GERAD

GROUPE D'ÉTUDES ET DE RECHERCHE
EN ANALYSE DES DÉCISIONS

Les Cahiers du GERAD

G-2012-08
March 2012

CITATION ORIGINALE / ORIGINAL CITATION

Audet, C., Dang, K.-C., Orban D.,
Optimization of algorithms with OPAL
Mathematical Programming
Computation, 6(3), 233-254, 2014.
Doi: 10.1007/s12532-014-0067-x

GERAD HEC Montréal
3000, ch. de la Côte-Sainte-Catherine
Montréal (Québec) Canada H3T 2A7

Tél. : 514 340-6053
Téléc. : 514 340-5665
info@gerad.ca
www.gerad.ca

Les Cahiers du GERAD ISSN: 0711–2440

Optimization of Algorithms with OPAL

C. Audet, K.-C. Dang, D. Orban

G–2012–08

March 2012

Optimization of Algorithms with OPAL

Charles Audet
Kien-Cong Dang
Dominique Orban

GERAD & Department of Mathematics and Industrial Engineering
École Polytechnique de Montréal
Montréal (Québec) Canada, H3C 3A7

charles.audet@gerad.ca
kien.cong.dang@gerad.ca
dominique.orban@gerad.ca

March 2012

Les Cahiers du GERAD
G–2012–08

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 handles a wide variety of parameter types, it can exploit multiple processors in parallel at different levels, and can take advantage of a surrogate blackbox.

Key Words: optimization, parameter optimization, automatic tuning.

Acknowledgments: The research is partially supported by the first author NSERC Discovery Grant 239436-05, Afosr FA9550-09-1-0160, and ExxonMobil Upstream Research Company EM02562. The research is partially supported by the third author NSERC Discovery Grant 299010-04.

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 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). 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., 2003) 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., 2009) on collections of unconstrained and constrained test problems. They introduce the first version of the OPAL package. Finally, Audet et al. (2011) 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 implementation 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. §2 describes a blackbox formulation of parameter-optimization problems. §3 describes the OPAL package, and illustrates its usage on well-known parameter optimization problems.

We conclude and look ahead in §4.

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 §2.2. A description of direct-search methods along with our method of choice are given in §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*.

2.1 Algorithmic Parameters

This 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 guaranteeting theoretical

[1] **OP**timization of **AL**gorithms

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 on/off parameters, 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 a binary parameter set at a particular value (say, *on*) is its complement (e.g., *off*). The situation may be substantially more complicated for general categorical parameters.

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 in 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, 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

$$\begin{array}{ll} \underset{p}{\text{minimize}} & \psi(p) \\ \text{subject to} & p \in \mathbf{P} \\ & \varphi(p) \in \mathbf{M}. \end{array} \tag{1}$$

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 (1) 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 (1). 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.

2.3 Blackbox Optimization by Direct Search

Opal allows the user to select a solver tailored to the parameter optimization problem (1). 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., 2009). 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 with either a failure or with 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. 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 the 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 (1) 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 upgraded 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 (1) 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 use 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 (2010) 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. 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,

