Math. Prog. Comp. (2014) 6:233–254
DOI 10.1007/s12532-014-0067-x

FULL LENGTH PAPER

Optimization of algorithms with OPAL

Charles Audet · Kien-Cong Dang · Dominique Orban

Received: 19 December 2012 / Accepted: 3 February 2014 / Published online: 6 March 2014

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. Several features of OPAL are illustrated on a problem consisting in the design of a hybrid sort strategy.

Keywords Nonsmooth optimization · Parameter optimization · Autotuning

Mathematics Subject Classification 65K05 · 90C56 · 90C90

Research partially supported by NSERC Discovery Grant 239436, Afosr FA9550-09-1-0160, and ExxonMobil Upstream Research Company EM02562.

Research partially supported by NSERC Discovery Grant 299010-04.

C. Audet · D. Orban (✉)
GERAD and Department of Mathematics and Industrial Engineering, Ecole Polytechnique, Montreal, QC, Canada
e-mail: dominique.orban@gerad.ca

C. Audet
e-mail: charles.audet@gerad.ca

K.-C. Dang
GERAD, Montreal, QC, Canada
e-mail: kien.cong.dang@gerad.ca

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., [14,35,38,40]. By contrast, OPAL uses an optimization method supported by a strong convergence theory, yielding solutions that are local minimizers in a meaningful sense. We illustrate the modeling facilities of OPAL on the problem of designing a hybrid search strategy able to outperform a prescribed set of standard sorts. Our results show that after a moderate number of blackbox evaluations, OPAL identifies a search strategy that differs from the preset sorts and strictly dominates all of them. This problem also provides an elegant model with a single categorical variable. OPAL is open-source software available from `dpo.github.com/opal`.

In [12], we study the four standard parameters of a trust region algorithm [25] for unconstrained nonlinear optimization. In particular, we study the question of minimizing the overall CPU time required to solve 55 test problems of moderate size from the CUTEr [24] 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) [9] 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 MADSalgorithm. The numerical experiments lead to a 25 % computing time reduction compared to the default parameters.

In [5], we extend the framework to make it more configurable, use it to tune parameters of the DFO algorithm [19] on collections of unconstrained and constrained test problems, and introduce the first version of the OPAL package. Finally, in [6], we illustrate usage of parallelism at various levels within the OPAL framework and its impact on the 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.

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

Section 2 describes a blackbox formulation of parameter-optimization problems. Section 3 describes the OPAL package, and Sect. 4 illustrates its usage the on well-known parameter-optimization problem of designing an efficient hybrid sort strategy. We conclude and look ahead in Sect. 5.

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

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 guaranteeting 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.

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. If the blackbox required to evaluate the objective or a constraint stalls, crashes or otherwise fails, an infinite value is assigned to this problem function.

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 interval of $\mathbb{R}$. Examples of such parameters include the step acceptance threshold in a trust-region method [12,25] 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 to be used in the construction of a heat shield (e.g., material A, B or C) [30], 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

$$\underset{p}{\text{minimize}}\ \psi(p) \quad \text{subject to}\ p \in \mathbf{P},\ \varphi(p) \in \mathbf{M}. \tag{1}$$

The set **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 **M** constrains the values of composite measures. OPAL allows virtually any composite measure as objective or 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 [19]. 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 **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 a case where the blackbox is an optimization solver and the input data consists in the entire 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}. The authors of [22] used this algorithm on one of the first digital computers.

This simple coordinate search algorithm was generalized in [36] to a broader framework of *pattern-search* methods, which includes the methods of [16] and [27]. Pattern-search methods introduce more flexibility in the construction of trial parameters and in the step size. Convergence analysis of pattern-search methods appears in [36] for unconstrained $\mathcal{C}^2$ functions, and the analysis was upgraded to nonsmooth functions in [8] using the Clarke generalized calculus [17]. Pattern-search methods were subsequently further generalized in [9] and [10] to handle general constraints in a way that is satisfactory in theory and in practice. The resulting method is the Mesh-Adaptive *Direct-Search* algorithm (MADS). It can be used on 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, MADS uses a mesh structure consisting of an implicit discretization of the search 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 a set of 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 $\hat{p}$.

