Selected Numerical Aspects of Non-Derivative Optimization

Margaret H. Wright

Computer Science Department

Courant Institute of Mathematical Sciences (CIMS)

New York University

Advances in Numerical Computing:

A Workshop in Honor of Sven Hammarling

Manchester Institute for Mathematical Sciences (MIMS)

5 July 2011

Thank you for inviting me to deliver this lecture.

It has two purposes:

1. To discuss numerical issues in non-derivative optimization;
2. To honor our dear friend and colleague Sven Hammarling.

Before turning to non-derivative optimization, a few words about Sven... in particular, about his remarkable BP connection.

Not the company formerly known as British Petroleum, but...

Beatrix Potter (1866–1943), famous English writer, illustrator, mycologist, and philanthropist.

What's the connection between our Sven and Beatrix Potter?

Sven's mother was one of the children of Annie Moore, who was Beatrix Potter's governess for a while.

The Tale of Peter Rabbit, Beatrix Potter's first published book, was originally written by her in 1893 as a picture letter to Sven's uncle Noel when he was ill.

The dedication of *The Pie and the Patty Pan* is "For Joan, to read to Baby"—Joan is Sven's aunt, and Baby is Sven's mother, also named Beatrix, the youngest of the Moore children.

Very interesting, I'm sure you will agree—but how is it relevant to advances in numerical computation, numerical issues in non-derivative optimization, and Sven's own scientific contributions?

Sven's work has covered many topics in numerical computation. Focusing on just two of them,

- parallel and high-performance computing, and
- reliability and robustness,

we'll consider how they, in the context of non-derivative optimization, are related to themes in Beatrix Potter's work.

To do this, we need some background about non-derivative optimization.

Consider unconstrained** local minimization of a nonlinear function:

$$\underset{x \in \mathbb{R}^n}{\text{minimize}} \quad f(x)$$

And not just any old f.

** Extensions to constraints are possible, but will not be discussed here.

As you all know, many real-world applications from science and engineering involve optimization problems in which f has one or more of the following properties:

- f is time-consuming or expensive to calculate, even on the highest-end machines or because it involves data collected from the real world (may take hours, days, weeks, . . .)
- f is unpredictably non-nice (e.g., undefined at certain points, discontinuous, non-smooth)
- f is evaluated by "black-box" software whose inner workings are not under the current user's control

- f is "noisy" because of
  - adaptivity in calculations
  - stochastic elements
  - uncontrollable variations (e.g., inclusion of real-world experimental data)

In cases like these, first derivatives are often difficult or impossible to obtain, even with today's most advanced automatic differentiation.

Unless we give up immediately, our only choice is a **non-derivative** method, i.e. a method that uses only function values.

A few examples, among hundreds, of such problems:

- Design of heating, air-conditioning, and ventilation (HVAC) systems;
- Modeling the population dynamics of the cannibalistic flour beetle;
- Estimating the structure of transmembrane proteins;
- Circuit design;
- Design of wireless systems, inside and outside;
- Drug selection during cancer chemotherapy, based on the patient's measured responses.

At least two broad classes of non-derivative methods:

1. **Model-based**

   - Iteratively create a model of f, usually linear or quadratic, based on interpolation or least-squares, and minimize the model (à la Newton's method or quasi-Newton methods)
   - Some smoothness assumed somewhere.

2. ** **"Direct search"** ** [my focus today]

