Global optimization using *Mathematica*: A test of software tools

Craig Loehle

Abstract

Mathematica provides a suite of built-in and 3rd party tools for nonlinear optimization. These tools are tested on a set of hard problems. The built-in *Mathematica* functions are tested as well as the tools in the MathOptimizer and Global Optimization packages. The problems tested represent classes of problems that cause difficulties for global solvers, including those with local minima, discontinuous and black box modules, problems with non-real regions, and constrained problems with complicated and wavy constraints. In addition, scaling of performance with problem size is tested. In general, no tool could solve all problems but all problems could be solved by at least on tool. All of the tools except the Global Optimization tools **`GlobalSearch`** and **`GlobalPenaltyFn`** were prone to returning infeasible solutions on discontinuous and black box modules, problems with non-real regions, and constrained problems with complicated and wavy constraints. The **`GlobalSearch`** and **`GlobalPenaltyFn`** tools were thus the most robust, and were in many cases also the fastest.

Introduction

Optimization can be done in many different programming languages such as FORTRAN, C++, and specialty languages. *Mathematica* is a high level programming language that offers many advantages for optimization. Very high precision math is standard. A huge library of advanced math functions is available. The notebook user interface is easy to use and interactive. Most critically for optimization, symbolic manipulation of expressions is possible. For example, the derivative of a function can be found exactly (symbolically) rather than by numerical approximation. It is thus interesting to see what optimization capabilities exist in *Mathematica*. There are several built-in and third-party tools for nonlinear optimization. These tools have broad capabilities, which are demonstrated in their respective user manuals and help files (see below), but in this report I explore some more difficult test problems than those shown in those places.

Unlike many other *Mathematica* functions, there is no general proof of correctness for nonlinear solvers. It is not entirely satisfying to test programs with a library of test problems, although this is often done. Instead, I here look at the general properties of hard problems. If a function is smooth and convex, many codes can solve it. If the function has local minima (is not convex), this causes difficulties because the usual tests for convergence are passed at a local minimum. A simple test of a global solver, then, is how it handles wavy functions. If there are many equivalent solutions, I consider it a success in this study if a valid solution is found. Since many functions have infinitely many true minima (e.g., $1 / \mathrm{Sin}(x)$ on $\{0, 1\}$), I do not consider it meaningful to try to find "all" solutions. Most of the tools reviewed here can be given different starting points so that multiple solutions can be found, except for the **`Minimize`** function in *Mathematica.*

The second type of problem that can cause difficulties is one that is nondifferentiable, because many optimization codes make use of derivatives (either exact or numerical). A stepwise discontinuity is an example of a point where derivatives are not defined. In *Mathematica*, a special kind of nondifferentiability exists. An expression like

```
x^2
```

Can be differentiated in *Mathematica* symbolically using

```
D[x^2, x]
```

which here yields $2x$ as the derivative. Expressions in *Mathematica*, however, can be in the form of a **Module** which can contain logical operations and procedural code. Any attempt to find a derivative of such an expression will yield invalid results.

Over certain regions, an expression might produce non-real results, which in *Mathematica* include `Infinity`, `Complex`, and `Indeterminate` data types. It is critical that a code handle such cases well.

Scaling is an issue for global optimization programs. An exhaustive search or grid search will work well for one or two dimensions but not for larger problems. Optimization algorithms are typically tested on two dimensional problems but this does not show whether they scale well. In this study I test several large problems.

Vanishing gradients are a big problem for optimization programs. They make it hard to detect when a minimum has been achieved. The high precision math in *Mathematica* should make it easier to solve such problems.

Constraints can make the solution of an optimization problem quite tough when the problem and/or constraints are nonlinear.

Representative cases of each of the above problem types are addressed in this study. No attempt is made to be exhaustive in testing. A few classic "hard" problems are used as test cases. It should be noted that all of the tools tested can solve many nonlinear problems. The focus here is on the particular issues that cause algorithms to fail. The problems used in this report are representative of a large number that the author has tested and no attempt is made here to do exhaustive testing.