In the smooth case, one expects the gradient of the objective function ψ at $\hat{p}$ to vanish when the problem is unconstrained, or all directional derivatives of ψ at $\hat{p}$ in feasible directions to be nonnegative in the presence of constraints. But in the nonsmooth case, the gradient, the directional derivatives and the cone of feasible directions may be undefined, or difficult to use within a rigorous convergence analysis. By using the nonsmooth calculus of [17], notions such as directional derivatives ψ' may be extended to generalized directional derivatives ψ° and tangent cones are generalized to hypertangent and contingent cones. These extensions lead to the following main convergence results for MADS

– $\hat{p}$ is the limit of mesh local optimizers on meshes that become 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 in [9] was augmented in [1] to the second-order and in [37] to discontinuous functions.

The authors of [10] propose a method to handle constraints that allows an infeasible starting point. In the situation where the limit point $\hat{p}$ is infeasible with respect to the composite measure, the method ensures that the above optimality conditions are satisfied for the problem

$$\underset{p}{\text{minimize}}\ h(p) \quad \text{subject to}\ \ p \in \mathbf{P},$$

where $h(p)$ is the constraint aggregation function of [23] and measures deviation from feasibility with respect to the general constraints $\varphi(p) \in \mathbf{M}$.

As an example of blackbox optimization, consider 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 [40]. 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

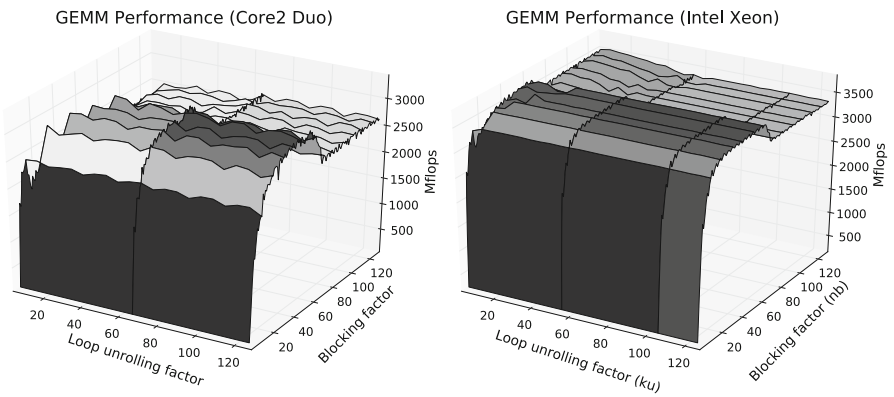

Fig. 1 Performance in MFlops of a particular implementation of the matrix–matrix multiply as a function of the loop unrolling factor and the blocking factor

noise, but has a certain regularity. In this sense, the MADSframework provides a family of methods that have the potential to identify meaningful minimizers rather than just stationary points.

Note that a method of the MADSframework yields reproducible results provided that the objective and constraint functions are not multi-valued or stochastic—such as would be the case with, e.g., CPU time measurements. In addition, the flexible MADSframework lends itself to global optimization extensions such as those based on the Variable Neighborhood Search [4,32]. Because such extensions are typically costly in function evaluations, they are disabled by default in the framework of Sect.3.

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 solid open-source scripting language. Running Python programs does not involve a compiler.
- Python is available on almost any platform.
- Python interoperates well with many other languages.
- Users can write Python programs much in the same way as shell scripts, or elect to use the full power of object-oriented programming.
- A wide range of numerical and scientific extensions is available for Python.
- Aside from scientific capabilities, Python is a full-fledged programming language with an extensive standard library.
- The Python syntax is human readable. A beginner 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 profuse documentation and ressources.

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 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.

Note that item 3 reflects the typical use case of algorithmic optimization: that of a target algorithm given as an opaque executable blackbox. If necessary, this executable should be wrapped in an outer layer whose role is to ensure smooth communication with OPAL, i.e., to read parameter values from file, pass them to the executable and write atomic measures to file.

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.

```
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

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 $\psi(h) = \left|\frac{1}{h}\big(\sin(\pi/4 + h) - \sin(\pi/4)\big) - \cos'(\pi/4)\right|$. 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, relevant modules are imported in lines 1–3, 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 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. In Python syntax, lines 11–19 are executed when the run file is called as an executable. 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. Typically, only lines 6–9 change across run files.

