NonOpt Manual

Frank E. Curtis*

May 31, 2020

Contents

1 Introduction

NonOpt (Nonlinear/nonconvex/nonsmooth Optimizer) is an open source software package for minimization. It is designed to locate a minimizer, or at least a stationary point, of

$$\min_{x\in\mathbb{R}^n} \; f(x), \tag{1}$$

where $f : \mathbb{R}^n \to \mathbb{R}$ is locally Lipschitz over $\mathbb{R}^n$ and continuously differentiable over a full-measure subset of $\mathbb{R}^n$. As indicated, the function f is allowed to be nonlinear, nonconvex, and/or nonsmooth.

1.1 Algorithm Overview

NonOpt implements a collection of algorithms, including bundle, gradient sampling, and quasi-Newton methods for solving problem (1). Users can choose between different strategies for the search direction computation, inverse Hessian approximation update, line search, point set update, and symmetric matrix storage. For more information on the algorithm, please see §4 and [4], with further details in [2, 3].

*Department of Industrial and Systems Engineering, Lehigh University, frank.e.curtis@gmail.com

1.2 Availability

NonOpt is written in C++ and is available on GitHub (see the NonOpt homepage for further information) under the MIT open source license. This puts very few restrictions on the use and reuse of the software. This license has been selected since the goal of open source software is continuous development and improvement. It is only asked that if you use the software in work that leads to a publication, then you cite the articles [1, 4, 2, 3]. Writing effective and efficient software requires a lot of time an effort, so citations are appreciated!

2 Installing NonOpt

NonOpt has been written with the intention of having it be Unix-based platform and compiler independent, but there are no guarantees. Comments and/or questions can be sent to Frank E. Curtis, although there are no guarantees that all issues will be resolvable. (Please do not e-mail any inquiries pertaining to installing NonOpt on Microsoft Windows! You are on your own for that. Sorry!)

2.1 Obtaining NonOpt

NonOpt is available on GitHub; see the NonOpt homepage for further information.

2.2 Compiling NonOpt

A C++ compiler is required to compile the NonOpt library. Building NonOpt also requires:

- BLAS (Basic Linear Algebra Subroutines). Many operating systems provide precompiled and optimized libraries for these subroutines. One could also obtain the source code from www.netlib.org. However, it is recommended to use an optimized BLAS implementation.
- LAPACK (Linear Algebra PACKage). As with BLAS, many operating systems provide precompiled and optimized libraries, but the source code can also be obtained from www.netlib.org.

After obtaining NonOpt, two environment variables need to be set:

- `$NONOPTDIR` should point to the NonOpt root directory of the distribution that you downloaded. (This root directory is the one containing the `README` file for the distribution.)
- `$LAPACKDIR` should point to your LAPACK root directory.

After these environment variables have been set, the NonOpt library can be built by issuing the following commands (here, and throughout this document, “`>`” indicates your terminal prompt):

```
> cd $NONOPTDIR/NonOpt
> make
```

This should `make` the libraries in each of the `src`, `problems`, and `tests` subdirectories. It should also `make` two executables, `runExperiment` and `solveNonsmoothProblem` in the `exes` subdirectory.

2.3 Testing the Installation

If the compilation has been successful, then a few unit tests and a minimization test can be run by issuing:

```
> cd $NONOPTDIR/NonOpt/tests
> ./testAll
```

Standard output will reveal the results of the unit tests. If all unit tests are successful, then the script will continue to run a minimization test for a test problem (implemented in `$NONOPTDIR/NonOpt/problems`). If a unit test fails, then the minimization test is not run. In this case, it is possible (although not necessarily the case) that the results you obtain using NonOpt will not be as good as expected.

More detailed output from each unit test can be obtained by running each unit test separately. For example, the dual active-set quadratic optimization (QP) subproblem solver can be tested by issuing:

```
> cd $NONOPTDIR/NonOpt/tests
> ./testQPSolver
```

The last line of the detailed output indicates whether or not the test was successful.

3 Interfacing to NonOpt

A NonOpt distribution comes with a set of test problems in the `$NONOPTDIR/NonOpt/problems` directory. The easiest way to interface your problem to NonOpt is to follow one of these as an example.

