NonOpt Manual

Frank E. Curtis

March 28, 2025

Contents

1 Introduction

NonOpt (Nonconvex, Nonsmooth Optimizer) is an open-source C++ software package for solving unconstrained minimization problems. It is designed to locate a minimizer, or at least a (nearly) stationary point, of an objective function $f : \mathbb{R}^n \to \mathbb{R}$, which is assumed to be locally Lipschitz. That is, given such an objective function f, the software contains algorithms that are designed to solve problems of the form

$$\min_{x \in \mathbb{R}^n} \; f(x).$$

The objective function f can be nonlinear, nonconvex, and/or nonsmooth. For further information, see [5].

1.1 Algorithm Overview

NonOpt contains implementations of a collection of algorithms, including gradient-sampling and proximal-bundle methods, each equipped with quasi-Newton techniques for improving practical performance. You can choose between different strategies for the search direction computation, approximate Hessian approximation update, line search, point-set update, symmetric matrix storage, and termination. For more information on the implemented algorithms, see [5], which in turn references the algorithms developed in [1, 2, 3, 4].

1.2 Availability

NonOpt is written in C++ and is available on GitHub here—see also the NonOpt homepage here—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 article [5], and potentially (a subset of) the articles [1, 2, 3, 4], when applicable. Writing effective and computationally efficient software requires a lot of time and effort, so citations are appreciated!

2 Installation

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 with 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 here; see also the NonOpt homepage here.

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, an environment variable needs to be set:

- `$NONOPTDIR` should point to the NonOpt root directory of the distribution that you obtained. This root directory is the one containing the `README` file for the distribution.

After these environment variables have been set, the NonOpt library can be built by issuing the commands below. 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 `solveProblem`, in the `exes` subdirectory, as well as a few executables in the `mpc` subdirectory, which are provided to reproduce the numerical results for the experiments in [5].

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 that is 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 NonOpt will not run properly, or at least the performance that you will obtain by 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 symmetric matrix class can be tested by issuing:

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

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

3 Interfacing to NonOpt

NonOpt 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 `$NONOPTDIR/NonOpt/problems`. This problem is scalable, with the input `n` to the constructor dictating the dimension of the problem to be solved. If your problem is not scalable, then the value of `number_of_variables` can be hard-coded. The remainder of an implementation of a problem can be understood in this example. Overall, the functions that need to be implemented are those dictated in the header file `$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 `$NONOPTDIR/NonOpt/problems`, then running `make` in this directory will add `YourProblem` to the `libNonOptProblems.a` library that is built in this directory. Your problem can then be solved by adding it to the list of problems in `solveProblem.cpp` in `$NONOPTDIR/NonOpt/exes`, then issuing:

```
> cd $NONOPTDIR/NonOpt/exes
> ./solveProblem 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 `$NONOPTDIR/NonOpt-Manual/options.txt` file. Some safeguards have been added so that you cannot set an option to an illegal value, although it is possible that you could set an option to a value that will cause the optimizer to behave in strange ways. If you provide 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 you attempt 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 (due to the provided value being illegal). 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 you 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 you 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 NonOpt Reporter Output

The option `print_level` determines the level of output to be printed to standard output. Setting this level to `0` means that no information will be printed. Setting this level to `1` means that a basic amount of information will be printed, such as in the example below. The information at this level is self-explanatory.

```
+------------------------------------------------------------+
|       NonOpt = Nonlinear/Nonconvex/Nonsmooth Optimizer     |
| NonOpt is released as open source code under the MIT License |
|     Please visit: https://github.com/frankecurtis/NonOpt   |
+------------------------------------------------------------+

This is NonOpt version 2.0

Number of variables.................. : 1000
Initial objective.................... : 6.908755e+00
Initial objective (unscaled)......... : 6.908755e+00
```

```
Approximate Hessian update strategy.. : BFGS
Derivative checker strategy.......... : FiniteDifference
Direction computation strategy....... : CuttingPlane
Line search strategy................. : WeakWolfe
Point set update strategy............ : Proximity
QP solver (small scale) strategy..... : DualActiveSet
QP solver (large scale) strategy..... : InteriorPoint
Symmetric matrix strategy............ : Dense
Termination strategy................. : Basic

EXIT: Stationary point found.

Objective............................ : 3.008077e-09
Objective (unscaled)................. : 3.008077e-09

Number of iterations................. : 22
Number of inner iterations........... : 25
Number of QP iterations.............. : 29
Number of function evaluations....... : 414
Number of gradient evaluations....... : 347

CPU seconds.......................... : 0.108261
CPU seconds in evaluations........... : 0.006546
CPU seconds in direction computations : 0.065522
CPU seconds in line searches......... : 0.008766
```

Setting `print_level` to `2` means that per-iteration output will be printed in addition to the basic output shown above. A header line is printed periodically to indicate the meaning of the values in each column of the per-iteration output. Further information about these quantities is provided in the list below. The per-iteration output when the `direction_computation` option is set to either `CuttingPlane` or `GradientCombination` is exactly the same, whereas when this option is set to `Gradient` the column `L?` is omitted since the small-scale QP subproblem solver is always employed for this strategy.

- `Iter.`: iteration index
- `Objective`: objective function value
- `St. Rad.`: stationarity radius
- `Tr. Rad.`: trust region radius
- `|Pts|`: cardinality of point set
- `In Its`: inner iteration index in direction computation method
- `QP Pts`: cardinality of points used to build QP subproblem
- `L?`: indicator that QP is large scale (`1`) or not (`0`)
- `QP Its`: iterations required to solve QP subproblem
- `S?`: termination status of QP subproblem solver (`0` indicates success)
- `QP KKT`: QP subproblem solution KKT error

- `|Step|`: norm of search direction
- `|Step|_H`: norm of search direction, with norm defined by approximate Hessian
- `|Grad.|`: norm of objective gradient
- `|G. Cmb.|`: norm of combination of gradients corresponding to QP subproblem solution
- `Stepsize`: step size from line search
- `Up. Fact.`: approximate Hessian update factor
- `U?`: indicator that approximate Hessian update is being updated (`1`) or not (`0`)

References

[1] Frank E. Curtis and Minhan Li. Gradient Sampling Methods with Inexact Subproblem Solutions and Gradient Aggregation. *INFORMS Journal on Optimization*, 4(4):347–445, 2022.

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

[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.

[5] Frank E. Curtis and Lara Zebiane. NonOpt: Nonconvex, Nonsmooth Optimizer, 2025.