Unconstrained Optimization with MINTOOLKIT for GNU Octave

Michael Creel*

March 2004

Abstract

The paper documents MINTOOLKIT for GNU Octave. MINTOOLKIT provides functions for minimization and numeric differentiation. The main algorithms are BFGS, LBFGS, and simulated annealing. Examples are given.

1 Introduction

Unconstrained optimization is a basic tool in many disciplines, and there is no need to discuss its importance. This paper discusses how unconstrained optimization may be done using GNU Octave (Eaton, www.octave.org) using the package MINTOOLKIT (Creel, http://pareto.uab.es/mcreel/MINTOOLKIT). If you would just like to see some examples of how to use the algorithms, skip to section 4. Otherwise, here's some introductory information that explains how algorithms we selected for inclusion into MINTOOLKIT.

1.1 Types of problems

We first briefly discuss types of optimization problems, in order to identify the cases where GNU Octave will be a good platform for analysis, and which of the algorithms in MINTOOLKIT will likely work well for given cases.

1.1.1 Large/small

What is the number of parameters to be minimized? Let k be the number of parameters. Memory usage of an algorithm will be some function $f(k)$. If this

*Dept. of Economics and Economic History, Universitat Autònoma de Barcelona. michael.creel@uab.es

function is growing rapidly for a certain algorithm, that algorithm will cease to be useful for ”large” problems, since memory will be exhausted. Of course, ”large” is a relative term that increases over time as memory becomes cheaper. For ”small” problems, the speed of convergence of the algorithm will be of primary importance, since memory resources will not be a bottleneck.

1.1.2 Continuous/discontinuous

Gradient-based methods such as Newton's algorithm or quasi-Newton methods rely on the function being differentiable. This will not hold if the function is not continuous. Search-type algorithms will be appropriate here.

1.1.3 Convex/nonconvex

A convex objective function will have a single global minimizer, whereas nonconvex functions may have additional local minima. Quasi-Newton methods only use local information in their updates, so they may well converge to a non-global minimum, depending upon starting values. A possible solution is to try a number of starting values. This is likely to work well if the nonconvexity problem is not too severe. When there are *many* local minima, a search-type algorithm may become more efficient, since the problem of local minima is dealt with automatically and doesn't require the analysts' intervention. After all, who's time is more important, yours, or your computer's?

1.1.4 Costly/cheap

Can the objective function be evaluated quickly, or is it time-consuming? Octave is an interpreted language, and is in general slower than FORTRAN, C, or similar. So expensive objective functions are best not implemented in pure Octave. One might go to a different environment for analysis, but in fact it is relatively easy to convert objective functions written in Octave to C++, and call them dynamically from Octave scripts. In this way, the expensive calculations are done using a fast language, while the user deals with the convenient, friendly Octave environment. In fact, C++ functions may make use of the Octave classes, so converting an Octave function to C++ is not very difficult. The algorithms in MINTOOLKIT serve as examples of how this may be done.

2 Algorithms

The algorithms in MINTOOLKIT were chosen based upon many sources of information, two of which are Nocedal (1992), for continuous, convex problems, and Mittelmann for global minimization. The goal of MINTOOLKIT is to be able to solve well-posed problems quickly and robustly, using the smallest set of algorithms possible. "Well-posed" is an important adjective here - MINTOOLKIT is not meant to be able to solve poorly conditioned problems or problems that can easily fall into numeric precision traps. Please pay attention to how your data is scaled, try to "bullet-proof" your objective function to avoid divisions by zero, etc. Nevertheless, if you find bugs, or have suggestions or comments, please contact the author.

2.1 BFGSMIN

The BFGS algorithm is probably the most widely-used quasi-Newton method for moderately-sized continuous problems that are not extremely nonconvex. It is more robust than other quasi-Newton methods such as DFP (Nocedal, 1992), and it is faster than Newton's method, since the Hessian matrix need not be calculated. One could easily create a Newton algorithm using the source code for `bfgsmin`.

2.2 LBFGSMIN

For large problems, the BFGS algorithm may not be feasible, since it requires storing a $k \times k$ matrix. The LBFGS ("L" is for "limited memory") method is able to store all information for updates in vectors, which substantially reduced memory requirements. It may also be faster than the BFGS algorithm in some circumstances, since it may use fewer floating point operations per iteration, depending upon the size of the problem. While it will usually require more iterations that BFGS, it may be faster if each iteration is faster. Liu and Nocecal 1989 is a reference.