3.1 Implementing a Problem

For example, consider `MaxQ` implemented in `MaxQ.hpp` and `MaxQ.cpp` in `$NONOPTDIR/NonOpt/problems`. This problem is scalable, with the input n to the constructor dictating the dimension of the problem to be solved. If your problem is not scalable, then the value of `number_of_variables_` can be hard-coded. The remainder of an implementation of a problem can be found in this example. Overall, the functions that need to be implemented are those dictated in the header file `$NONOPTDIR/NonOpt/src/NonOptProblem.hpp`.

Note: It is *extremely* easy to set up NonOpt for failure if you have bugs in your objective and/or derivative implementations. NonOpt comes equipped with a derivative checker that you can use to test your implementation of your `evaluateGradient` function with respect to your `evaluateObjective` function. See §3.3 for instructions on turning on the derivative checker when your problem is being solved.

3.2 Solving a Problem

Suppose your problem is implemented in `YourProblem.hpp` and `YourProblem.cpp`. If these files are located in `$NONOPTDIR/NonOpt/problems`, then running `make` in this directory will add `YourProblem` to the `libNonOptProblems.a` library that is built in this directory. Your problem can then be solved by adding it to the list of problems in `solveNonsmoothProblem.cpp` in `$NONOPTDIR/NonOpt/exes`, then issuing:

```
> cd $NONOPTDIR/NonOpt/exes
> ./solveNonsmoothProblem YourProblem
```

You might also implement your own executable to set up and solve your problem.

3.3 Changing Options

NonOpt is designed to be flexible, offering various options, each of which can be changed from its default value, if desired. For a list of options, along with their types and default values, please see the `NonOpt-Manual/options.txt` file in the `NonOpt` distribution. Some safeguards have been added so that a user cannot set an option to an illegal value, although it is possible that a user could set an option to a value that will cause the behavior of the optimizer to behave in strange ways. If the user provides a legal value (and the `reporter` is issuing output to a report at a sufficient level), then a note will be printed (to a report) that the value for an option has been changed. If the user attempts to set an option to an illegal value, then similarly a note will be printed saying that a change was attempted, but the attempt was not successful. Options should be set with caution, or at least with sufficient knowledge of the underlying algorithms.

The value for an option can be changed in one of two ways:

- In an executable. After a `NonOptSolver` object has been declared (via `NonOptSolver nonopt;`), one can modify the value of an option via a statement such as:

  ```
  nonopt.options()->modifyDoubleValue(nonopt.reporter(), "cpu_time_limit", 600.0);
  ```

  The functions `modifyBoolValue`, `modifyDoubleValue`, `modifyIntegerValue`, and `modifyStringValue` all have the same form, with the last two arguments indicating the name of the option and the value to which the user would like to set it, respectively.

- In an options file. After a `NonOptSolver` object has been declared, one can modify the value of *multiple* options via a statement such as:

  ```
  nonopt.options()->modifyOptionsFromFile(nonopt.reporter(), "nonopt.opt");
  ```

  Here, `nonopt.opt` represents the name of a file containing a list of options and the values to which a user would like to set them. For example, the contents of the file might be:

  ```
  cpu_time_limit 600.0
  iteration_limit 1e+4
  ```

  Each line should contain only the name of an option followed by the desired value.

As previously mentioned, a useful option is to turn on the derivative checker, which is off by default. The derivative checker can be turned on by the statement:

```
nonopt.options()->modifyBoolValue(nonopt.reporter(), "check_derivatives", true);
```

or within an options file by including the line:

```
check_derivatives true
```

The derivative checker can be computationally expensive, so it is not recommended to run the derivative checker when performing timed experiments. But it is the best first step for debugging!

4 Algorithm

An overview of the algorithmic framework implemented in NonOpt is written as Algorithm 1. In each iteration, a search direction is computed through an inner loop. At the heart of each iteration of the inner loop is a primal-dual quadratic optimization (QP) subproblem defined by (sub)gradient and other information corresponding to a *point set* represented by $\mathcal{X}$. The specific nature of the *step acceptance* and *local model improvement* steps in the algorithm depend on the `direction_computation` option. The other major components of the code are the line search, (inverse) hessian approximation update, and point set update, the strategies for which can be modified by changing options.