```
1  from opal.core.io import read_params_from_file, write_measures_to_file
2  from fd import fd               # Target algorithm.
3  from math import pi, sin, cos   # Used to measure errors.
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

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*, materialized by the `ModelStructure` class, 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*, materialized by the `ModelData` class, 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. Both are combined into a *model* by way of the `Model` class. 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.

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 in [15], 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.

```
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 measures["ERROR"]
8
9  # Define parameter optimization problem.
10 data = ModelData(FD)
11 struct = ModelStructure(objective=get_error)  # Unconstrained by default.
12 model = Model(modelData=data, modelStructure=struct)
13
14 # Create solver instance and solve.
15 NOMAD = NOMADSolver()
16 NOMAD.solve(blackbox=model)
```

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

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 MADSframework, local quadratic surrogates are proposed in [18] and global treed Gaussian process surrogates in [26].

In OPAL, surrogates are typically used in two ways. Firstly, OPAL can use a surrogate problem as if it were the true optimization problem. 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 in [12]. 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 in [12] and [6]. 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.

```
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 cuter_unc 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.

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 $\varphi(p) \leq 0$, 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 and that there are no constraints. 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.

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 [33] uses categorical variables to represent a loop order parameter and compiler options in a standard matrix multiply.

The authors of [3] discuss strategies to select the best sort 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 sort algorithm to use. Listing 5 gives the OPAL declaration of a simplistic categorical parameter representing the choice of a sort strategy. Note however that the ultimate goal of [3] 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.

```
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

MADSeasily handles integer variables by exploiting their inherent ordering. This is done by making sure that the step size parameter Δ mentioned in Sect. 2.3 is integer. Furthermore, a natural stopping criterion triggers when an iteration fails to improve p^{best} with a unit step size.

Categorical variables cannot be handled as easily as integer variables. They do not posses any ordering properties, and therefore need to be accompanied by a neighborhood structure, such as the one illustrated in Listing 5. In the example, insertion sort has quicksort as sole neighbor. In turn, quicksort has three neighbors: insertion sort, radix sort and merge sort. This neighborhood structure may be thought of as a directed graph where arcs indicate neighbors–see Fig. 2. Each iteration of the MADSalgorithm constructs two sets of tentative trial parameters. One set keeps the categorical variables fixed and modifies only the continuous and integer variables using the same technique as when categorical variables are not present. The other set is constructed using the user-provided

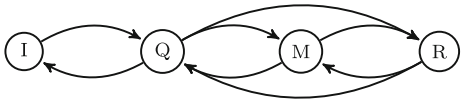

Fig. 2 Oriented graph illustration of the neighborhood structure of Listing 5. *I* stands for insertion sort, *Q* for quicksort, *M* for merge sort and *R* for radix sort.

set of categorical neighbors. Categorical variables have lower priority in the sense that an improving iterate in continuous or integer space will be accepted before an improving iterate in categorical space. A precise description of how this is accomplished for the pattern search algorithm is presented in [2], and the method is illustrated in [30] on an optimization problem where the neighborhood structure is such that changes in some of the categorical variables alters the number of optimization variables of the problem. It is important to note that the neighborhood structure accompanying categorical variables influences the search, and therefore the solution eventually identified.

In Sect. 4, we revisit the example of Listing 5 and propose a model based on a single categorical variable that does not alter the problem dimension.

3.5 Parallelism at different levels

OPAL can exploit architectures with several processors or several cores at different levels. In [6], we compare three ways of using parallelism within OPALThe first strategy consists in the blackbox solver evaluating the quality of trial parameters in parallel, the second exploits the structure of (1) and consists in launching the target algorithm to solve test problems concurrently, and the third 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 [20] considers the optimization of six real parameters from IPOPT, a nonlinear constrained optimization solver described in [39]. 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 [24]. 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 concurrent 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 [28,29,34]. 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 NOMAD [31]. It is a robust code, implementing the MADS algorithm for nonsmooth constrained optimization. NOMAD can be used in conjunction with a surrogate optimization problem. Among others, quadratic model surrogates can be generated automatically [18].

NOMAD handles all the variable types enumerated in Sect. 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, i.e., the objective function ψ is replaced with $\hat{\psi}(p)$, which takes the value $\psi(p)$ if p is feasible or $+\infty$ otherwise. 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 in [11], together with applications to engineering blackbox problems.