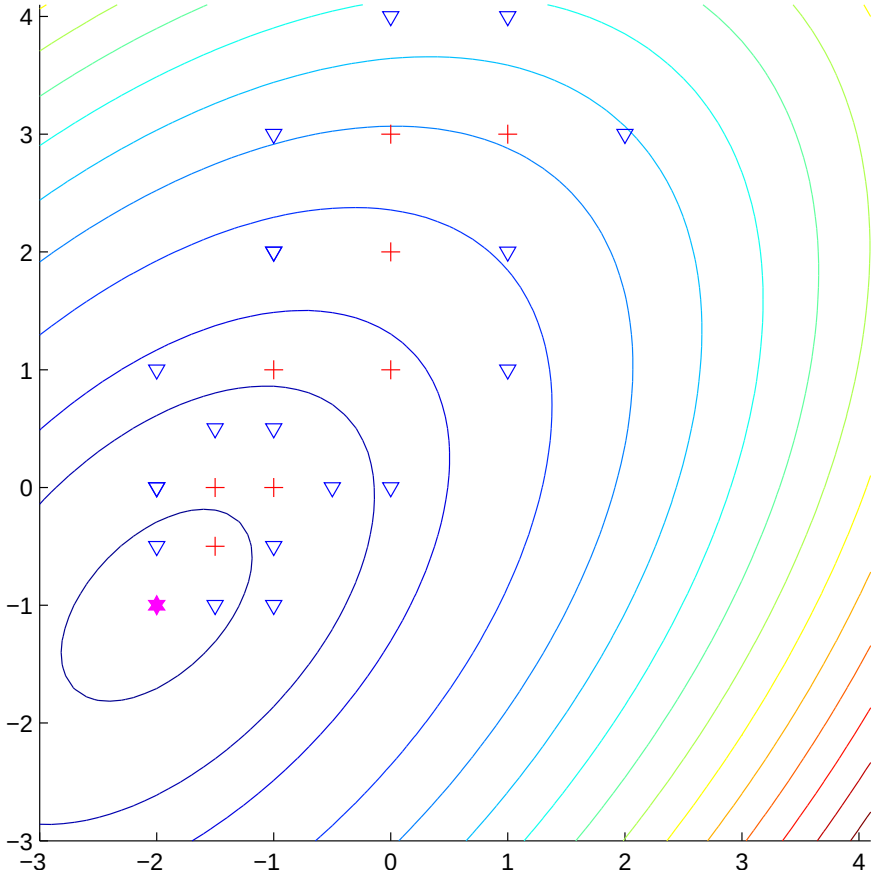
   - No *explicit* model of f, but sometimes used with "surrogate" models

NB: Simulated annealing, genetic, evolutionary, and swarming algorithms will not be considered.

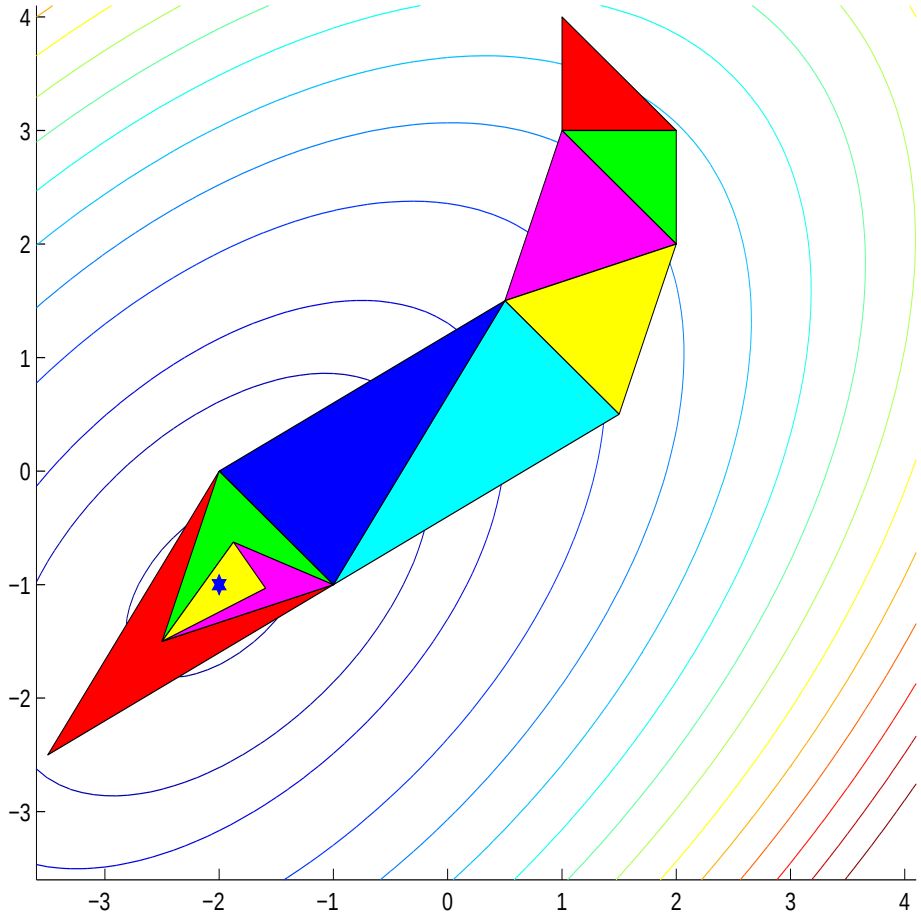
A sketchy history of non-derivative methods:

- Started in 1950s (or before)—Fermi and Metropolis applied coordinate search in a 1952 paper.
- Loved and widely used by practitioners from the beginning, especially the direct search "simplex" method of Nelder and Mead (1965).
- Often were simple and easy to understand, based on **low-dimensional** intuition.