Mathematica provides several functions for optimization. For convex, unconstrained nonlinear problems, *Mathematica* provides **FindMinimum**, which is very fast. For more general problems, **NMinimize** and **Minimize** are available. **NMinimize** has a default method selection, which is usually the **NelderMead** method. Other solution methods are **RandomSearch**, **SimulatedAnnealing**, and **DifferentialEvolution**. The **Minimize** function uses the most sophisticated methods of solution, but cannot handle multiple-valued functions (e.g., Sin) in an expression. The MathOptimizer package (Pinter Consulting, www.pinterconsulting.com or www.wolfram.com/products/applications /mathoptimizer/) provides three functions. **MS** is a scoping tool that narrows down the search region. **CNLP** is a local solver that handles constraints. To find a global solution, **Optimize** is used, which first calls **MS** and then **CNLP**. **Optimize** and **MS** require bounds but **CNLP** can search outside of the bounds given. The Global Optimization package (Loehle Enterprises, developed by the author, www.wolfram.com/products/applications/globalopt/ or www.loehleenterprises.com) provides two core functions: **GlobalSearch** and **GlobalpenaltyFn**. **GlobalSearch** solves constrained or unconstrained problems. Bounds are needed but are only a suggestion for getting started and they need not bound the true solution. If hard bounds are required they are entered as constraints. **GlobalpenaltyFn** is used if constraints are too complicated. These functions were the ones tested. MathOptimizer-Pro is a high-end product that was not tested here due to cost. All tests were conducted with Mathematica 5.1, Global Optimization 5.1, and MathOptimizer on a Pentium IV 3.4 GHz machine with 3Gb

RAM under Windows XP with no other applications running. Note that because **MS** and **Optimize** only search within the given bounds, all problems had bounds that covered the true solution, unless otherwise stated. This is a restrictive assumption, because users often do not know what variable range will cover the true solution. For all *Mathematica* and Global Optimization procedures, the input range is not binding, but is a suggestion for starting the search. For **GlobalSearch** and **GlobalPenaltyFn** the default is 3 random starts, and this was used unless otherwise stated. The local solver in *Mathematica* **FindMinimum** was not tested on most problems, because while it is fast it can not handle constraints and is easily trapped by local minima. In all tests, methods **SimulatedAnnealing**, **DifferentialEvolution** and **RandomSearch** are abbreviated **SA**, **DE**, and **RS**, respectively. All of the software tested uses proprietary methods, and thus the exact algorithms used can not be described here. The reader is referred to the respective web sites for whatever details they provide.

Unconstrained problem scaling

Convex functions are those that are differentiable and that have only a single minimum (i.e., no local minima). Such functions are the easiest to solve and should be solvable by any of the tools under consideration. The simplest example is a sum of squared terms. All functions tested could solve this problem, of course. What is interesting here is to see how speed scales with problem size. Timing was tested for different size problems. Bounds were set from −100 to 100.

Table 1 Timing, in seconds, for quadratic function.

Variables	10	100	400	4000
FindMinimum	0.016	0.015	0.344	414.8
GlobalSearch	0.05	0.25	2.5	320.
GlobalPenaltyFn	0.06	0.31	2.9	386.5
NMinimize	0.22	1.5	16.6	1152.
NMinimize_SA	0.17	1.5	12.8	Memory
NMinimize_DE	0.17	1.5	12.8	1167.6
NMinimize_RS	0.17	1.5	12.8	1167.6
Minimize	0.16	1.5	13.5	1137.1
CNLP	0.08	0.06	1.	114.2
MS	Failed	Failed	Failed	Failed
Optimize	8.7	2447.	Time	Time

For comparison, **FindMinimum** is used, which is a local solver. As expected, **FindMinimum** is fastest for the problem up to 400 variables, but for very large problems **CNLP** is fastest (Table 1) and **GlobalSearch** and **GlobalPenaltyFn** next fastest. **GlobalSearch** (and **GlobalPenaltyFn**, which behave almost identically on unconstrained problems) scales well with problem size, as do **NMinimize** and **Minimize. Optimize** with standard defaults (calling first **MS** and then **CNLP**) scales very badly with problem size. For the 400 variable test it ran all night and had to be cancelled in the morning. **MS** alone is too slow to use at all above 10 variables, and could not even reduce this quadratic function below 20 000 for the 10 variable problem. **GlobalSearch** is 4 times faster than **NMinimize** (all methods) and **Minimize**, which have similar performance. At 4000 variables, **SA** ran out of memory. At 10 000 variables, the other **NMinimize** options ran out of memory, so only **FindMinimum**, **GlobalSearch**, **GlobalPenaltyFn**, and **CNLP** could solve such a large problem.