4 An application: the cooperative sort

The objective of this section is to illustrate the use of OPAL on a convenient model of the hybrid sort algorithm example using a surrogate and a single categorical variable. The elegant model used below removes the difficulty related to the varying dimensionality alluded to earlier.

4.1 Three sorts

Numerous algorithms can be used to sort lists of numbers. For the purpose of our illustration, we limit ourselves to three well-known methods. The plot to the left of Fig. 3 shows the computational time in seconds required to sort lists of integers of length up to 10,000 using recursive implementations of merge sort, quick sort and radix sort. The computational time is computed as the average of 100 repeats. Each list is generated by rounding numbers drawn from a normal distribution with mean 2,000 and standard deviation 25. The histogram of one such list is presented in the right part of Fig. 3. Note that with this specific distribution, no sort dominates the others over all list

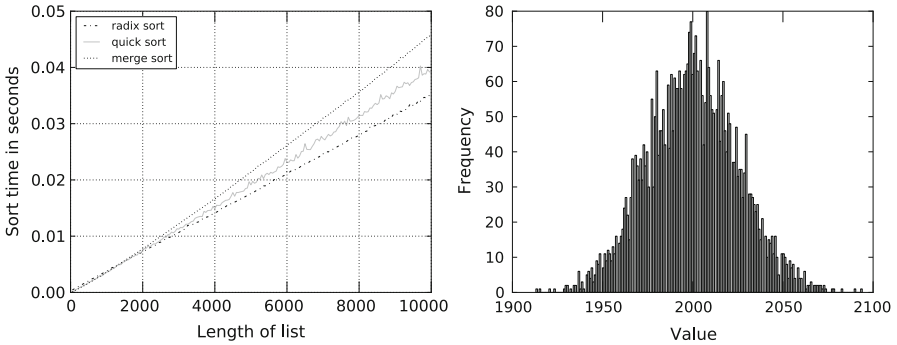

Fig. 3 Time to sort 100 lists, and histogram of one such list

lengths. Radix sort is the best for short-sized lists, but the worst for large ones. This suggests that there might be room for improvement by combining the sort strategies.

4.2 A model using a categorical variable

We construct a hybrid algorithm called Coop Sort based on the three sorts illustrated in Fig. 3 exploiting the fact that several sorts proceed in three steps. Firstly, the input list is partitioned into two sublists. Secondly, each sublist is recursively sorted. Finally, the two sorted sublists are merged together. Quick sort and merge sort partition differently but both recursively call themselves on the sublists. However, any other sort algorithm could be used on the sublists. Consider the two following strategies to partition a given list of length ℓ into two sublists A and B:

- Partitioning I consists in defining A to be the $\lceil \frac{\ell}{2} \rceil$ first elements of the list, and to place the remaining ones in B
- Partitioning II consists in reading the first element a of the list, and to place in A every element whose value is less than or equal to a, and the others in B.

Partitioning II, typically used in quick sort, is obviously more expensive than Partitioning I, but subsequently, merging the sublists A and B after they have been sorted is trivial. The merge associated to partitioning I can be performed in linear time. Our hybrid sort can be compactly modeled by a string of integers. We first assign integer values to the two partitioning strategies and to the three sorts as shown in Table 1.

Table 1 Numerical codes associated to sorts and partitioning strategies.

Code	Strategy
1	Merge sort
2	Quick sort
3	Radix sort
4	Partitioning I
5	Partitionning II

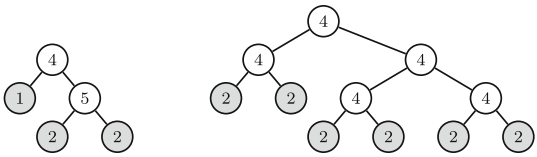

Fig. 4 Tree representation of the two hybrid sorts 41522 and 44224422422

A categorical variable can be used to represent any hybrid strategy. The categorical variable is an integer whose digits are between 1 and 5, and represents the nodes of a binary tree in *preorder,* i.e., as explored by a depth-first search. Leaves correspond to one of the three sorts 1, 2 or 3, and all other nodes are either 4 or 5. Figure 4 illustrates two different hybrid sorts. The one on the left corresponds to partitioning the list using partitioning I. The first sublist is processed by merge sort, and the second uses partitioning II. Both remaining partitions are processed by quick sort.