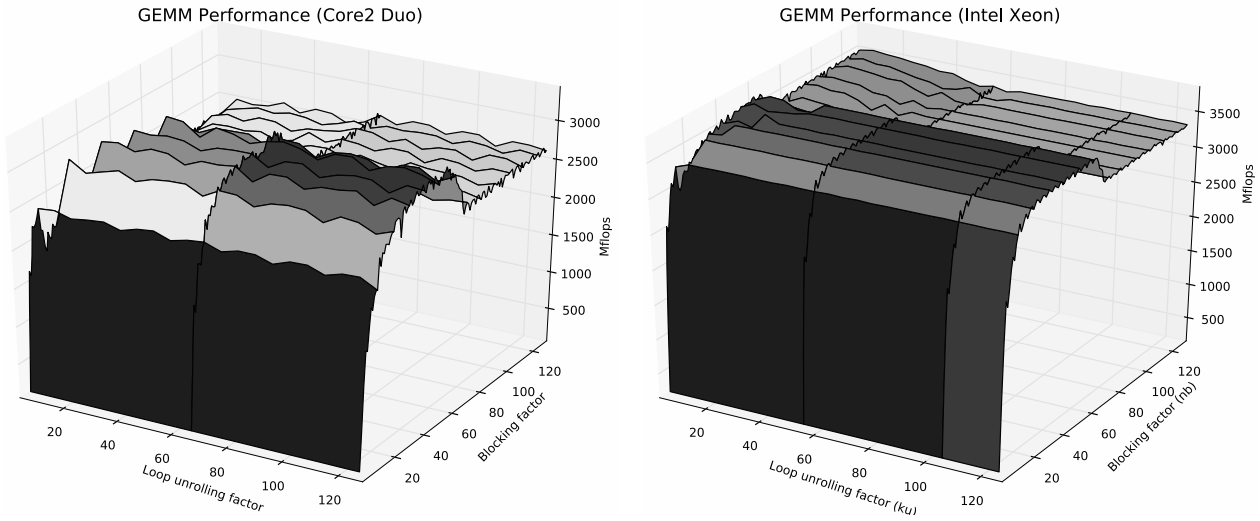

Figure 1: Performance in MFlops of a particular implementation of the matrix-matrix multiply as a function of the loop unrolling factor and the blocking factor.

the MADS framework provides a family of methods that have the potential to identify meaningful minimizers rather that just stationary points.

3 The OPAL Package

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

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 accomodate this diversity. It should also be understood that 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 (www.python.org) for the following reasons:

- Python is a rock-solid open-source scripting language. Python has been in constant developement 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 (www.scipy.org/numpy), an extension providing the *array* type and vector operations, Scipy (www.scipy.org), a general-purpose library of scientific extensions akin to Matlab toolboxes, and SAGE (www.sagemath.org), a symbolic computation package akin to Mathematica, to name only a few, as well as state-of-the art plotting packages such as Matplotlib (matplotlib.sf.net).
- 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 (www.enthought.com) and Python(x,y) (code.google.com/p/pythonxy) 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 resouces to get a glimpse of the expanse of Python-based research and projects is the Python Package Index webiste pypi.python.org/pypi.

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

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 (1) 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 (1) 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 1, which represents the contents of the *declaration file*. In Listing 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 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 the direct solver requests a run with given parameters, the executable command specified on line 6 of Listing 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

```
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', description='Error in derivative')
13 FD.add_measure(error)
```

Listing 1: `fd_declaration.py`: Declaration of the forward-difference algorithm

```
1  from opal.core.io import *
2  from fd import fd               # Target algorithm.
3  from math import pi, sin, cos
4
5  def run(param_file, problem):
6      "Run FD with given parameters."
7      params = read_params_from_file(param_file)
8      h = params['h']
9      return {'ERROR': abs(cos(pi/4) - fd(sin,pi/4,h))}
10
11 if __name__ == '__main__':
12     import sys
13     param_file  = sys.argv[1]
14     problem     = sys.argv[2]
15     output_file = sys.argv[3]
16
17     # Solve, gather measures and write to file.
18     measures = run(param_file, problem)
19     write_measures_to_file(output_file, measures)
```

Listing 2: `fd_run.py`: Calling the blackbox

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 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 (1) itself is modeled. OPAL separates the optimization problem into two components: the model *structure* and the model *data*. The *structure* represents the abstract problem (1) 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 3. The most important part of Listing 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.

```
1  from fd_declaration import FD
2  from opal import ModelStructure, ModelData, Model
3  from opal.Solvers import NOMADSolver
4
5  # Return the error measure.
6  def get_error(parameters, measures):
7      return sum(measures["ERROR"])
8
9  # Define parameter optimization problem.
10 data = ModelData(FD)
11 struct = ModelStructure(objective=get_error)  # Unconstrained
12 model = Model(modelData=data, modelStructure=struct)
13
14 # Create solver instance.
15 NOMAD = NOMADSolver()
16 NOMAD.solve(blackbox=model)
```

