled mean value. This treatment is expensive in terms of computational resources. A more efficient way is to think of each test problem as a sample drawn from a problem population; and that an analysis over a good enough sample set can achieve accurate descriptions of dependences between performance and the parameters. Answers to the question of how to get an accurate description are summarized in Kleijnen (2010) and result in a methodology called *design of experiments*. Statistical design of experiments is the process of first planning an experiment so that appropriate data is collected and analyzed using statistical methods so that useful conclusions can be drawn. For example,

13. `http://www.cuter.rl.ac.uk/`

CALIBRA (Adenso-Diaz and Laguna, 2006) employs fractional factorial designs to draw conclusions based on a subset of experiments from a set of p^k possible combinations of parameters where k is the number of parameters and p is number of critical values of each parameter. The strategy for selecting the subset from the total set is that of Taguchi *et al.* (2004), which uses orthogonal arrays to lay out the experiments. CALIBRA uses the $L_9(3^4)$ orthogonal array that can handle up to 4 parameters with 3 critical values by just 9 experiments. More examples can be found in Bartz-Beielstein *et al.* (2010a), a book on experiment design techniques specialized to parameter tuning with a particular focus on sequential techniques that instruct how to select new promising points based on information obtained from previous experiments.

Another research area that can be involved in empirical evaluation is the concept of performance and data profiles of Dolan and Moré (2002) and Moré and Wild (2009) for comparing solvers through a set of test problems. In the context of parameter tuning, the target algorithm associated with each parameter setting is regarded as a solver. These profiles provide a visual qualitative information, and hence a method to get a quantitative output from performance profiles can be a good empirical assessment method, for example, the area of the region below a profile curve.

In addition to techniques of treating measures and observation, selecting a good test problem set can significantly improve the extrapolatory quality of the evaluations. Although the set of test problems is pre-selected, having awareness of the influences of the test problems is necessary for a good empirical result analysis. More detailed arguments can be found in Auger *et al.* (2009) and Hutter *et al.* (2010b).

2.2.3 Search methods for exploring parameter space

In any method of empirical quality evaluation, there is an assumed model that expresses the relation between parameter value combinations and observations or quality. The model can be formulated implicitly as a blackbox (Audet and Orban, 2006) or explicitly as a stochastic model (Bartz-Beielstein *et al.*, 2010b) or as a deterministic model (Yotov, 2006). Search strategies must take into account the parameter space representation as well as the method of quality assessment.

Most parameter tuning projects choose heuristic approaches. Targeting a specific problem, taking advantage of particular descriptions about the parameters, a heuristic may get good performance. However, the flexibility is often reduced as efficiency is gained. There are some projects that develop their heuristic from a meta-heuristic; if the development does not involve too much particularities, it can be extended or be modified to deploy in other projects. For example, ParamILS (Hutter *et al.*, 2007) is developed from the ILS (Lourenço *et al.*, 2010) meta-heuristic, which proposes iterative jumps to other regions after reaching a local optimum of the current region. ILS leaves users free to choose a method to find local minima as well as how to jump to another region. ParamILS approaches parameter tuning with the two most basic ingredients: search a better solution by a simple procedure and as soon as finding it, jump to another region by a random move.

Besides the dependence of performance on particularities, most heuristics work with finite discrete sets, which means that a tuning algorithm based on these heuristics can only work on categorical parameters or integer parameters with bounds. For real parameters, these methods require a discretization without loosing information phase that may be costly. Moreover, heuristics can only handle simple parameter spaces that are composed of a small number of variables and each variable may have few values.

As mentioned previously, an experiment design method includes not only techniques for drawing conclusions from experimental results, but also techniques to set up or control experiments. In the context of parameter tuning, the latter techniques will figure out potential parameter settings where the tests can manifest all the characteristics of the target algorithm, hence most simply, we can choose one of suggested settings as the solution. There is a class of techniques called the *sequences of experiments* that suggest the next settings to examine based on the result of experiments performed. The SPO search strategy is built based on this theory.

In reality, three basic questions are tightly corded. However, previous projects have focused on each question separately or not paid enough attention to the relations between them. This is one possible reason why the proposed tuning techniques in the literature have remained non-systematic approaches. In the next section, we study the parameter tuning problem from the optimization point of view where the three basic questions are examined in a unique model.

2.3 Automatic parameter tuning as an optimization problem

In the optimization community, the automated tuning parameter question may be formulated as an optimization problem where variables are the involved parameters and the tuning objective and the context are expressed by objective function and constraints, respectively. By formulating as a blackbox problem, minimal information on problem structure is required. This means that users can easily and quickly define a parameter optimization problem and rely on the chosen solver. Nonetheless, it does not prevent the possibility of providing more specific information to accelerate the search process.

2.3.1 Formulation of a parameter optimization problem

We formulate the parameter tuning problem as a blackbox optimization problem. The simplest statement of a parameter optimization problem is

$$\begin{array}{ll} \underset{p}{\text{minimize}} & \psi(p) \\ \text{subject to} & p \in \mathbf{P} \\ & \varphi(p) \in \mathbf{M} \end{array} \tag{2.1}$$

where p denotes a parameter vector; $\mathbf{P}$ represents the valid parameters region; the objective function $\psi(p)$ expresses the tuning objective; and general constraints $\varphi(p) \in \mathbf{M}$ encode the restrictions of the tuning process. Note that the elements of a vector parameter p are not necessary of the same type. An element can be one of three following types: a real number (type $\mathbb{R}$), an integer number (type $\mathbb{Z}$) or a categorical value (type $\mathbb{C}$) (Audet and Dennis, Jr., 2000). Hence, if a target parameter set has n parameters of type $\mathbb{R}$, m parameters of type $\mathbb{Z}$ and l categorical parameters, a vector p is an element of $\mathbb{R}^n \times \mathbb{Z}^m \times \mathbb{C}^l$. The valid parameter region, $\mathbf{P}$, is a subset of $\mathbb{R}^n \times \mathbb{Z}^m \times \mathbb{C}^l$ that models the parameter region reacts such as a positive number or a real number in the interval $(0, 1)$. The target algorithm assessment result is expressed in the objective function $\psi(p)$ and the general constraints $\varphi(p) \in \mathbf{M}$. The reason why we split the constraints in two categories $p \in \mathbf{P}$ and $\varphi(p) \in \mathbf{M}$ is that the former is used to validate a parameter setting and decide if we need to launch the target algorithm over the test problems while the latter is only verified if all runnings are terminated.

Another attempt to bring automated tuning into the optimization community is proposed by Balaprakash *et al.* (2011b) where the parameter optimization problem is stated simply as

$$\min_{x}\{f(x) : x = (x_{\mathcal{B}}, x_{\mathcal{I}}, x_{\mathcal{C}}) \in \mathcal{D} \subset \mathbb{R}^n\} \tag{2.2}$$

where $x_{\mathcal{B}}$, $x_{\mathcal{I}}$ and $x_{\mathcal{C}}$ correspond to the binary, integer and continuous parameters, and $f(.)$ is some performance measure. The efficiency of solving 2.2 depends on the domain $\mathcal{D}$ whose the construction requires expertise knowledge on the target algorithm. In other words, this formulation does not give much information to a solving method until the domain $\mathcal{D}$ is well established.

A blackbox optimization problem can be solved by using a direct-search solver or a heuristic. The heuristic efficiency depends strongly on the expert knowledge. In the general case of a blackbox optimization problem, we assume that there is no information except for function values at certain points; this implies the inefficiency of heuristic methods. Thus, we reserve the next subsection for discussing only direct-search methods.

2.3.2 Direct-search methods for solving blackbox optimization problem

Direct-search solvers comprise all methods that use only functions values (objective and constraints) to search for a local optimum. These methods are distinguished from classical methods that require first order-information (derivative, gradient), such as gradient-based methods or even second order-information (Hessian matrix), such as Newton methods. Direct-search methods form only a subset of derivative-free methods that includes methods that approximate derivatives or use derivative-like concepts such as sub-gradients (Conn *et al.*, 2009b). Focusing only on methods that work well for blackbox optimization, we review results of direct-search methods. There are two main ideas for direct-search methods. The first one is to use a model to guide iterates approaching a local optimum, the methods are classified as *model-based* methods. The model can be a local approximation of the objective function and its precision is improved from iteration to iteration. Another option is stochastic models that capture the global characteristics of the functions. The second idea is based on sampling variable domains; at each iteration, the variable domain is sampled at certain points depending on a sampling strategy to evaluate the objective function. This branch is in turn divided into two sub-categories, such divide-and-conquer and pattern-based are sometimes referred to as directional search. Figure 2.1 illustrates some state-of-art methods that are considered as fundamental ideas; variants and derived methods now constitute a rich set of direct-search methods.

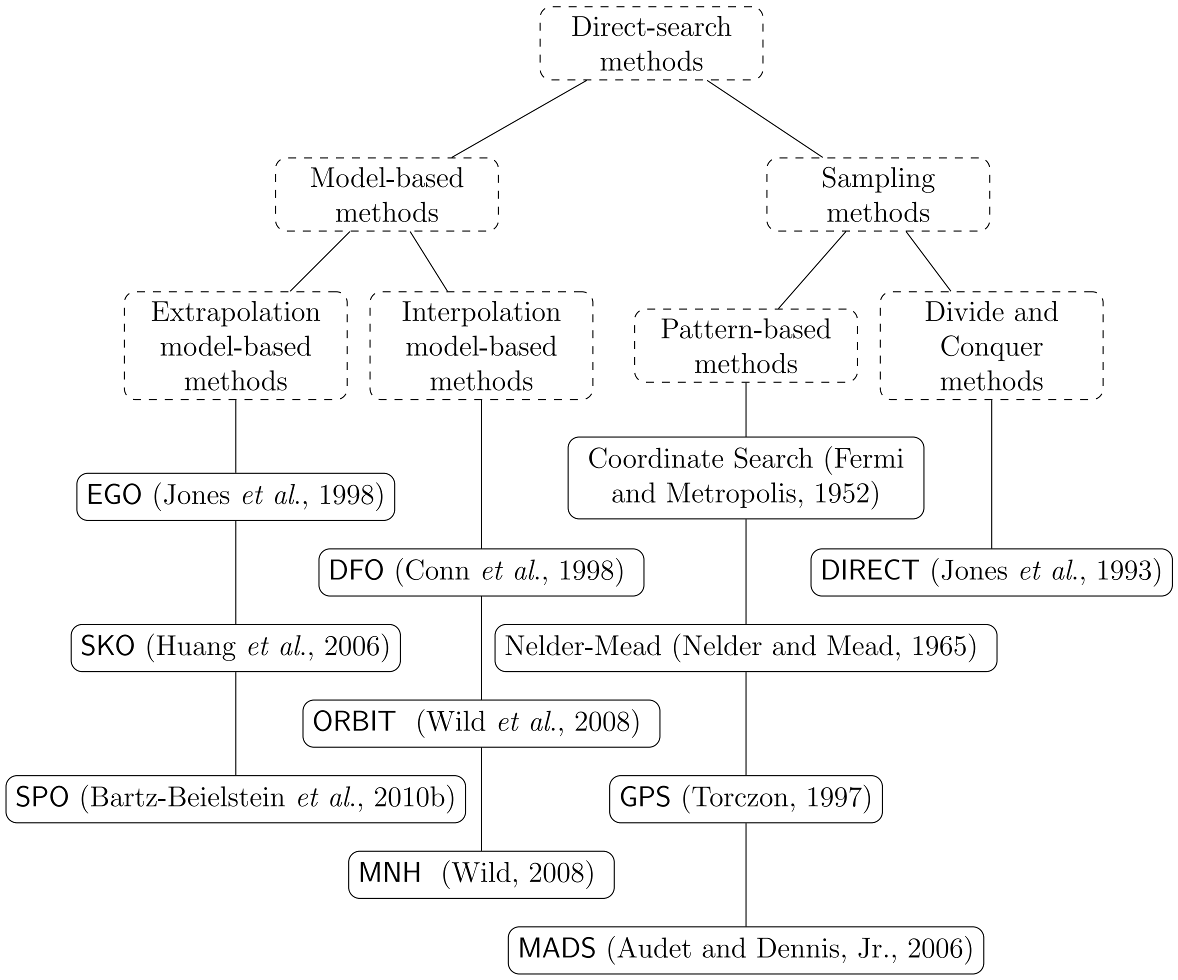

Direct-search methods
Model-based methods
Sampling methods
Extrapolation model-based methods
Interpolation model-based methods
Pattern-based methods
Divide and Conquer methods
EGO (Jones *et al.*, 1998)
SKO (Huang *et al.*, 2006)
SPO (Bartz-Beielstein *et al.*, 2010b)
DFO (Conn *et al.*, 1998)
ORBIT (Wild *et al.*, 2008)
MNH (Wild, 2008)
Coordinate Search (Fermi and Metropolis, 1952)
Nelder-Mead (Nelder and Mead, 1965)
GPS (Torczon, 1997)
MADS (Audet and Dennis, Jr., 2006)
DIRECT (Jones *et al.*, 1993)

Figure 2.1 Classification of direct-search methods

The DFO (Derivative-Free Optimization) (Conn *et al.*, 1998) method locally models the objective function by quadratic interpolation and uses this local model to find the next iterate. In each iteration, the local optimum of the model within a trust region is chosen as the next iterate if the reduction of the model and the reduction of the objective/merit function at this point are compatible. Otherwise, the method remains at the incumbent point and tries to find a local minimum of the model in a smaller region. In both cases, the model is updated in order to improve its quality by adding to the interpolation set a new point satisfying a *well-poisedness* condition (for example Λ-poisedness, where Λ is a constant related to geometry of the interpolation set). In practice, DFO uses a quadratic model that is not expensive to construct and optimize. The idea behind this procedure is that interpolation models can accurately represent the objective function that can be a smooth (twice differentiable) function over a small region. However, the interpolation can suffer from issues on a practical engineering blackbox or stochastic blackbox problems. Furthermore, interpolation for a full quadratic model in n-dimension space requires an interpolation set of $\frac{(n+1)(n+2)}{2}$ points that mentions a non-realistic condition for a computationally expensive blackbox and hence, an addition mechanism to build models using fewer points is necessary. Such mechanisms can be MFN (Minimum Frobenius Norm) Conn *et al.* (2009b) (used in DFO package) or MNH (Minimal Norm Hessian) (Wild, 2008) that build *underdetermined quadratic models.* EGO (Efficient Global Optimization) (Jones *et al.*, 1998) handles problems with noisy blackboxes using a stochastic model and a Bayesian-based update mechanism. In order to deal with the issue of computational expense, ORBIT (Optimization by Radial basis function Interpolation in Trust region) (Wild *et al.*, 2008) uses a radial basis function (RBF) interpolation model that is constructed from a flexible set of data points, which does not have too many requirements.

ORBIT (Wild *et al.*, 2008; Wild and Shoemaker, 2011) approaches the blackbox optimization problem in a similar way to DFO with an interpolation model and the trust-region framework. However, using RBF instead of quadratic models (polynomial models in general) gives it some advantages: it does not need a large initial set of base points; and has a more flexible update mechanism because the set of interpolation points can freely vary.

EGO uses a kriging model to capture a function “shape” in a region of interest. It determines the next iterate by the *expected improvement* procedure. The new iterate is added to the set of intrapolation points in order to build a new model in the next iteration. SKO (Sequential Kriging Optimization) modifies the selection of the next iterate; the new principle called *augmented expected improvement* is able to adapt more smoothly to stochastic blackbox problems.

Model-based methods differ by their model, their strategy to select new iterates and their updating mechanism. The divergence of these elements shown in the previous examples is not merely a small modification to improve, to adapt to each specific use-case, they originate from assumptions about the blackboxes with which they work. EGO and SKO use kriging models because they focus on the shape of functions in a larger region instead of focusing on local function behavior like DFO or ORBIT do. In other words, DFO targets deterministic blackboxes that can produce a smooth output, meanwhile EGO and SKO are for noisier models. This means that information about the blackbox is necessary to select a model.

In contrast, sampling-based methods do not usually need assumptions on the blackbox because they concentrate on potential regions instead of function behavior. One of the oldest and most popular methods is the Nelder-Mead method (Nelder and Mead, 1965), which is based on a geometry concept called the *simplex*. A simplex in n-dimensional space is the convex hull of a set of $n + 1$ vertices in this space. The Nelder-Mead method transforms the simplex by replacing the worst vertex, in the sense of objective function value, by a new, better one. Although the idea is very simple, convergence properties are only studied for strictly convex functions in dimensions 1 and 2 (Lagarias *et al.*, 1998), but the method is intuitive and efficient in practice. The idea of Coordinate Search is introduced in the work of Fermi and Metropolis (1952). In Coordinate Search, a set of sampling points, also called a *pattern*, is defined along the coordinate axes, and GPS (Generalized Pattern Search) (Torczon, 1997), where the patterns are fixed in some predefined directions. The patterns tied to fixed directions prevent these two methods from exploring thoroughly the space; certain regions can never be reached. Examples illustrating this issue can be found in Abramson (2002), Audet (2004) or Audet and Dennis, Jr. (2006). This implies that we still need more information to assure that the pattern-based algorithms work, because there are only finitely many prefixed search directions.

Considered as the most recent evolution in the pattern-based branch, MADS (Mesh Adaptive Direct Search) Audet and Dennis, Jr. (2006) overcomes most obstacles encountered by its predecessors. It not only removes many assumptions related to the blackbox, but also relies on a solid hierarchical convergence analysis that guarantees convergence to a first-order point. We describe in more detail this algorithm in the next subsection.

DIRECT (DIviding RECTangles) (Jones *et al.*, 1993) targets bound-constrained, non-smooth, Lipschitz-continuous problems. Its convergence is analyzed in Finkel and Kelley (2004). The sampling procedure of DIRECT is simple: at each iteration, it samples the function at the centers of hyperrectangles to determine the hyperrectangles with the most potential. These hyperrectangles are divided into smaller hyperrectangles in next iterations and the sampling process is repeated.

2.3.3 The MADS algorithm and the NOMAD solver

The MADS algorithm repeatedly samples the domain of a problem by patterns built on *integer lattices* called a *mesh*. Mathematically, at the k^{th} iteration, the set of sampling points, called the *poll set*, denoted as P_k, is defined as:

$$P_k = \{x_k + \Delta_k^m d : d \in D_k\},$$

where

- x_k is the current incumbent that plays the role of *poll center*
- $\Delta_k^m \geq 0$ is the *mesh size*
- D_k is the set of *poll directions* at the k^{th} iteration and has to satisfy three conditions:
    - *(i)* sampling points are laid on a mesh predefined at the beginning of the iteration,
    - *(ii)* distance between a poll point and the poll center does not exceed a constant times of the poll size denoted as Δ_k^p,
    - *(iii)* D_k is a positive spanning set of $\mathbb{R}^n$.

The first condition imposed on sampling points means that $P_k \in M_k$ with M_k is mathematically defined as

$$M_k = \{x + \Delta_k^m Dz : x \in V_k, z \in \mathbb{N}^n\},$$

where

- V_k is the set of examined points,
- $D = GZ \in \mathbb{R}^{n\times p}$ is a positive spanning set with $G \in \mathbb{R}^{n\times n}$ being nonsingular and $Z \in \mathbb{Z}^{n\times p}$.

Thus, the condition $P_k \in M_k$ can be expressed more specifically as $\forall d \in D_k, \exists u \in \mathbb{N}^p$ such that $d = Du$, the condition on the sampling size is

$$dist(x_k, x_k + \Delta_k^m d) = \Delta_k^m ||d||_\infty \leq \Delta_k^p \max\{||d'||_\infty : d' \in D\}.$$

Intuitively, relations between P_k and M_k are shown by examples illustrated in Figure 2.2 where patterns are represented by arrows with one end at the poll center and the other at a poll point.

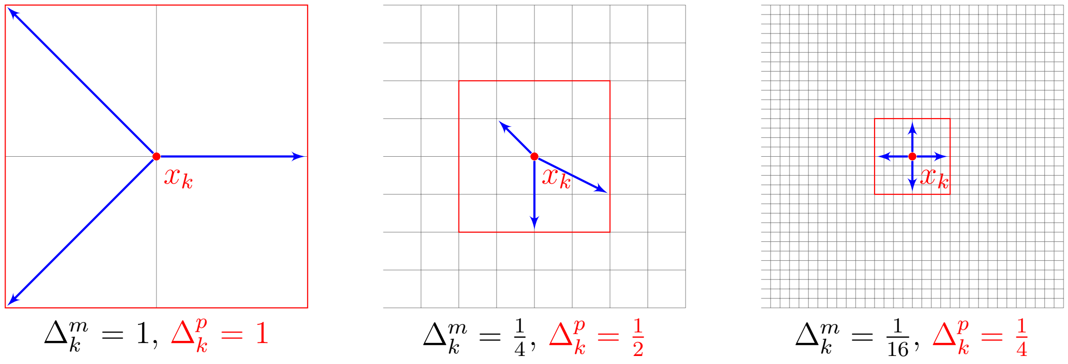

x_k
$\Delta_k^m = 1$, $\Delta_k^p = 1$
x_k
$\Delta_k^m = \frac{1}{4}$, $\Delta_k^p = \frac{1}{2}$
x_k
$\Delta_k^m = \frac{1}{16}$, $\Delta_k^p = \frac{1}{4}$

Figure 2.2 Meshes and poll sets with different sizes

In the definition of a polling set, the introduction of two size-related parameters Δ_k^m and Δ_k^p is very important. In GPS the pattern size is totally controlled by only one parameter that converges to zero when the algorithm samples enough points. In MADS, we control the minimal size (or size unit) by Δ_k^m and the maximal size by Δ_k^p; the restrictions on pattern size now are

(i) At all iterations, $\Delta_k^m \leq \Delta_k^p$

(ii) $\lim_{k\in K} \Delta_k^m = 0 \Leftrightarrow \lim_{k\in K} \Delta_k^p = 0$

The new pattern size control principle does not prohibit of pattern size convergence to zero; furthermore, as a result, the MADS poll direction set D_k is no longer a subset of a predefined set D. In consequence, all the poll directions can form a dense set that indicates that MADS studies thoroughly the neighborhood of the final incumbent.

At the k^{th} iteration, MADS samples the space by performing two steps, SEARCH and POLL. The latter is the crucial step where sampling points are defined by P_k; this step guarantees the convergence to a first-order local optimum. In the former, a finite set of points on M_k is considered; this is an optional step whose aim is to search for a global optimum, or to accelerate a solving process by integrating a heuristic based on particular knowledge of the problem. An example of the SEARCH can be found in the work of Audet *et al.* (2008a).

The convergence of MADS on the blackbox problem

$$\underset{x \in \Omega}{\text{minimize}} \quad f(x)$$

is analysed in Audet and Dennis, Jr. (2006) based on the generalized derivatives $f^\circ(x;d)$ in a direction d, the generalized gradients $\partial f(x)$ and three types of tangent cones (hypertangent cone $T_\Omega^H(\hat{x})$, Clarke tangent cone $T_\Omega^{Cl}(x)$, contingent cone $T_\Omega^{Co}(\hat{x})$) defined by Clarke (1983). The analysis shows that MADS generates a converging sequence $\{x_k\}$ that contains a subsequence $\{x_k\}_{k \in K}$, called the *refining subsequence* that satisfies the following conditions:

(i) $\forall k \in K$ we have $f(x_k) \leq f(x) \ \forall x \in P_k$,

(ii) $\liminf_{k \in K} \Delta_k^p = \liminf_{k \in K} \Delta_k^m = 0$,

(iii) The normalized directions of $\hat{D} = \bigcup_{k \in K} D_k$ are dense in the unit sphere.

Thus, the solution $\hat{x}$ is the limit point of a refining subsequence, $\hat{x} = \lim_{k \in K} x_k$.