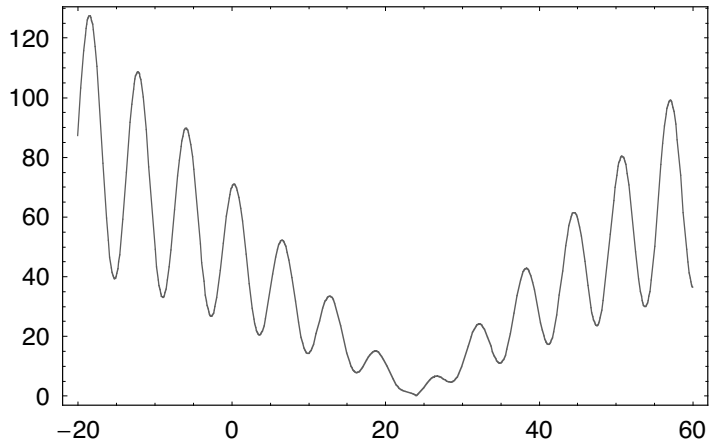
Overall the **MathOptimizer** functions **MS** and **Optimize** do not scale well at all with problem

size. It is also noteworthy that all **MathOptimizer** functions require bounds that encompass the true solution and a good guess starting point. Users can rarely determine correct bounds on problems, so this is really restrictive.

Wavy functions

If a function is wavy, it has local minima that will trap the solution algorithm for smooth solvers. A simple problem is:

```
In[1]:= Plot[Abs[2 (x - 24) + (x - 24) * Sin[x - 24]], {x, -20, 60}];
```



which has a minimum of 0 at $x \to 24$. **GlobalSearch** and **GlobalPenaltyFn** solved it with almost any starting range. The **MathOptimizer** functions will only solve this if given bounds that cover the true solution, so they were given bounds {−10, 100}. **NMinimize** fails with default parameters, but solves it with certain range values given to it. For comparison, all tests in Table 3 except **MathOptimizer** runs are based on bounds of {60, 65}. **Minimize** can not even get started, and exits with an error message. Two of the four **NMinimize** options succeed with these bounds. The min for **Optimize** is 0.028, which is not very good. **MS** does okay but takes 500 times longer than **GlobalSearch**. **CNLP** fails. Table 2 shows results.

Table 2 Wavy function results.

	minimum	time (s)
GlobalSearch	0	0.08
GlobalPenaltyFn	0	0.03
NMinimize	36.	0.05
NMinimize_SA	0	0.09
NMinimize_DE	36.1	0.09
NMinimize_RS	0	0.09
Minimize	Failed	Failed
CNLP	Failed	Failed
MS	0.00001	40.1
Optimize	0.028	0.16

Optimize can solve other wavy problems, but only if the bounds actually contain the solution. Only in extreme cases can **GlobalSearch** or **GlobalPenaltyFn** be made to fail on a wavy problem.

A classic hard non-convex problem was tested. The Lennard-Jones atom packing problem is given by:

```
In[2]:= dim = 10;
        vars = Table[ToExpression[StringJoin["x", ToString[i]]],
          {i, 1, 3 * dim}];

        f = 4 Sum[Sum[
            (1 / Sqrt[(vars[[i]] - vars[[j]]) ^ 2 + (vars[[i + 1]] - vars[[j + 1]]) ^
                   2 + (vars[[i + 2]] - vars[[j + 2]]) ^ 2]) ^ 12 -
             (1 / Sqrt[(vars[[i]] - vars[[j]]) ^ 2 + (vars[[i + 1]] -
                    vars[[j + 1]]) ^ 2 + (vars[[i + 2]] - vars[[j + 2]]) ^ 2]) ^ 6,
            {i, 1, j - 3, 3}], {j, 4, 3 * dim - 2, 3}];
```

Here dim is 10 (number of atoms) and **vars** is the **xi** terms, which makes this a 30 variable problem. The minimum is -28.4225 for this problem. There are a huge number of local minima.

Table 3 Lennard-Jones atom-packing results showing min and timing.

	minimum	time (s)
GlobalSearch	−28.4225	3.4
GlobalPenaltyFn	−28.4225	3.2
NMinimize	−3.	0.7
NMinimize_SA	−27.5	7.7
NMinimize_DE	−26.4	3.4
NMinimize_RS	−28.4225	13.6
Minimize	Memory	Failed
CNLP	−27.5	0.4
MS	−27.7	107.2
Optimize	−27.6	1358.9

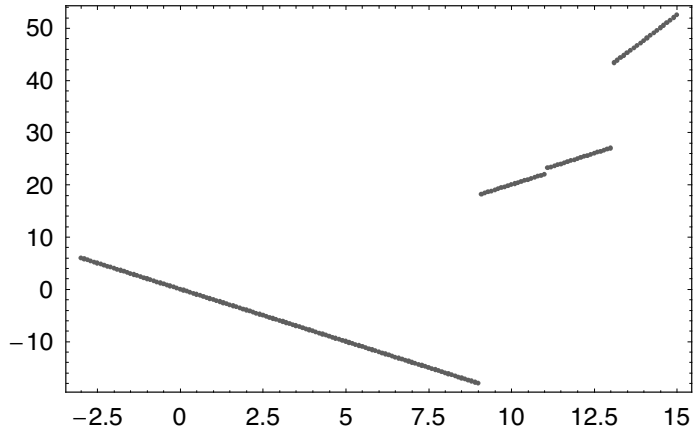
With 6 starts **GlobalSearch** and **GlobalPenaltyFn** solved it (Table 3). **NMinimize** solved it using **RandomSearch**, but not with the default or other methods. **Minimize** grabbed all available memory and failed. **CNLP** was run with six random starts, but did not find the minimum. **Optimize** found a suboptimal result and took a very very long time to do it. On this problem, **MS** was actually faster than **Optimize**, but was still quite slow, and only found a close suboptimal result. For comparison, **FindMinimum** was very fast on this problem but with 20 random starts the best solution found was −21.686.