Listing 3: `fd_optimize.py`: Statement of the problem and solution

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), and are used by the solver as substitutes for the optimization problem. A fundamental property of surrogate problems is that their objective and constraints need to be less expensive to evaluate than the objective and constraints of (1). They need to share some similarities with (1), 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 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 starting point for (1). Secondly, both the surrogate and true optimization problems can be supplied to the blackbox solver, and the mechanisms in the solver decide which problem is solved. Surrogates are then 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 4 in the context of the trust-region algorithm examined by Audet and Orban (2006) and Audet et al. (2011). 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 4 we define two measures; ψ is represented by the function `sum_heval()` which computes the toal 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

```
1  from trunk_declaration import trunk            # Target algorithm.
2  from opal import ModelStructure, ModelData, Model
3  from opal.Solvers import NOMADSolver
4  from opal.TestProblemCollections import CUTEr  # The CUTEr test set.
5
6  def sum_heval(parameters, measures):
7      "Return total number of Hessian evaluation across test set."
8      return sum(measures["HEVAL"])
9
10 def get_error(parameters,measures):
11     "Return number of nonzero error codes (failures)."
12     return len(filter(None, measures['ECODE']))
13
14 cuter_unc = [p for p in CUTEr    if p.ncon == 0]   # Unconstrained problems.
15 smaller   = [p for p in problems if p.nvar <= 100] # Smaller problems.
16
17 # Define (constrained) parameter optimization problem.
18 data   = ModelData(algorithm=trunk, problems=cuter_unc)
19 struct = ModelStructure(objective=sum_heval, constraints=[(None,get_error,0)])
20 model  = Model(modelData=data, modelStructure=struct)
21
22 # Define a surrogate (unconstrained).
23 surr_data   = ModelData(algorithm=trunk, problems=smaller)
24 surr_struct = ModelStructure(objective=sum_heval)
25 surr_model  = Model(modelData=surr_data, modelStructure=surr_struct)
26
27 NOMAD = NOMADSolver()
28 NOMAD.solve(blackbox=model, surrogate=surr_model)
```

Listing 4: Definition of a surrogate model.

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

Listing 5: Example use of categorical variables in OPAL

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.

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.

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 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 § 2.3 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 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 variablesgalters the number of optimization variables of the problem.

3.5 Parallelism at Different Levels

OPAL can exploit architectures with several processors or several cores at different levels. Audet et al. (2011) 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 (1) and consists in launching the target algorithm to solve test problems concurrently, and the third simultaneously 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.

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. Dang (2012) 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. (2003)). 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.

Dang uses clustering 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 $\hat{p}$.

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, quadratic model surrogates can be generated automatically (Conn and Le Digabel, 2011).

NOMAD handles all the variable types enumerated in §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.

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.

References

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

M.A. Abramson and C. Audet. Convergence of mesh adaptive direct search to second-order stationary points. *SIAM Journal on Optimization*, 17(2):606–619, 2006. DOI: 10.1137/050638382.

J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, and S. Amarasinghe. Petabricks: A language and compiler for algorithmic choice. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, Dublin, Ireland, Jun 2009.

C. Audet and J. E. Dennis, Jr. Analysis of generalized pattern searches. *SIAM Journal on Optimization*, 13(3): 889–903, 2003. DOI: 10.1137/S1052623400378742.

C. Audet and J. E. Dennis, Jr. Mesh adaptive direct search algorithms for constrained optimization. *SIAM Journal on Optimization*, 17(1):188–217, 2006. DOI: 10.1137/040603371.

C. Audet and J. E. Dennis, Jr. A progressive barrier for derivative-free nonlinear programming. *SIAM Journal on Optimization*, 20(4):445–472, 2009. DOI: 10.1137/070692662.

C. Audet and D. Orban. Finding optimal algorithmic parameters using derivative-free optimization. *SIAM Journal on Optimization*, 17(3):642–664, 2006. DOI: 10.1137/040620886.

C. Audet, C.-K. Dang, and D. Orban. Algorithmic parameter optimization of the DFO method with the OPAL framework. In J. Cavazos K. Naono, K. Teranishi and R. Suda, editors, *Software Automatic Tuning: From Concepts to State-of-the-Art Results*, pages 255–274. Springer, New-York, NY, 2010a.