The convergence hierarchy states that

(i) if $\Omega = \mathbb{R}^n$ (unconstrained optimization):
  – if the function f is strictly differentiable near $\hat{x}$, then $\nabla f(\hat{x}) = 0$;
  – if the function f is convex, then $0 \in \underline{\partial} f(\hat{x})$, where $\underline{\partial} f(\hat{x})$ is subgradient;
  – if the function f is Lipschitz continuous near $\hat{x}$, then $0 \in \partial f(\hat{x})$;

(ii) if hypertangent cone $T_\Omega^H(\hat{x})$ is non-empty:
  – then $\hat{x}$ is a Clarke stationary point of f over Ω: $f^\circ(\hat{x};d) \geq 0, \ \forall d \in T_\Omega^{Cl}(\hat{x})$;
  – if f is strictly differentiable at $\hat{x}$ and if Ω is regular at $\hat{x}$, then $\hat{x}$ is a contingent KKT stationary point of f over Ω: $\nabla f(\hat{x})^T d \geq 0, \ \forall d \in T_\Omega^{Co}(\hat{x})$.

A complete description and analysis of MADS can be found in Audet and Dennis, Jr. (2006) while some examples of its extensions can be found in Abramson *et al.* (2009a); Audet and Le Digabel (2012); Audet *et al.* (2010b).

NOMAD[14] (Le Digabel, 2011) is a C++ software that implements the MADS algorithm for blackbox optimization under general nonlinear constraints. NOMAD is provided as an executable program or a library corresponding to two modes: batch and library. In the batch mode, users must define their blackbox in the form of an executable that returns output as a list of function values; this mode is intended for

14. `http://www.gerad.ca/nomad`

a basic and simple usage. Library mode targets advanced users who require a flexible solver.

Chapter 3

ORGANIZATION OF THE THESIS

The contributions of this thesis are presented through three papers corresponding to the three following chapters. The present chapter summarizes the works of the three papers in such a way that readers can see our approaches aiming at a parameter tuning framework.

From the reviews of related works in the previous chapter, we can see that the question of parameter tuning is always an important concern; there are many projects but none of them aim at a general framework or a systematical methodology. Hence, our motivation is to propose a framework general enough to apply to virtually all situations, sophisticated enough to take maximum advantage of knowledge of a particular case and flexible to work with other systems. The methodology is initiated by Audet and Orban (2006) with impressive numerical results. The works of this thesis concentrate on developing a framework based on this methodology with three intentions: generality, sophistication and flexibility.

Chapter 4, which corresponds to the publication (Audet *et al.*, 2010a) describes three basic elements that allow launching a tuning session. Although this is a paper that officially introduces the framework, the idea of framework was suggested by Audet and Orban (2006). The contribution of this paper is that this is the first time that the three fundamental questions of an empirical parameter optimization are studied. The basic elements are next identified; they include parameter description, elementary (atomic) measures, algorithm wrapper, simple parameter constraints, composite measures, model data and model structure. As a consequence, to optimize any algorithm, users only need to specify these elements. Within OPAL, these elements are defined by Python syntax in a natural way and a tuning task is described as an optimization model composed of variables, model data and model structure. In addition to the framework description, some simple examples of optimizing the DFO algorithm and numerical results are selectively presented to illustrate OPAL usage and efficiency.

After a framework is established, chapter 5 investigates particularities in order to improve the framework. The second paper published in Audet *et al.* 2011a, illustrates the extensions that target improving framework performance through parallelism and interruption of unnecessary tasks. In the opening part, we show our motivations for parallelizing the tasks. The parallelism is naturally deployed by some particularities of methodology: the core of assessment is to apply the target algorithm over a list of test problems; these applications are independent, thus we can start as many applications as possible at a time; the only constraint is the availability of computational resource. The second place where parallelism can be deployed is the parameter search; although its feasibility depends strongly on the search strategy used. Using NOMAD as the default solver whose parallel working mode is always available, OPAL absolutely has a parallel solver working mode. Taking the advantage of the independence of two stages, assessing the target algorithm and searching the parameter space, OPAL also gives users the possibility of combining two parallelization mechanisms to increase speedup. We deploy the parallelism into OPAL with three working modes and implement it with many techniques behind relating to different parallel platforms such as MPI, LSF or Multi-Threading. However, for OPAL users, parallelism is merely an option in problem definition; that means users can activate by specifying this option a suitable value corresponding to the desired strategy. Besides parallelism, OPAL has another opportunity to accelerate its tuning process with an idea inspired from branch-cutting techniques. We interrupt the target algorithm as soon as an infeasibility is detected. For example, if a parameter optimization requires that the target algorithm returns no error on all 10 test problems, but the target algorithm returned an error on the third problem, there is no need to continue solving the 7 remaining ones, and the entire process may be interrupted. In practice, this technique is neither deterministic nor universal; this means it depends on each concrete problem; it can work with one problem but not with others. Numerical experiment on a trust-region solver, called TRUNK is presented. In the discussion, we propose some directions to apply parallelism more smoothly and more efficiently as well as techniques to increase the probability of interruptions.

Chapter 6 that is in progress paper describes OPAL as a parameter tuning framework, as a Python package implementing the framework. In addition to systematically recalling the main characteristics and features, a new feature relating to the integration mechanisms of OPAL with other systems is introduced. The new feature is introduced through a case-study whose numerical results are shown in the Appendix of this thesis. The case-study shows a cooperation between OPAL and a classification tool based on SOM (Self-Organizing Map, Kohonen 1997) to solve a parameter optimization problem. The difficulty is that the set of test problems is so large that it takes too much time for an assessment, it can also prevent the searching strategy from heading to a promising region in the parameter space. Thus, we desire to extract a good subset of test problems for defining the parameter optimization problem. A SOM-based clustering algorithm is involved in order to get the subset based on the atomic measures obtained from running the target algorithm with the default parameter setting. The obtained subset of test problems is then used to define a surrogate or a parameter optimization subproblem that can guide the search approaching a promising trajectory within an acceptable restriction on tuning time.

Finally, chapter 7 discusses the contributions of the thesis and suggests possible extensions of the framework.

Chapter 4

ALGORITHMIC PARAMETER OPTIMIZATION OF THE DFO METHOD WITH THE OPAL FRAMEWORK

Charles Audet **Cong-Kien Dang** **Dominique Orban**

Abstract

We introduce the Opal framework in which the identification of good algorithmic parameters is interpreted as a black box optimization problem whose variables are the algorithmic parameters. In addition to the target algorithm, the user of the framework must supply or select two components. The first is a set of metrics defining the notions of *acceptable* parameter values and of *performance* of the algorithm. The second is a collection of representative sets of valid input data for the target algorithm. Opal may be applied to virtually any context in which parameter tuning leads to increased performance. The black box optimization problem is solved by way of a direct-search method which provides local optimality guarantees and offers a certain flexibility. We illustrate its use on a parameter-tuning application on the DFO method from the field of derivative-free optimization.

This chapter corresponds to the paper published in Audet *et al.* (2010a)

4.1 Introduction

Most computational tasks depend on a set of parameters. Such tasks include running numerical methods on input problems with intent to identify a solution. The choice of a sparse matrix storage format influences the speed of dot products. The grain in a grid computing environment directly affects throughput and performance. The choice of an adequate preconditioner for a given class of problems affects performance and may even make the difference between solving the problem and not solving it. The list goes on and is not limited to scientific computing applications. Compilers generate machine code, the efficiency of which depends on loop unrolling levels, loop blocking and other parameters. Network parameters influence throughput. A natural question ensues: how can we tune those parameters so as to increase the performance of our computational tasks? In this paper, we describe a flexible practical environment in which to express parameter tuning problems and solve them using nondifferentiable optimization tools. Our environment is independent of the application area and runs on almost any platform.

Typically, computational tasks do not depend *smoothly* on their parameters. Jumps in performance may occur when the value of a parameter is changed. In many cases, the performance measure cannot be expressed in analytical form. Worse yet, it may not even be a *function*, i.e., it may yield different readings when read twice with the same parameter values. Examples of this phenomenon include CPU time and any measure that is inherently inaccurate. Computational tasks come in such wide diversity and in a multitude of programming languages that any language-dependent attempt at tackling the parameter-tuning problem is bound to fail. For these reasons, in the work of Audet and Orban (2006) the problem of identifying locally-optimal parameters is formulated as a black box optimization problem, i.e., one in which we seek to maximize performance while at the same time constraining all parameters to remain within the limits allowed by the application. More precisely, the problem can be stated as

$$\underset{p \in \mathbb{R}^n}{\text{minimize}} \;\; \psi(p) \quad \text{subject to } p \in \mathbf{P}, \tag{4.1}$$

where $p \in \mathbb{R}^n$ is the vector of parameters, $\mathbf{P} \subseteq \mathbb{R}^n$ is the set of acceptable parameter values, and ψ is a performance measure. By convention, we state (4.1) as a minimization problem but it could equally be stated as a maximization problem by flipping the sign of ψ. A typical property of parameter-turning problems is that ψ

may be nonsmooth, even discontinuous, and that the set $\mathbf{P}$ is not normally described by smooth functions. The problem is also usually nonconvex. Descent methods or derivative-free methods that assume the existence of $\nabla\psi(p)$ are not viable options to solve (4.1) since we must rely on function values only. For these reasons, direct methods are employed—see for instance Kolda *et al.* (2003) for an overview and pointers to the literature. Besides only requiring the evaluation of ψ at a number of different values of p, a well-chosen direct method will offer certain optimality guarantees upon termination, which is in contrast with heuristic methods.

In a typical situation, inexperienced users trust the default parameter settings and never change them. Even the most experienced users may find it challenging to set parameters to *better* values for the problem at hand. The task is made more arduous by the fact that simple algorithms can depend on 5 to 20 parameters or more. This combines with the computational cost of evaluating the worthiness of a given set of parameter values to make it impossible to thoroughly explore the search space.

In the black box optimization framework of Audet and Orban (2006), two questions must be answered before parameter tuning can take place:

1. "What are the *acceptable* parameter values?" This question is usually partially answered by the specifications of the computational task, e.g., the step length in a linesearch may not become negative. The user may include additional conditions for specific purposes. For instance, the number of iterations performed by the method must not exceed a specified threshold and it is understood that this number of iterations depends implicitly on the parameter settings.
2. "In what sense is a set of parameter values *better* than another?" This question defines the notion of *performance*. Simple performance measures are the total CPU time, the number of iterations, the number of linear systems solved, the success or failure to solve a given problem, etc. We call such measures *atomic* because they are normally read directly from the output of the computation. More elaborate, *compound*, measures are typically used, such as weighted combinations of atomic measures.

In this paper, we implement and extend the framework of Audet and Orban (2006) by generalizing the black box formulation of the parameter tuning problem, and by providing an environment that is flexible enough to encompass a wide range of application areas while at the same time retaining ease of use and efficiency. Our extension to the framework consists in the utilization of atomic and compound measures to define the objective and constraint functions, and in flexibility in the selection of the optimization solver.

Our environment is written in the Python programming language which is easy to learn by example for the newcomer, natural to use for the fluent programmer, and flexible and powerful for the advanced user. Some distinctive aspects of our procedure are that it is non intrusive, it requires no modification of the code or algorithm being tuned, and it does not require knowledge of the internals of this code or algorithm or of the programming language in which it is written. We illustrate the use of this environment on a test case from the field of derivative-free optimization.

The use of optimization methods for stability analysis of computational methods may be traced back to the mid-70's. In Miller (1975) and Larson and Sameh (1980) languages are devised in which numerical algorithms are to be implemented. Upon compilation, a descent method exercises the algorithm by varying its input so as to maximize an error measure with the intent of assessing the numerical stability of the method as implemented. The programming languages impose a number of stringent rules on the implementation, which, for instance, may not make use of loops.

The more recent literature on parameter tuning include the description of the CALIBRA system of Adenso-Diaz and Laguna (2006), based on fractional factorial experimental designs coupled with a local search. Major limitations of this system are that it only handles up to five parameters and does not offer optimality guarantees. On the contrary, ParamILS (Hutter *et al.*, 2007) avoids the pitfalls of over-training by taking on a stochastic approach and provably converges to a local optimum in a statistical sense.

Some parameter-tuning applications have had a major impact on the efficiency of modern numerical methods. The best example is surely the ATLAS[1] library of automatically-tuned linear algebra software of Whaley and Dongarra (1998) which adds a parameter-tuning layer over the standard BLAS library of critical linear algebra kernels (Lawson *et al.*, 1979; Blackford *et al.*, 2002). The addition of such a parameter-tuning layer is a paradigm termed AEOS—Automated Empirical Optimization of Software (Whaley *et al.*, 2001). The heuristic search used in the ATLAS is a coordinate search. The report of Seymour *et al.* (2008) provides a comparison of various search strategies, not including direct search, to the study of automatic code optimization. The PHiPAC project Bilmes *et al.* (1998) aims to provide near-peak performance

1. `http://math-atlas.sourceforge.net`

in linear-algebra kernels by way of parametrized code generators. The parametrized code is then explored by heuristic search procedures. Similar functionality is provided by the OSKI library (Vuduc *et al.*, 2005).

The black box optimization framework for the identification of locally-optimal algorithmic parameters proposed by Audet and Orban (2006) was used to tune the four parameters of a trust-region solver for unconstrained optimization with respect to two performance measures—the total computing time and the number of function evaluations on test problems taken from the CUTEr collection (Gould *et al.*, 2003a). Using a surrogate function to guide the optimization, the authors identify parameter values that reduce the computing time by approximately 25% over the default values. The final parameter values obtained are very close to those identified by a nearly-exhaustive exploration of the search space (Gould *et al.*, 2005). This gives us reason to believe that there is a lot to be gained in using proven optimization methods backed by a solid convergence theory to tackle parameter-tuning applications.

The rest of this paper is organized as follows. Section 4.2 describes the relevance of black box optimization and direct-search methods to parameter-tuning problems and the various ingredients necessary to completely specify a given application. Section 4.3 covers the details of our parameter-tuning package. In §4.4 we work through a practical application in which the parameters of a derivative-free solver for optimization are optimized. We finish with a discussion of further research in §4.5.

4.2 Optimization of Algorithmic Parameters

Optimizing parameters is tightly linked to the type of input that will be fed to the algorithm or computational task that is to be carried out. For a given collection of sets of input data (such as for example, test problems), certain locally-optimal parameter values may be found but these may differ if the collection of sets of input data is changed. Therefore, the input data is a defining component of the parameter-tuning problem. For the purposes that the user has in mind, adequate input data must be used and the final parameter values must be interpreted in the context of this input data.

Throughout the remainder of this paper we restrict our attention to real parameters and use the following notation. We denote by $\mathcal{A}$ the algorithm whose parameters are to be optimized, by $\mathcal{L}$ a finite collection of representative sets of input data, such as test problems, for Algorithm $\mathcal{A}$ and by $p \in \mathbb{R}^n$ the vector of parameters that we wish to optimize. Finally, let $\mathbf{P} \subseteq \mathbb{R}^n$ denote the domain from which p must be selected. The definition of $\mathbf{P}$ usually follows from the specification of Algorithm $\mathcal{A}$.

For future reference, a (very) high-level structure of our parameter optimization framework is illustrated in Figure 4.1.

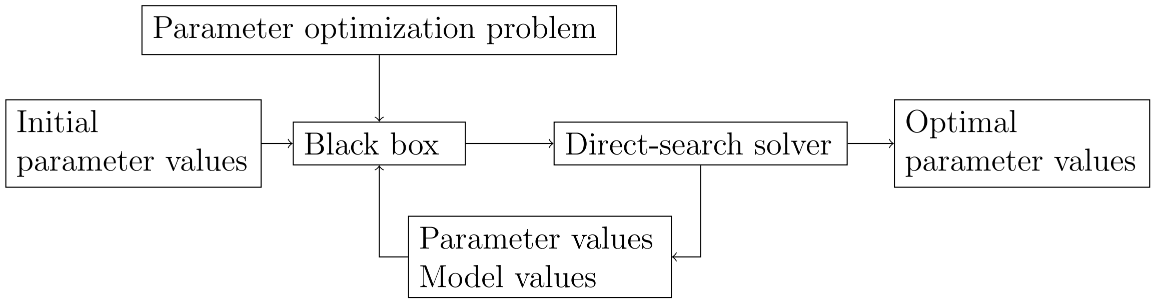


Figure 4.1 Schematic Algorithmic Parameter Optimization Framework

4.2.1 Black Box Construction

In the following, and for consistency with our implementation described in §4.3, a black box optimization problem representing a parameter-tuning problem will be called a *model*. We divide the specification of the model into two components. The first component is the *model structure* and specifies the fundamental abstract aspects of the problem: the performance measure and the constraints. The model structure is the skeleton of the problem. The same structure might apply to various parameter-tuning problems. In this sense, it does not fully characterize the model until we specify the *model data*, which is the second component. It specifies which algorithm or computational task is concerned, which collection of sets of input data will be fed to the computational task, as well as a description of the parameters of this task and a description of $\mathbf{P}$.

In order to define a performance measure and constraints, it is important to describe and collect all the relevant *measures* reported by our computational task when it is fed a valid set of input data. Those measures usually provide statistics on the run and an assessment of the quality of the final result. For instance, a typical algorithm for smooth optimization, when fed a test problem, will return the computing time, the number of iterations, the number of objective and constraint functions evaluations, the number of linear systems solved, the accuracy of the solution identified, an exit code, etc. Such *atomic* measures are readily accessible from the solver's output. It is those atomic measures that are used to define a performance measure and constraints in our model.

For a given test problem $\ell \in \mathcal{L}$, the i-th atomic measure may be viewed as a function of the parameters of the algorithm or computational task, i.e., as a function $p \mapsto \mu_\ell^i(p)$ from $\mathbb{R}^n$ into $\mathbb{R} \cup \{\infty\}$. We gather the, say, q atomic measures reported by Algorithm $\mathcal{A}$ into the vector-valued function $\mu_\ell : \mathbb{R}^n \to (\mathbb{R} \cup \{\infty\})^q$. A run of Algorithm $\mathcal{A}$ essentially gives access to a measure matrix from which compositions of atomic measures—or *compound* measures—may be constructed. For example, if the i-th measure is the CPU time, a typical performance measure is then $\psi(p) = \sum_\ell \alpha_\ell \mu_\ell^i(p)$ for certain weights $\alpha_\ell \geq 0$. Arbitrary compound measures may be used to define performance and constraints in the model without concern for continuity or smoothness.

The black box optimization problem may now be stated as

$$\begin{array}{ll} \underset{p}{\text{minimize}} & \psi(p) \\ \text{subject to} & p \in \mathbf{P} \\ & \varphi(p) \in \mathbf{M}, \end{array} \tag{4.2}$$

where ψ and $\varphi = (\varphi_1, \ldots, \varphi_s)$ are compound measures and $\mathbf{M}$ is a user-defined feasible set. For example, the user may wish to minimize the CPU time $\psi(p)$ while requiring that at least 90% of the test problems be solved to within an accuracy of 10^{-6}. Problem (4.2) generalizes problem (1) of Audet and Orban (2006) by allowing constraints involving atomic and compound measures, rather than simply allowing the domain to be entirely defined by $\mathbf{P}$.

The components of the black box are thus as follows. The user supplies two ingredients: The model structure and the model data. The first represents the black box problem (4.2) while the second contains Algorithm $\mathcal{A}$ along with its full specification and a collection $\mathcal{L}$ of test problems.

4.2.2 Direct Search Algorithms

The nonsmooth optimization problem (4.2) represents the question of identifying good algorithmic parameters. It is nonsmooth because the evaluation of the objective function and constraints relies on performance measures obtained by launching algorithm $\mathcal{A}$ on a collection of test problems. It is worth repeating that technically, the measures may not even be functions since, for example, the CPU time required to solve a problem may differ slightly from one run to another.

As suggested by Figure 4.1, problem (4.2) is treated as a black box. Direct search algorithms, designed for nonsmooth optimization problems, rely on function values only at sample points to search for an optimal solution. They do not require knowledge or even existence of any derivatives to explore the space of variables. In the present work, we use the Mesh Adaptive Direct Search (MADS) algorithm (Audet and Dennis, Jr., 2006). The reasons motivating our choice are that MADS is supported by a hierarchical convergence analysis based on Clarke's nonsmooth calculus (Clarke, 1983), and has been successfully applied to parameter-optimization problems (Audet and Orban, 2006).

In order to solve (4.2), a MADS algorithm generates trial points on an underlying mesh in the domain space. A mesh is an enumerable subset of the domain space whose coarseness is driven by an iteration-dependent mesh size parameter $\Delta_k > 0$. At each iteration, the algorithm attempts to improve the current best solution, called the incumbent, by evaluating the objective and constraint functions at finitely many trial points on the mesh. Trial points that violate the constraints are either simply rejected from consideration, or handled by a progressive barrier (Audet and Dennis, Jr., 2009). This last strategy allows an infeasible starting point. If an improved solution is found, the mesh size can be increased to allow sampling further away and thus promote fast progress towards promising regions. Otherwise, the incumbent is a local minimizer with respect to the neighbouring poll points. The mesh size parameter is reduced and another cycle begins on the finer mesh.

As the algorithm unfolds, the mesh size parameter satisfies $\liminf \Delta_k = 0$ under the assumption that all trial points remain in a bounded set. Thus, regardless of the smoothness or lack thereof of the functions defining the problem, MADS generates a convergent subsequence of trial points, each of which is a local mesh-minimizer in a certain sense, on a sequence of meshes that become infinitely fine. Adding more assumptions on the smoothness leads to a hierarchical convergence analysis.

If the objective function is Lipschitz continuous, the Clarke generalized derivatives are nonnegative in the hypertangent directions to the feasible region. The analysis also states that if the objective is strictly differentiable, and if the domain is regular, then the limit point is a KKT stationary point. The interested reader can consult Audet and Dennis, Jr. (2006) for a complete description of MADS and its convergence analysis.

4.3 The OPAL Package

We propose the OPAL package as an implementation of the framework detailed in the previous sections. The name stands for **Op**timization of **Al**gorithms. In this initial version, only real algorithmic parameters are allowed, although our framework makes provision for integer and categorical parameters. Work is under way to permit usage of those more general parameter types.

Computational tasks in need of parameter tuning come in infinite variety on widely different platforms and in vastly different environments and languages. It seems *à priori* arduous to design a parameter-tuning environment that is both sufficiently portable and sufficiently flexible to accommodate this diversity. Moreover not all users are computer programmers and therefore any general tool seeking to meet the above flexibility requirements must be as easy to use as possible without sacrificing expandability and advanced usage. In our opinion, the latter constraints rule out all low-level programming languages. There remains a handful of options that are portable, flexible, expandable and user friendly. Among those, our option of choice is the Python programming language[2].

Python is an open-source scripting language in constant development which has evolved through its thriving user community to become a standard. It is available on almost any imaginable platform. Users can write Python programs much in the same way as shell scripts, batch scripts or Apple scripts, or elect to use the full power of object-oriented programming. A wide range of numerical and scientific extensions is available for Python. In addition, Python is a full-fledged programming language with an extensive standard library.

2. http://www.python.org

4.3.1 The OPAL Structure

Within the Opal Python environment for algorithmic parameter optimization, a model is represented by the same two ingredients as described in §4.2: a model structure and model data. Once a model has been defined, it may be solved with any direct-search solver available. The whole environment is decomposed into a number of modules that help users describe a model in a natural manner. There are thus two main components to the parameter-tuning problem: the *Black-Box Model* and the *Direct-Search Solver*. Those two components of Opal along with a few other can be combined to form a fully-specified parameter optimization problem.

For now, the *Direct-Search Solver* component contains a single specific instance: the NOMAD implementation (Abramson *et al.*, 2004) of the Mads family of algorithms.

The *Black-Box Model* component contains the two main ingredients that constitute a model. The *Model Structure* component lets users specify a high-level description of Problem (4.2). It gives access to atomic measures and gives the possibility to build arbitrary compound measures. The *Model Data* component contains the problem-specific information necessary to start solving (4.2). It allows users to specify an algorithm from the *Algorithms* component. It lets users choose corresponding input data from the *Test Problems* component. Finally, it offers a selection of preprogrammed compound measures that are likely to be useful in many contexts, such as the total cpu time, the total number of function evaluations, and the termination code, to name a few.

Opal is build with easy expandability in mind thanks to object-oriented programming. Users can define new algorithms, compound measures, test data sets and solvers by specializing—or *subclassing*—high level conceptual classes that abstract such objects.