"Opportunistic" coordinate search (Fermi and Metropolis); keep looking for a new best point by searching along the coordinate directions.



Nelder–Mead: iteratively update a simplex by moving away from the *worst vertex*



An interesting part of the history of optimization. . .

Starting in the mid-1960s, non-derivative methods became essentially invisible within the mainstream optimization community.

There were at least two reasons for this:

1. Essentially no theory had been developed for non-derivative methods, and optimization was becoming "mathematicized";
2. Non-derivative methods often experienced difficulties in practice, even on problems viewed as easy.

Nonetheless, despite a lack of theory and well known occasional poor behavior, through the 1980s and into the 1990s, direct search methods (almost always Nelder-Mead) were by far the most popular optimization routines in software libraries.

This popularity probably arose because users did not want to write code to evaluate derivatives. The biggest single error in using optimization software was programming the derivatives incorrectly.

But. . . this should no longer be an issue because we have reliable and sophisticated software for automatic differentiation.

What about today???

Non-derivative methods remain very popular with practitioners, who have never stopped needing to solve nasty problems.

And . . . non-derivative methods have undergone a major renaissance within the optimization research community, returning to grace and favor.

A significant factor in this sea change: Torczon's (1989) PhD thesis (Rice University), which presented a new direct search method, multidirectional search, with a convergence proof.

Once an initial proof existed (acting like the smell of blood to a shark), mathematicians began to generalize, producing new classes of direct search methods.

Rigorous mathematical analysis has been and is important and useful, providing understanding as well as added confidence of success in practice.

Tools and approaches from convex analysis and approximation have broadened the applicable class of functions to include (for example) specific forms of discontinuity in f or its derivatives.

Possibly inevitably, newer direct search methods are becoming much more complicated and much less appealing to intuition. No one could describe some of the recently defined direct search methods as "simple and easy to understand".

A broad-brush view of advances in theory:

Convergence proofs have been derived (with varying definitions of "convergence") for the following categories of direct search methods, and more:

- pattern search and generalized pattern search,
- generating set search,
- adaptive pattern search,
- mesh-adaptive direct search (MADS),
- frame-based methods,
- grid-restrained methods,
- . . .

The proof techniques used are typically closely related to derivative-based optimization.

But because this workshop is about advances in numerical computation, let me now cite one of my all-time favorite quotes, from Donald Knuth, renowned computer scientist:

My main conclusion after spending ten years of my life working on the TEX project is that software is hard. It's harder than anything else I've ever had to do.

Given this evident truth, we can pat ourselves on the back and concentrate henceforth on software-related topics in direct search methods.

What many users care about, more than anything else, is solving their problems efficiently.

Because this involves several imprecise and ambiguous terms ("solving", "efficiently"), there are many unresolved numerical issues in direct search methods—and I can't discuss, or even mention, all of them today.

So I've selected two.

The first—parallel and high-performance computing—has been a longstanding feature of Sven's research.

How can direct search methods be implemented effectively on evolving high-performance (parallel) machines?

Many people in scientific computing regard this question as extremely urgent. According to Jim Demmel (2011),

> *A large change in the computing world has started in the last few years: not only are the fastest computers parallel, but nearly all computers will soon be parallel. . . . So all programs that need to run faster will have to become parallel programs.*

And now we call on Beatrix Potter.

MANY THANKS TO PROJECT GUTENBERG for making her works freely available.

THE TAILOR OF GLOUCESTER

BY

BEATRIX POTTER

F. WARNE & Co

There lived a tailor in Gloucester. . . . He lived alone with his cat; it was called Simpkin.

Although he sewed fine silk for his neighbours, he himself was very, very poor. He cut his coats without waste; they were very small ends and snippets that lay about upon the table.