2.3 SAMIN

When problems are mildly nonconvex, the quasi-Newton methods above, combined with a number of trial start values, may be the fastest way to find the global minimum. This solution is implemented in `battery.m`. But when the problem becomes less smooth, with many local minima, this solution may fail. Simulated annealing is one of the algorithms that works well in this case. The

implementation by Goffe has been used widely, and is the basis for the version included in MINTOOLKIT. An additional reference is Goffe (1996).

`samin` differs from the Goffe code in two important ways:

1. The "temperature" is determined automatically. In a first stage, the temperature is increased until the active search region covers the entire parameter space defined as the k-dimensional rectangle $\times_{j=1}^{k}(lb_j, ub_j)$ where lb_j and ub_j are the jth elements of the LB and UB vectors that are the lower and upper bounds for the parameters (these are user-specified arguments to samin. Once this is achieved, the temperature decreases as usual.

2. Convergence is defined as *two* conditions holding simultaneously.

   (a) The last NEPS best function values cannot differ by more than FUNCTOL. This is as in Goffe's code.

   (b) The width of the search interval must be less than PARAMTOL for each parameter. This allows to avoid accepting points on a flat plateau.

3 Obtaining the code

- MINTOOLKIT is available directly from the author, at http://pareto.uab.es/mcreel/MINTOOLK If you get it this way, uncompress the file where you like, change to the MINTOOLKIT directory, and compile by typing "make all" (this supposes that you have Octave installed already). Make sure that Octave knows where the MINTOOLKIT directory is. This option guarantees that you have the most recent version.

- Otherwise, you can obtain MINTOOLKIT as part of the octave-forge package.

- If you happen to be running Debian Linux, you can install a pre-compiled version of octave-forge and all required files by typing "apt-get install octave-forge". This is the easiest (and recommended) option.

4 Examples

MINTOOLKIT contains some functions for use by users, and some other functions that users can ignore. The functions for users are

Function	Purpose
bfgsmin	Ordinary BFGS algorithm
battery	Calls bfgsmin with a set of starting values
lbfgsmin	Limited-memory BFGS, for large problems
samin	Simulated annealing, for global minimization
numgradient	numeric first derivative of vector-valued function
numhessian	numeric second derivative matrix

This section gives some very simple examples of the use of the algorithms and functions in MINTOOLKIT. The first examples are intended to clearly illustrate how to use the algorithms. Realism is not important. Then some more difficult problems are considered.

The functions in MINTOOLKIT allow minimization or differentiation with respect to any of the arguments of a function, holding the other arguments fixed. The other arguments can include data or fixed parameters of the function, for example. The argument with respect to which minimization or differentiation is done is denoted by *minarg*, which by default is equal to 1. Any function to be minimized or differentiated by algorithms in MINTOOLKIT must follow one of the forms

$$\begin{aligned} value &= f(arg_1, arg_2, ..., arg_p) \\ [value, return_2, ..., return_n] &= f(arg_1, arg_2, ..., arg_p) \end{aligned}$$

Special case: If the second form is used *and* $return_2$ is a $k \times 1$ vector, where k is the dimension of *minarg*, then is assumed to be the gradient of f with respect to *minarg*, if the algorithm called uses the gradient. Otherwise, it (and any other returns from f) are ignored by MINTOOLKIT.

4.1 Minimization

4.1.1 `bfgsmin`

`bfgsmin` is called as

$$[theta, value, convergence] = bfgsmin("f", \{args\}, \{control\})$$

The first argument "f" is a string variable that holds the name of the function to be minimized. The second argument, *args*, is a cell array that hold the arguments of f. The third argument *control* is an optional cell array of 4 elements. The elements of *control* are described in Table 1. The outputs of `bfgsmin` are

Table 1: Controls for `bfgsmin`

Element	Purpose	Default Value	Other possible values
1	maxiters	-1 (infinity)	any positive integer
2	verbosity	0	1: summary every iteration; 2: only final summary
3	criterion	1: strict convergence (f, g, Δp)	2: weak convergence (only f)
4	minarg	1: first argument	int: $1 \leq minarg \leq k$, $k = \#args$

obvious, except the code values that *convergence* can take on. These are -1 for no convergence, maxiters exceeded; 1: convergence according to the specified strong or weak criterion; 2: no convergence due to failure of the algorithm (*e.g.*, the gradient calculation fails, or a stepsize cannot be found).