4.3.2 Usage of OPAL

We now briefly describe, by way of an example, a few implementation details regarding some of the above-mentioned components. This will give a glimpse of the conciseness of working examples and of how the power of the Python language is harnessed in Opal. The example concerns the algorithm DFO described in the next section. Knowledge of DFO is not necessary however to work through the example.

The final code is given in Listing 4.1. As is customary in Python, but not mandatory, we gather all `import` commands at the top of our script. Note that thanks to object-oriented constructs and Python's human-readable syntax, the code is relatively close to natural language and is intuitive. We now describe its various statements.

Listing 4.1 Minimal Realistic Example

```
 1 from opal.Algorithms import DFO
 2 from opal.TestProblemCollections import CUTEr
 3 from opal.Solvers import NOMAD
 4 from opal import StatisticalMeasure as stats
 5 from opal import ModelStructure, ModelData, BlackBox
 6
 7 # Select real parameters for DFO
 8 params = [par for par in DFO.parameters if par.is_real]
 9
10 # Select tiny unconstrained HS problems
11 probs = [pb for pb in CUTEr.HS if pb.nvar<=5 and pb.ncon==0]
12
13 # Build (unconstrained) model structure and model data
14 data = ModelData(DFO, probs, params)
15 structure = ModelStructure(objective=stats.average('CPU'))
16
17 blackbox = BlackBox(modelData=data, modelStructure=structure)
18 NOMAD.solve(blackbox)
```

An important component is the *parameter*. In OPAL, parameters are represented by abstract objects that have a name, a kind and a default value. Parameters are intrinsically tied to the computational task whose performance is to be optimized. In OPAL, computational tasks are generically referred to as *algorithms*.

The `DFO` object exposed in the current workspace by the `import` command on line 1 is a compound object containing certain *members*. The set of parameters associated with DFO is one such member and is accessible by typing `DFO.parameters`. If we wanted to work on all parameters of DFO irrespective of their kind, we would supply `DFO.parameters` as an argument when we build the model data. However, the Python syntax lets us easily extract parameters of interest only using *list comprehension* as in the statement of line 8. As expected, this statement builds a list of the *real* parameters only. Another, longer to type, possibility would be to select the parameters by name. The only missing ingredient to the model data is the set of input test problems. Because DFO is an optimization algorithm, we select optimization test problems from the Hock and Schittkowski ("HS") collection (Hock and Schittkowski, 1981). Since the latter is an integral part of the CUTEr collection (Gould *et al.*, 2003a), it is defined as a subcollection of the CUTEr problems in Opal. The CUTEr problems are imported via the command of line 2 and the subcollection of HS problems, being a member of `CUTEr`, is accessed via `CUTEr.HS`. For our minimal example, we illustrate another usage of list comprehension to select only a few HS problems in line 11. This effectively restricts our test set to unconstrained HS problems that have at most 5 variables. We have all the elements to assemble our model data. Line 5 imports the definition of the abstract template representing the data of a model, along with similar templates for the model structure and the black box solver, to be used later. Line 14 creates an instance by populating the abstract template with our selections.

The model structure of the minimal example does not have any constraints, for simplicity, aside from those defining the set $\mathbf{P}$. The latter set is defined in the specifications of the `DFO` object. The objective function of the problem is chosen in line 15 to be the average time. This simple predefined measure was imported on line 4. Note that the `ModelStructure` template was imported at the same time as `ModelData` above. The final step is to use our complete model to define a black box, and solve it using NOMAD. This is performed in lines 17 and 18.

In the second part of this paper, we work through a more realistic parameter tuning of DFO and compare our results with those corresponding to default parameters.

4.4 Application to Derivative-Free Optimization

In this section, we illustrate the usage of our software package to determine suitable parameter values in the derivative-free optimization solver DFO (Conn *et al.*, 1998).

4.4.1 General Description of DFO

DFO is the implementation of an algorithm for constrained and unconstrained problems which does not rely upon availability of the derivatives of the objective and constraint functions. It does however assume that they exist. The method is said to be *model-based* because at each iteration, a quadratic model of the objective function is computed and approximately minimized within a trust region. In the presence of linear constraints or simple bounds, the model is minimized over the intersection of the trust region and the portion of the feasible set described by those constraints.

If more general constraints are present and are not simply true/false constraints, they are combined with the objective function by way of a penalty term. The latter depends on a penalty parameter which is updated as the algorithm proceeds. DFO also allows boolean constraints which simply indicate whether a given point is feasible or not, without returning any measure of infeasibility.

The premise of DFO is that an evaluation of the objective and nonlinear constraints is expensive enough that building a quadratic model from an interpolation set and minimizing this model over a trust region has negligible cost. At each iteration, the algorithm stores a set of feasible points arranged so that computing a quadratic interpolant is a well-posed problem—this set is said to be *poised*.

4.4.2 Two DFO Parameter Optimization Problems

DFO depends on the set of algorithmic parameters described in Table 4.1. We restrict our attention to the real parameters, holding the others fixed at their default value and use the tools described in the previous sections to identify parameter values that approximately minimize various performance measures.

Table 4.1 Algorithmic Parameters of DFO.

Name	Type	Domain	Purpose
MAXIT	integer	$\mathbb{N}$	Maximum number of iterations
MAXNF	integer	$\mathbb{N}$	Maximum number of evaluations
STPCRTR	categorical	$\{1, 2\}$	Stopping criterion
DELMIN	real	$\mathbb{R}_+$	Smallest trust-region radius
STPTHR	real	$\mathbb{R}_+$	Slow progress threshold
DELTA	real	$\mathbb{R}_+$	Initial trust region radius
CNSTOL	real	$\mathbb{R}_+$	Feasibility tolerance
PP	real	$\mathbb{R}_+$	Initial penalty parameter
SCALE	categorical	$\{\text{True}, \text{False}\}$	Scaling

We use sets of test problems extracted from the CUTEr (Gould *et al.*, 2003a) and HS collections (Hock and Schittkowski, 1981). The sets consist of equality-constrained, inequality-constrained and unconstrained problems, respectively. The name and dimension of these problems are presented in Tables 4.2 and 4.3. Our test problems are the same as those of Conn *et al.* (1998) except for some differences in the selection of HS problems. Observe that in the latter paper, problem HS26 is misclassified as an inequality-constrained problem. Furthermore, our tests differ from those of Conn *et al.* (1998) as we use IPOPT (Wächter and Biegler, 2006) to minimize the quadratic model at each iteration. In our experiments, the test problems are partitioned into two sets, one to optimize the parameters—the *training* set—and the other for cross-validation tests.

Table 4.2 Unconstrained problems from the CUTEr collection; n is the number of variables.

Name	n	Name	n	Name	n	Name	n
AKIVA	2	ALLINITU	4	BEALE	2	BIGGS6	6
BOX3	3	BRKMCC	2	BROWNAL	10	BROWNBS	2
BROWNDEN	4	BRYBND	10	CLIFF	2	CRAGGLVY	10
CUBE	2	DENSCHNA	2	DENSCHNB	2	DENSCHNC	2
DENSCHND	3	DENSCHNE	3	DENSCHNF	2	DIXMAANK	15
DJTL	2	DQRTIC	10	EIGENALS	6	ENGVAL2	3
EXPFIT	2	FMINSURF	16	GROWTHLS	3	GULF	3
HAIRY	2	HATFLDD	3	HATFLDE	3	HEART6LS	6
HEART8LS	8	HELIX	3	HIELOW	3	HILBERTA	2
HILBERTB	10	HIMMELBB	2	HIMMELBF	4	HIMMELBG	2
HIMMELBH	2	HUMPS	2	JENSMP	2	KOWOSB	4
LOGHAIRY	2	MANCINO	10	MARATOSB	2	MEXHAT	2
MEYER3	3	MOREBV	10	OSBORNEA	5	OSBORNEB	11
OSCIPATH	15	PALMER1C	8	PALMER1D	7	PALMER2C	8
PALMER3C	8	PALMER4C	8	PALMER5C	6	PALMER6C	8
PALMER7C	8	PALMER8C	8	PARKCH	15	PFIT1LS	3
PFIT2LS	3	PFIT3LS	3	PFIT4LS	3	POWER	10
ROSENBR	2	S308	2	SINEVAL	2	SISSER	2
SNAIL	2	SROSENBR	10	STRATEC	10	TRIDIA	10
VARDIM	20	VIBRBEAM	8	WATSON	12	WOODS	12
YFITU	3	ZANGWIL2	2				

For conciseness, we will use the following notation when referring to the test problems and parameters. The list of test problems is denoted by $\mathcal{L}$, and will either contain all 82 unconstrained problems, or the 125 constrained ones. Let $p = (p_1, p_2, p_3, p_4, p_5) = (\text{DELMIN}, \text{STPTHR}, \text{CNSTOL}, \text{DELTA}, \text{PP})$ denote the real-valued parameters of DFO. Following the recommendations from the DFO User's Manual

Conn *et al.* (2009a), we define the feasible region to be the set $\mathbf{P}$ of vectors $p \in \mathbb{R}^5$ that satisfy the following linear and bound constraints

$$10^{-8} \le p_1 \le 10^{-3}, \quad 0 \le p_2 \le 1, \quad 0 \le p_3 \le 0.1, \quad p_1 \le p_4, \quad \text{and} \quad 1 \le p_5.$$

The default parameter values are $p^0 = (10^{-4}, 10^{-3}, 10^{-5}, 1, 10^3)$.

Note that while the DFO documentation does not explicitly recommend a value for $p_5 = \text{PP}$, the example driver sets it to 1000. We thus selected the latter value as default.

We next define atomic measures associated to a specific test problem ℓ from one of the test sets $\mathcal{L}$ presented in the above tables.

– $\mu_\ell^{\text{EVAL}} : \mathbf{P} \to \mathbb{N}$ returns the number of function evaluations required by DFO to solve problem ℓ with parameters p,
– $\mu_\ell^{\text{SOLVED}} : \mathbf{P} \to \{-9, -8, \ldots, -1, 0, 1, 2\}$ returns the DFO exit code when solving problem ℓ with parameters p. A zero exit code means that the problem ℓ was solved successfully. All other exit codes indicate a failure,
– $\mu_\ell^{\text{QUALITY}} : \mathbf{P} \to \mathbb{R} \cup \{+\infty\}$ returns $+\infty$ if $\mu_\ell^{\text{SOLVED}} \neq 0$. Otherwise, it returns the final objective function value produced by DFO using the parameters p on problem ℓ.

From the atomic measures, we define the following compound measures to construct our objective function and constraints. The proportion of problems solved is

$$\varphi^{\text{SOLVED}}(p) = \frac{|\mathcal{S}(p)|}{|\mathcal{L}|} \in [0, 1],$$

where $\mathcal{S}(p) = \{\ell \in \mathcal{L} \mid \mu_\ell^{\text{SOLVED}}(p) = 0\}$ is the set of indices of problems solved when using parameter p. For comparison with the default values, we also define $\mathcal{S}(p, p^0) = \mathcal{S}(p) \cap \mathcal{S}(p^0)$ to be the set of problems successfully solved with both parameters p and p^0.

The average normalized reduction in the number of evaluations with respect to the default parameter p^0 on problems that were successfully solved with both parameters p and p^0 is written

$$\varphi^{\text{EVAL}}(p) = \frac{1}{|\mathcal{S}(p, p^0)|} \sum_{\ell \in \mathcal{S}(p, p^0)} \frac{\mu_\ell^{\text{EVAL}}(p) - \mu_\ell^{\text{EVAL}}(p^0)}{\mu_\ell^{\text{EVAL}}(p) + \mu_\ell^{\text{EVAL}}(p^0)}.$$

Table 4.3 Constrained problems from the Hock-Schittkowski collection. Here, n is the number of variables and m is the number of constraints.

Name	n	m	Name	n	m	Name	n	m	Name	n	m
HS1	2	0	HS10	2	1	HS100	7	4	HS100LNP	7	2
HS100MOD	7	4	HS101	7	5	HS102	7	5	HS103	7	5
HS104	8	5	HS105	8	1	HS106	8	6	HS107	9	6
HS108	9	13	HS109	9	10	HS11	2	1	HS110	10	0
HS111	10	3	HS111LNP	10	3	HS112	10	3	HS113	10	8
HS114	10	11	HS116	13	14	HS117	15	5	HS118	15	17
HS119	16	8	HS12	2	1	HS13	2	1	HS14	2	2
HS15	2	2	HS16	2	2	HS17	2	2	HS18	2	2
HS19	2	2	HS2	2	0	HS20	2	3	HS21	2	1
HS21MOD	7	1	HS22	2	2	HS23	2	5	HS24	2	3
HS25	3	0	HS26	3	1	HS268	5	5	HS27	3	1
HS28	3	1	HS29	3	1	HS3	2	0	HS30	3	1
HS31	3	1	HS32	3	2	HS33	3	2	HS34	3	2
HS35	3	1	HS35I	3	1	HS35MOD	3	1	HS36	3	1
HS37	3	2	HS38	4	0	HS39	4	2	HS3MOD	2	0
HS4	2	0	HS40	4	3	HS41	4	1	HS42	4	2
HS43	4	3	HS44	4	6	HS44NEW	4	6	HS45	5	0
HS46	5	2	HS47	5	3	HS48	5	2	HS49	5	2
HS5	2	0	HS50	5	3	HS51	5	3	HS52	5	3
HS53	5	3	HS54	6	1	HS55	6	6	HS56	7	4
HS57	2	1	HS59	2	3	HS6	2	1	HS60	3	1
HS61	3	2	HS62	3	1	HS63	3	2	HS64	3	1
HS65	3	1	HS66	3	2	HS67	3	14	HS68	4	2
HS69	4	2	HS7	2	1	HS70	4	1	HS71	4	2
HS72	4	2	HS73	4	3	HS74	4	5	HS75	4	5
HS76	4	3	HS76I	4	3	HS77	5	2	HS78	5	3
HS79	5	3	HS8	2	2	HS80	5	3	HS81	5	3
HS83	5	3	HS84	5	3	HS85	5	21	HS86	5	10
HS87	6	4	HS88	2	1	HS89	3	1	HS9	2	1
HS90	4	1	HS91	5	1	HS92	6	1	HS93	6	2
HS95	6	4	HS96	6	4	HS97	6	4	HS98	6	4
HS99	7	2									

By convention, we set $\varphi^{\text{EVAL}}(p) = +\infty$ when $\mathcal{S}(p, p^0) = \emptyset$. We average $\varphi^{\text{EVAL}}(p)$ over the problems successfully solved, since the in the course of the direct method iterations, we may encounter trial parameters for which not all problems can be solved. It would however be desirable that all, or sufficiently many, problems be solved with the optimized parameters.

Similarly, the average normalized solution quality improvement with respect to the default parameter p^0 is expressed as

$$\varphi^{\text{QUALITY}}(p) = \frac{1}{|\mathcal{S}(p,p^0)|} \sum_{\ell \in \mathcal{S}(p,p^0)} \frac{\mu_\ell^{\text{QUALITY}}(p) - \mu_\ell^{\text{QUALITY}}(p^0)}{|\mu_\ell^{\text{QUALITY}}(p)| + |\mu_\ell^{\text{QUALITY}}(p^0)|}.$$

A negative value of $\varphi^{\text{QUALITY}}(p)$ indicates that, on average, the parameters p produce an improvement over p^0. If ever both the numerator and denominator of the ℓ-th term of the sum vanish, we reset this term to zero.

The above compound measures allow us to formulate the two following parameter tuning problems.

$$\begin{aligned} \underset{p\in\mathbf{P}}{\text{minimize}} \quad & \varphi^{\text{EVAL}}(p) \\ \text{subject to} \quad & \varphi^{\text{SOLVED}}(p) = 1, \\ & \varphi^{\text{QUALITY}}(p) \le 0, \\ & p_2 = p_2^0, \\ & p_5 = p_5^0, \end{aligned} \tag{4.3}$$

$$\begin{aligned} \underset{p\in\mathbf{P}}{\text{minimize}} \quad & \varphi^{\text{EVAL}}(p) \\ \text{subject to} \quad & \varphi^{\text{SOLVED}}(p) = 1, \\ & \varphi^{\text{QUALITY}}(p) \le 0, \\ & p_2 = p_2^0, \\ & p_3 = p_3^0 \end{aligned} \tag{4.4}$$

Problem (4.3) will be used to tune parameters on the unconstrained test problems of Table 4.2 while (4.4) will be used on the constrained problems of Table 4.3. In both problems we fix p_2 = STPTHR to its default value because this parameter is a stopping tolerance for DFO. Allowing it to vary while minimizing the number of evaluations would not make sense, since the optimizer would simply increase its value. For the same reason, the feasibility tolerance p_3 = CNSTOL is fixed in (4.4). However, p_3 also plays other roles related to management of the trust region in the DFO implementation, and this is why it is allowed to vary in (4.3). Since the penalty parameter p^5 = PP has no effect for unconstrained problems, we fix it in the constraints of (4.3) and thereby reduce the dimension of the search space.

4.4.3 Numerical Results

We perform three sets of numerical experiments. In each experiment, the test problem set is divided into two subsets: a training subset and a cross-validation subset. We apply the OPAL framework to the training set to identify good algorithmic parameters. The quality of those parameters is then measured on the cross-validation subset. In order to determine what proportion of problems should be considered in the training phase, we select training subsets consisting in 50%, 25% and 10% of the test problems. The training problems are selected by listing all problems alphabetically (as in Tables 4.2 and 4.3) and selecting every second, fourth or tenth problem. The remaining 50%, 75% and 90% of the problems constitute the cross-validation subset.

The optimization in the training phase is performed with the NOMAD direct-search method, using all default parameters. The termination criteria is set to 500 evaluations, i.e., DFO is launched on the training set at most 500 times. In addition, NOMAD performs an automatic diagonal scaling of the variables by dividing $p \in \mathbb{R}^5$ by $(10^{-5}, 1.0, 1.0, 10^{-4}, 10^2)$.

Table 4.4 illustrates the training phase. The table shows the default and optimized parameters for the unconstrained and constrained cases. The column BB gives the number of black-box evaluations—i.e., the number of times that φ^{EVAL} was evaluated—that were necessary to identify the final parameter values. Recall that one evaluation of φ^{EVAL} requires a run of DFO on each training problem.

Table 4.4 Optimized Parameters from Training Phase for (4.3) on Unconstrained Test Problems (left) and for (4.4) on Constrained Test Problems (right). For each training phase, the column BB gives the number of black-box evaluations.

	DELMIN	CNSTOL	DELTA	BB	DELMIN	DELTA	PP	BB
Deflt	1.000e−4	1.000e−5	1.000e+0		1.000e−4	1.000e+0	1.000e+3	
50-50	1.075e−4	2.225e−4	9.688e−1	339	9.984e−4	1.265e−1	1.000e+0	474
25-75	5.250e−5	2.500e−5	1.250e+0	318	3.294e−4	1.785e+0	1.051e+0	302
10-90	2.041e−6	4.600e−4	1.000e+0	325	9.994e−4	1.722e−1	1.729e+0	420

In both the unconstrained and constrained cases, a locally optimal parameter set p^* is identified and is feasible for (4.3) and (4.4), i.e., $\varphi^{\text{SOLVED}}(p^*) = 1$ and $\varphi^{\text{QUALITY}}(p^*) \leq 0$. In the unconstrained case, the parameters generated during the training phase differ only slightly from the default ones. There is however no clear tendency. For each of the three parameters, there is a training case where it is increased, and another where it is decreased. This suggests that the default parameter recommendations are good for unconstrained problems.

In the constrained case, the DELMIN parameter is increased by a factor ranging from 3 to 10 in all three training phases. The DELTA parameter is decreased in two cases, and increased in another. The most noticeable variation is to the penalty parameter. In all three cases, it is reduced by three orders of magnitude.

In order to measure the quality of the sets of parameters given in Table 4.4, we run DFO on the corresponding cross-validation subsets. Tables 4.5 and 4.6 summarize the comparison with the default parameters. The tables report the objective function value φ^{EVAL} as well as the constraint value φ^{QUALITY} for both the training and cross-validation phases.

Table 4.5 Cross-Validation Results on Unconstrained Test Problems

	Training		Cross-validation	
	φ^{EVAL}	φ^{QUALITY}	φ^{EVAL}	φ^{QUALITY}
50-50	−5.68e−2	−5.22e−3	6.96e−4	1.16e−2
25-75	−4.95e−2	−1.89e−2	2.45e−3	1.27e−1
10-90	−1.21e−1	−6.82e−2	4.22e−3	−1.91e−2

Table 4.6 Cross-Validation Results on Constrained Test Problems

	Training		Cross-validation	
	φ^{EVAL}	φ^{QUALITY}	φ^{EVAL}	φ^{QUALITY}
50-50	−2.91e−1	−7.73e−2	−2.91e−1	−8.04e−2
25-75	−3.04e−1	−1.60e−1	−1.46e−1	−2.75e−1
10-90	−2.94e−1	−1.03e−1	−2.53e−1	−1.18e−1

The function values values φ^{EVAL} and φ^{QUALITY} are all negative on the training sets. This is not surprising, since the first function was the objective, and the second is constrained to be non-positive. In the cross-validation phase on unconstrained problems, some values are positive, but very small, and others are negative, which suggests that both sets of parameters produce a comparable average normalized number of evaluations. In the constrained case, all values are negative which suggests that the optimized parameters yield an improvement over the default values.

The quantitative data of Tables 4.5 and 4.6 are adequately complemented by the more qualitative performance profiles (Dolan and Moré, 2002). If we denote by p^0 the default parameters and by p^* the optimized parameters, the profile corresponding to $p \in \{p^0, p^*\}$ represents the step function

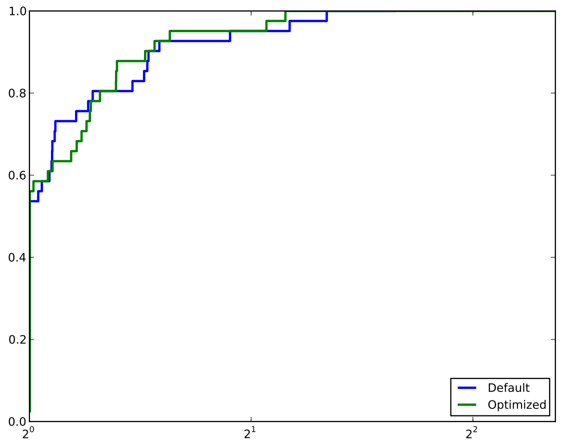
$$\alpha \mapsto \frac{1}{|\mathcal{L}_{\mathrm{C}}|} \cdot \left| \left\{ \ell \in \mathcal{L}_{\mathrm{C}} \text{ such that } \mu_\ell^{\mathrm{EVAL}}(p) \leq \alpha \min \left[\mu_\ell^{\mathrm{EVAL}}(p^0), \mu_\ell^{\mathrm{EVAL}}(p^*) \right] \right\} \right|,$$

where $\alpha \geq 1$ and $\mathcal{L}_{\mathrm{C}}$ is the relevant cross-validation subset. For $\alpha = 1$, the above value is the proportion of problems on which the method with parameters p was the best in terms of number of function evaluations. For $\alpha = 2$, we obtain the proportion of problems on which the method with parameters p was within a factor 2 of the best. For $\alpha \to \infty$, we obtain the proportion of problems solved.