Discontinuous and black-box functions

Discontinuous functions arise in various contexts such as economics and engineering. A simple example is given here.

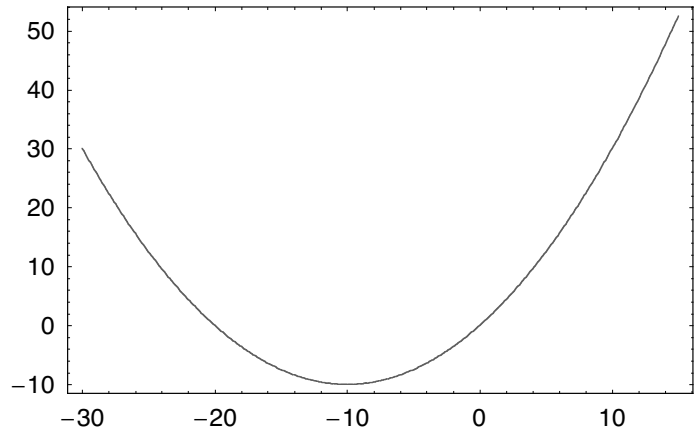
```
In[5]:= g[z_] := Module[{}, If[z ≤ 9, res = -2 z, Null];
          If[z > 9, res = 2 z, Null]; If[z > 11, res = 2 z + 1, Null];
          If[z > 13, res = 2 z + .1 z^2, Null]; Return[res]]
```

```
In[6]:= dat = {};
        Do[AppendTo[dat, {i, g[i]}], {i, -3, 15, .1}]

In[8]:= ListPlot[dat, PlotJoined → False];
```



GlobalSearch and **GlobalPenaltyFn** solve this problem. **NMinimize** (all methods) and **Minimize** give the correct min value, but the *x* value is wrong for any parameter range for *x*. **MS**, **Optimize**, and **CNLP** all fail on this problem (min is correct but *x* is 20, the input starting point), and probably for the same reason. The same result obtains for a continuous black-box function. The function **g** is continuous but there are If statements used in its computation.

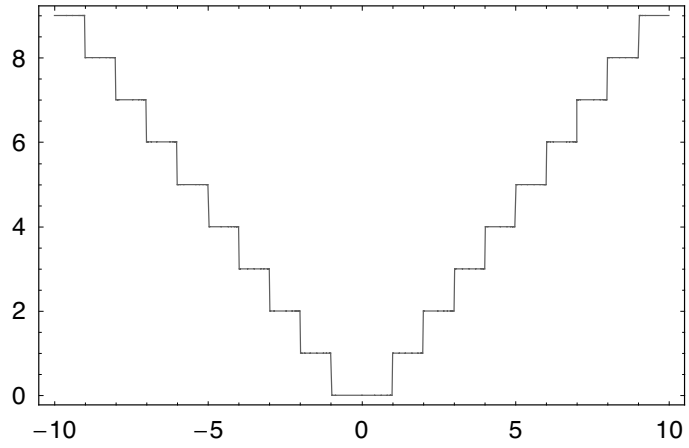
```
In[9]:= g[z_] := Module[{}, If[z > 1, z2 = z, z2 = z];
          res = 2 z2 + .1 z2^2; Return[res]]

In[10]:= dat = {}; Do[AppendTo[dat, {i, g[i]}], {i, -30, 15, .1}]

In[11]:= ListPlot[dat];
```



In the above cases, all of the **MathOptimizer** and **NMinimize** options give either the *x* value or function value wrong, probably because the function is being treated as analytic when it is not.

A discontinuous function that is not black-box is given by:

```
In[12]:= Plot[Abs[IntegerPart[x]], {x, -10, 10}];
```



As above, the initial range was set to {x, 60, 65} for all except the **MathOptimizer** functions, which were set to −40 to 40. The solution is 0 at any x such that $\{-1 < x < 1\}$. **GlobalSearch**, **GlobalPenaltyFn**, and **NMinimize** solve this for any starting range (Table 4). **SA**, **DE**, and **RS** fail (do not proceed much past the initial range of {60, 65}. **Minimize** does not execute. **Optimize** requires bounds that cover the region −1 to 1 in order to solve it, as does **MS**. **CNLP** fails (won't execute).