4.3 Neighborhood of a categorical variable

In the presence of categorical variables, NOMAD requires that the optimization problem be accompanied by a neighborhood structure, indicating how to change the value of the categorical variable in a way that increases the chances of achieving descent. In our implementation, a given hybrid sort modeled as a binary tree possesses the neighbors defined by the following four rules:

1. (Change a sort) a leaf $a \in \{1, 2, 3\}$ is replaced by another leaf $b \in \{1, 2, 3\}$ with $b \neq a$
2. (Change a partition) a node $a \in \{4, 5\}$ is replaced by its complement in $\{4, 5\}$
3. (Increase recursion) a leaf $a \in \{1, 2, 3\}$ is replaced by a node $b \in \{4, 5\}$ whose children are the two leaves c and d such that $c < d$ and $\{a, c, d\} = \{1, 2, 3\}$
4. (Decrease recursion) a node $a \in \{4, 5\}$ with two children $b, c \in \{1, 2, 3\}$ is replaced by a single leaf $d \in \{1, 2, 3\}$ with $d \neq b$ and $d \neq c$.

The motivation behind this neighborhood structure is to favor variety in the trees generated during an exploration.

We illustrate the above four rules by computing the neighbors of the leftmost tree in Fig. 4. We underline the node modified. The first rule yields the neighbors

$$\{4\underline{2}522, 4\underline{3}522, 415\underline{1}2, 415\underline{3}2, 4152\underline{1}, 4152\underline{3}\}.$$

The second rule yields

$$\{\underline{5}1522, 41\underline{4}22\}.$$

The third rule yields

$$\{4\underline{4}23522, 4\underline{5}23522, 415\underline{4}132, 415\underline{5}132, 4152\underline{4}13, 4152\underline{5}13\}.$$

Finally, the last rule yields the neighbors

$$\{41\underline{1}, 41\underline{3}\}.$$

Clearly, more elaborate rules can be devised by, e.g., allowing to change several nodes at a time but the above simple rules provide sufficient flexibility for our purposes.

4.4 An optimization problem with a surrogate

In (1), p is the categorical variable describing a hybrid sort, $\mathbf{P}$ is the set of all valid strings of digits between 1 and 5, our objective $\psi(p)$ is the average time required to sort a randomly-generated list over 100 repeats, as detailed in Sect. 4.1, and there are no general constraints $\varphi(p)$. Instead of generating a random list once and for all, the list changes at each function evaluation. As a consequence, our objective function is not deterministic and this should help ensure that any improving hybrid sort can be expected to perform well on average on such randomly-generated lists. This also helps illustrate how a method of the MADSfamily of algorithms works on this type of non-deterministic—or noisy—objective.

At a given iterate, the order in which the neighbors are evaluated may influence the overall optimization. We order those neighbors using a surrogate model that consists in computing the average time required to sort a list of similarly-generated random numbers over 10 repeats. The objective $\psi(p)$ is then first evaluated at neighbors sorted faster by the surrogate model, i.e., the most promising ones. The cost of evaluating this surrogate model is thus approximately one tenth of that to evaluate the actual objective function.

4.5 Numerical results

Figure 3 shows that, among our selection of sorts, the best method to sort lists of size 4,000 is radix sort ($p = 3$). In order to challenge the optimizer, we choose $p = 5522522$ as the starting solution. Notice that this is equivalent to $p = 522$ and to $p = 2$, but possesses a richer neighborhood.

OPAL produces the solution $\hat{p} = 44224422422$ after a total of 55 evaluations of ψ and 349 evaluations of the surrogate model. By taking into account the fact that a call to the surrogate is approximately ten times faster than a call to the original objective, it follows that the total computational effort required by the optimization of the blackbox problem is approximately equivalent to 90 objective evaluations. This proposed strategy, illustrated in the right part of Fig. 4, consists in applying quick sort on lists that are recursively cut in two using the partitioning strategy II and is not identical to using quick sort on the original list.