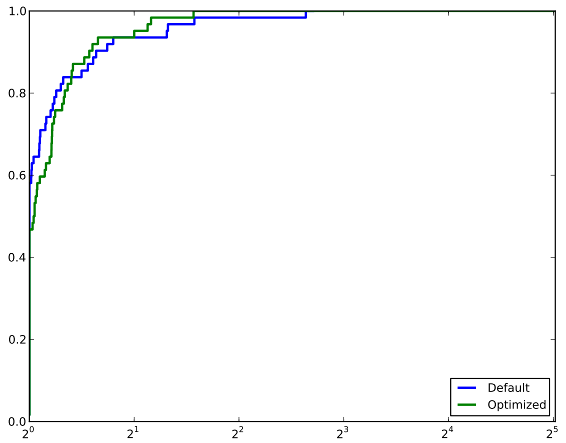
Figures 4.2 and 4.3 show performance plots for the optimized parameters obtained from the three training phases. Because the objective function of (4.3) and (4.4) average the number of evaluations over the cross-validation subset, a fourth plot is introduced for a finer look at the quality of the optimized parameters. The vertical axis refers to a proportion of the cross-validation test problems. The horizontal axis refers to the values

$$\rho_\ell^{\mathrm{EVAL}}(p^*) = \frac{\mu_\ell^{\mathrm{EVAL}}(p^*) - \mu_\ell^{\mathrm{EVAL}}(p^0)}{\mu_\ell^{\mathrm{EVAL}}(p^*) + \mu_\ell^{\mathrm{EVAL}}(p^0)}$$

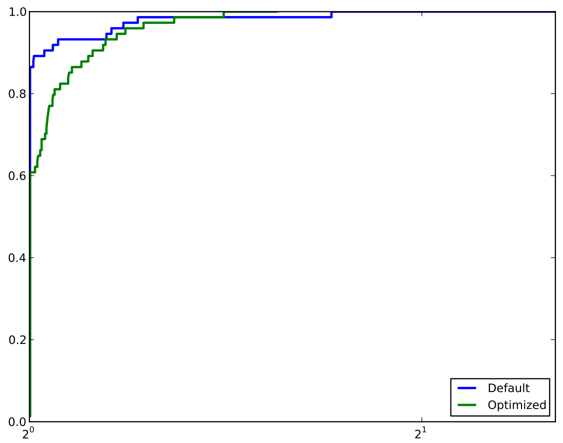
present in the sum used in the definition of the compound measure $\varphi^{\mathrm{EVAL}}(p^*)$. From the 10–90 training curve of Figure 4.2(d), we see that $\rho_\ell^{\mathrm{EVAL}}(p^*) \leq 0$ for approximately 60% of the unconstrained test problems, i.e., the optimized parameters did equally well or better than the default ones on 60% of the cross-validation problems, and this results from tuning the parameters on a training sample of 10% of the problems. Notice that this value of 60% can also be seen on the vertical axis of Figure 4.2(c). Of course, exactly which 10% of the problems appear in the training subset influences the results and as a general rule, a *representative* subset should be chosen. The 50–50 and 25–75 curves do not yield as much improvement as one might expect and suggest that there is no significant difference between the performance of the default and optimized



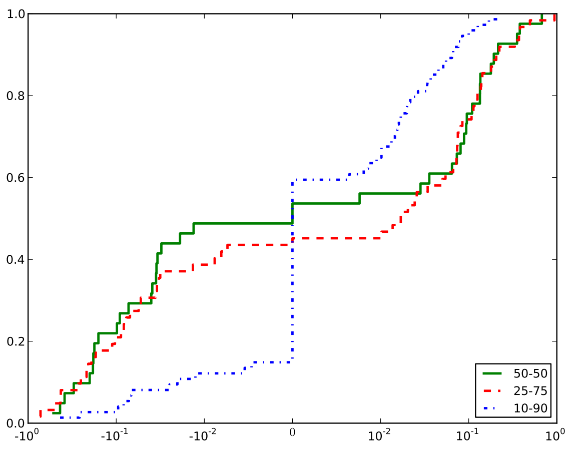
(a) Partition 50–50



(b) Partition 25–75



(c) Partition 10–90

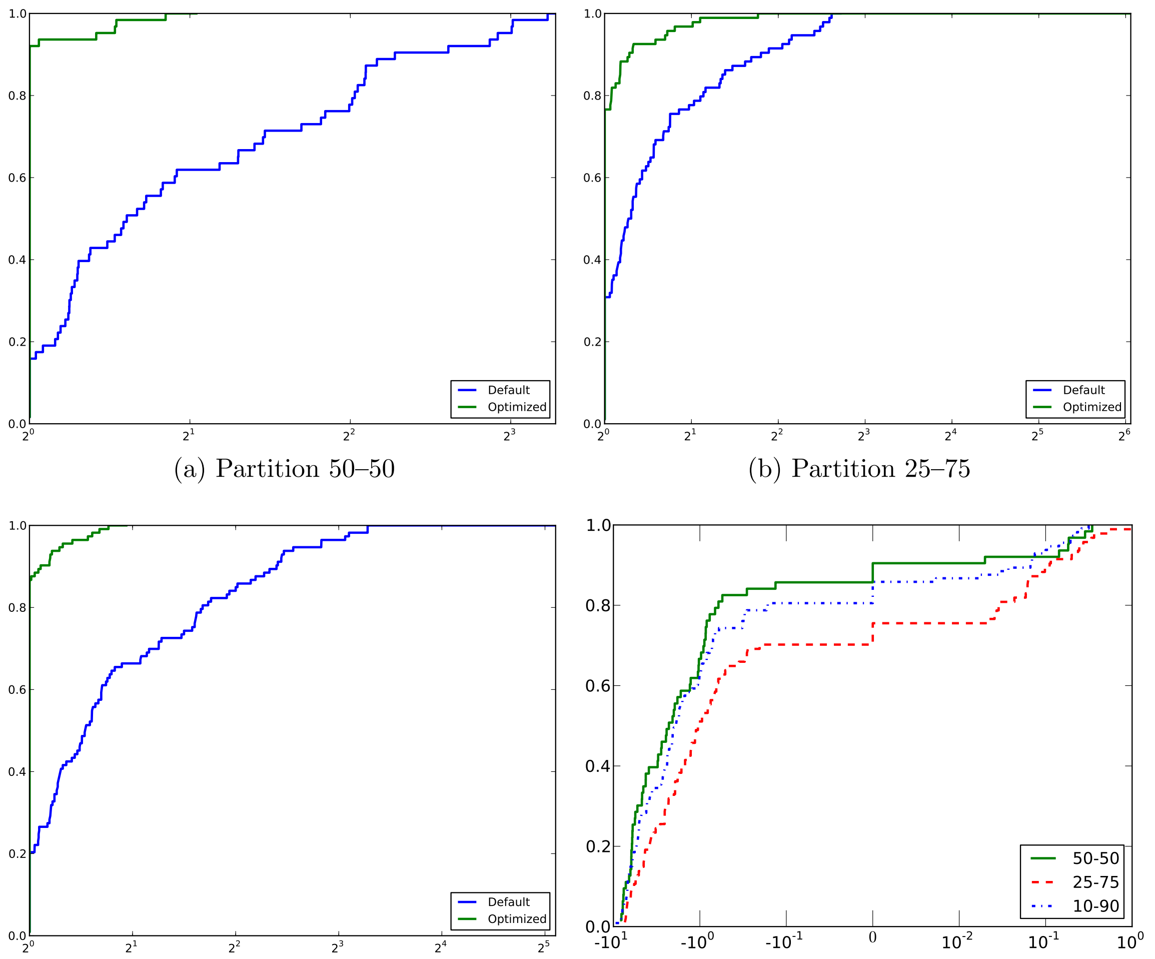


(d) Comparison of the Number of Evaluations

Figure 4.2 Profiles for (4.3) on Each Cross-Validation Set.

parameters in the unconstrained case. The leftmost part of Figure 4.2(d) indicates that the number of evaluations required by DFO with the optimized parameters is less than or equal to that with the default parameters on 60% of the problems. Conversely, the number of evaluations required with the default parameters was less than or equal to that with the optimized ones on 85% of the problems. We deduce that both variants of DFO required the same number of evaluations on 45% of the test problems, the default parameters were (strictly) preferable on 40% and that the optimized ones are (strictly) preferable on only 15% of the problems.

On the other hand, Figure 4.3 is more clear-cut and suggests a marked difference in quality between p^0 and p^*. In all three performance plots, the optimal parameters



(a) Partition 50–50 (b) Partition 25–75

(c) Partition 10–90 (d) Comparison of the number of evaluations

Figure 4.3 Profiles for (4.4) on Each Cross-Validation Set.

clearly dominate the default ones. Figure 4.3(d) confirms that for all three training scenarios, the optimized parameters (strictly) improved the number of evaluations on more than 70% of the cross-validation problems.

As this case study illustrates, black-box optimization offers a convenient, non-intrusive, mechanism for parameter tuning. In all cases, the choice of training set influences the results directly. Moreover, the precise formulation of the performance criterion is determinant and dictates how the results should be interpreted.

4.5 Discussion

The Opal system is a general framework for algorithmic parameter optimization and is an implementation of the groundwork laid out in Audet and Orban (2006). Opal effectively acts as a modeling and solution environment for parameter tuning problems. The modeling stage involves the declaration of the algorithm to be tuned and the definition of meaningful metrics used to formulate the notions of performance and of feasibility of parameters. The solution phase consists in selecting an appropriate black-box optimizer and adequate sets of input data. All such tasks are performed by way of natural Python commands. The lack of assumptions on the nature of the algorithm being tuned and on the input data provides maximum flexibility and we hope that Opal will be used in a wide range of parameter-tuning applications.

We have studied the optimization of some of the DFO parameters. Our study confirms that the default values proposed in the documentation are well-chosen in the unconstrained case. If the training set used to tune the parameters is too small or ill chosen, then the resulting parameters may not perform well on a larger or more representative test set. On the constrained optimization problems, our study identified alternate DFO parameters that lead to a important decrease in the number of evaluations on the cross-validation sets. In all cases, the direct-search solver required between about 300 and 500 black-box evaluations, which may be expensive in some applications. Current research is focusing on decreasing this number with the choice of a better initial guess than the default parameter values. A good initial guess may for instance be identified via a simplified version of (4.3) and (4.4).

At present Opal is only able to work with real parameters but the generalization to integer and categorical parameters is the subject of ongoing research. Additional ongoing improvements include the use of surrogates to guide the local search in hopes to identify promising regions or search directions quickly. An example of such a surrogate in the context of tuning the parameters of another optimization algorithm is given in Audet and Orban (2006). Finally, identifying *robust* locally-optimal parameters—i.e., for which small perturbations lead to comparable performance—remains an open question.

Chapter 5

EFFICIENT USE OF PARALLELISM IN ALGORITHMIC PARAMETER OPTIMIZATION APPLICATIONS

Charles Audet   Cong-Kien Dang,   Dominique Orban

Abstract

In the context of algorithmic parameter optimization, there is much room for efficient usage of computational resources. We consider the Opal framework in which a nonsmooth optimization problem models the parameter identification task, and is solved by a mesh adaptive direct search solver. Each evaluation of trial parameters requires the processing of a potentially large number of independent tasks. We describe and evaluate several strategies for using parallelism in this setting. Our test scenario consists in optimizing five parameters of a trust-region method for smooth unconstrained minimization.

This chapter corresponds to the publication (Audet *et al.*, 2011a)

5.1 Introduction

The Opal framework (Audet *et al.*, 2010a) for automated algorithmic parameter optimization identifies locally optimal parameter settings by formulating and solving a nonsmooth constrained optimization problem in which evaluating the objective and constraints consists in running a target algorithm on a given training set of test problems. The search for local optimality is performed by the Nomad software (Le Digabel, 2011) – an implementation of the mesh adaptive direct search family of algorithms (Mads) for nonsmooth constrained optimization (Audet and Dennis, Jr., 2006; Abramson *et al.*, 2009b; Audet and Dennis, Jr., 2009). The structure of both the optimization procedure and the objective and constraints evaluations creates opportunities for parallelism at various levels on several types of commodity hardware.

The goal of the present paper is to describe and study various strategies for using parallelism opportunities efficiently in an algorithmic parameter optimization application. Our discussion focuses on the Opal framework but most ideas developed here can be adapted to other contexts.

Audet and Orban (2006) proposed a methodology to optimize real algorithmic parameters and reported numerical experience in a serial environment on a standard trust-region algorithm (Conn *et al.*, 2000) for unconstrained optimization. A strategy involving a surrogate model used to guide the search led to an overall 25% decrease in average cpu time on a training set of 163 test problems. In such a context, evaluating the objective and constraints can be particularly costly in terms of computational effort and time. The cumulative computing resources necessary to perform the optimization can thus be very large. In this work (Audet and Orban, 2006), the reported cumulative cpu time is approximately 18 days on a sequential machine.

At each iteration, the mechanism of Mads consists in evaluating the quality of each vector of trial parameters from a finite set. This is done by running the target algorithm on a collection of test problems with each of those trial parameters in turn. For each trial parameter vector, the target algorithm returns a number of metrics which are the constituents of the parameter optimization problem. With n parameters, a set of $2n$ vectors of parameters may need to be evaluated at each iteration. For each one of these parameters, the target algorithm must work through the training set of test problems. Different vectors of parameters are non-correlated in the sense that their quality is assessed independently. Similarly, solving each test problem in the training set is a self-contained and independent task.

The present paper compares three ways of using parallelism within Opal. The first strategy consists in relying on the blackbox solver to evaluate the quality of the trial parameters in parallel. The second one exploits the structure of the optimization problem and consists in launching the target algorithm to solve the test problems concurrently on a number of different compute nodes or processors. The third strategy is a combination of the first two: several trial parameters are treated in parallel, and for each of them, the test problems are solved in parallel by the target algorithm.

The paper is divided as follows. Section 5.2 gives a high-level description of the Opal framework and Section 5.3 describes in more detail the three strategies to use parallelism. Section 6.3.6 presents numerical results on the trust-region target algorithm using the Nomad blackbox solver. Concluding remarks are presented in Section 5.5.

5.2 The Opal framework

Opal is used to optimize the performance of a given target algorithm with respect to (some of) its parameters, where *performance* is a context-dependent concept defined by the user. In order to achieve this goal, the user provides the target algorithm, a collection of representative test problems $\mathcal{L}$, specifies which parameters p will participate in the optimization and the domain P of these parameters. The user must also specify various measures of the quality of any given realization of the parameters p. These measures can be combined in arbitrary ways into *composite measures* to define the objective function and, possibly, additional constraints of the parameter optimization problem. The latter is treated as a *blackbox problem*, i.e., one in which the structure is not exploited and only the value of the objective and constraints can be computed. Schematically, a realization of the parameters is fed as input to the blackbox, which returns the value of the objective and the constraints corresponding to the input parameters.

5.2.1 Parameter optimization as a blackbox optimization problem

A blackbox solver iteratively generates a parameter value p and inputs it into the blackbox. If p lies in the domain P, the blackbox launches the target algorithm on each test problem $\ell \in \mathcal{L}$ in turn to collect *atomic measures* $\mu_\ell(p)$. The atomic measures can be viewed as the log produced by the application of the target algorithm to problem ℓ using the value p as algorithmic parameters. Typical atomic measures include the CPU time to solve the problem, the accuracy of the final solution produced by the algorithm or a measure of the amount of work required to solve the problem. A flag indicating whether p lies in P or not is returned to the blackbox solver.

Once all test problems have been processed and all atomic measures collected, the *score* $\Psi(p)$ is computed and returned to the solver. The score is composed of a series of *composite measures*, and contains an objective function value together with values indicating whether the constraints are satisfied. A typical objective function might be the sum over all test problems of the CPU-time atomic measure. Constraints might require, for example, that at least 90% of the test problems be solved to within a precision of 10^{-3}. The score is then recorded by the solver, and a new parameter value is supplied to the blackbox, initiating the next iteration. The blackbox solver terminates when appropriate optimality criteria are met.

In this paper, we are only concerned with real parameters and simple composite measures.

5.2.2 Computational cost reduction with OPAL

Parameter optimization in the OPAL framework may be a time-consuming and computationally-intensive task. Fortunately, the typical mechanism of a direct-search solver and of the nonsmooth problem guiding the optimization suggest various ways to reduce the computational effort.

Firstly, parallelism may be exploited at either the solver level, the target algorithm level, or both. Such strategies are detailed in §5.3.

Secondly, the specifics of the constraints may enable OPAL to avoid unnecessary runs entirely. OPAL distinguishes two types of constraints on the parameters. The first type are *a priori* constraints defining the domain of definition P, and are typically bounds or simple linear constraints that are easily verified independently of the list

$\mathcal{L}$ of test problems. An example of such a constraint might require the parameters to remain positive and their sum to remain below 1. Some of these constraints can be supplied directly to the blackbox solver; most solvers perform better when bounds on the variables are supplied, as it allows them to scale the variables and functions appropriately. *A priori* constraints are checked prior to launching the target algorithm on the test problem collection. If they are not satisfied, i.e., if $p \notin P$, the entire scoring process will be bypassed and a flag indicating infeasibility will be returned to the solver within the score. This strategy, known as the *extreme barrier* (Audet and Dennis, Jr., 2003), significantly reduces the computational effort.

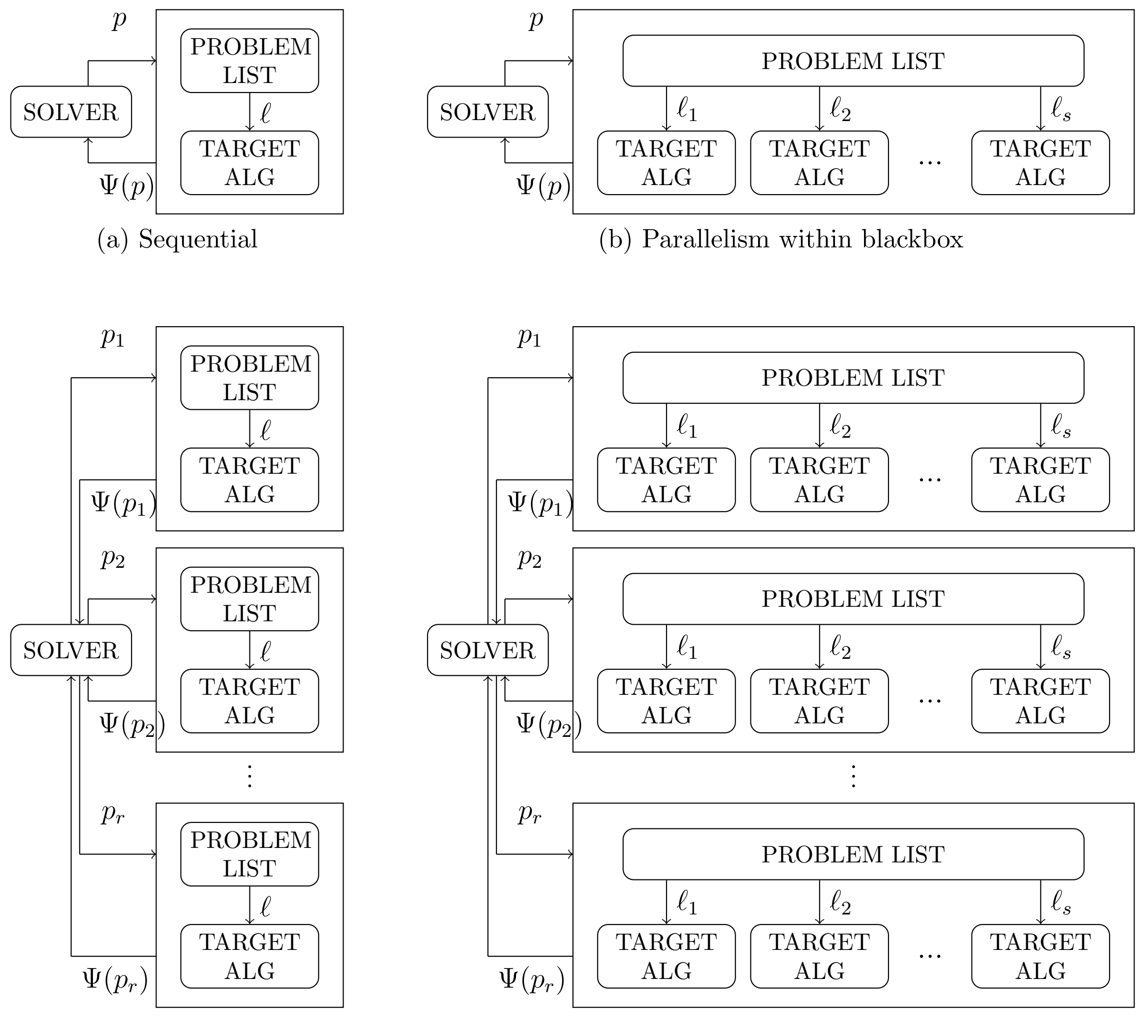
Constraints of the second type, said to be *a posteriori*, are modeled as $\varphi(p) \in M$, where $\varphi(p)$ is a vector of composite measures. They represent restrictions that can be measured only after some or all atomic measures are computed. The constraint stating that 90% of the test problems need to be solved with a precision of 10^{-3} is an example. *A posteriori* constraints can either be handled by the extreme or the progressive barrier (Audet and Dennis, Jr., 2009).

5.3 Parallelism in algorithmic parameter optimization

A sequential way to solve a parameter optimization problem is illustrated in Figure 5.1a. The solver sends parameter values p to the blackbox, which assigns a score $\Psi(p)$ by launching the target algorithm on each problem $\ell \in \mathcal{L}$ and analyzing the atomic measures $\mu_\ell(p)$. The next subsections present three strategies of using parallelism to solve this optimization problem.

5.3.1 The blackbox solver handles the parallelism

Some direct search optimization algorithms are designed to handle parallelism by generating a list of trial parameters to be assessed concurrently. Each process is given specific values for the parameters, and launches the target algorithm on every test problems from the collection $\mathcal{L}$. The direct search solver deals with the synchronization issues, as it is most likely that the CPU time will differ from one blackbox evaluation to another. Figure 5.1c illustrates this type of parallelism. When a total of r processors are available, the solver can send up to r evaluations in parallel



(a) Sequential

(b) Parallelism within blackbox

(c) Parallel solver

(d) Mixed parallelism

Figure 5.1 High level representation of the sequential and parallel strategies in OPAL

with parameters $p_1, \ldots, p_r$, and wait for the scores $\Psi(p_1), \ldots, \Psi(p_r)$. In the numerical experiments of §6.3.6, r is set to the number of available processors.

Our numerical experiments are performed using NOMAD (Le Digabel, 2011) as the blackbox optimization solver, which is the default solver in the OPAL framework. Other implementations of parallel direct search solvers include APPSPACK (Griffin *et al.*, 2008; Gray and Kolda, 2006) and IFFCO (Gilmore *et al.*, 1999). NOMAD is an implementation of the mesh adaptive direct search (MADS) framework (Audet and

Dennis, Jr., 2006) designed for blackbox optimization, and is supported by a rigorous hierarchical convergence analysis based on Clarke's nonsmooth calculus (Clarke, 1983). It is designed to exploit parallelism synchronously or asynchronously (Audet *et al.*, 2008b; Le Digabel, 2009) and handles general, hidden and non-relaxable constraints (Audet and Dennis, Jr., 2009; Choi and Kelley, 2000; Conn *et al.*, 2009b) by the extreme barrier, the progressive barrier, the filter or combinations of these strategies. NOMAD has been used successfully on a wide range of test problems (Audet *et al.*, 2010b; Le Digabel, 2011) and is freely available (Abramson *et al.*, 2004) under the LGPL license.

MADS is an iterative algorithm for constrained optimization. At each iteration it generates a finite set of trial points in the solution space, and sends them to the blackbox for evaluation. MADS then studies the scores associated to these trial points in order to determine the set of points to be used in the next iteration. MADS does not rely on any sufficient decrease condition and, under reasonable assumptions, produces a limit point satisfying necessary optimality conditions that depend on the local smoothness of the objective function and on local properties of the feasible region.

NOMAD is able to evaluate the trial points scores synchronously or asynchronously. The synchronous version evaluates the points in parallel and waits for all evaluations to be completed before processing a new batch. The advantage of this strategy is that it performs identically to—but more rapidly than—a sequential run. However, it does not exploit the available resources efficiently since some processors may remain idle for extended periods of time. The asynchronous version is a simplified version of the asynchronous parallel pattern search algorithm APPSPACK (Gray and Kolda, 2006) and allows to terminate an iteration as soon as a new success is recorded. Evaluations still in progress are not terminated and if one of them later results in an improvement over the current best point, the algorithm will backtrack and consider this point as the new incumbent. The numerical experiments of §6.3.6 use the asynchronous strategy.

5.3.2 Parallelism within the blackbox

A different way to exploit parallelism is to use a sequential blackbox solver, but to take advantage of the structure of the blackbox itself by having the target algorithm process the test problems in parallel. This situation is depicted in Figure 5.1b. The solver evaluates the score of one trial point at a time, but the blackbox launches the target algorithm on a problem $\ell \in \mathcal{L}$ as soon as a compute node is available. In the numerical results, we allocate a fixed number s of processors to the blackbox evaluations. This strategy requires synchronization, as all test problems from the collection $\mathcal{L}$ need to be solved before returning the score $\Psi(p)$ to the solver.