Table 4 Results for the step function problem.

	Minimum
GlobalSearch	0
GlobalPenaltyFn	0
NMinimize	0
NMinimize_SA	55.
NMinimize_DE	41.
NMinimize_RS	60.
Minimize	Failed
CNLP	Failed
MS	0
Optimize	0

Problems with non-real regions

Some functions have regions in which they do not return real results. A very simple example is:

```
In[13]:= f = x^0.5;
```

with solution 0 at {0}. With any starting range, including negative (e.g., {−6, −3}), **GlobalSearch** and **GlobalPenaltyFn** solve it. **Minimize** can solve it if the exponent is in the form of a ratio of whole numbers (1/2) but not as a rational number. **NMinimize** can not solve it using any option or bounds, and fails to execute at all if even the lower range value is negative. With bounds that cover the true solution, **Optimize** and **MS** make progress but do not get very close (min of 0.08), and otherwise they fail. **CNLP** fails to execute. Similar results obtain with other such problems.

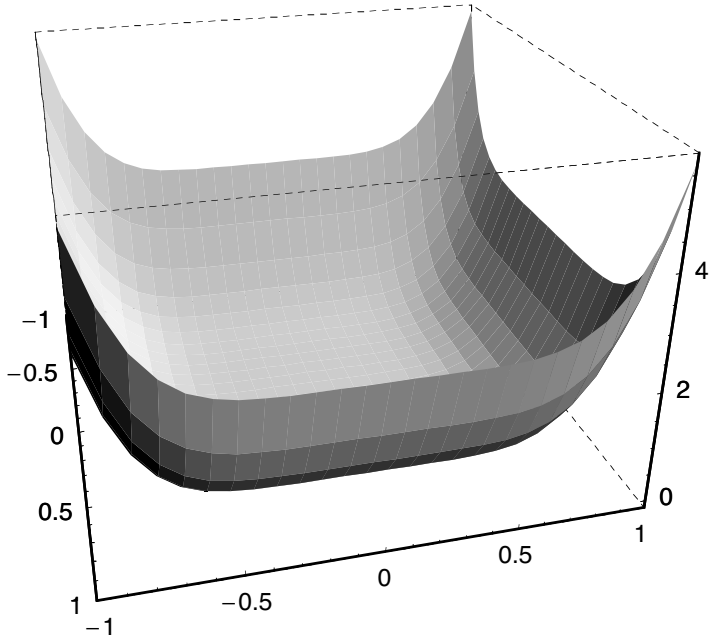
Problems with vanishing gradients

Some functions have a gradient that vanishes near the solution. For such functions, it can be hard to tell if the minimum has been reached. An example is:

```
In[14]:= f = x^6 (2.0 + Sin[1.0/x]) + y^6 (2.0 + Sin[1.0/y]);
```

Unfortunately, at exactly 0, this function is `Indeterminate` due to division by 0. A slight modification of the function retains it's difficulty but retains the solution 0 at {0}.

```
In[15]:= f = x^6 (2.0 + Sin[1.0/Max[.0000000001, Abs[x]]]) +
          y^6 (2.0 + Sin[1.0/Max[.0000000001, Abs[y]]]);

In[16]:= Plot3D[f, {x, -1., 1.}, {y, -1., 1.}];
```



The criterion for success here is how close to 0 the x and y values are. The input range was $\{-5, 5\}$ for both x and y. While it might be thought that **FindMinimum** would be able to solve this, it actually does not get that close, with a solution of $\{x \to 0.0115656, y \to -0.012471\}$.

The Loehle Enterprises functions were given a tolerance of 10^{-60} while other functions used **Automatic**. **AccuracyGoal** in **NMinimize** can not be made so small because it requires **Real** values.

Table 5 Results for the ashtray function. Order of magnitude results are shown (powers of 10) to focus on the precision of the result.

	obj	x	y
GlobalSearch	$1.\times10^{-68}$	$1.\times10^{-12}$	$1.\times10^{-12}$
GlobalPenaltyFn	0	0	0
NMinimize	$1.\times10^{-29}$	$1.\times10^{-6}$	0.00001
NMinimize_SA	$1.\times10^{-11}$	0.01	0.01
NMinimize_DE	$1.\times10^{-34}$	$1.\times10^{-6}$	$1.\times10^{-6}$
NMinimize_RS	$1.\times10^{-10}$	0.01	0.01
Minimize	$1.\times10^{-71}$	$1.\times10^{-12}$	$1.\times10^{-12}$
CNLP	Failed	Failed	Failed
MS	$1.\times10^{-13}$	0.001	0.001
Optimize	$1.\times10^{-9}$	−0.03	0.03

GlobalPenaltyFn found the exact solution while **GlobalSearch** and **Minimize** got extremely close. **CNLP** exited with a gradient error message before making any progress.