Consider a simple example - minimizing a quadratic function. The program bfgsmin-example.m follows:

- The first example uses numeric derivatives, and minimizes with respect to x, the first argument of the objective function. The second argument, y, is treated as fixed.
- The second example uses analytic derivatives, since it calls objective2, and minimizes with respect to x, the first argument of the objective function. The second argument, y, is treated as fixed.
- The third example uses numeric derivatives, and minimizes with respect to y, the second argument of the objective function, since the 4th element of `control`, `minarg`, is 2 . The first argument, x, is treated as fixed.

The output of running this example is

Notice that analytic gradients lead to faster convergence that do numeric gradients. Also note in the third example, where `minarg=2`, that minimization can be with respect to any of the arguments of the objective function.

4.1.2 `lbfgsmin`

When the problem is very large, a limited-memory bfgs algorithm may be needed, if `bfgsmin` is not feasible due to memory limitations. `lbfgsmin` is called as

$$[theta, value, convergence] = lbfgsmin("f", \{args\}, \{control\})$$

The first argument "f" is a string variable that holds the name of the function to be minimized. The second argument, *args*, is a cell array that hold the

Table 2: Controls for `lbfgsmin`

Element	Purpose	Default Value	Other possible values
1	maxiters	-1 (infinity)	any positive integer
2	verbosity	0	1: summary every iteration; 2: only final summary
3	criterion	1: strict convergence (f, g, Δp)	2: weak convergence (only f)
4	minarg	1: first argument	int: $1 \leq minarg \leq k$, $k = \#args$
5	memory	5	any positive integer

arguments of f. The third argument *control* is an optional cell array of 5 elements. The elements of *control* are the same as for `bfgsmin`, except that there is one more element that controls how many iterations are used to form the quasi-Hessian matrix (this is the *memory* of the method). The control vector is fully described in Table 3. You can easily modify the above example to use the `lbfgsmin` method.

It is possible that `lbfgsmin` can outperform `bfgsmin` even when memory is not an issue. Remember that both of these algorithms are *approximating* the Hessian matrix using previous gradient evaluations. If the true Hessian is changing rapidly, then a limited memory approximation may be better than a long memory approximation. The Rosenbrock function is such a case. The program lbfgsmin-example.m minimizes a 200-dimensional Rosenbrock function using both algorithms. The outputshows that the limited memory algorithm uses significantly more iterations that the ordinary BFGS algorithm, but it is almost 4 times as fast. In general, though, the ordinary BFGS algorithm is recommended when memory limitations are not a problem.

4.1.3 `samin`

For discontinuous and/or seriously nonconvex problems, the quasi-Newton methods are not likely to work well. `samin` is called as

$$[theta, value, convergence] = samin("f", args, control)$$

The controls for `samin` are summarized in Table

The example program sa-example.m is listed here:The objective function is the sum of k exponentiated cosine waves, each shifted down so the minimum is zero, with some curvature added in to create a global minimum of $f(x) = 0$ at $x = (0, 0, ..., 0)$. The (edited to shorten) output of the example is here:

You can see that the minimum was found correctly.

Table 3: Controls for `samin`

Element	Name	Purpose	Description
1	lb	lower bounds	vector of lower bounds for parameters
2	ub	upper bounds	vector of upper bounds for parameters
3	nt	control looping	loops per temp. reduction, e.g., nt=20
4	ns	control looping	loops per stepsize adjustment, e.g., ns=5
5	rt	reduce temp.	0 < rt < 1, e.g., rt=0.75
6	maxevals	limit evaluations	usually, a large number, unless just exploratory, e.g., 1e10
7	neps	convergence	positive integer. Higher is stricter criterion, e.g., neps=5
8	functol	convergence	last neps function values must be this close to eachother
9	paramtol	convergence	width of search interval must be less than this value
10	verbosity	output	0: no outout; 1: intermediate; 2: only final
11	minarg	arg. for min.	which arg to min. w.r.t., usually = 1