In Opal, different paradigms may be used to parallelize the blackbox. The first one is the Message Passing Interface (MPI) (Gropp *et al.*, 1994) via the Python module `mpi4py`[1]. Although there is typically no communication between processes at the blackbox level, this paradigm is useful because of its ubiquity. The second one is the Load Sharing Facility (LSF) job scheduler[2], better suited to distributed-memory concurrent computation environments such as blade centers and networks with slow interconnections. Finally, the third paradigm is the Symmetric Multi Processing (SMP) architecture which is suitable for multicore environments and shared-memory platforms.

5.3.3 Mixed parallelism

A third way to handle parallelism is a combination of the previous two in which the scores of trial points are evaluated in parallel over a certain number r of processors and each blackbox evaluation also occurs in parallel over s processors. As depicted in Figure 5.1d, the solver launches up to r blackboxes in parallel, each with a different parameter value $p_1, \ldots, p_r$. Each blackbox uses a subset of processors to solve in parallel the problems from the list $\mathcal{L}$.

In the numerical experiments, we set r to be equal to the number of parameters n over which the optimization occurs. In the context of optimizing the trust-region parameters, this number is $r = n = 5$. The blackboxes compete for all remaining processors. In our case, because Nomad itself parallelizes the processing of trial points using MPI, the blackbox is parallelized via SMP.

1. `http://code.google.com/p/mpi4py`
2. `http://www.platform.com/workload-management/high-performance-computing`

5.4 Numerical results

5.4.1 The target algorithm: A trust-region method for unconstrained optimization

Trust-region methods provide a mechanism for ensuring global convergence in the unconstrained minimization of an objective function f. Our target algorithm only concerns the minimization of a twice-continuously differentiable objective but trust-region methods are sufficiently flexible to be adapted to other cases (Conn *et al.*, 2000). At the k-th iteration of a trust-region method, a quadratic model m_k of the objective is approximately minimized over a ball of radius Δ_k centered at the current iterate x_k—the *trust region*. Based on whether the decrease δm_k in the model accurately reflects the decrease δf_k actually achieved in the objective, the size of the trust region is adjusted and the trial step is accepted or rejected. For the purposes of this paper, we need only be concerned with the management of the trust region and the parameter that it involves. The step proposed by the minimization of the model is accepted if $\delta f_k > \eta_1 \delta m_k$ for some fixed $\eta_1 \in (0,1)$ and rejected otherwise. The update of the trust-region depends on the adequacy between the model and the objective. If $\delta f_k > \eta_2 \delta m_k$ for some fixed $\eta_2 \in (\eta_1, 1)$ the model is considered very accurate and Δ_{k+1} is set to $\gamma_3 \Delta_k$ for a given $\gamma_3 > 1$. If the step is rejected, the adequacy is poor and we set $\Delta_{k+1} = \gamma_1 \Delta_k$ for a given $\gamma_1 \in (0,1)$. In the intermediate situation, $\Delta_{k+1} = \gamma_2 \Delta_k$ for some $\gamma_2 \in (\gamma_1, 1]$. Shrinking the trust region when a step is rejected guarantees that a step will eventually be accepted once Δ_k has become sufficiently small and progress will be achieved because m_k is required to coincide with f up to first order at x_k.

The parameter η_1 determines how demanding we are in the adequacy between the model and objective before accepting a step. Consider the limiting case $\eta_1 = 0$ where we are satisfied with a simple decrease in f—a strategy which does not yield global convergence. The parameter η_2 determines how eager we are to increase the size of the trust region and promote larger steps with the aim of making faster progress. However, using too large a trust region is risking a potentially long sequence of rejected steps if x_k lies in a region where f cannot be accurately modeled by a quadratic over a wide domain. The meaning of the parameters γ_1, γ_2 and γ_3 is easier to grasp; they simply represent the factor by which we decrease or increase the size of the domain in which we believe that a quadratic model of f can be trusted. To summarize, the five real parameters of a trust-region algorithm are

$$0 < \eta_1 < \eta_2 < 1, \quad \text{and} \quad 0 < \gamma_1 < \gamma_2 \leq 1 < \gamma_3.$$

These conditions represent *a priori* constraints on our parameter optimization problem.

Our Fortran 95 implementation of the trust-region method uses a second-order Taylor expansion of f about x_k as model m_k and computes an approximate minimizer of m_k within $\{x_k + s \mid \|s\| \leq \Delta_k\}$ using the generalized Lanczos method for trust-region subproblems GLTR (Gould *et al.*, 1999) as implemented in the GALAHAD library (Gould *et al.*, 2003b). In the absence of preconditioning, the main computational cost of GLTR is matrix-vector products $H_k v$ where H_k is the Hessian matrix of f at x_k and v is some vector of appropriate size. In theory, it is possible to require up to n of those products to compute a single trust-region step, where n is the number of variables of f. The total number of such products is a measure of the overall work performed to solve a given problem. The algorithm stops with a success if it identifies an iterate x_k such that

$$\|\nabla f(x_k)\|_2 \leq \max(10^{-5}, 10^{-6} \|\nabla f(x_0)\|_2)$$

or declares failure when it reaches the limit of 500 iterations.

5.4.2 The parameter optimization problem

An important feature of optimization methods for nonlinear problems is their ability to reliably solve a large class of problems within a reasonable amount of *work*, where *work* may be a function of the number of objective evaluations—the premise being that the objective function may be costly to evaluate—, of the number of matrix-vector products—which determines the effort expended on solving subproblems iteratively—, or of other related quantities. This prompts us to consider the atomic measure $\mu_\ell^{\mathrm{H}}(p)$ defined as the number of matrix-vector products with the Hessian of the objective necessary to solve problem ℓ using parameters p if the solver was indeed able to identify an optimal solution to problem ℓ to within the prescribed tolerance. In case the problem failed to be solved to optimality, the maximum allowed number of products, iterations or evaluations has likely been reached and this problem will contribute adversely to the minimization of the objective of the parameter optimization problem. We choose however to impose that all problems in $\mathcal{L}$ be solved to optimality as an *a posteriori* constraint via the atomic measure $\mu_\ell^{\mathrm{E}}(p)$, which is set to zero if problem ℓ was solved to optimality in 500 iterations or less and to 1 otherwise.

Our parameter optimization problem is thus to

$$\underset{p \in P}{\text{minimize}} \sum_{\ell \in \mathcal{L}} \mu_\ell^{\mathrm{H}}(p) \quad \text{subject to} \sum_{\ell \in \mathcal{L}} \mu_\ell^{\mathrm{E}}(p) \leq 0. \tag{5.1}$$

Table 5.1 records our test problems along with their number of variables. The test problems are a subset of those used in Audet and Orban (2006) chosen because of their widely different typical solve times. We hope those differences will help contrast the benefits of each type of parallelization. All test problems are available as part of the CUTEr collection (Gould *et al.*, 2003a). The table also shows the atomic measures for two sets of parameters: p_0, the standard values often found in the literature—see for example, Conn *et al.* (2000)—and p_{CPU}, an alternate set of values identified as a good initial guess in the work of Audet and Orban (2006) by minimizing the total CPU time of the trust-region method for solving a collection of 54 *easy* test problems with dimension $2 \leq n \leq 500$ and small run times. The precise values of p_0 and p_{CPU} are given in the next section.

5.4.3 Comparative study of parallelism within Opal

The parameter optimization problem (5.1) is in general a highly nonconvex problem with many local minima. Our experimental tests solve (5.1) using the parallel strategies detailed in Section 5.3 from the two feasible initial parameter values p_0 and p_{CPU}. The strategies are labeled as follows: Solver denotes the parallel blackbox solver with sequential blackbox evaluations, Mixed denotes the parallel blackbox solver with parallel blackbox evaluations, Blackbox denotes the sequential blackbox solver with parallel blackbox evaluations, and Sequential denotes the sequential blackbox solver with sequential blackbox evaluations. Both p_0 and p_{CPU} are given in Tables 5.2 and 5.3. Note that the present tests differ from those in Audet and Orban (2006), where the objective function depends on the CPU time, in that the objective

Table 5.1 Test set.

Name	n	$\mu_\ell^H(p_0)$	$\mu_\ell^E(p_0)$	$\mu_\ell^H(p_{cpu})$	$\mu_\ell^E(p_{cpu})$
BDQRTIC	5000	91	0	120	0
BROYDN7D	5000	11135	0	7503	0
BRYBND	5000	79	0	85	0
CRAGGLVY	5000	251	0	135	0
CURLY10	10000	147345	0	127478	0
DIXON3DQ	10000	51264	0	44187	0
EIGENALS	2550	5509	0	5408	0
FMINSRF2	5625	3953	0	3874	0
FMINSURF	5625	3104	0	2999	0
GENROSE	500	15387	0	14530	0
HIELOW	3	31	0	24	0
MANCINO	100	60	0	20	0
NCB20	5010	2392	0	2036	0
NCB20B	5000	5953	0	4390	0
NONDQUAR	5000	8211	0	5471	0
POWER	10000	1338	0	1374	0
SENSORS	100	169	0	138	0
SINQUAD	5000	35	0	40	0
TESTQUAD	5000	2539	0	2243	0
TRIDIA	5000	1537	0	1639	0
WOODS	4000	283	0	271	0

function of (5.1) is deterministic.

All tests are performed on a 64-bit computer with two 6-core Core i7 processors and 12Gb of RAM. A single processor is used for the sequential run. All 12 cores are used by NOMAD-MPI in the SOLVER strategy. The solver is asynchronous, and no processor remains idle as NOMAD proposes new parameters as soon as a blackbox evaluation terminates. The BLACKBOX case imposes a form of synchronization. The problems from the test collection are solved by the trust-region algorithm concurrently, and the blackbox must wait for the last one to be processed before returning the atomic measures to the solver. Consequently, if the runtimes of the individual problems differ sufficiently, there are situations where most processors are idle. Finally, in the MIXED case, we assign five processors to NOMAD. The test problem collection is solved in parallel by the trust region algorithm using all the 12 available processors. This is possible because on a given processor, the NOMAD process does not compete with the blackbox process—it merely waits for the result of the evaluation.

For comparison purposes, the termination criteria for each run is fixed to a precision of three decimals in the parameter values.

Table 5.2 summarizes the results of the four runs using the standard trust-region parameter values p_0 as initial guess. The value of p_0 appears in the last line of the table, together with the value of the objective function of problem (5.1) at p_0. The four other lines of the table list the final values of the trust-region parameters at the end of the optimization process, together with their corresponding objective function value, and wall-clock time required by the run. Table 5.3 displays similar results, generated by taking the solution p_{CPU} as a starting guess for the optimization.

Table 5.2 Solutions produced from the initial point p_0

Strategy	η_1	η_2	γ_1	γ_2	γ_3	Obj	Time
SOLVER	0.046875	0.9640625	0.346875	1	8.03125	158634	11h 40m
MIXED	0.04609375	0.751953125	0.203125	1	3.01171875	191516	8h 23m
BLACKBOX	0.05	0.65	0.5	1	3	214309	16h 05m
SEQUENTIAL	0.05	0.65	0.5	1	3	214309	> 24h
p_0	0.25	0.75	0.5	1	2	260666	-

Table 5.3 Solutions produced from the initial point p_{CPU}

Strategy	η_1	η_2	γ_1	γ_2	γ_3	Obj	Time
SOLVER	0.121625	0.95207031	0.38996094	1	8.2792969	183 145	10h 07m
MIXED	0.121625	0.90207031	0.13996094	1	6.2792969	166 854	7h 35m
BLACKBOX	0.01091333	0.95211303	0.2897259546	0.99987182	8.2780151	165 792	12h 19m
SEQUENTIAL	0.01091333	0.95211303	0.2897259546	0.99987182	8.2780151	165 792	> 24h
p_{CPU}	0.221625	0.90207031	0.38996094	1	2.2792969	223 965	-

Inspection of these two tables and the logs of the runs leads to the following observations. In both tables, the SEQUENTIAL and the BLACKBOX strategies perform exactly the same steps. This is because NOMAD with default parameters is a deterministic method and the blackbox returns the same values regardless of whether it is evaluated in parallel or sequentially. This behavior is apparent on Figures 5.2 and 5.3, where the objective function values are plotted for each of the four strategies versus the wall clock time. From the starting point p_0, these two strategies were not able to reduce the objective function value as much as with the other ones. Further analysis of the logs reveals that the reason is that most of the computational effort was deployed – and wasted– around an infeasible solution with a very low objective function value. This could be corrected by adjusting the constraints-handling parameters in NOMAD, but we preferred to focus on parallelism rather than to tune NOMAD itself in the present study.

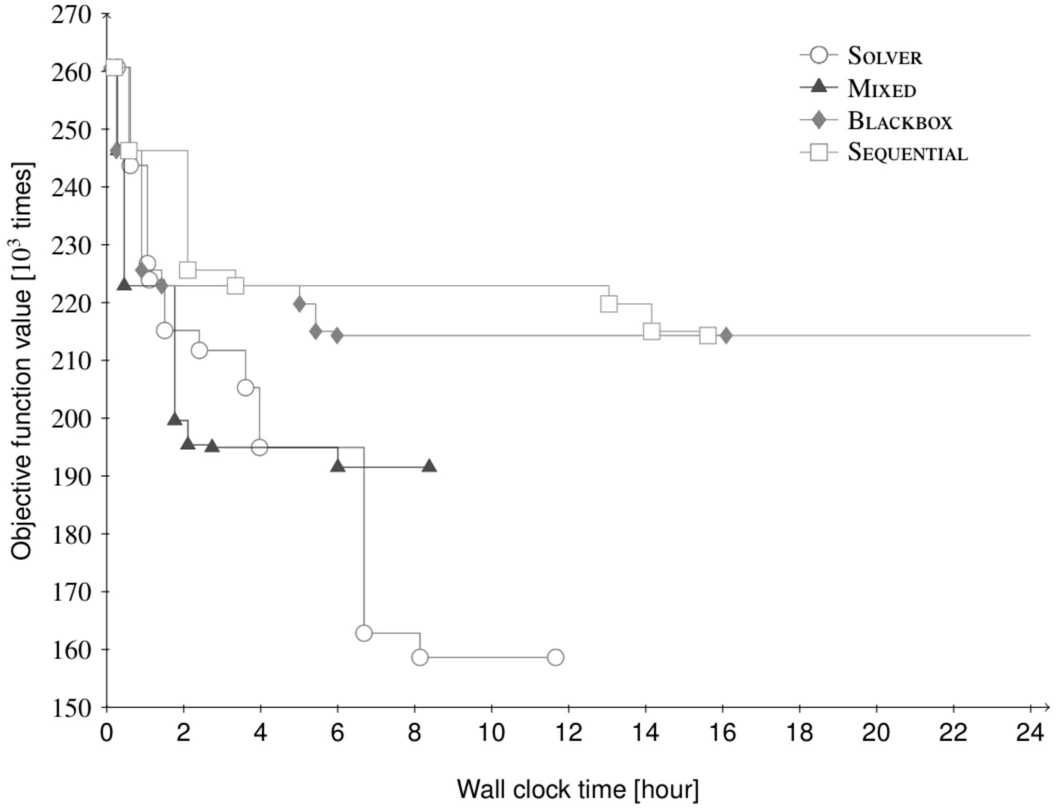


Figure 5.2 Objective function value versus wall clock time from the initial point p_0.

The parallel runs from both starting points all lead to different final solutions, having objective function values that range from 158 634 to 191 516. This suggests that (5.1) possesses several local optimal solutions, and that the starting point location has an important influence on where the algorithm converges to. Some of NOMAD's options (that are not activated in the default settings) such as the Variable Neighborhood Search (Audet *et al.*, 2008a) could be used to attempt to escape from local solutions. However, we do not consider such options in the present research.

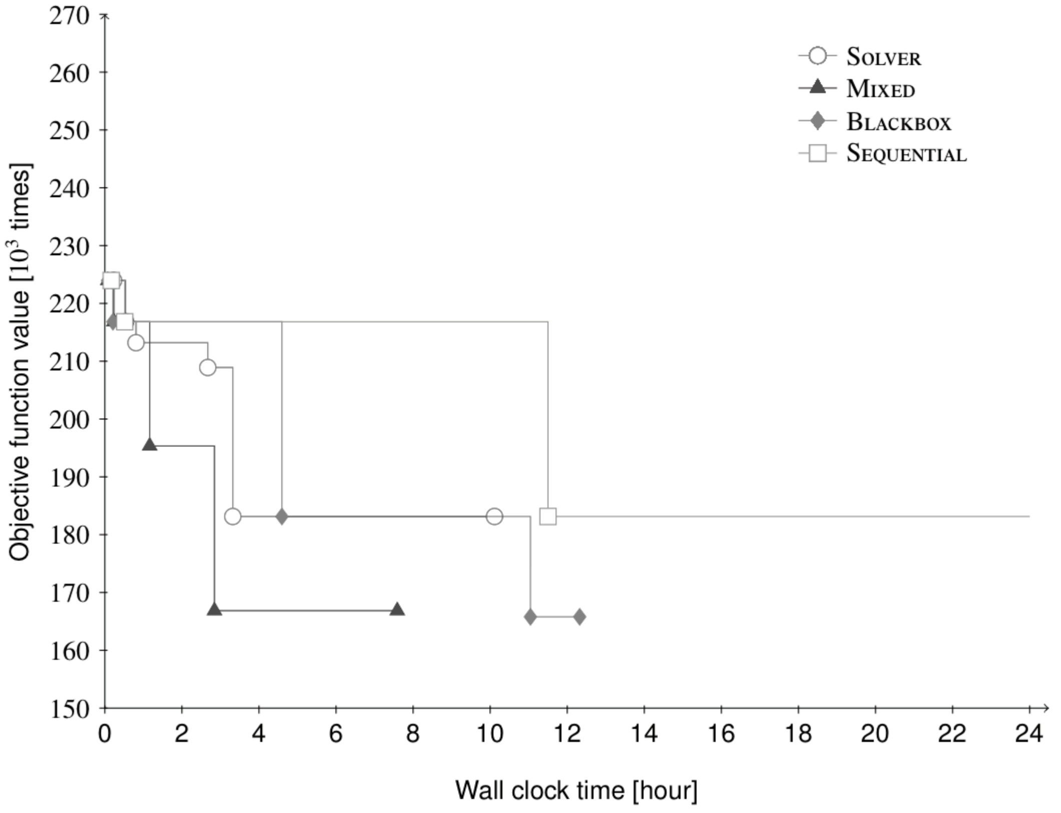


Figure 5.3 Objective function value versus wall clock time from the initial point p_{CPU}.

If the computing resources were limited to a few hours, both figures suggest that the Mixed strategy is the one for which the objective function value decreases the most rapidly. It is also able to find the second best value from both initial guesses. It thus appears to be the most promising of all.

The final parameter values are consistent with those of Audet and Orban (2006). This suggests that the results obtained in Audet and Orban (2006) are not particular to the test problems used there; they also apply to other test problems. It is interesting however to note that the direct solver left γ_2 at its initial value, even though γ_2 is an additional parameter in comparison with Audet and Orban (2006). This is satisfactory in our view as it confirms that the intuitive choice $\gamma_2 = 1$ made in most implementations of the trust-region method is a sound one—see, e.g., Conn *et al.* (2000).

From both starting points, the largest speedup due to paralellism occurs with the MIXED strategy, followed by the SOLVER strategy, i.e., when NOMAD handles all or part of the parallelism. This is certainly due to the fact that NOMAD is asynchronous while the BLACKBOX strategy alone requires synchronization at the end of each blackbox evaluation. In the present context, the test problem CURLY10 is expensive in terms of number of Hessian-vector products—and therefore in terms of CPU time. As we see from Table 5.1, this problem contributes more than twice the number of Hessian-vector products of all the other problems combined.

The starting point p_{CPU} was obtained in Audet and Orban (2006) in a parameter optimization study to minimizes the overall CPU time required by the trust region algorithm to solve a large collection of easy test problems. There is a strong correlation between the CPU time and the objective function considered in the present paper. This translates in a significant time reduction in the last column of Tables 5.2 and 5.3.

5.5 Outlook

The OPAL framework shows great promise as a general tool for the automatic tuning of algorithmic parameters and allows for parallelism at several levels. As such we believe its importance will continue to grow as an aid in the design of efficient numerical methods.

In addition to the synchronous and asynchronous strategies discussed in the present work, NOMAD can exploit parallelism in two other ways. PSD-MADS (Audet *et al.*, 2008b) explores in parallel various subspaces of variables while COOP-MADS (Le Digabel, 2011) launches concurrent executions of MADS with different seeds. These two implementations are not directly integrated inside the NOMAD package, but are separated programs using NOMAD as a library. In future work, we plan to have OPAL exploit these alternate parallel strategies.

There is more room for parallelism in algorithmic parameter optimization contexts. It is typical to use a *surrogate model* to guide and accelerate the search in applications. The freedom allowed to devise this surrogate model can be used advantageously to benefit from a parallel computing environment. For instance, during the so-called *search step*, the surrogate evaluates the quality of parameters which may be far from the current best choice and this task may be performed in parallel. However, constructing the surrogate can obviously be done in parallel as well, e.g., if constructing an interpolatory or least-squares model, or if the structure of the surrogate is similar to that of the blackbox problem. Finally, the target algorithm itself may be able to run in parallel.

With a heterogeneous set of test problems such as those of Table 5.1 where one problem is twice as costly as all the others combined, proper load balancing between the processors is required. In the future, OPAL should be able to gather test problems into *pools* of roughly equivalent cost and reallocate those pools dynamically.

On-the-fly job interruption is another worthwhile mechanism in the context of algorithmic parameter optimization. As the blackbox is being evaluated, the solver might infer that the point being evaluated will not result in an improvement based on the partial information accumulated so far. Resources would be best used by interrupting this evaluation and moving on to the next candidate.

In the presence of a fixed architecture and a fixed set of resources, the question of how to best exploit these resources given the various components that are able to run in parallel is open.

Chapter 6

OPTIMIZATION OF ALGORITHMS WITH OPAL

Charles Audet Cong-Kien Dang Dominique Orban

Abstract

OPAL is a general-purpose system for modeling and solving algorithm optimization problems. OPAL takes an algorithm as input, and as output it suggests parameter values that maximize some user-defined performance measure. In order to achieve this, the user provides a Python script describing how to launch the target algorithm, and defining the performance measure. OPAL then models this question as a blackbox optimization problem which is then solved by a state-of-the-art direct search solver. OPAL can handle a wide variety of parameter types, exploit multiple processors in parallel at different levels and take advantage of a user-defined surrogate for blackbox optimization problem.

6.1 Introduction

Parameter tuning has widespread applications because it addresses a widespread problem: *improving performance*. Evidently, this is by no means a new problem and it has been addressed in the past by way of various procedures that we briefly review

This chapter corresponds to a technical report (Audet *et al.*, 2011b) and has been submitted for publication.

below. In this paper, we describe a flexible practical environment in which to express parameter tuning problems and solve them using nondifferentiable optimization tools. Our environment, named OPAL[1], is independent of the application area and runs on most platforms supporting the Python language and possessing a C++ compiler. OPAL is non-intrusive in the sense that it treats the target application as a blackbox and does not require access to its source code or any knowledge about its inner mechanisms. All that is needed is a means to request a run for a given set of parameters. At the heart of OPAL is a derivative-free optimization procedure to perform the hard work. Surprisingly, the literature reveals that other so-called *autotuning* frameworks use heuristics, unsophisticated algorithms such as coordinate search or the method of Nelder and Mead, or even random search to perform the optimization—see, e.g., Seymour *et al.* (2008); Whaley *et al.* (2001); Bilmes *et al.* (1998); Vuduc *et al.* (2005); Balaprakash *et al.* (2011a). By contrast, OPAL uses a solid optimization method supported by a strong convergence theory, yielding solutions that are local minimizers in a meaningful sense.