Constrained problems

When constraints are added to nonlinear problems or nonlinear constraints are added to linear problems, problem difficulty goes up considerably. In the following tests, I consider a constraint to be satisfied if it has less than a 10^{-6} violation.

The simplest equality constraint is simply a linear term. A quadratic problem was set up with every set of three terms summing to 1, as $\{x1 + x2 + x3 = 1, \ldots\}$ with 100 terms. All methods solved the problem correctly except **MS** which even with parameters {MaxIterations→1000,MaxSample→1000} could not find a feasible solution. **Minimize** solved this correctly in about 0.9 sec. With **NMinimize**, the default, **SA**, **DE**, and **RS** options took 0.78, 13.4, 4.6, and 1.7 seconds, respectively. **GlobalSearch** and **GlobalPenaltyFn** took 0.3 and 0.2 seconds. **CNLP** took 0.3 seconds. The largest problem tried, 50 variables, took 807 seconds with **Optimize**, and a 100 variable problem was stopped after 3 hours. Bounds had to cover the true solution to get any answer with **Optimize**.

A second constrained problem comes to us from finance (supplied by a customer), where interest rate terms produce fractional exponents. It is given by:

```
In[17]:= f = -(1927.86 * q^0.37 (-110 + s + 100 * (1 + t))^0.1743) /
          (-100 * q * t + 27.54 * s (-110 + s + 100 * (1 + t))^0.21)^0.2;
```

The list of inequality terms is

```
In[18]:= ineqs = {-10000 - 100 * q * t + 27.54 * s (-110 + s + 100 * (1 + t))^0.21,
    -100 + 27.54 (-110 + s + 100 * (1 + t))^0.21,
    139.338 - 350 / ((100*q (1+t)+27.54*(110-s) (-110+s+100*(1+t))^0.21 / (q+27.54*(-110+s+100*(1+t))^0.21))^0.2,
    110 - s - 100 * (1 + t),
    -0.3 + t, -t, -s, -q, -110 + s,
    -27.54 * (-110 + s + 100 * (1 + t))^0.21};
```

where each inequality term is in standard form, and ≤ 0 is implied. The single equality term is:

```
In[19]:= eq = {-q - 27.54 * (-110 + s + 100 * (1 + t))^0.21 +
            350 / ((100*q*(1+t)+27.54*(110-s) (-110+s+100*(1+t))^0.21) / (q+27.54*(-110+s+100*(1+t))^0.21))^0.2};
```

where it is implied that **{x + y}** is the same as **{x + y == 0}**. The equality constraint here is such that **GlobalSearch** can not run, and it gives an error message to use **GlobalPenaltyFn**, which solved this problem (min value -5321.65 at $\{t \to 0.29949, s \to 43.446, q \to 73.5118\}$). **NMinimize** does not even execute with defaults. With an input range for $\{t, s, q\}$, **NMinimize** finds a solution but the solution violates the constraints with a value of around 85 for the equality constaint violation. This is true for all options (simulated annealing, etc). **Minimize** also finds an equally bad infeasible solution. **Optimize**, **MS**, and **CNLP** failed to execute.

An example from the **MathOptimizer** user manual is a concave quadratic programming problem with multiple points that have a minimum $f = -310$. Results are shown in Table 6.

```
In[20]:= f = -25 (x1 - 2)^2 - (x2 - 2)^2 -
           (x3 - 1)^2 - (x4 - 4)^2 - (x5 - 1)^2 - (x6 - 4)^2;
```

The list of inequality terms is

```
In[21]:= ineqs = {-x1, x1 - 6, -x2, x2 - 6, 1 - x3, x3 - 5, -x4, x4 - 6,
           1 - x5, x5 - 5, -x6, x6 - 10, 1/4 * (-(x3 - 3)^2 - x4 + 4),
           1/4 * (-(x5 - 3)^2 - x6 + 4), 1/2 * (x1 - 3 x2 - 2),
           1/2 * (-x1 + x2 - 2), 1/6 * (x1 + x2 - 6), 1/2 * (-x1 - x2 + 2)};
```

Table 6 Results for quadratic programming problem.