4.1.4 A more difficult problem

The Moré-Garbow-Hillstrom test suite contains some relatively difficult minimization problems. `bfgsmin` by itself can solve some of these problems, but not all of them, since some have multiple local minima, or completely flat regions where a gradient-based method will not be able to find a decreasing direction of search. The "Biggs EXP6" problem #18 is one for which `bfgsmin` fails to find the global minimum. This program shows how the global minimum may be found by combining an initial search that uses `samin` to find good starting values with refinement using `bfgsmin` to sharpen up the final results. The `samin` results from running this program, which use a fast temperature reduction and a fairly low limit on function evaluations are:

```
NO CONVERGENCE: MAXEVALS exceeded
Stage 2, decreasing temperature
Obj.  fn.  value 0.000006
parameter search width
9.844228 0.000000
4.294696 0.000000
-5.231675 0.000000
-3.114330 0.000000
1.076916 0.000000
1.118343 0.000000
```

Then come the BFGS iterations to sharpen up the results. The final BFGS results are:

```
BFGSMIN intermediate results:  Iteration 33
Stepsize 0.0000000
Using analytic gradient
Objective function value 0.0000000000
Function conv 1 Param conv 1 Gradient conv 1
```

```
params gradient change
10.0000 0.0000 0.0000
4.0000 -0.0000 -0.0000
-5.0000 0.0000 0.0000
-3.0000 -0.0000 0.0000
1.0000 -0.0000 0.0000
1.0000 0.0000 0.0000
```

- The minimum is found, but note that the solution values are in a different order than those given on the SolvOpt web page, with some negative signs. This problem suffers from a lack of identification - there are multiple values that give exactly the same value of zero.

An alternative which will often be faster, but is less sure to find the global minimum, is to call `bfgsmin` with many random starting values and a limited number of iterations. This is implemented in `battery.m`. You can see an example in This program. This leads to the results

```
BFGSMIN intermediate results:  Iteration 130
Stepsize 0.0000000
Using analytic gradient
Objective function value 0.0000000000
Function conv 1 Param conv 1 Gradient conv 1
params gradient change
4.0000 -0.0000 0.0000
10.0000 0.0000 -0.0000
3.0000 0.0000 0.0000
5.0000 -0.0000 0.0000
1.0000 -0.0000 0.0000
1.0000 0.0000 0.0000
```

The minimum is found correctly, and you can see that the problem is not identified.

4.1.5 Tips for successful minimization

Scaling Scaling the data and other constant parameters of the objective function so that the elements of the gradient are of approximately the same order of magnitude will help improve accuracy of the Hessian approximation. This can help a lot in obtaining convergence, and the results will have higher accuracy. This program illustrates. The output is You can see that the scaled data gives a more accurate solution, using less than half the iterations.

Bullet-proofing Writing your objective function so that it cannot return NaN or otherwise crash can save a lot of grief. Insert something like `if (((abs(obj_value) == Inf)) || (isnan(obj_value)))`

```
   obj_value = realmax;
endif
```

at the end of the objective function, and then return `obj_value`. This way, parameter values that lead to crashes are penalized, and will be avoided automatically.

4.2 Numeric differentiation

`numgradient` and `numhessian` can be used for numeric differentiation. `numgradient` returns the derivative of an $n \times 1$ vector-valued function with respect to a $k \times 1$ vector in a $n \times k$ matrix. `numhessian` returns the derivative of a real-valued function with respect to a $k \times 1$ vector in a $k \times k$ matrix. Both functions are quite accurate. numderivatives.m, which follows, shows how it can be done.

The results are:

5 Testing the code

The program mgh-test.m allows testing the algorithms using the Moré-Garbow-Hillstrom test suite, obtained from the SolvOpt source code. You can compare the output with these results, if you like. Note that simply applying BFGS with a single start value will sometimes lead to a failure of convergence, or convergence to a non-global minimum. This is expected, considering the nature of the problems. See section 4.1.4 for an appropriate means of proceeding with these problems.

If you find any bugs in the code, please contact me.

References

[1] Eaton, J.W., `http://www.octave.org/`

[2] Liu and Nocedal, `http://www.ece.northwestern.edu/~nocedal/PDFfiles/limited-m`

[3] Mittelmann, `http://plato.asu.edu/topics/problems/global.html`

[4] Nocedal (1992), `http://www.ece.northwestern.edu/~nocedal/PDFfiles/acta.pdf`

[5] Goffe, `http://www.netlib.no/netlib/opt/simann.f`

[6] Goffe, "SIMANN: A Global Optimization Algorithm using Simulated Annealing " Studies in Nonlinear Dynamics & Econometrics, Oct96, Vol. 1 Issue 3.