Audet and Orban (2006) study the four standard parameters of a trust region algorithm (Gould *et al.*, 2005) for unconstrained nonlinear optimization. In particular, they study the question of minimizing the overall CPU time required to solve 55 test problems of moderate size from the CUTEr (Gould *et al.*, 2003a) collection. The question is reformulated as a blackbox optimization problem, with four variables representing the four parameters, subject to bounds, and a strict linear inequality constraint. An implementation of the mesh adaptive direct search (MADS) (Audet and Dennis, Jr., 2006) family of blackbox optimization methods is used to solve the problem. In addition, a surrogate function obtained by solving a subset of the trust region test problems is used to guide the MADS algorithm. The numerical experiments lead to a 25% computing time reduction compared to the default parameters.

Audet *et al.* (2010a) extend the framework to make it more configurable, and use it to tune parameters of the DFO algorithm (Conn *et al.*, 2009b) on collections of unconstrained and constrained test problems. They introduce the first version of the OPAL package. Finally, Audet *et al.* (2011a) illustrate usage of parallelism at various levels within the OPAL framework and illustrate its impact on performance of the algorithm optimization process.

The present paper presents extensions to the OPAL framework, discusses its imple-

1. **OP**timization of **AL**gorithms

mentation and showcases usage on a few example applications. A goal of the present work is also to illustrate how Opal interacts with other tools that may be useful in parameter optimization applications. The rest of this paper is divided as follows. §4.2 describes a blackbox formulation of parameter-optimization problems. §5.2 describes the Opal package, and illustrates its usage on well-known parameter optimization problems. We conclude and look ahead in §6.4.

6.2 Optimization of Algorithmic Parameters

In this section, we formalize the key aspects of the parameter-tuning problem in a way that enables us to treat it as a blackbox optimization problem. We then explain how direct-search methods go about solving such blackbox problems. The precise construction of the blackbox is detailed in §6.2.2. A description of direct-search methods along with our method of choice are given in §6.2.3.

Throughout this paper we refer to the particular code or algorithm whose performance is to be optimized, or *tuned*, as the *target algorithm*.

6.2.1 Algorithmic Parameters

The target algorithm typically depends on a number of *parameters*. The defining characteristic of algorithmic parameters is that, in theory, the target algorithm will execute correctly when given valid input data regardless of the value of the parameters so long as those values fall into a preset range guaranteeing theoretical correctness or convergence. The performance may be affected by the precise parameter values but the correctness of the output should not. In practice, the situation is often more subtle as certain valid parameter values may cause the target algorithm to stall or to raise numerical exceptions when given certain input data. For instance, a compiler still produces a valid executable regardless of the level of loop unrolling that it is instructed to perform. The resulting executable typically takes more time to be produced when more loop unrolling, or more sophisticated optimization, is requested. However, an implementation of the Cholesky factorization may declare failure when it encounters a pivot smaller than a certain positive threshold. Regardless of the value of this threshold, it may be possible to adjust the elements of a perfectly valid input matrix so that by cancellation or other finite-precision effects, a small pivot is produced. Because such behavior is possible, it becomes important to select sets of algorithmic parameters in a way that maximizes the performance of the target algorithm, in a sense defined by the user. We may want, for example, to select the appropriate preconditioner so as to minimize the number of iterations required by a Krylov method to solve a large system of linear equations, or adjust the memory of a limited-memory quasi-Newton method so as to minimize a combination of the CPU time and the computer memory used to solve a set of optimization problems.

It is important to stress that our framework does not assume correctness of the target algorithm, or even that it execute at all. Failures are handled in a very natural manner thanks to the nondifferentiable optimization framework.

Algorithmic parameters come in different kinds, or types, and their kind influences how the search space is explored. Perhaps the simplest and most common kind is the *real* parameter, representing a finite real number which can assume any value in a given subset of $\mathbb{R}$. Examples of such parameters include the step acceptance threshold in a trust-region method (Gould *et al.*, 2005; Audet and Orban, 2006), the initial value of a penalty parameter, a particular entry in an input matrix, etc. Other parameters may be *integer*, i.e., assume one of a number of allowed values in $\mathbb{Z}$. Such parameters include the number of levels of loop unrolling in a compiler, the number of search directions in a taboo search, the blocking factor in a matrix decomposition method for specialized architectures, and the number of points to retain in a geometry-based derivative-free method for nonlinear optimization. *Binary* parameters typically represent on/off states and, for this reason, do not fit in the integer category. Such parameters can be used to model whether a preconditioner should be used or not in a numerical method for differential equations, whether steplengths longer than unity should be attempted in a Newton-type method for nonlinear equations, and so on. Finally, other parameters may be *categorical*, i.e., assume one of a number of discrete values on which no particular order is naturally imposed. Examples of such parameters include the type of model to be used during a step computation in a trust-region method (e.g., a linear or a quadratic model), the preconditioner to be used in an iterative linear system solve (e.g., a diagonal preconditioner or an SSOR preconditioner), the insulation material (Kokkolaras *et al.*, 2001) to be used in the construction of a heat shield (e.g., material A, B or C), and so forth. Though binary parameters may be considered as special cases of categorical parameters, they are typically modeled differently because of their simplicity. In particular, the only neighbor of an on/off parameter at a particular value (say, *on*) is its complement (*off*). The situation may be substantially more complicated for general categorical parameters.

6.2.2 A Blackbox to Evaluate the Performance of Given Parameters

Let us denote the vector of parameters of the target algorithm by p. The *performance* of the target algorithm is typically measured on the basis of a number of specific metrics reported by the target algorithm after it has been run on valid input data. Specific metrics pertain directly to the target algorithm and may consist of the number of iterations required by a nonlinear equation solver, the bandwidth or throughput in a networking application, the number of objective gradient evaluations in an optimization solver, and so forth. Performance may also depend on external factors, such as the CPU time required for the run, the amount of computer memory used or disk input/output performed, or the speedup compared to a benchmark in a parallel computing setting. Specific metrics are typically observable when running the target algorithm or when scanning a log file, while external factors must be observed by the algorithm optimization tool. Both will be referred to as *atomic measures* in what follows, and the notation $\mu_i(p)$ will often be used to denote one of them. Performance, however, does not usually reduce to an atomic measure, but is normally expressed as a function of atomic measures. We will call such a function a *composite measure* and denote it $\psi(p)$ or $\varphi(p)$. Composite measures can be as simple as the average or the largest of a set of atomic measures, or might be more technical, e.g., the proportion of problems solved to within a prescribed tolerance. Most of the time, atomic and composite measures may only be evaluated after running the target algorithm on the input data and the parameter values of interest. It is important to stress at this point that they depend on the input data. Technically, their notation should reflect this but we omit the explicit dependency in the notation for clarity.

The parameter optimization problem is formulated as the optimization—by default, we use the minimization formulation—of an objective function $\psi(p)$ subject to constraints. The generic formulation of the blackbox optimization problem is

$$\begin{array}{ll} \underset{p}{\text{minimize}} & \psi(p) \\ \text{subject to} & p \in \mathbf{P} \\ & \varphi(p) \in \mathbf{M}. \end{array} \tag{6.1}$$

The set $\mathbf{P}$ represents the domain of the parameters, as described in the target algorithm specifications. Whether or not $p \in \mathbf{P}$ can be verified without launching the target algorithm. The set $\mathbf{M}$ constrains the values of composite measures. Opal allows the user to use virtually any composite measure to define an objective or a constraint.

A typical use of (4.2) to optimize algorithmic parameters consists in training the target algorithm on a list of representative sets of input data, e.g., a list of representative test problems. The hope is then that, if the representative set was well chosen, the target algorithm will also perform well on new input data. This need not be the only use case for (4.2). In the optimization of the blocking factor for dense matrix multiplication, the input matrix itself does not matter; only its size and the knowledge that it is dense.

6.2.3 Blackbox Optimization by Direct Search

Opal allows the user to select a solver tailored to the parameter optimization problem (4.2). Direct-search solvers are a natural choice, as they treat an optimization problem as a blackbox and aim to identify a local minimizer, in a meaningful sense, even in the presence of nonsmoothness. Direct-search methods belong to the more general class of derivative-free optimization methods (Conn *et al.*, 2009b). They are so named because they work only with function values and do not compute, nor do they generally attempt to estimate, derivatives. They are especially useful when the objective and/or constraints are expensive to evaluate, are noisy, have limited precision or when derivatives are inaccurate.

In the Opal context, consider a situation where the user wishes to identify the parameters so as to allow an algorithm to solve a collection of test problems to within an acceptable precision in the least amount of time. The objective function in this case is the time required to solve the problems. To be mathematically precise, this measure is not a function, since two runs with the exact same input parameters will most likely differ slightly. The gradient does not exist, and its approximation may point in unreliable directions. For our purposes, a blackbox is an enclosure of the target algorithm that, when supplied with a set of parameter values p, returns either a failure or a *score* consisting of the values of $\psi(p)$, $\varphi(p)$ and all relevant atomic measures $\mu_j(p)$.

The optimization method that we are interested in iteratively calls the blackbox with different inputs. In the present context, the direct-search solver proposes a trial parameter p. The first step is to verify whether $p \in \mathbf{P}$. In the negative, control is returned to the direct-search solver, the trial parameter p is discarded, and the cost of launching the target algorithm is avoided. If all runs result in such a failure, either the set $\mathbf{P}$ is too restrictive or an initial feasible set of parameters should be supplied by the user. Otherwise, a feasible parameter $p \in \mathbf{P}$ is eventually generated and the blackbox computes the composite measures $\psi(p)$ and $\varphi(p)$. This is typically a time-consuming process that requires running the target algorithm on all supplied input data. Consider for instance a case where the blackbox is an optimization solver and the input data consists in the entirety of the CUTEr collection—over 1000 problems for a typical total run time of several days. The composite measures are then returned to the direct search solver.

Direct-search solvers differ from one another in the way they construct the next trial parameters. One of the simplest methods is Coordinate Search, which simply consists in creating $2n$ trial parameters (where n is the dimension of the vector p) in hopes of improving the current best known parameter, say p^{best}. These $2n$ tentative parameters are

$$\{p^{\text{best}} \pm \Delta e_i \;\mid\; i = 1, 2, \ldots, n\}$$

where e_i is the i-th coordinate vector and $\Delta > 0$ is a given step size, also called a *mesh size*. Each of these $2n$ trial parameters is supplied in turn to the blackbox for evaluation. If one of them is feasible for (4.2) and produces an objective function value $\psi(p) < \psi(p^{\text{best}})$, then p^{best} is reset to p and the process is reiterated from the new best incumbent. Otherwise, the step size Δ is shrunk and the process is reiterated from p^{best}. Fermi and Metropolis (1952) used this algorithm on one of the first digital computers.

This simple coordinate search algorithm was generalized by Torczon (1997) in a broader framework of *pattern-search* methods, which also include the methods of Box (1957) and Hooke and Jeeves (1961). Pattern-search methods introduce more flexibility in the construction of the trial parameters and in the variation of the step size. Convergence analysis of pattern-search methods was conducted by Torczon (1997) for unconstrained $\mathcal{C}^2$ functions, and the analysis was extended to nonsmooth functions by Audet and Dennis, Jr. (2003) using the Clarke (1983) generalized calculus.

Pattern-search methods were subsequently further generalized by Audet and Dennis, Jr. (2006) and Audet and Dennis, Jr. (2009) to handle general constraints in a way that is both satisfactory in theory and in practice. The resulting method is called the *Mesh-Adaptive Direct-Search* algorithm (MADS). It can be used to solve problems such as (4.2) even if the initial parameter p does not satisfy the constraints $\varphi(p) \in \mathbf{M}$.

Like the coordinate search, MADS is an iterative algorithm generating a sequence $\{p_k\}_{k=0}^{\infty}$ of trial parameters. At each iteration, attempts are made to improve the current best parameter p_k. However, instead of generating tentative parameters along the coordinate directions, the MADS algorithm uses a mesh structure, consisting of a discretization of the space. The union of all normalized directions generated by MADS is not limited to the coordinate directions, but instead grows dense in the unit sphere.

The convergence analysis considers the iterations that are unsuccessful in improving p_k. At these iterations, p_k satisfies some discretized optimality conditions relative to the current mesh. Any accumulation point $\hat{p}$ of the sequence of unsuccessful parameters p_k for which the mesh gets infinitely fine satisfies optimality conditions that are tied to the local smoothness of the objective and constraints near $\hat{p}$. The convergence analysis relies on the Clarke (1983) nonsmooth calculus. Some of the main convergence results are

- $\hat{p}$ is the limit of mesh local optimizers on meshes that get infinitely fine;
- if the objective function ψ is Lipschitz near $\hat{p}$, then the Clarke generalized directional derivative satisfies $f^\circ(\hat{p}; d) \geq 0$ for any direction d hypertangent to the feasible region at $\hat{p}$;
- if the objective function ψ is strictly differentiable near $\hat{p}$, then $\nabla\psi(\hat{p}) = 0$ in the unconstrained case, and $\hat{p}$ is a contingent KKT stationary point, provided that the domain is regular.

The detailed hierarchical presentation of the convergence analysis given by Audet and Dennis, Jr. (2006) was augmented by Abramson and Audet (2006) to the second-order and by Vicente and Custódio (2012) for discontinuous functions. One of these additional results shows that unlike gradient-based methods for unconstrained C^2 optimization (such as Newton's method), Mads cannot stagnate at a strict local maximizer or at a saddle point. This is somewhat counterintuitive that a method that does not compute nor require derivatives has stronger convergence properties than a method exploiting first and second derivatives for C^2 functions.

It is however interesting in our opinion to use a solver capable of guaranteeing—admittedly at some cost—that a local minimizer will be identified when the problem is sufficiently smooth, and not only a stationary point. Consider for example the objective $\psi(p)$ depicted in Fig. 6.1, which represents the performance in MFlops of a specific implementation of the matrix-matrix multiply kernel for high-performance linear algebra. The implementation used here is from the ATLAS library (Whaley *et al.*, 2001). The function ψ was sampled over a two-dimensional domain for two types of architecture; an Intel Core2 Duo and an Intel Xeon processor. The two parameters are, in this case, integers. One represents the loop unrolling level in the three nested loops necessary to perform the multiply. The other is the *blocking factor* and controls the block size when the multiply is computed blockwise rather than elementwise. Though the graph of ψ is a cloud of points rather than a surface in this case, it is quite apparent that the performance is not an entirely erratic function of the parameters, even though it appears to be affected by noise, but has a certain regularity. In this sense, the Mads framework provides a family of methods that have the potential to identify meaningful minimizers rather that just stationary points.

6.3 The OPAL Package

We propose the Opal package as an implementation of the framework detailed in the previous sections.

6.3.1 The Python Environment

Computational tasks in need of parameter tuning come in infinite variety on widely different platforms and in vastly different environments and languages. It seems *à priori* arduous to design a parameter-tuning environment that is both sufficiently portable and sufficiently flexible to accommodate this diversity. It should also be understood that not all users are computer programmers, and therefore any general

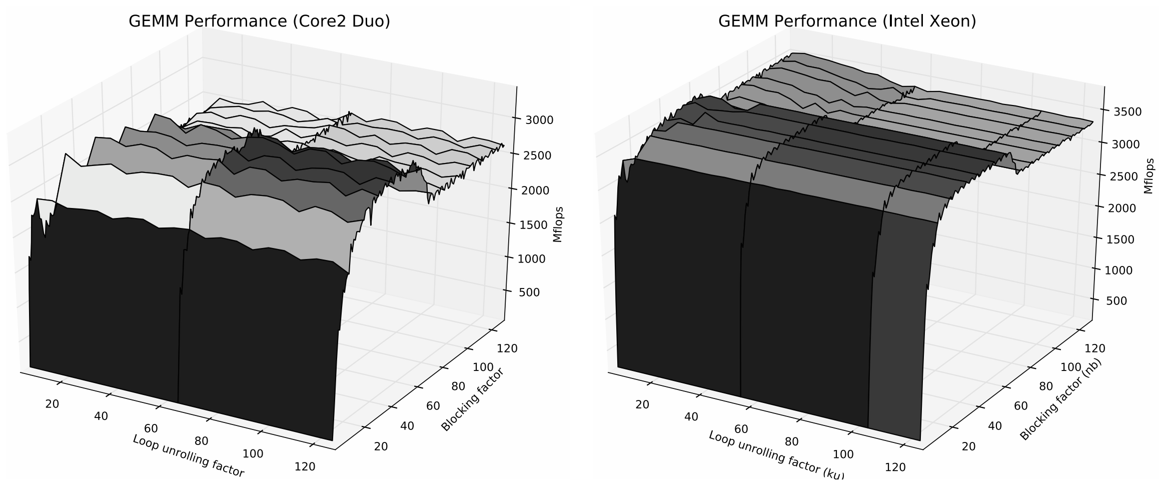


Figure 6.1 Performance in MFlops of a particular implementation of the matrix-matrix multiply as a function of the loop unrolling factor and the blocking factor.

tool seeking to meet the above flexibility requirements must be as easy to use as possible without sacrificing expandability and advanced usage. In our opinion, the latter constraints rule out all low-level programming languages. There remains a handful of options that are portable, flexible, expandable and user friendly. Among those, our option of choice is the Python programming language (`www.python.org`) for the following reasons:

– Python is a rock-solid open-source scripting language. Python has been in constant development since about 1990 and has evolved through its thriving user community to become a standard. Because it is open source, it may be freely shared and distributed for both commercial and non-commercial purposes. Since it is a scripting language, running Python programs does not involve a compiler. It is accompanied nevertheless by a sophisticated debugger.
– Python is available on almost any imaginable platform. Besides covering the three major families, UNIX, OSX and Windows, Python programs are entirely portable to many other platforms, such as OS/2, Amiga, Java VM, including portable devices.
– Python interoperates well with many other languages. A standard C/C++ API combines with automatic interface-generation tools to make interfacing Python and C/C++ programs a breeze. Interfacing Fortran presents no particular difficulty save perhaps for some more recent Fortran 95 features.

- Users can write Python programs much in the same way as shell scripts, batch scripts or Apple scripts, or elect to use the full power of object-oriented programming. Object orientation is by no means a requirement so that users can get started fast and efficiently. For more elaborate purposes, object-oriented programming quickly becomes more convenient, but it is also very natural.
- A wide range of numerical and scientific extensions is available for Python. Among them are Numpy [2], an extension providing the *array* type and vector operations, Scipy [3], a general-purpose library of scientific extensions akin to Matlab toolboxes, and SAGE [4], a symbolic computation package akin to Mathematica, to name only a few, as well as state-of-the art plotting packages such as Matplotlib [5].
- Aside from scientific capabilities, Python is a full-fledged programming language with an extensive standard library that is able to satisfy the most demanding needs, including cryptography, networking, data compression, database access and a lot more.
- The Python syntax is human readable. A user ignorant of the Python syntax is usually able to understand most of what a Python program does simply by reading it.
- It is possible to get up and running on Python programming in one day, thanks to well-designed tutorials and a profusion of documentation and ressources.
- Python comes with "batteries included" on many platforms. For instance, the Enthought Python Distribution [6] and Python(x,y) [7] come with numerous extensions pre-installed. It should be noted that they also come with licensing terms to abide by.
- A fast-paced and fast-increasing body of work has been and is being developed in Python. The best resources to get a glimpse of the expanse of Python-based research and projects is the Python Package Index website [8].

We urge the reader to visit `www.python.org` to learn more and get started with Python programming.

2. `www.scipy.org/numpy`
3. `www.scipy.org`
4. `www.sagemath.org`
5. `matplotlib.sf.net`
6. `www.enthought.com`
7. `code.google.com/p/pythonxy`
8. `pypi.python.org/pypi`

6.3.2 Interacting with Opal

One of the goals of Opal is to provide users with a set of programmatic tools to aid in the modeling of algorithmic parameter optimization problems. A complete model of a problem of the form (4.2) consists in

1. declaring the blackbox and its main features; this includes declaring the parameters p, their type, their domain $\mathbf{P}$, a command that may be used to run the target algorithm with given parameters, and registering those parameters with the blackbox;
2. stating the precise form of the parameter optimization problem (4.2) by defining the objective and constraints as functions of atomic and composite measures;
3. providing an executable that may be run by the direct-search solver and whose task is to read the parameter set proposed by the solver, pass them to the blackbox, and retrieve all relevant atomic measures.

Other ingredients may be included into the complete model. We provide a general description of the modeling task in this section and leave additions for later sections. For illustration, we use an intentionally simplistic problem consisting in finding the optimal stepsize in a forward finite-difference approximation to the derivative of the sine function at $x = \pi/4$. The only parameter is the stepsize $p = h$. The objective function is $\psi(h) = |(\sin(\pi/4 + h) - \sin(\pi/4))/h - \cos(\pi/4)|$. It is well known that in the absence of noise, the optimal value for h is approximately a constant multiple of $\sqrt{\varepsilon_{\mathrm{M}}}$ where ε_{M} is the machine epsilon. Although intuitively, only small values of h are of interest, the domain $\mathbf{P}$ could be described as $(0, +\infty)$. Note that $\mathbf{P}$ is open in this case and although optimization over non-closed sets is not well defined, the barrier mechanism in the direct solver ensures that values of h that lie outside of $\mathbf{P}$ are rejected. The declaration of the blackbox and its parameter is illustrated in Listing 6.1, which represents the contents of the *declaration file.* In Listing 6.1, a new algorithm is declared on line 5, an executable command to be run by Opal every time a set of parameters must be assessed is given on line 6, the parameter h is declared and registered with the algorithm on lines 8–10 and the sole measure of interest is declared and registered with the algorithm on lines 12–13. We believe that Listing 6.1 should be quite readable, even without prior knowledge of the Python language.

For maximum portability, information about parameter values and measure values are exchanged between the blackbox and the direct solver by way of files. Each time

Listing 6.1 `fd_declaration.py`: Declaration of the forward-difference algorithm

```
 1 from opal.core.algorithm import Algorithm
 2 from opal.core.parameter import Parameter
 3 from opal.core.measure   import Measure
 4
 5 FD = Algorithm(name=’FD’, description=’Forward Finite Differences’)
 6 FD.set_executable_command(’python fd_run.py’)
 7
 8 h = Parameter(kind=’real’, default=0.5, bound=(0, None),
 9               name=’h’, description=’Step size’)
10 FD.add_param(h)
11
12 error = Measure(kind=’real’, name=’ERROR’,
13                 sdescription=’Error in derivative’)
14 FD.add_measure(error)
```

the direct solver requests a run with given parameters, the executable command specified on line 6 of Listing 6.1 will be run with three arguments: the name of a file containing the candidate parameter values, the name of a problem that acts as input to the blackbox and the name of an output file to which measure values should be written. The second argument is useful when each blackbox evaluation consists in running the target algorithm over a collection of sets of input data, such as a test problem collection. In the present case, there is no such problem collection and the second argument should be ignored. The role of the *run file* is to read the parameter values proposed by the solver, pass them to the blackbox, retrieve the relevant measures and write them to file. An example run file for the finite-differences example appears in Listing 6.2.

The run file must be executable from the command line, i.e., it should contain a `__main__` section. Parameters are read from file using an input function supplied with Opal. The parameters appear in a dictionary of name-value pairs indexed by parameter names, as specified in the declaration file. The `run()` function returns measures—here, a single measure representing $\psi(h)$—as a dictionary. Again the keys of the latter must match measures registered with the blackbox in the declarations file. Finally, measures are written to file using a supplied output function. It is worth stressing that typically, only lines 6–9 change across run files. The rest stays the same, with a few variations in the *import* section (lines 2 and 3).

There remains to describe how the problem (4.2) itself is modeled. Opal separates the optimization problem into two components: the model *structure* and the

Listing 6.2 `fd_run.py`: Calling the blackbox

```python
from opal.core.io import *
from fd import fd                  # Target algorithm.
from math import pi, sin, cos

def run(param_file, problem):
    "Run FD with given parameters."
    params = read_params_from_file(param_file)
    h = params['h']
    return {'ERROR': abs(cos(pi/4) - fd(sin,pi/4,h))}

if __name__ == '__main__':
    import sys
    param_file  = sys.argv[1]
    problem     = sys.argv[2]
    output_file = sys.argv[3]

    # Solve, gather measures and write to file.
    measures = run(param_file, problem)
    write_measures_to_file(output_file, measures)
```

model *data*. The *structure* represents the abstract problem (4.2) independently of what the target algorithm is, what input data collection is used at each evaluation of the blackbox, if any, and other instance-dependent features to be covered in later sections. It specifies the form of the objective function and of the constraints. The *data* instantiates the model by providing the target algorithm, the input data collection, if any, and various other elements. This separation allows the solution of closely-related problems with minimal change, e.g., changing the input data set, removing a constraint, and so forth. The *optimize file* for our example can be found in Listing 6.3. The most important part of Listing 6.3 is lines 10–12, where the actual problem is defined. In the next section, the flexibility offered by this description of a parameter optimization problem allows us to define surrogate models using the same concise syntax.