	minimum	time (s)	
GlobalSearch	−310	8.4	
GlobalPenaltyFn	−310	15.6	
NMinimize	−171.	0.05	
NMinimize_SA	−120	0.99	
NMinimize_DE	−310.56	3.6	0.001 violation
NMinimize_RS	−274.1	6.4	
Minimize	Failed	Time	
CNLP	−∞	15	Infeasible
MS	−∞	100	Infeasible
Optimize	−∞	35	Infeasible

GlobalSearch and **GlobalPenaltyFn** solved this with 3 multiple starts. **NMinimize** did not get close with either the default or a range for the variables, or with any solution method except for

DifferentialEvolution, which had a slightly infeasible solution. **Minimize** timed out after 15 minutes. **CNLP**, **Optimize**, and **MS** all had solutions running off to $-\infty$ unless the inequality constraints were strongly conditioned (multiplied by a large number). Some of the inequality constraints had values of $+\infty$. Of course, typical users are unable to tell when they need to condition an equation.

The next constrained example illustrates how algorithms may be sensitive to starting points and bounds. The problem is test problem 62 from [1].

```
In[22]:= f = -32.174 *
           (255. * Log[(x1 + x2 + x3 + 0.03) / (0.09 * x1 + x2 + x3 + 0.03)] +
             280 * Log[(x2 + x3 + 0.03) / (0.07 * x2 + x3 + 0.03)] +
             290 * Log[(x3 + 0.03) / (0.13 * x3 + 0.03)]);

         eqs = {x1 + x2 + x3 - 1};
```

The implicit positivity of the variables is made explicit with inequality constraints. The default bounds are rather narrow at 0 to 1. Table 7 shows the results. The true solution is−26272.5 at {0.0617813, 0.328202, 0.0539851}. Two other sets of initial ranges are shown.

Table 7 Results for testing problem 62 showing minimum found for various initial bounds. **Minimize** can not be given starting range values, so **NA** is put for the second two columns.

	bounds\n {0,1}	bound\n {-20,-4}	bound\n {-113,4}
GlobalSearch	−26272.5	−26272.5	−26272.5
GlobalPenaltyFn	−26272.5	−26272.5	−26272.5
NMinimize	−26272.5	Failed	−17373.7
NMinimize_SA	−26272.5	Failed	−26272.5
NMinimize_DE	−26272.5	Failed	−25560.3
NMinimize_RS	−26272.5	Failed	−17374.6
Minimize	−17374.7	NA	NA
CNLP	Infeasible	Infeasible	Infeasible
MS	Infeasible	Infeasible	Infeasible
Optimize	Infeasible	Infeasible	Infeasible

GlobalSearch and **GlobalPenaltyFn** solve this for all initial ranges. **NMinimize** with the default (no range given) and **Minimize** both found the suboptimal result −17374.7. **NMinimize** only succeeds if given very narrow, positive bounds. **Optimize** and **CNLP** always give an infeasible result (equality constraint value of 0.5 to 1.0). It is very bad to return an infeasible result from an optimization problem.

A very difficult problem is created when the constraints have an oscillatory term. In the following problem, the solution is 0 at {0, 0}.

```
In[24]:= f = (2 * x1^2 - x2^2)^2 + (x2 - 6 * x1^2)^2;
         ineqs = {x2 + x1 - 2};
         eqs = {x1 - 10 * x2 - 100 * Sin[2 * x1 + 3 * x2]};
```

The problem was tested with initial range {−10, 10} for both variables. **CNLP** was tested with 10 random starts within this range and **GlobalPenaltyFn** with 3 starts. **MS** was run with {MaxIterations→1000,MaxSample→1000}. Results are shown in Table 8.

Table 8 Results for wavy equality constraint problem.

	minimum	time (s)
GlobalSearch	NA	NA
GlobalPenaltyFn	0	3.2
NMinimize	504.8	0.3
NMinimize_SA	77.	0.25
NMinimize_DE	0.13	0.88
NMinimize_RS	2408.5	5.5
Minimize	Failed	Failed
CNLP	2.05	1.75
MS	31708.8	85.6
Optimize	0	0.23

GlobalSearch printed an error message that **GlobalPenaltyFn** should be used, and **GlobalPenaltyFn** solved it from various starting ranges. **Optimize** also solved it, and quickly, but only if the bounds covered the true solution. **CNLP** could solve the problem with a sufficiently close guess, but 10 random tries did not succeed. **Minimize** would not execute with this function. **RandomSearch** with default did not succeed. **RandomSearch** with 100 search points solved it in 26 seconds (but not with 25 search points), but this brute force method may not work well with larger problems and clearly takes a while. **DifferentialEvolution** got pretty close.