C. Audet, J. E. Dennis, Jr., and S. Le Digabel. Globalization strategies for mesh adaptive direct search. *Computational Optimization and Applications*, 46(2):193–215, June 2010b. DOI: 10.1007/s10589-009-9266-1.

C. Audet, C.-K. Dang, and D. Orban. Efficient use of parallelism in algorithmic parameter optimization applications. *Optimization Letters*, pages 1–13, 2011. DOI: s11590-011-0428-6. Online First.

J. Bilmes, K. Asanović, C.-W. Chin, and J. Demmel. The PHiPAC v1.0 matrix-multiply distribution. Technical Report TR-98-35, International Computer Science Institute, CS Division, University of California, Berkeley CA, 1998.

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

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

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

A. R. Conn and S. Le Digabel. Use of quadratic models with mesh adaptive direct search for constrained black box optimization. Technical Report G-2011-11, Les cahiers du GERAD, 2011. To appear in Optimization Methods and Software.

A. R. Conn, K. Scheinberg, and L. N. Vicente. *Introduction to Derivative-Free Optimization*. MPS/SIAM Book Series on Optimization. SIAM, Philadelphia, 2009. ISBN 978-0-898716-68-9.

C.-K. Dang. *Optimization of Algorithms with the OPAL Framework*. Ph.D. Thesis, École Polytechnique de Montréal, Montréal, Québec, Canada, 2012.

E. Fermi and N. Metropolis. Numerical solution of a minimum problem. Los Alamos Unclassified Report LA–1492, Los Alamos National Laboratory, Los Alamos, USA, 1952.

N. I. M. Gould, D. Orban, and Ph.L. Toint. CUTEr (and SifDec): a Constrained and Unconstrained Testing Environment, revisited. *ACM Transactions on Mathematical Software*, 29(4):373–394, 2003. DOI: 10.1145/962437.962439.

N. I. M. Gould, D. Orban, A. Sartenaer, and Ph. L. Toint. Sensitivity of trust-region algorithms on their parameters. *4OR*, 3(3):227–241, 2005.

R. B. Gramacy and S. Le Digabel. The mesh adaptive direct search algorithm with treed gaussian process surrogates. Technical Report G-2011-37, Les cahiers du GERAD, 2011.

R. Hooke and T. A. Jeeves. Direct search solution of numerical and statistical problems. *Journal of the Association for Computing Machinery*, 8(2):212–229, 1961. DOI: 10.1145/321062.321069.

T. Kohonen. The self-organizing map. *Neurocomputing*, 21(1-3):1–6, November 1998. ISSN 09252312. DOI: 10.1016/S0925-2312(98)00030-7.

T. Kohonen and P. Somervuo. How to make large self-organizing maps for nonvectorial data. *Neural Networks*, 15(8-9):945–952, 2002. DOI: 10.1016/S0893-6080(02)00069-2.

M. Kokkolaras, C. Audet, and J. E. Dennis, Jr. Mixed variable optimization of the number and composition of heat intercepts in a thermal insulation system. *Optimization and Engineering*, 2(1):5–29, 2001.

S. Le Digabel. Algorithm 909: NOMAD: Nonlinear optimization with the MADS algorithm. *ACM Transactions on Mathematical Software*, 37(4):44:1–44:15, 2011. DOI: 10.1145/1916461.1916468.

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

S. Pantazi, Y. Kagolovsky, and J. R. Moehr. Cluster analysis of wisconsin breast cancer dataset using self-organizing maps. *Stud Health Technol Inform*, 90:431–436, 2002. ISSN 0926-9630.

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

V. Torczon. On the convergence of pattern search algorithms. *SIAM Journal on Optimization*, 7(1):1–25, 1997. DOI: 10.1137/S1052623493250780.

L.N. Vicente and A.L. Custódio. Analysis of direct searches for discontinuous functions. *To appear in Mathematical Programming*, 2010. DOI: 10.1007/s10107-010-0429-8.

R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Proceedings of SciDAC 2005*, Journal of Physics: Conference Series, San Francisco, CA, USA, June 2005. Institute of Physics Publishing.

A. Wächter and L. T. Biegler. On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Math. Program.*, 106(1):25–57, 2006.

R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.