6.3.3 Surrogate Optimization Problems

An important feature of the OPAL framework is the use of surrogate problems to guide the optimization process. Surrogates were introduced by Booker *et al.* (1999) for pattern search, and are used by the solver as substitutes for the optimization problem. A fundamental property of surrogate problems is that their objective and

Listing 6.3 `fd_optimize.py`: Statement of the problem and solution

```python
from fd_declaration import FD
from opal import ModelStructure, ModelData, Model
from opal.Solvers import NOMADSolver

# Return the error measure.
def get_error(parameters, measures):
    return sum(measures["ERROR"])

# Define parameter optimization problem.
data = ModelData(FD)
struct = ModelStructure(objective=get_error)  # Unconstrained
model = Model(modelData=data, modelStructure=struct)

# Create solver instance.
NOMAD = NOMADSolver()
NOMAD.solve(blackbox=model)
```

constraints need to be less expensive to evaluate than the objective and constraints of (4.2). They need to share some similarities with (4.2), in the sense that they should indicate promising search regions, but do not need to be an approximation.

In the parameter optimization context, a static surrogate might consist in solving a small subset of test problems instead of solving the entire collection. In that case, if the objective consists in minimizing the overall CPU time, then the surrogate value will not even be close to being an approximation of the time to solve all problems. Section 6.3.6 suggests a strategy to construct a representative subset of test problems by using clustering tools from data analysis. Another type of surrogate can be obtained by relaxing the stopping criteria of the target algorithm. For example, one might terminate a gradient-based descent algorithm as soon as the gradient norm drops below 10^{-2} instead of 10^{-6}. Another example would be to use a coarse discretization in a Runge-Kutta method.

Dynamic surrogates can also be used by direct search methods. These surrogates are dynamically updated as the optimization is performed, so that they model more accurately the functions that they represent. In the MADS framework, local quadratic surrogates are proposed by Conn and Le Digabel (2011) and global treed Gaussian process surrogates by Gramacy and Le Digabel (2011).

In OPAL, surrogates are typically used in two ways. Firstly, OPAL can use a surrogate problem as if it were the true optimization problem, and optimize it with the blackbox solver. The resulting locally optimal parameter set can be supplied as a starting point for (4.2). Secondly, surrogates are used by the solver to order tentative parameters, to perform local descents and to identify promising candidates.

A more specific description of the usage of surrogate functions within a parameter optimization context is given by Audet and Orban (2006). In essence, when problems are defined by training the target algorithm on a list of sets of input data, such as test problems, a surrogate can be constructed by supplying a set of simpler test problems. An example of how OPAL facilitates the construction of such surrogates is given in Listing 6.4 in the context of the trust-region algorithm examined by Audet and Orban (2006) and Audet *et al.* (2011a). This example also illustrates how to specify constraints. The syntax of line 19 indicates that there is a single constraint whose body is given by the function `get_error()` with no lower bound and a zero upper bound. If several constraints were present, they should be specified as a list of such triples.

In Listing 6.4 we define two measures; ψ is represented by the function `sum_heval()`, which computes the total number of Hessian evaluations and the constraint function φ is represented by the function `get_error()`, which returns the number of failures. The parameter optimization problem, defined in lines 18–20 consists in minimizing $\psi(p)$ subject to $\varphi(p) \geq 0$, which simply expresses the fact that we require all problems to be processed without error. A surrogate model is defined to guide the optimization in lines 23–25. It consists in minimizing the same $\psi(p)$ with the difference that the input problem list is different. For the original problem, the input problem list consists in all unconstrained problems from the `CUTEr` collection—see line 14. The surrogate model uses a list of smaller problems and can be expected to run much faster—see line 15. In line 19, the syntax for specifying constraints is to provide a list of triples. Each triple gives a lower bound, a composite measure and an upper bound. In this example, a single constraint is specified.

6.3.4 Categorical Variables

Several blackbox optimization solvers can handle continuous, integer and binary variables, but fewer have the capacity to handle categorical ones. Orban (2011) uses categorical variables to represent a loop order parameter and compiler options in a standard matrix multiply.

Listing 6.4 Definition of a surrogate model.

```
from trunk_declaration import trunk             # Target algorithm.
from opal import ModelStructure, ModelData, Model
from opal.Solvers import NOMADSolver
from opal.TestProblemCollections import CUTEr  # The CUTEr test set.

def sum_heval(parameters, measures):
    "Return total number of Hessian evaluation across test set."
    return sum(measures["HEVAL"])

def get_error(parameters,measures):
    "Return number of nonzero error codes (failures)."
    return len(filter(None, measures['ECODE']))

cuter_unc = [p for p in CUTEr    if p.ncon == 0]
# Unconstrained problems.
smaller   = [p for p in problems if p.nvar <= 100] # Smaller problems.

# Define (constrained) parameter optimization problem.
data   = ModelData(algorithm=trunk, problems=cuter_unc)
struct = ModelStructure(objective=sum_heval,
                        constraints=[(None,get_error,0)])
model  = Model(modelData=data, modelStructure=struct)

# Define a surrogate (unconstrained).
surr_data   = ModelData(algorithm=trunk, problems=smaller)
surr_struct = ModelStructure(objective=sum_heval)
surr_model  = Model(modelData=surr_data, modelStructure=surr_struct)

NOMAD = NOMADSolver()
NOMAD.solve(blackbox=model, surrogate=surr_model)
```

Ansel *et al.* (2009) discuss strategies to select the best sorting algorithm based on the input size. They state that insertion sort is adapted to small input sizes, quicksort to medium sizes, and either radix or merge sort is suitable for large inputs. With OPAL, a categorical parameter may be used to select which sorting algorithm to use. Listing 6.5 gives the OPAL declaration of a categorical parameter representing the choice of a sort strategy. Note however that the ultimate goal of Ansel *et al.* (2009) is different in that they exploit the fact that most sort strategies are recursive by nature. They are interested in determining the fastest sort strategy as a function of the input size so as to be able to determine on the fly, given a certain input size, what type of sort is best. To achieve this, their parameters are the sort type to be used at any given recursive level. Thus if the variable `sort_type` ever takes the value `quick`, it gives rise to two new categorical variables in the problem, each determining the type of sort to call on each half of the array passed as input to quicksort. This is an example where the dimension of the problem is not known beforehand.

MADS easily handles integer variables by exploiting their inherent ordering. This is done by making sure that the step size parameter Δ mentioned in § 4.2.2 is integer. Furthermore, a natural stopping criteria triggers when an iteration fails to improve p^{best} with a unit step size.

Categorical variables cannot be handled as easily as integer ones. They do not posses any ordering properties, and they need to be accompanied by a neighborhood structure, such as the one illustrated in Listing 6.5. Each iteration of the MADS algorithm constructs two sets of tentative trial parameters. One set retains the same categorical values as those of p^{best} and modifies only the continuous and integer variables using the same technique as without categorical variables. The other set is constructed using the user-provided set of categorical neighbors. A precise description of how this is accomplished for the pattern search algorithm is presented by Abramson *et al.* (2007), and the method is illustrated by Kokkolaras *et al.* (2001) on an optimization problem where the neighborhood structure is such that changes in some of the categorical variables alter the number of optimization variables of the problem.

6.3.5 Parallelism at Different Levels

OPAL can exploit architectures with several processors or several cores at different levels. Audet *et al.* (2011a) compare three ways of using parallelism within OPAL. The first strategy consists in the blackbox solver evaluating the quality of trial parameters in parallel, the second strategy exploits the structure of (4.2) and consists in launching the target algorithm to solve test problems concurrently, and the third simultaneously

Listing 6.5 Example use of categorical variables in OPAL

```
sort_type = Parameter(kind='categorical', default='quick',
                      neighbors={'insertion': ['quick'],
                                 'quick': ['insertion', 'radix',
                                           'merge'],
                                 'radix':  ['quick', 'merge'],
                                 'merge':  ['quick', 'radix']})
```

applies both strategies. The blackbox solver is parallelized by way of MPI and can be set to be synchronous or asynchronous. When parallelizing the blackbox itself, Opal supports MPI, SMP, LSF and SunGrid Engine.

6.3.6 Combining Opal with Clustering Tools

In this section, we briefly illustrate how Opal may be combined with external tools to produce effective surrogate models. The experimental test illustrated in the appendix considers the optimization of six real parameters from IPOPT, a nonlinear constrained optimization solver described by Wächter and Biegler (2006). The objective to be minimized is the total number of objective and constraint evaluations, as well as evaluations of their derivatives. The only constraint requires that all the test problems be solved successfully. The testbed $\mathcal{L}$ contains a total of 730 test problems from the CUTEr collection (Gould *et al.*, 2003a). The objective function value with the default parameters p_0 is $\psi_{\mathcal{L}}(p_0) = 207,866$. The overall computing time required for solving this blackbox optimization problem is 27h55m, and produces a set of parameters $\hat{p}$ with an objective function value of $\psi_{\mathcal{L}}(\hat{p}) = 198,615$. Paralellism is used by allowing up to 10 concurring function evaluations on multiple processors.

Clustering is used to generate a surrogate model with significantly less test problems than the actual blackbox problem. More specifically, he performs a clustering analysis on the cells of a self-organizing map based on the work of Kohonen (1998); Kohonen and Somervuo (2002) and Pantazi *et al.* (2002). The self-organizing map partitions the testbed into clusters sharing similar values of the objective and constraints. A representative problem from each cluster is identified by the clustering scheme, resulting in a subset $\mathcal{L}_1$ of 41 test problems from $\mathcal{L}$. Opal is then launched on the minimization of $\psi_{\mathcal{L}_1}(p)$ subject to the same no-failure constraint. This surrogate problem is far easier to solve, as it requires only 4h17m and produces a solution p_1 which is close to $\hat{p}$.

6.3.7 The Blackbox Optimization Solver

The default blackbox solver used by Opal is the Nomad software (Le Digabel, 2011). It is a robust code, implementing the Mads algorithm for nonsmooth constrained optimization of Audet and Dennis, Jr. (2006), which is supported by a rigorous nonsmooth convergence analysis. Nomad can be used in conjunction with a surrogate optimization problem. Among others, dynamic quadratic model surrogates can be generated automatically (Conn and Le Digabel, 2011).

Nomad handles all the variable types enumerated in §6.2.1, and in addition allows subsets of variables to be free, fixed or periodic. It also allows the possibility of grouping subsets of variables. In the Opal context, consider for example an algorithm that has two embedded loops, and a subset of parameters that relates to the inner loop, while another subset relates to the outer loop. It might be useful to declare these subsets as two groups of variables as it would allow Nomad to conduct its exploration in smaller parameter subspaces.

Nomad is designed to handle relaxable constraints by a progressive barrier or by a filter, and non-relaxable constraints by the extreme barrier, which means that the objective function ψ is replaced with

$$\hat{\psi}(p) := \begin{cases} \psi(p) & \text{if } p \text{ is feasible,} \\ +\infty & \text{otherwise.} \end{cases}$$

It is also robust to hidden constraints (i.e., constraints that reveal themselves by making the simulation fail). A discussion of these types of constraints and approaches to handle them are described by Audet *et al.* (2010b), together with applications to engineering blackbox problems.

6.4 Discussion

In designing the Opal framework, our goal is to provide users with a modeling environment that is intuitive and easy to use while at the same time relying on a state-of-the-art blackbox optimization solver. It is difficult to say whether the performance of an algorithm depends continuously on its (real) parameters or not. Since parameters may also often be discrete, a nonsmooth optimization solver seems to be the best choice.

Algorithmic parameter optimization applications are in endless supply and there is often much to gain when there are no obvious dominant parameter values. The choice of the Python language maximizes flexibility and portability. Users are able to combine Opal with other tools, whether implemented in Python or not, to generate surrogate models or run simulations. Opal also makes it transparent to take advantage of parallelism at various levels. It has been used in several types of applications, including code generation for high-performance linear algebra kernels to the optimization of the performance of optimization solvers. It is however not limited to computational science—any code depending on at least one parameter could benefit from optimization.

OPAL is non intrusive, which could make it a good candidate for legacy code that should not be recompiled or for closed-source proprietary applications.

Much remains to be done in the way of improvements. Among other aspects, we mention the identification of *robust* parameter values—values that would remain nearly optimal if slightly perturbed—and the automatic identification of the most influential parameters of a given target algorithm.

Chapter 7

GENERAL DISCUSSION

The main contribution of this thesis is not only a framework for optimization of algorithms but also a Python package implementing the framework. Three papers in previous chapters reveal the development of these two main contributions. In each paper some conclusions have been made, hence we reserve this chapter to review the general aspects of the two achievements: a general, flexible, efficient framework and an easy-to-use, extensible, integrable package.

Although the parameter tuning question has been raised for a long time and there have been many attempts to address it, the focus on a particular target algorithm prevents the popularity of these attempts. By approaching the problem of empirical parameter tuning from two points of view, OPAL becomes a general, flexible and efficient framework for parameter tuning or algorithm optimization. Having identified and answered the crucial questions, the OPAL framework satisfies the requirements of a versatile tool for parameter tuning. Within OPAL, a tuning parameter question is modelled as a blackbox optimization problem. The versatility is confirmed in the sense that any type of parameter is accepted and can be easily defined; any objective and restriction can be specified by the composite measures; no particular prerequisite is needed to solve defined problem. Users can experiment with the flexibility in activating parallel mode, in defining a surrogate for accelerating the search or in interacting with other systems. Finally, efficiency is illustrated by numerical experiments illustrated in the thesis.

The Python package OPAL includes components that facilitate defining a parameter optimization and invoking a solver. Besides essential components of an automated tuning system, OPAL works as an interface that gathers external components such as a target algorithm, a direct solver or a benchmarking system, etc. This characteristic requires OPAL components to be fairly independent but can communicate to each other easily. Therefore, we design the package in such a way that every component is autonomous and communicates to each other by exchanging text messages through a message pool. As a result, our package possesses high extensibility and integrability. An arbitrary component such as a software or, a Python-based module can be deployed programmatically into OPAL if it is equipped with an inter-process communication mechanism and is able to understand OPAL message as well as to throw the OPAL-understandable messages to the message pool. Otherwise, OPAL can be integrated in a passive way through the text-based logging files and Python scripts; the host system will read log messages of OPAL and write Python scripts to control OPAL.

Chapter 8

CONCLUSION AND RECOMMENDATIONS

The main contribution of this thesis is not only a framework for optimization of algorithms but also a Python package implementing the framework. The main features are discussed in the previous chapter. In this chapter, there are discussions on perspectives regarding the framework.

OPAL is a general, flexible and efficient framework for parameter tuning or algorithm optimization. Having identified and answered the crucial questions, the OPAL framework satisfies the requirements of a versatile tool for parameter tuning. Additionally, the Python package OPAL includes autonomous components that make OPAL be integrated easily with other systems. Although satisfying the requirements of a versatile framework, OPAL can still be developed further. The ideas behind any extension are to exploit as many particularities as possible of a parameter optimization problem. Some ideas are already realized by the extensions presented in our work, others are still in discussion.

Solving parameter optimization problems with NOMAD, users can use surrogate models to accelerate search or to guide to a promising region. A surrogate model can be anything that can simulate the behaviour of the blackbox but has a lower computational cost. OPAL allows users to freely define a surrogate and integrate it into a tuning process in a simple way. In our previous experiments, surrogate models were mainly defined as other parameter optimization problems whose set of test problems is small. The set of test problems is normally pre-selected based on knowledge of the test problems. Hence in future works, we would like to develop a feature of automated constructing of a surrogate based on information gathered since the launching of the tuning process. For example, users can select the problems by a classification tool or a machine learning techniques.

The second paper showed two techniques for improving performance. However, these techniques are realized at fairly simple levels; they can not adapt to changes in computational resources availability. Hence, a load balancing strategy can be considered. We can selectively launch test problems in such a way that the test problems having similar computing time are executed in parallel at the same time. Alternatively an asynchronous strategy of executing test problems can be used to take maximal advantage of the computing capacity of the system. Another possibility is that we can reorder the list of test problems in such a way that the probability of interruption is increased. For example, suppose we have a parameter optimization problem that requires no failure in the test problems and in a previous iteration, we find a test problem failed, we obviously want to execute this test problem first at the next iteration to avoid wasting time when an interruption occurs.

Besides concrete techniques targeting performance issues, we also suggest another perspective on the parameter optimization problem. The objective function is normally an aggregate function that synthesizes the results on multiple test problems such as the sum or the mean. In fact, each test problem or each class of test problems has a different influence on a target algorithm, and hence can direct the search towards different regions. In other words, the aggregate functions eliminate the role of test problem structure while it is one of the most important factors dominating target algorithm behaviors. Intuitively, we want to find a parameter setting where the target algorithm works well on most test problems. As a result, parameter optimization can be studied as a multi-objective optimization problem. However, in practice if we consider each test problem to represent for an objective, the multi-objective parameter optimization problem becomes unrealistic and infeasible. Therefore, the idea of grouping test problem such that each group represents an objective for multi-objective parameter optimization problem is out of the question. Clearly, clustering or classification techniques can be considered for grouping test problems. Following this approach, we give more opportunities for test problems to "communicate" with one another.

Continuing to exploit the roles of test problems, we consider MDO (Multidisciplinary Design Optimization) (Cramer *et al.*, 1994). MDO uses optimization methods to solve design problems incorporating multiple disciplines; the idea behind this is to exploit the interaction between disciplines during the optimization to get an optimum superior to solutions obtained by optimizing each discipline sequentially. In the case of a numerical algorithm that is usually sensitive to problem structure, each class of test problems can be considered as a discipline because it will cause different reactions and behaviors from target algorithm. In other words, different classes of test problems require different optimal settings and will take the tuning process in different directions. Hence, application techniques from MDO to the problem of empirical parameter optimization could be a relevant direction of research.

Bibliography

Abramson, M., Audet, C., Chrissis, J. and Walston, J. (2009a). Mesh adaptive direct search algorithms for mixed variable optimization. *Optimization Letters*, 3, 35–47.

Abramson, M., Audet, C. and Dennis, Jr., J. E. (2007). Filter pattern search algorithms for mixed variable constrained optimization problems. *Pacific Journal on Optimization*, 3, 477–500.

Abramson, M. A. (2002). *Pattern Search Algorithms for Mixed Variable General Constrained Optimization Problems.* PhD Thesis, Department of Computational and Applied Mathematics, Rice University.

Abramson, M. A. and Audet, C. (2006). Convergence of mesh adaptive direct search to second-order stationary points. *SIAM Journal on Optimization*, 17, 606–619.

Abramson, M. A., Audet, C., Couture, G., Dennis, Jr., J. E. and Le Digabel, S. (2004). *The NOMAD project.* Master thesis, GERAD.

Abramson, M. A., Audet, C., Dennis, Jr., J. E. and Le Digabel, S. (2009b). OrthoMADS: A deterministic MADS instance with orthogonal directions. *SIAM Journal on Optimization*, 20, 948–966.

Adenso-Diaz, B. and Laguna, M. (2006). Fine-Tuning of Algorithms Using Fractional Experimental Designs and Local Search. *Operations Research*, 54, 99–114.

Ansel, J., Chan, C., Wong, Y. L., Olszewski, M., Zhao, Q., Edelman, A. and Amarasinghe, S. (2009). PetaBricks: a language and compiler for algorithmic choice. *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation.* ACM, New York, NY, USA, PLDI '09, 38–49.

Ansel, J., Won, Y. L., Chan, C., Olszewski, M., Edelman, A. and Amarasinghe, S. (2011). Language and compiler support for auto-tuning variable-accuracy algorithms. *The International Symposium on Code Generation and Optimization.* Chamonix, France, 85–96.

AUDET, C. (2004). Convergence Results for Pattern Search Algorithms are Tight. *Optimization and Engineering*, 5, 101–122.

AUDET, C., BÉCHARD, V. and LE DIGABEL, S. (2008a). Nonsmooth optimization through mesh adaptive direct search and variable neighborhood search. *Journal of Global Optimization*, 41, 299–318.

AUDET, C., DANG, C.-K. and ORBAN, D. (2010a). Algorithmic parameter optimization of the DFO method with the OPAL framework. J. C. K. Naono, K. Teranishi and R. Suda, editors, *Software Automatic Tuning: From Concepts to State-of-the-Art Results*, Springer, New-York, NY. 255–274.

AUDET, C., DANG, C.-K. and ORBAN, D. (2011a). Efficient use of parallelism in algorithmic parameter optimization applications. *Optimization Letters*, 1–13. Online First.

AUDET, C., DANG, C.-K. and ORBAN, D. (2011b). Optimization of algorithms with OPAL. Technical report G-2011-08, Les cahiers du GERAD. In Progress.

AUDET, C. and DENNIS, JR., J. E. (2000). Pattern search algorithms for mixed variable programming. *SIAM Journal on Optimization*, 11, 573–594.

AUDET, C. and DENNIS, JR., J. E. (2003). Analysis of generalized pattern searches. *SIAM Journal on Optimization*, 13, 889–903.

AUDET, C. and DENNIS, JR., J. E. (2006). Mesh adaptive direct search algorithms for constrained optimization. *SIAM Journal on Optimization*, 17, 188–217.

AUDET, C. and DENNIS, JR., J. E. (2009). A progressive barrier for derivative-free nonlinear programming. *SIAM Journal on Optimization*, 20, 445–472.

AUDET, C., DENNIS, JR., J. E. and LE DIGABEL, S. (2008b). Parallel space decomposition of the mesh adaptive direct search algorithm. *SIAM Journal on Optimization*, 19, 1150–1170.

AUDET, C., DENNIS, JR., J. E. and LE DIGABEL, S. (2010b). Globalization strategies for mesh adaptive direct search. *Computational Optimization and Applications*, 46, 193–215.

AUDET, C. and LE DIGABEL, S. (2012). The mesh adaptive direct search algorithm for periodic variables. *Pacific Journal of Optimization*, 8, 103–109.

AUDET, C. and ORBAN, D. (2006). Finding optimal algorithmic parameters using derivative-free optimization. *SIAM Journal on Optimization*, 17, 642–664.

AUGER, A., HANSEN, N., PEREZ ZERPA, J. M., ROS, R. and SCHOENAUER, M. (2009). Experimental comparisons of derivative free optimization algorithms experimental algorithms. J. Vahrenhold, editor, *Experimental Algorithms*, Springer Berlin / Heidelberg, Berlin, Heidelberg, vol. 5526 of *Lecture Notes in Computer Science*, chapter 3. 3–15.

BALAPRAKASH, P., WILD, S. M. and HOVLAND, P. D. (2011a). Can search algorithms save large-scale automatic performance tuning? Technical report ANL/MCS-P1823-0111, Argonne National Laboratory, Mathematics and Computer Science Division.

BALAPRAKASH, P., WILD, S. M. and NORRIS, B. (2011b). Spapt: Search problems in automatic performance tuning. Technical report ANL/MCS-1956-0911, Argonne National Laboratory, Mathematics and Computer Science Division.

BARTZ-BEIELSTEIN, T. (2010). Spot: An R package for automatic and interactive tuning of optimization algorithms by sequential parameter optimization. *CoRR*, abs/1006.4645.

BARTZ-BEIELSTEIN, T., CHIARANDINI, M., PAQUETE, L. and PREUSS, M., editors (2010a). *Experimental Methods for the Analysis of Optimization Algorithms*. Springer, Germany.

BARTZ-BEIELSTEIN, T., LASARCZYK, C. and PREUSS, M. (2010b). The sequential parameter optimization toolbox experimental methods for the analysis of optimization algorithms. T. Bartz-Beielstein, M. Chiarandini, L. Paquete and M. Preuss, editors, *Experimental Methods for the Analysis of Optimization Algorithms*, Springer Berlin Heidelberg, Berlin, Heidelberg, chapter 14. 337–362.