A very difficult problem is Alkyl, given by

```
In[27]:= f = 5.04 * x0 + x1 + 3.36 * x12 + 0.35 * x13 - (6.3 * x2 * x4);

         ineqs = {-x0, -x1, -x2, 0.85 - x3, 0.9 - x4, 3. - x5,
           1.2 - x6, 1.45 - x7, 0.99 - x8, 0.99 - x9, 0.9 - x10,
           0.99 - x11, -x12, -x13, -2 + x0, -1.2 + x1, -5 + x2,
           -0.93 + x3, -0.95 + x4, -12 + x5, -4 + x6, -1.62 + x7,
           -1.0101010101 + x8, -1.0101010101 + x9, -1.11111111111 + x10,
           -1.0101010101 + x11, -2 + x12, -1.6 + x13};

         eq = {-.8196721311475 * x0 + x2 - .8196721311475 * x12,
           -x3 * (0.01 * x2 * x6 + x1) + 0.98 * x1, x12 + 10 * x13 - x0 * x5,
           -x0 * (.13167 * x5 - 0.0067 * x5 * x5 + 1.12) + x2 * x8,
           -0.325 * x3 + (0.00038 * x5 - .01098) * x5 + x4 * x9 - 0.57425,
           22.2 * x7 + x6 * x10 - 35.82, -3 * x4 + x7 * x11 + 1.33};
```

which has solution −1.756. The cross product terms in the equality constraints are the principle source of difficulty here. Results (Table 9) show that several tools failed on this problem.

Table 9 Results for problem Alkyl.

	minimum	time (s)
GlobalSearch	−1.756	24.7
GlobalPenaltyFn	−0.99	655
NMinimize	−1.39	8.8
NMinimize_SA	−1.359	6.8
NMinimize_DE	−1.39	6.9
NMinimize_RS	−1.32	168.9
Minimize	−1.39	8.7
CNLP	Infeasible	6.4
MS	Infeasible	63.9
Optimize	Infeasible	220

Only **GlobalSearch** is able to find the solution. **GlobalPenaltyFn** took much too long. The MathOptimizer function solutions violate every equality constraint and several inequalities by 0.2 to 2 in magnitude.

Conclusions

There are several types of problems that pose particular difficulties for nonlinear optimization. If functions are discontinuous or procedural, they are nondifferentiable. For this reason all solvers tested here failed on such problems except the Loehle Enterprises solvers **GlobalSearch** and **GlobalPenaltyFn**. These tests included black-box functions which can not be differentiated (even if continuous), functions with non-real regions, and step functions.

A second class of difficult functions is those that are wavy. **GlobalSearch** and **GlobalPenaltyFn** are very robust to such problems. **NMinimize** usually failed with the defaults. It could succeed with certain ranges and certain options (e.g., **RandomSearch**) but was sensitive to the range. **Minimize** usually failed to execute. The MathOptimizer functions **MS** and **Optimize** could always solve such problems but only if the bounds covered the true solution, which is a severe limitation since users rarely can define good bounds. **CNLP** and **FindMinimum** found local minima.

Scaling of speed with problem size is a serious concern. **CNLP** scales even better than **FindMinimum**, and can handle constraints, but it is still a local solver and fails on many problems. **Optimize** does not scale well and a 100 variable problem may run for hours. **GlobalSearch** and **GlobalPenaltyFn** scale similarly to **NMinimize** and are often much faster. On larger problems the *Mathematica* functions can grab all the memory and crash the kernel.

The **Minimize** function has the virtue of finding exact solutions but only for a restricted set of problems. If exponents are exact (Integer or ratios of whole numbers), it will work, but if they are Real, **Minimize** will actually call **NMinimize**. If there are multivalued terms such as Sin, it will not run.

Constrained problems are one of the greatest challenges for global optimization. Many problems can be solved by all the tools tested. However, it is easy to find constrained problems which can not be solved by **NMinimize**, **Minimize**, **Optimize**, **MS**, or **CNLP**, which often return infeasible results. **GlobalSearch** (or **GlobalPenaltyFn** for complicated constraints) are very robust to such problems and return an error message if no feasible solution can be found.

While *Mathematica* is not as fast as C++, the algorithms tested turned in quite good times on some

challenging problems. No single tool could solve all the problems tested, but all problems could be solved by at least one of the tools. One or more of the tools tested should be able to solve most global optimization problems.

Literature cited

[1] W. Hock and K. Schittkowski, *Test Examples for Nonlinear Programming Codes*. Springer-Verlag, Berlin / Heidelberg / New York, 1981.

About the author

The author is a senior research scientist with NCASI. He has published extensively on modeling methods, spatial statistics, the philosophy of science, and applied mathematics. His book "Thinking Strategically" is published by Cambridge Press.

Craig Loehle, Ph.D.
1258 Windemere Ave.
Naperville, IL 60564 USA
info@loehleenterprises.com