To measure the performance of this hybrid sort, we compare $\hat{p}$ to the three initial sorts on lists of length up to 10,000. The left plot in Fig. 5 illustrates the fact that $\hat{p}$ strictly dominates any of the other three sorts, not only on lists of length 4,000 but on all such list lengths—a lucky but welcome bonus! Note that the plot is not identical to that of Fig. 3 since new random lists were generated.

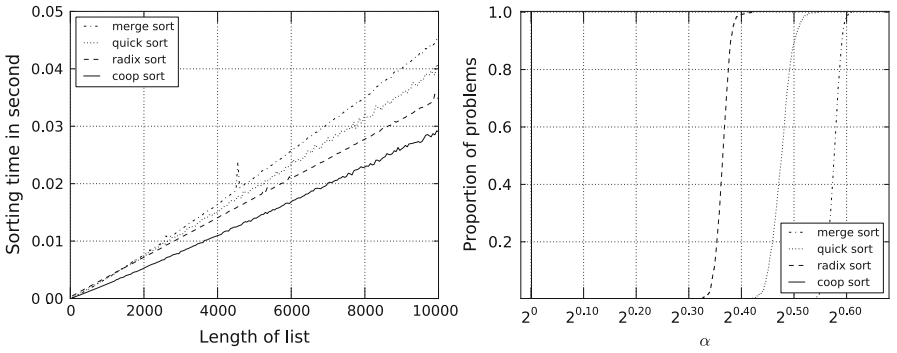

Fig. 5 Comparison of the hybrid Coop sort with merge, quick and radix sort

As a cross validation, we sort 10,000 lists of size 4,000 with each of the four sorts and record the computational time required by each sort on each list. The right part of Fig. 5 presents the results in the form of a performance profile [21] that indicates that $\hat{p}$ dominates the other three sorts. The horizontal axis of the profile is a dimensionless factor. A point on a curve, say the quick sort curve, with x-coordinate α and y-coordinate t indicates that on 100 % of the lists, quick sort is within a factor α of the best sort. As the curve corresponding to coop sort is essentially a vertical line at abscissa 1, it consistently outperforms the other three sorts.

We repeated the experiment above with radix sort replaced with heapsort. Starting from $p = 44114411411$, OPAL produces $\hat{p} = 44552422224423554523253 44$ after 81 black box evaluations and 729 surrogate evaluations. This solution is no longer a variation on the quick sort, features both types of partitioning and mixes the quick sort with the radix sort.

We note that this example problem and sample results are by no means an exhaustive exploration of the hybrid sort problem, not even among the three sorts considered. For instance, our pivot choice in quick sort is simply the first element. Other choices are possible and influence the performance of quick sort. Similarly, whether we use a sequential or recursive implementation of quick sort matters—our tests use a sequential implementation. The stable sort used inside radix sort is also important—our tests use a simple count sort. Nevertheless, the fact that our results are conclusive suggests that, by considering additional types of sorts along with competing implementations and by defining an appropriate neighborhood structure, it should be possible to reach similar conclusive results. In addition, the modeling is not more complicated; all that is required is a richer set of symbols in Table 1.

5 Discussion

In designing the OPAL framework, our goal is to provide users with a modeling environment that provides a reasonable balance of ease of use and power given the complexity of problems that it allows users to model, while relying on a state-of-the-art blackbox optimization solver. It is difficult to say if the performance of an algorithm depends continuously on its (real) parameters. As parameters may 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.

OPAL has been used in several types of applications, including code generation for high-performance linear algebra kernels and 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.

Algorithmic parameter optimization problems may be naturally formulated as multiobjective problems—e.g., minimize the total CPU time and the memory consumption of an application. The extension of MADSto biobjective optimization proposed in [13] is featured in NOMAD and could also be used as suggested in [7] to perform trade-off studies between various constraints. Exploiting such capabilities is left for future research.