One bitter cold day near Christmas the tailor began to make a coat. . . for the Mayor of Gloucester.

There were twelve pieces for the coat and four pieces for the waistcoat; and there were pocket flaps and cuffs, and buttons all in order. For the lining of the coat there was fine yellow taffeta; and for the button-holes of the waistcoat, there was cherry-coloured twist. And everything was ready to sew together in the morning, all measured and sufficient.

The tailor was very tired and beginning to be ill.

All that day he was ill, and the next day, and the next; and what should become of the cherry-coloured coat? In the tailor's shop in Westgate Street the embroidered silk and satin lay cut out upon the table—one-and-twenty button-holes—and who should come to sew them, when the window was barred, and the door was fast locked?

The little mice came out again, and listened to the tailor; they took notice of the pattern of that wonderful coat. They whispered to one another about the taffeta lining, and about little mouse tippets.

From the tailor's shop in Westgate came a glow of light; . . . there was a snippeting of scissors, and snappeting of thread.

”Alack,” said the tailor, “I have . . . no more strength—nor time—than will serve to make me one single button-hole; for this is Christmas Day in the Morning! The Mayor of Gloucester shall be married by noon—and where is his cherry-coloured coat?”

He unlocked the door of the little shop in Westgate Street, and Simpkin ran in. . . . But there was no one there! Not even one little brown mouse! The boards were swept clean; the little ends of thread and the little silk snippets were all tidied away, and gone from off the floor.

But upon the table—oh joy! the tailor gave a shout—there, where he had left plain cuttings of silk—there lay the most beautifullest coat and embroidered satin waistcoat that ever were worn by a Mayor of Gloucester.

And from then [NB: because of parallel computing] began the luck of the Tailor of Gloucester; he grew quite stout, and he grew quite rich.

Never were seen such ruffles, or such embroidered cuffs and lappets! But his button-holes were the greatest triumph of it all.

The stitches of those button-holes were so small–*so* small–they looked as if they had been made by little mice!

And what about direct search methods and parallel computing?

An interesting sidebar: the concept of using direct search methods as a tool to do automatic performance tuning on high-end machines. Direct search can be (sensibly) adapted for the non-nice functions representing performance; accurate solutions are not needed or possible.

See, for example,

Seymour, You, and Dongarra (2008);

Balaprakash, Wild, and Hovland (2011);

Karcher and Pankratius (2011).

NB: The idea is not to take advantage of parallelism to make the direct search method efficient—it's to use a direct search method to improve/optimize parallel performance.

But how about creating parallel-efficient direct search methods?

With apologies to anyone who is an expert in direct search methods, here is a super-terse (and not entirely accurate) overview of generalized pattern search methods.

Definition: A set $\mathcal{D}$ of vectors in $\mathcal{R}^n$, $\{d_\ell\}$, is a **positive spanning set** if every $v \in \mathcal{R}^n$ can be written as a **nonnegative** linear combination of its elements.

Standard examples in $\mathcal{R}^2$:

$$\mathcal{D} = \begin{pmatrix} 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix} \quad \text{and} \quad \mathcal{D} = \begin{pmatrix} 1 & 0 & -1 \\ 0 & 1 & -1 \end{pmatrix}$$

Ingredients in a generalized pattern search method:

- A fixed positive spanning set $\mathcal{D}$ that can be written as $\mathcal{D} = GZ$, where G is a fixed nonsingular matrix and the elements of Z are integers.
- A rational mesh update parameter, $\tau > 1$. (Usually, $\tau = 2$.)
- A strategy for strictly decreasing a step control parameter Δ when x_k is mesh-locally optimal (defined in a moment), and, optionally, for increasing Δ when x_k is better than x_{k-1}.
- A strategy for performing some optional "exploratory" (sometimes confusingly called "search") moves
- A strategy for choosing the "poll directions" from $\mathcal{D}$ at every iteration.

In the polling step at x_k, we choose a set of polling directions $\mathcal{D}_k \subseteq \mathcal{D}$, whose columns form a positive spanning set, and we evaluate f at some, possibly all, of the mesh-neighboring points

$$x_k + \Delta_k d^{(\ell)},$$

where $d^{(\ell)} \in \mathcal{D}_k$.

If f at all of these points is not better than $f(x_k)$, x_k is said to be mesh-locally optimal and Δ_k is reduced in a controlled fashion.