References

[1] Frank E. Curtis and Minhan Li. Gradient Sampling Methods with Inexact Subproblem Solutions and Gradient Aggregation. arXiv 2005.07822, 2020.

[2] Frank E. Curtis and Xiaocun Que. An Adaptive Gradient Sampling Algorithm for Nonsmooth Optimization. *Optimization Methods and Software*, 28(6):1302–1324, 2013.

Algorithm 1 NonOpt Algorithm Framework

```
Require: $x_0 \in \mathbb{R}^n$; $H_0 \in \mathbb{R}^{n\times n}$ with $H_0 \succ 0$; $W_0 \leftarrow H_0^{-1}$; $\epsilon_0 \in \mathbb{R}_{>0}$; $\delta_0 \in \mathbb{R}_{>0}$
 1: Set $\mathcal{X}_0 \leftarrow \{x_0\}$
 2: for all $k \in \mathbb{N}$ do
 3:     loop
 4:         Denote $\{x_{k,0}, x_{k,1}, \dots, x_{k,m}\} := \overline{\mathcal{X}}_k$ (with $x_{k,0} \equiv x_k$) where $\overline{\mathcal{X}}_k \subseteq \mathcal{X}_k \cap \mathbb{B}_{\epsilon_k}(x_k)$
 5:         Denote $\{f_{k,j}\}_{j=0}^m$ and $\{g_{k,j}\}_{j=0}^m$ such that
                $f_{k,j} \in \mathbb{R}$  and  $g_{k,j} \in \partial f(x_{k,j})$  for all  $j \in \{1,\dots,m\}$
 6:         Compute $(d_k, \omega_k, \gamma_k)$ by (approximately) solving the primal-dual subproblem pair
                $\min_{d\in\mathbb{R}^n} \left(\max_{j\in\{1,\dots,m\}}\{f_{k,j} + g_{k,j}^T(d + x_k - x_{k,j})\} + \tfrac{1}{2}d^T H_k d\right)$  s.t.  $\|d\|_\infty \le \delta_k$
                $\sup_{(\omega,\gamma)\in\mathbb{R}^m_{\ge 0}\times\mathbb{R}^n} \left(-\tfrac{1}{2}(G_k\omega + \gamma)^T W_k (G_k\omega + \gamma) + b_k^T\omega - \delta_k\|\gamma\|_1\right)$  s.t.  $\mathbb{1}^T\omega = 1$
 7:         if $(d_k, \omega_k, \gamma_k)$ is deemed acceptable then          // step acceptance
 8:             break
 9:         else
10:             add new points to $\mathcal{X}$          // local model improvement
11:         end if
12:     end loop
13:     if $\|d_k\|_\infty$, $\|G_k\omega_k\|_\infty$, and $\|G_k\omega_k + \gamma_k\|_\infty$ are sufficiently small then
14:         if $\epsilon_k$ is sufficiently small then
15:             terminate
16:         else
17:             Set $\epsilon_{k+1} < \epsilon_k$
18:         end if
19:     else
20:         Set $\epsilon_{k+1} \leftarrow \epsilon_k$
21:     end if
22:     Set $\alpha_k \leftarrow \mathbb{R}_{>0}$ by a line search at $x_k$ along $d_k$          // line search
23:     Set $x_{k+1} \leftarrow x_k + \alpha_k d_k$
24:     Set $H_{k+1}$ and $W_{k+1}$ such that $H_{k+1} \succ 0$ and $W_{k+1} = H_{k+1}^{-1}$          // hessian approximation update
25:     Set $\delta_{k+1} \in \mathbb{R}_{>0}$
26:     Set $\mathcal{X}_{k+1}$ (as a set of previously encountered points, including $x_{k+1}$)          // point set update
27: end for
```

[3] Frank E. Curtis and Xiaocun Que. A Quasi-Newton Algorithm for Nonconvex, Nonsmooth Optimization with Global Convergence Guarantees. *Mathematical Programming Computation*, 7(4):399–428, 2015.

[4] Frank E. Curtis, Daniel P. Robinson, and Baoyu Zhou. A Self-Correcting Variable-Metric Algorithm Framework for Nonsmooth Optimization. *IMA Journal of Numerical Analysis*, 40(2):1154–1187, 2020.