BAZ, M., BROOK, P. J., GOSAVI, A. and HUNSAKER, B. (2007). Automated tuning of optimization software parameters. Technical report, University of Pittsburgh Department of Industrial Engineering.

BAZ, M., HUNSAKER, B. and PROKOPYEV, O. (2009). How much do we "pay" for using default parameters? *Computational Optimization and Applications*, 44.

BILMES, J., ASANOVIĆ, K., CHIN, C.-W. and DEMMEL, J. (1998). The PHiPAC v1.0 matrix-multiply distribution. Technical Report TR-98-35, International Computer Science Institute, CS Division, University of California, Berkeley CA.

BLACKFORD, L. S., DEMMEL, J., DONGARRA, J. J., DUFF, I. S., HAMMARLING, S., HENRY, G., HEROUX, M., KAUFMAN, L., LUMSDAINE, A., PETITET, A., Pozo, R. and K. Remington, R. C. W. (2002). An updated set of basic linear algebra subprograms (BLAS). *ACM Transactions on Mathematical Software*, 28, 135–151.

Booker, A. J., Dennis, Jr., J. E., Frank, P. D., Serafini, D. B., Torczon, V. and Trosset, M. W. (1999). A rigorous framework for optimization of expensive functions by surrogates. *Structural Optimization*, 17, 1–13.

Box, G. E. P. (1957). Evolutionary operation: A method for increasing industrial productivity. *Appl. Statist.*, 6, 81–101.

Cartis, C., Gould, N. I. M. and Toint, P. L. (2012). An adaptive cubic regularization algorithm for nonconvex optimization with convex constraints and its function-evaluation complexity. *IMA Journal of Numerical Analysis*.

Cavazos, J. and O'Boyle, M. F. P. (2005). Automatic tuning of inlining heuristics. *Proceedings of the 2005 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, Washington, DC, USA, SC '05, 14–.

Choi, T. D. and Kelley, C. T. (2000). Superlinear convergence and implicit filtering. *SIAM Journal on Optimization*, 10, 1149–1162.

Clarke, F. H. (1983). *Optimization and Nonsmooth Analysis.* Wiley, New York. Reissued in 1990 by SIAM Publications, Philadelphia, as Vol. 5 in the series Classics in Applied Mathematics.

Conn, A. R., Gould, N. I. M. and Toint, P. L. (2000). *Trust-Region Methods.* MPS-SIAM Series on Optimization 1. SIAM.

Conn, A. R. and Le Digabel, S. (2011). Use of quadratic models with mesh adaptive direct search for constrained black box optimization. Technical report G-2011-11, Les cahiers du GERAD.

Conn, A. R., Scheinberg, K. and Toint, P. L. (1998). A derivative free optimization algorithm in practice. *Proceedings the of 7th AIAA/USAF/NASA/ISSMO Symposium on Multidisciplinary Analysis and Optimization.* St. Louis, Missouri.

Conn, A. R., Scheinberg, K. and Toint, P. L. (2009a). DFO. Web page.

Conn, A. R., Scheinberg, K. and Vicente, L. N. (2009b). *Introduction to Derivative-Free Optimization.* MPS/SIAM Book Series on Optimization. SIAM, Philadelphia.

CRAMER, E. J., DENNIS, J. E., FRANK, P. D., LEWIS, R. M. and SHUBIN, G. R. (1994). Problem formulation for multidisciplinary optimization. *SIAM Journal on Optimization*, 4, 754–776.

DOLAN, E. D. and MORÉ, J. J. (2002). Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91, 201–213.

FERMI, E. and METROPOLIS, N. (1952). Los Alamos unclassified report LS–1492. Technical report, Los Alamos National Laboratory, Los Alamos, New Mexico.

FESTA, P., GONÇALVES, J., RESENDE, M. and SILVA, R. (2010). Automatic tuning of GRASP with Path-Relinking heuristics with a biased Random-Key genetic algorithm. P. Festa, editor, *Experimental Algorithms*, Springer Berlin / Heidelberg, Berlin, Heidelberg, vol. 6049 of *Lecture Notes in Computer Science*, chapter 29. 338–349–349.

FINKEL, D. and KELLEY, C. (2004). Convergence analysis of the direct algorithm. Technical report CRSC-TR04-28, Center for Research in Scientific Computation.

FRIGO, M. and JOHNSON, S.-G. (2005). The design and implementation of FFTW3. *Proceedings of the IEEE*, 93, 216–231.

GILMORE, P., CHOI, T., ESLINGER, O., KELLEY, C. T., PATRICK, H. A. and GABLONSKY, J. (1999). IFFCO (Implicit Filtering For Constrained Optimization). Software available at `http://www4.ncsu.edu/~ctk/iffco.html`.

GOULD, N., ORBAN, D. and TOINT, P. L. (2003a). CUTEr (and SifDec): a constrained and unconstrained testing environment, revisited. *ACM Transactions on Mathematical Software*, 29, 373–394.

GOULD, N. I. M., LUCIDI, S. and TOINT, P. L. (1999). Solving the trust-region subproblem using the lanczos method. *SIAM Journal on Optimization*, 9, 504–525.

GOULD, N. I. M., ORBAN, D., SARTENAER, A. and TOINT, P. L. (2005). Sensitivity of trust-region algorithms on their parameters. *4OR*, 3, 227–241.

GOULD, N. I. M., ORBAN, D. and TOINT, P. L. (2003b). Galahad, a library of thread-safe fortran 90 packages for large-scale nonlinear optimization. *ACM Transactions on Mathematical Software*, 29, 353–372.

GRAMACY, R. B. and LE DIGABEL, S. (2011). The mesh adaptive direct search algorithm with treed gaussian process surrogates. Technical report G-2011-37, Les cahiers du GERAD.

Gray, G. A. and Kolda, T. G. (2006). Algorithm 856: APPSPACK 4.0: Asynchronous parallel pattern search for derivative-free optimization. *ACM Transactions on Mathematical Software*, 32, 485–507.

Grefenstette, J. (1986). Optimization of control parameters for genetic algorithms. *IEEE Trans. Syst. Man Cybern.*, 16, 122–128.

Griffin, J. D., Kolda, T. G. and Lewis, R. M. (2008). Asynchronous parallel generating set search for linearly constrained optimization. *SIAM Journal on Scientific Computing*, 30, 1892–1924.

Gropp, W., Lusk, E. and Skjellum, A. (1994). *Using MPI: portable parallel programming with the message-passing interface*. Scientific And Engineering Computation Series. MIT Press, Cambridge, MA, USA, second edition.

Hartono, A., Norris, B. and Sadayappan, P. (2009). Annotation-based empirical performance tuning using orio. *IPDPS*. IEEE, 1–11.

Higham, N. J. (2002). *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, second edition.

Hock, W. and Schittkowski, K. (1981). *Test Examples for Nonlinear Programming Codes*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Hooke, R. and Jeeves, T. A. (1961). Direct search solution of numerical and statistical problems. *Journal of the Association for Computing Machinery*, 8, 212–229.

Huang, D., Allen, T. T., Notz, W. I. and Zeng, N. (2006). Global optimization of stochastic Black-Box systems via sequential kriging Meta-Models. *J. of Global Optimization*, 34, 441–466.

Hutter, F., Bartz-Beielstein, T., Hoos, H.-H., Leyton-Brown, K. and Murphy, K.-P. (2009). An experimental investigation of model-based parameter optimisation: SPO and beyond. *Proc. of GECCO-09*. 271–278.

Hutter, F., Hoos, H., Leyton-Brown, K. and Murphy, K. (2010a). Time-Bounded sequential parameter optimization. C. Blum and R. Battiti, editors, *Learning and Intelligent Optimization*, Springer Berlin / Heidelberg, Berlin, Heidelberg, vol. 6073 of *Lecture Notes in Computer Science*, chapter 30. 281–298–298.

Hutter, F., Hoos, H. H. and Brown, K. L. (2010b). Tradeoffs in the empirical evaluation of competing algorithm designs. *Annals of Mathematics and Artificial Intelligence*, 60, 65–89.

Hutter, F., Hoos, H. H. and Stützle, T. (2007). Automatic algorithm configuration based on local search. *Proc. of the Twenty-Second Conference on Artifical Intelligence (AAAI '07).* 1152–1157.

Im, E. J. and Yelick, K. (1998). Model-based memory hierarchy optimizations for sparse matrices. *In Workshop on Profile and Feedback-Directed Compilation.*

Jones, D. R., Perttunen, C. D. and Stuckman, B. E. (1993). Lipschitzian optimization without the lipschitz constant. *J. Optim. Theory Appl.*, 79, 157–181.

Jones, D. R., Schonlau, M. and Welch, W. J. (1998). Efficient global optimization of expensive black-box functions. *Journal of Global Optimization*, 13, 455–492.

Kleijnen, J. P. (2010). Design and analysis of computational experiments: Overview. T. Bartz-Beielstein, M. Chiarandini, L. Paquete and M. Preuss, editors, *Empirical Methods for the Analysis of Optimization Algorithms*, Heidelberg: Springer-Verlag. 51–72.

Kohonen, T., editor (1997). *Self-organizing maps.* Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Kohonen, T. (1998). The self-organizing map. *Neurocomputing*, 21, 1–6.

Kohonen, T. and Somervuo, P. (2002). How to make large self-organizing maps for nonvectorial data. *Neural Networks*, 15, 945–952.

Kokkolaras, M., Audet, C. and Dennis, Jr., J. E. (2001). Mixed variable optimization of the number and composition of heat intercepts in a thermal insulation system. *Optimization and Engineering*, 2, 5–29.

Kolda, T. G., Lewis, R. M. and Torczon, V. (2003). Optimization by direct search: new perspectives on some classical and modern methods. *SIAM Review*, 45, 385–482.

Lagarias, J. C., Reeds, J. A., Wright, M. H. and Wright, P. E. (1998). Convergence properties of the nelder-mead simplex method in low dimensions. *SIAM Journal of Optimization*, 9, 112–147.

Larson, J. L. and Sameh, A. H. (1980). Algorithms for roundoff error analysis—a relative error approach. *Computing*, 24, 275–297.

Lawson, C. L., Hanson, R. J., Kincaid, D. and Krogh, F. T. (1979). Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5, 308–323.

Le Digabel, S. (2009). NOMAD user guide. Les cahiers du GERAD G-2009-37, GERAD, Montréal, Canada.

Le Digabel, S. (2011). Algorithm 909: NOMAD: Nonlinear optimization with the MADS algorithm. *ACM Trans. Math. Softw.*, 37, 44:1–44:15.

Lourenço, H. R., Martin, O. C. and Stützle, T. (2010). Iterated local search: Framework and applications. M. Gendreau and J.-Y. Potvin, editors, *Handbook of Metaheuristics*, Springer US, Boston, MA, vol. 146 of *International Series in Operations Research & Management Science*, chapter 12. 363–397.

Milder, P. A. (2010). *A Mathematical Approach for Compiling and Optimizing Hardware Implementations of DSP Transforms.* PhD Thesis, Electrical and Computer Engineering, Carnegie Mellon University.

Miller, W. (1975). Software for roundoff analysis. *Transactions of the ACM on Mathematical Software*, 1, 108–128.

Moré, J. J. and Wild, S. M. (2009). Benchmarking derivative-free optimization algorithms. *SIAM J. Optimization*, 20, 172–191.

Nelder, J. A. and Mead, R. (1965). A simplex method for function minimization. *The Computer Journal*, 7, 308–313.

Orban, D. (2011). Templating and automatic code generation for performance with python. Technical report G-2011-30, Les cahiers du GERAD.

Pantazi, S., Kagolovsky, Y. and Moehr, J. R. (2002). Cluster analysis of wisconsin breast cancer dataset using self-organizing maps. *Stud Health Technol Inform*, 90, 431–436.

Preuss, M. and Bartz-Beielstein, T. (2007). Sequential parameter optimization applied to self-adaptation for binary-coded evolutionary algorithms. F. Lobo, C. Lima and Z. Michalewicz, editors, *Parameter Setting in Evolutionary Algorithms*, Springer. 91–119.

Püschel, M., Franchetti, F. and Voronenko, Y. (2011). *Encyclopedia of Parallel Computing*, Springer, chapter Spiral.

Püschel, M., Moura, J. M. F., Johnson, J., Padua, D., Veloso, M., Singer, B., Xiong, J., Franchetti, F., Gacic, A., Voronenko, Y., Chen, K., Johnson, R. W. and Rizzolo, N. (2005). SPIRAL: Code generation for DSP transforms. *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, 93, 232– 275.

Ridge, E. and Kudenko, D. (2007). Tuning the performance of the MMAS heuristic. *Proceedings of the 2007 international conference on Engineering stochastic local search algorithms: designing, implementing and analyzing effective heuristics.* Springer-Verlag, Berlin, Heidelberg, SLS'07, 46–60.

Seymour, K., You, H. and Dongarra, J. J. (2008). A comparison of search heuristics for empirical code optimization. *Proceedings of the 2008 IEEE International Conference on Cluster Computing.* Tsukuba International Congress Center, EPOCHAL TSUKUBA, Japan, Third international Workshop on Automatic Performance Tuning (iWAPT 2008), 421–429.

Smit, S. K. and Eiben, A. E. (2010). Using entropy for parameter analysis of evolutionary algorithms experimental methods for the analysis of optimization algorithms. T. Bartz-Beielstein, M. Chiarandini, L. Paquete and M. Preuss, editors, *Experimental Methods for the Analysis of Optimization Algorithms*, Springer Berlin Heidelberg, Berlin, Heidelberg, chapter 12. 287–310.

Taguchi, G., Chowdhury, S. and Wu, Y. (2004). *Taguchi's Quality Engineering Handbook.* Wiley-Interscience.

Torczon, V. (1997). On the Convergence of Pattern Search Algorithms. *SIAM Journal on Optimization*, 7, 1–25.

Vicente, L. N. and Custódio, A. L. (2012). Analysis of direct searches for discontinuous functions. *Math. Program.*, 133, 299–325.

Vuduc, R., Demmel, J. W. and Yelick, K. A. (2005). Oski: A library of automatically tuned sparse matrix kernels. *Journal of Physics: Conference Series*, 16, 521.

Wächter, A. and Biegler, L. T. (2006). On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Math. Program.*, 106, 25–57.

Whaley, R. C. and Dongarra, J. J. (1998). Automatically tuned linear algebra software. *Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM).* IEEE Computer Society, Washington, DC, USA, Supercomputing '98, 1–27.

Whaley, R. C., Petitet, A. and Dongarra, J. J. (2001). Automated empirical optimization of software and the atlas project. *Parallel Computing*, 27, 3–35.

Wild, S. M. (2008). Mnh: A derivative-free optimization algorithm using minimal norm hessians. Technical report ORIE-1466, Cornell University, School of Operations Research and Information Engineering. Submitted to the Tenth Copper Mountain Conference on Iterative Methods, January 2008.

Wild, S. M., Regis, R. G. and Shoemaker, C. A. (2008). ORBIT: optimization by radial basis function interpolation in trust-regions. *SIAM J. on Scientific Computing*, 30, 3197–3219.

Wild, S. M. and Shoemaker, C. A. (2011). Global convergence of radial basis function trust region derivative-free algorithms. *SIAM J. Optimization*, 21, 761–781.

Yotov (2006). *On the Role of Search in Generating High-Performance BLAS Libraries.* PhD Thesis, Cornell University.

Yuan, Z., Montes De Oca, M. A., Birattari, M. and Stützle, T. (2010). Modern continuous optimization algorithms for tuning real and integer algorithm parameters. *Proceedings of the 7th international conference on Swarm intelligence.* Springer-Verlag, Berlin, Heidelberg, ANTS'10, 203–214.

Appendix A

Case-study on tuning IPOPT parameters

In this appendix, OPAL is used together with a clustering tool to identify good algorithmic parameters at a reasonable cost. The target algorithm considered here is IPOPT, a nonlinear constrained optimization solver described by Wächter and Biegler (2006). In our setup, we analyze the effect of six parameters on the computational effort. The parameters, their bounds, type, default value, scale and context of utilization are summarized in Table A.1.

Table A.1 Six IPOPT parameters

Variable and bounds	Type	Default value	Scale	Context
$0 < \tau_{\min} < 1$	Real	0.99	0.05	fraction-to-boundary parameter update $\alpha_k^{\max} = \max\{\alpha \in (0,1] : x_k + \alpha d_k^x \geq (1-\tau_j)x_k\}$
$0 < s_\theta < \infty$	Real	1.1	5	switch condition in a search step $\alpha_{k,l}[-\nabla\varphi_{\mu_j}(x_k)^T d_k^x]^{s_\varphi} \leq \delta[\theta(x_k)]^{s_\theta}$
$0 < s_\varphi < \infty$	Real	2.3	5	
$0 < \delta < \infty$	Real	1.0	5	
$0 \leq p^{max} < \infty$	Integer	4	8	maximal number of second order corrections
$0 < \kappa_{soc} < 1$	Real	0.99	0.05	minimal reduction for second order correction step

In order to evaluate the quality of a prescribed set of parameters, IPOPT is launched on a testbed $\mathcal{L}$ containing a total of 730 test problems from the CUTEr collection (Gould *et al.*, 2003a). The application of IPOPT with the default parameter values reveals that:

- 11 problems return code 1 indicating that the algorithm did not converge to the desired tolerance levels, but produced a point satisfying other weaker tolerance levels.

– 25 problems return code 2, which means that the restoration phase converged to a minimizer for the constraint violation function, which is not feasible for the original problem. This suggests that the problems may be locally infeasible.
– 3 problems return code 4 indicating that the problems may be unbounded as the iterates grow unbounded.
– 3 problems return code 6, which means that the problem has as many equality constraints as free variables, and a feasible point was found.
– The remaining 688 problems are solved with return code 0, indicating that a locally optimal point within the desired tolerances was found.
– None of the problems returned a negative return code, which would indicate failure.

A.1 Direct Optimization of some IPOPT Parameters

The optimization problem considered by OPAL consists in minimizing the objective function $\psi_{\mathcal{L}}$:

$$\begin{aligned} &\psi_{\mathcal{L}}(p) = \sum_{\ell \in \mathcal{L}} \psi_\ell(p) \\ \text{where} \quad &\psi_\ell(p) := \mu_{\ell,p}^{\mathsf{FEVAL}} + \mu_{\ell,p}^{\mathsf{GEVAL}} + \mu_{\ell,p}^{\mathsf{EQCVAL}} + \mu_{\ell,p}^{\mathsf{INCVAL}} + \mu_{\ell,p}^{\mathsf{EQJVAL}} + \mu_{\ell,p}^{\mathsf{INJVAL}} \end{aligned}$$

subject to the constraint:

$$\varphi_{\mathcal{L}}(p) := \left|\{\ell \ : \ \mu_{\ell,p}^{\mathsf{ECODE}} < 0\}\right| \le 0$$

and where $\mu_{\ell,p}^{\mathsf{FEVAL}}, \mu_{\ell,p}^{\mathsf{GEVAL}}, \mu_{\ell,p}^{\mathsf{EQCVAL}}, \mu_{\ell,p}^{\mathsf{INCVAL}}, \mu_{\ell,p}^{\mathsf{EQJVAL}}$ and $\mu_{\ell,p}^{\mathsf{INJVAL}}$ represent the number of objective, gradient, equality constraints, inequality constraints, equality Jacobian and inequality Jacobian function evaluations, respectively, and where $\mu_{\ell,p}^{\mathsf{ECODE}}$ returns the exit code of solving problem ℓ using the parameter p. The constraint requires that all the test problems be solved by IPOPT. The constraints defining the domain $\mathbf{P}$ are simply the bounds and types reported in Table A.1. The objective function value with the default parameters p_0 is $\psi_{\mathcal{L}}(p_0) = 207866$.

The overall computing time required for solving this blackbox optimization problem with OPAL is 27h55m, and produces a set of parameters $\hat{p}$ with an objective function value of $\psi_{\mathcal{L}}(\hat{p}) = 198615$.

A.2 Combining OPAL with a clustering tool

An alternate and less expensive way to use the OPAL framework for this problem is by using clustering analysis on the cells of a self-organizing map based on the work of Kohonen (1998); Kohonen and Somervuo (2002) and Pantazi *et al.* (2002). The self-organizing map partitions the testbed into clusters sharing similar values of the objective and constraints. A representative problem from each cluster is identified by the clustering scheme, resulting in a subset $\mathcal{L}_1$ of 41 test problems from $\mathcal{L}$. OPAL is then launched on the minimization of $\psi_{\mathcal{L}_1}(p)$ subject to the same no-failure constraint. This surrogate problem is far easier to solve, as it requires only 4h17m and produces a solution p_1 which is close to $\hat{p}$.

Table A.2 lists the three solutions with their corresponding objective function values $\psi_{\mathcal{L}_1}(p)$ and $\psi_{\mathcal{L}}(p)$, together with the overall computational time required to generate them. The value $\psi_{\mathcal{L}}(p_1)$ is computed and inserted in the table for comparison purposes. The output codes generated by IPOPT on the entire testbed $\mathcal{L}$ with p_1 are identical to those produced by p_0 and $\hat{p}$.

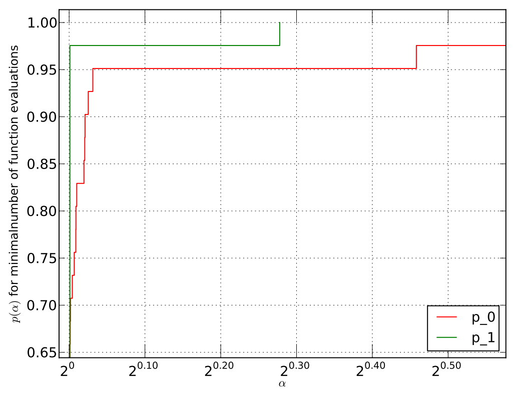
Table A.2 Default and optimized parameters for IPOPT

Solution	Parameter value						$\psi_{\mathcal{L}_1}(p)$	$\psi_{\mathcal{L}}(p)$	Time
p_0	0.99	1.1	2.3	1.0	4	0.99	51233	207866	-
$\hat{p}$	0.99	1.1	2.3	1.0	4	0.927548828125		198615	27h55m
p_1	0.99	1.1	2.3	1.0	4	0.94	49633	198663	4h17m

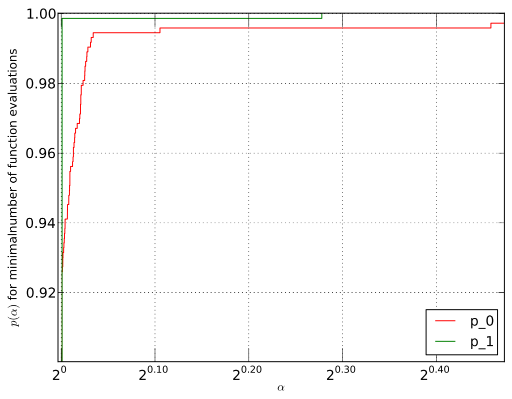
Inspection of the table reveals that the default IPOPT parameter values are well chosen. The only modification that the tests suggest is to slightly reduce the value of the parameter $\kappa_{soc} < 1$, used to determine the second order correction step constraint violation reduction. But with the value of $\kappa_{soc} = 0.94$ instead of 0.99, the number of function evaluations required by IPOPT drops by approximately 4.4% on the entire collection of 730 test problems, even if the optimization is conducted on a subset of only 41 problems.

Figure A.1 compares the two sets of parameters p_0 and p_1 from a different perspective. These performance profiles plot the proportion of problems solved with p_0 or p_1

within a factor of α (on the horizontal axis) of the best strategy. In both subplots the optimized parameters p_1 dominates the default ones p_0. The differences between the two parameter settings are more pronounced in subfigure (a). This is due to the fact that the optimization was conducted on the list $\mathcal{L}_1$. Figure (b) confirms that the combination of the clustering and self-organizing maps produced a representative subset of the collection of test problems.



(a) Reduced test problem set $\mathcal{L}_1$



(b) Original test problem set $\mathcal{L}$

Figure A.1 Performance profiles for the sets of parameters p_0 and p_1