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

1. Abramson, M.A., Audet, C.: Convergence of mesh adaptive direct search to second-order stationary points. SIAM J. Optimization **17**(2), 606–619 (2006). doi:10.1137/050638382
2. Abramson, M.A., Audet, C., Dennis, J.E Jr.: Filter pattern search algorithms for mixed variable constrained optimization problems. Pac. J. Optimization **3**(3), 477–500 (2007)
3. Ansel, J., Chan, C., Wong, Y.L., Olszewski, M., Zhao, Q., Edelman, A., Amarasinghe, S.: Petabricks: A Language and Compiler for Algorithmic Choice. In: ACM SIGPLAN Conference on Programming Language Design and Implementation. Dublin, Ireland (2009). doi:10.1145/1542476.1542481
4. Audet, C., Béchard, V., Le Digabel, S.: Nonsmooth optimization through mesh adaptive direct search and variable neighborhood search. J. Glob. Optimization **41**(2), 299–318 (2008). doi:10.1007/s10898-007-9234-1
5. Audet, C., Dang, C.K., Orban, D.: Algorithmic parameter optimization of the DFO method with the OPAL framework. In: Naono, J.C.K., Teranishi, K., Suda, R. (eds.) Software Automatic Tuning: From Concepts to State-of-the-Art Results, pp. 255–274. Springer, New York (2010). doi:10.1007/978-1-4419-6935-4_15
6. Audet, C., Dang, C.K., Orban, D.: Efficient use of parallelism in algorithmic parameter optimization applications. Optimization Lett. **7**(3), 421–433 (2013). doi:10.1007/s11590-011-0428-6
7. Audet, C., Dennis, J. Jr.: Le Digabel, S.: Trade-off studies in blackbox optimization. Optimization Methods Softw. **27**(4–5), 613–624 (2012). doi:10.1080/10556788.2011.571687
8. Audet, C., Dennis, J.E Jr.: Analysis of generalized pattern searches. SIAM J. Optimization **13**(3), 889–903 (2003). doi:10.1137/S1052623400378742
9. Audet, C., Dennis, J.E Jr.: Mesh adaptive direct search algorithms for constrained optimization. SIAM J. Optimization **17**(1), 188–217 (2006). doi:10.1137/040603371
10. Audet, C., Dennis, J.E Jr.: A progressive barrier for derivative-free nonlinear programming. SIAM J. Optimization **20**(1), 445–472 (2009). doi:10.1137/070692662
11. Audet, C., Dennis, J.E Jr.: Digabel, S.L.: Globalization strategies for mesh adaptive direct search. Comput. Optimization Appl. **46**(2), 193–215 (2010). doi:10.1007/s10589-009-9266-1