Termination criteria—often based on the size of Δ_k.

NB: Searches are made along a *set of directions*.

Steps of the standard Nelder–Mead algorithm:

Given: the $n+1$ vertices x_i, $i = 1 \ldots n+1$, of a nondegenerate simplex in $\mathcal{R}^n$ and the associated values $\{f_i\}$ of f.

1. Order the vertices to satisfy $f_1 \leq \cdots \leq f_n \leq f_{n+1}$, using an appropriate tie-breaking rule.

2. Calculate $\bar{x} = \sum_{i=1}^{n} x_i$ (the average of all the points except the worst).

3. *Reflection.* Compute $x_r = 2\bar{x} - x_{n+1}$ and evaluate $f_r = f(x_r)$. If $f_1 \leq f_r < f_n$, accept x_r and terminate the iteration. NB: x_r is better than the worst vertex.

4. *Expansion.* If $f_r < f_1$, calculate $x_e = \bar{x} + 2(x_r - \bar{x})$ and evaluate $f_e = f(x_e)$. If $f_e < f_r$, accept x_e; otherwise, accept x_r. Terminate the iteration.

5. *Contraction.* If $f_r \geq f_n$, perform a *contraction*.

   a. *Outside.* If $f_n \leq f_r < f_{n+1}$, calculate $x_{oc} = \bar{x} + \frac{1}{2}(x_r - \bar{x})$ and evaluate $f_{oc} = f(x_{oc})$. If $f_{oc} \leq f_r$, accept x_{oc} and terminate the iteration; otherwise, do a shrink.

   b. *Inside.* If $f_r \geq f_{n+1}$, calculate $x_{ic} = \bar{x} - \frac{1}{2}(\bar{x} - x_{n+1})$ and evaluate $f_{ic} = f(x_{ic})$. If $f_{ic} \leq f_{n+1}$, accept x_{ic} and terminate the iteration; otherwise, do a shrink.

6. *Shrink.* Evaluate f at the n points $v_i = x_1 + \sigma(x_i - x_1)$, $i = 2, \ldots, n+1$. The vertices of the simplex at the next iteration are x_1, v_2, $\ldots$, v_{n+1}.

NB: Except for shrinks (which hardly ever happen), all "searches" are along a single line, joining the worst point to $\bar{x}$.

Parallelism has been inherent in (and, in fact, was the motivation for) pattern search methods ever since Torczon's 1989 definition of multidirectional search, which was developed at the Center for Research on Parallel Computing (CRPC) at Rice University.

Asynchronous parallel pattern search (APPS) was first proposed in 2001 by Hough, Kolda, and Torczon.

A newer version of APPS was published in 2006, and HOPSPACK is now considered (by its Sandia authors) to have replaced APPS; `https://software.sandia.gov/trac/hopspack/`

The idea for exploiting parallelism in all these instances is based on separately performing the needed searches along the set of search directions.

Nelder–Mead is much more problematic, since in its original form it appears to be inherently sequential.

In recent years, however, there have been several proposals that offer promise for an effective "parallel Nelder–Mead", although of course they are variants of the original algorithm.

Because of time limitations here, only one example (and its application) will be (briefly) mentioned.

The reference: Lee and Wiswall (from NYU's Department of Economics), "A Parallel Implementation of the Simplex Function Minimization Routine", *Computational Economics* 30 (2007).

Their strategy is to split up the Nelder–Mead algorithm into separate searches that can be assigned to p processors, assuming that $n \geq p$.

1. Given $p < n$, a processor is assigned to each of the p worst points.

2. Each processor i, $i = 1, \ldots, p$, then calculates its own "search direction" by reflecting its given "worst point" through the centroid of the $n - p + 1$ points that have better function values.

3. Each processor proceeds independently as in the non-parallel NM algorithm, trying reflection, expansion, contraction, and shrinkage.

4. The result is a set of p updated "worst points", which are then used, along with the best $n - p + 1$ points, to create a new simplex.

Lee and Wiswall report the results of several hundred Monte Carlo experiments, where the starting vertices are chosen at random, to minimize sums of squares and sums of absolute values up to dimension 200, for degrees of parallelism from $p = 1$ to $p = 0.9n$.

Not surprisingly,

> *the performance varied substantially across experiments and degrees of parallelization.*

The authors conclude (and I agree) that their experiments show the potentially large gains from parallelizing the Nelder–Mead method—and that much more remains to be done.

In a June 2011 final project, "Parallel Optimization and its Application to Earnings Inequality", in a parallel programming class taught by John Gilbert at the University of California, Santa Barbara, Klein and Neira show that parallel Nelder–Mead (meaning the method of Lee and Wiswall) is good for problems with a "cheap objective function but many variables".

Klein and Neira differ from Lee and Wiswall by processing k points ($k > 1$) on p/k processors, rather than one point on each of p processors. They also show how to implement parallel Nelder–Mead with distributed memory.

Their application is to model human capital accumulation, in a function with 400 variables.

Very encouraging results!

The second issue that (in my view) has not been sufficiently well addressed in direct search methods involves reliability.

Ideally, optimization software will, with the highest possible probability, produce a "correct" solution, using a reasonable definition of "correct". If this is not possible (e.g., the tolerances are too small), the software should produce information to guide the user to reformulate the problem, change a parameter, and so on.

Before discussing how this property arises in direct search methods, we invoke Beatrix Potter's most famous tale.

Probably everyone knows the story, but just in case. . .

THE TALE OF PETER RABBIT

BY

BEATRIX POTTER

F. WARNE & Co

“Now my dears, you may go into the fields or down the lane, but don’t go into Mr. McGregor’s garden: your father had an accident there; he was put into a pie by Mrs. McGregor”.

Flopsy, Mopsy, and Cotton-tail, who were good little bunnies, went down the lane to gather blackberries.

But Peter, who was very naughty, ran straight away to Mr. McGregor's garden and squeezed under the gate!

I am sorry to say that Peter was not very well during the evening. His mother put him to bed and made some camomile tea; and she gave a dose of it to Peter!

But Flopsy, Mopsy, and Cotton-tail had bread and milk and blackberries for supper.

What's the "takeaway" for numerical computing from *The Tale of Peter Rabbit*?

Peter is, of course, the character that readers love, even though he is disobedient, reckless, and not reliable at all.

His siblings Flopsy, Mopsy, and Cottontail are (perhaps unfairly) perceived as annoyingly reliable, and readers tend to dismiss them as a bit smug, a bit Goody Two-Shoes.

But *The Tale of Peter Rabbit* is fiction!

As Miss Prism comments in Oscar Wilde's *The Importance of Being Earnest*, "The good ended happily and the bad unhappily. That is what fiction means".

If Peter, Flopsy, Mopsy, and Cotton-tail were writing numerical software, we would have quite a different story. Readers would not smile indulgently and forgive Peter's recklessness if his software failed regularly.

What do we need to make direct search software reliable?

Convergence proofs are important, but not nearly enough.

One reason: to prove anything, assumptions must be made about f—but the alleged virtue of direct search methods is supposed to be that they will optimize the proverbial "any function". And of course this is nonsense.

The already-mentioned January 2011 Argonne report by Balaprakesh, Wild, and Hovland, "Can Search Algorithms Save Large-scale Automatic Performance Tuning?", comments

> *Despite potentially sharp discontinuities and disconnected feasible regions, our experience suggests that there will almost always be some structure in the objectives of interest.*

To believe that direct search software is reliable, it helps to know that it has correctly solved many problems!

There is a strong sense of *déjà vu* in thinking about testing optimization software, which has been a recurring contentious issue for more than 30 years.

> The standards expected for mathematical exposition are *only rarely applied* to the reporting of computational experimentation. . . . Part of the problem stems from the fact that *there are no standards* for reporting computational experiments.

H. Crowder, R. Dembo, and J. Mulvey, On reporting computational experiments with mathematical software, *ACM Transactions on Mathematical Software*, 1979.

> Engineers want to know which optimization methods can be applied to their [simulation] models, and *under what circumstances*. This question is difficult to answer in a satisfactory way. . . each design problem poses different goals.

R. Barton, Testing strategies for simulation optimization, *Winter Simulation Conference*, 1987.

Despite general agreement on the principles involved, many open questions remain. Numerical experiments are needed:

1. To help us make informed judgments about which methods tend to be most effective for problems in general and specific problem classes in particular;
2. To reveal algorithmic strengths and weaknesses that may not be obvious from theory, and to confirm (or deny) expectations of performance derived from theory;
3. To suggest guidelines (aka rules of thumb) for choosing optional algorithmic parameters.