12. Audet, C., Orban, D.: Finding optimal algorithmic parameters using derivative-free optimization. SIAM J. Optimization **17**(3), 642–664 (2006). doi:10.1137/040620886
13. Audet, C., Savard, G., Zghal, W.: Multiobjective optimization through a series of single-objective formulations. SIAM J. Optimization **19**(1), 188–210 (2008). doi:10.1137/060677513
14. Bilmes, J., Asanović, K., Chin, C.W., Demmel, J.: The PHiPAC v1.0 matrix-multiply distribution. Technical Report TR-98-35, International Computer Science Institute, CS Division, University of California, Berkeley (1998)
15. Booker, A.J., Dennis, J.E Jr.: Frank, P.D., Serafini, D.B., Torczon, V., Trosset, M.W.: A rigorous framework for optimization of expensive functions by surrogates. Struct. Optimization **17**(1), 1–13 (1999). doi:10.1007/BF01197708
16. Box, G.E.P.: Evolutionary operation: a method for increasing industrial productivity. Appl. Stat. **6**(2), 81–101 (1957)
17. Clarke, F.H.: Optimization and Nonsmooth Analysis. Reissued in 1990 by SIAM Publications, Philadelphia, as, Vol. 5 in the series Classics in Applied Mathematics. Wiley, New York (1983)
18. Conn, A.R., Le Digabel, S.: Use of quadratic models with mesh-adaptive direct search for constrained black box optimization. Optimization Methods Softw. **28**(1), 139–158 (2013). doi:10.1080/10556788.2011.623162
19. Conn, A.R., Scheinberg, K., Vicente, L.N.: Introduction to Derivative-Free Optimization. MPS/SIAM Book Series on Optimization. SIAM, Philadelphia (2009)
20. Dang, C.K.: Optimization of algorithms with the OPAL framework. Ph.D. Thesis, École Polytechnique de Montréal, Montréal, Québec, Canada (2012)
21. Dolan, E.D., Moré, J.J.: Benchmarking optimization software with performance profiles. Math. Program. **91**(2), 201–213 (2002). doi:10.1007/s101070100263
22. Fermi, E., Metropolis, N.: Numerical solution of a minimum problem. Los Alamos Unclassified Report LA-1492. Los Alamos National Laboratory, Los Alamos (1952)
23. Fletcher, R., Leyffer, S.: Nonlinear programming without a penalty function. Math. Program. Ser. A **91**, 239–269 (2002). doi:10.1007/s101070100244
24. Gould, N., Orban, D., Toint, P.: CUTEr (and SifDec): a constrained and unconstrained testing environment, revisited. ACM Trans. Math. Softw. **29**(4), 373–394 (2003). doi:10.1145/962437.962439
25. Gould, N.I.M., Orban, D., Sartenaer, A., Toint, P.L.: Sensitivity of trust-region algorithms on their parameters. 40R **3**(3), 227–241 (2005). doi:10.1007/s10288-005-0065-y
26. Gramacy, R.B., Le Digabel, S.: The Mesh Adaptive Direct Search Algorithm with Treed Gaussian Process Surrogates. Tech. Rep. G-2011-37, Les cahiers du GERAD (2011)
27. Hooke, R., Jeeves, T.A.: Direct search solution of numerical and statistical problems. J. Assoc. Comput. Mach. **8**(2), 212–229 (1961). doi:10.1145/321062.321069
28. Kohonen, T.: The self-organizing map. Neurocomputing **21**(1–3), 1–6 (1998). doi:10.1016/S0925-2312(98)00030-7
29. Kohonen, T., Somervuo, P.: How to make large self-organizing maps for nonvectorial data. Neural Netw. **15**(8–9), 945–952 (2002). doi:10.1016/S0893-6080(02)00069-2
30. Kokkolaras, M., Audet, C., Dennis, J.E Jr.: Mixed variable optimization of the number and composition of heat intercepts in a thermal insulation system. Optimization Eng. **2**(1), 5–29 (2001). doi:10.1023/A:1011860702585
31. Le Digabel, S.: Algorithm 909: NOMAD: nonlinear optimization with the MADS algorithm. ACM Trans. Math. Softw. **37**(4), 44:1–44:15 (2011). doi:10.1145/1916461.1916468
32. Mladenović, N., Hansen, P.: Variable neighborhood search. Comput. Oper. Res. **24**(11), 1097–1100 (1997). doi:10.1016/S0305-0548(97)00031-2
33. Orban, D.: Templating and automatic code generation for performance with python. Tech. Rep. G-2011-30, Les cahiers du GERAD (2011)
34. Pantazi, S., Kagolovsky, Y., Moehr, J.R.: Cluster analysis of wisconsin breast cancer dataset using self-organizing maps. In: Surjàn, G., Engelbrecht, R., Mcnair, P. (eds.) ealth Data in the Information Society, no. 90 in Technology and Informatics. International Congress on Medical Informatics, pp. 431–436. IOS Press, Amsterdam (2002)
35. Seymour, K., You, H., Dongarra, J.J.: 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), pp. 421–429. Tsukuba International Congress Center, EPOCHAL TSUKUBA, Japan (2008). doi:10.1109/CLUSTR.2008.4663803

36. Torczon, V.: On the convergence of pattern search algorithms. SIAM J. Optimization **7**(1), 1–25 (1997). doi:10.1137/S1052623493250780
37. Vicente, L.N., Custódio, A.L.: Analysis of direct searches for discontinuous functions. Math. Program. **133**(1–2), 299–325 (2012). doi:10.1007/s10107-010-0429-8
38. Vuduc, R., Demmel, J.W., Yelick, K.A.: OSKI: a library of automatically tuned sparse matrix kernels. In: Proceedings of SciDAC 2005, Journal of Physics: Conference Series. Institute of Physics Publishing, San Francisco (2005). doi:10.1088/1742-6596/16/1/071
39. Wächter, A., Biegler, L.T.: On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. Math. Program. **106**(1), 25–57 (2006). doi:10.1007/s10107-004-0559-y
40. Whaley, R.C., Petitet, A., Dongarra, J.J.: Automated empirical optimization of software and the ATLAS project. Parallel Comput. **27**(1–2), 3–35 (2001). doi:10.1016/S0167-8191(00)00087-9