The controversies begin as soon as one tries to implement these principles. Should we:

1. Test problems that are difficult?
2. Test problems that are typical?
3. Test real applications, or problems that are "like" real applications?
4. Test for reliability, speed, or . . . ?
5. Compare existing software?

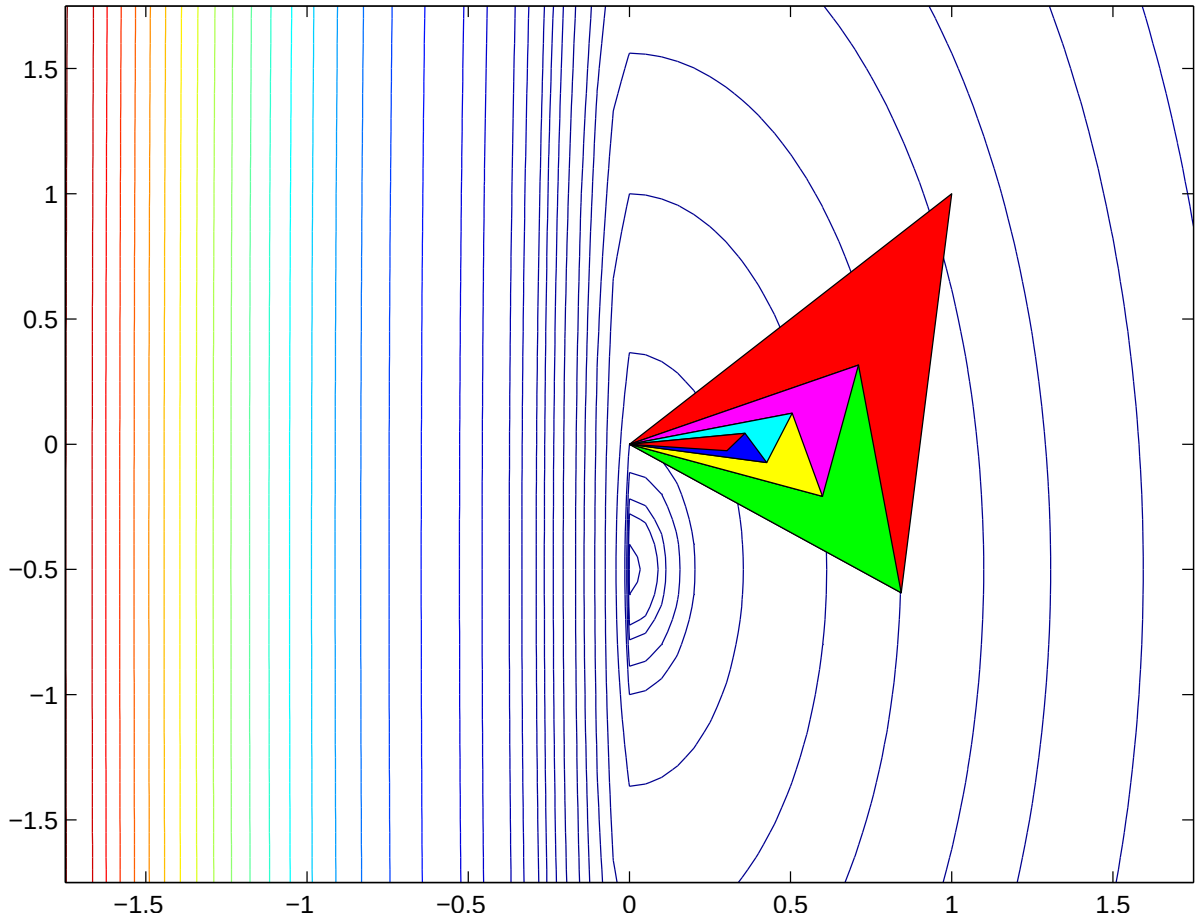
All of these involve imprecise and ambiguous terms.

What's the role of "difficult" problems?

Obviously it's pointless to test methods on problems that are known to be "easy" for all methods.

Optimization researchers care a lot about extreme and/or pathological situations, which can provide insights into the modes of failure and poor performance.

And all mathematicians love counterexamples!

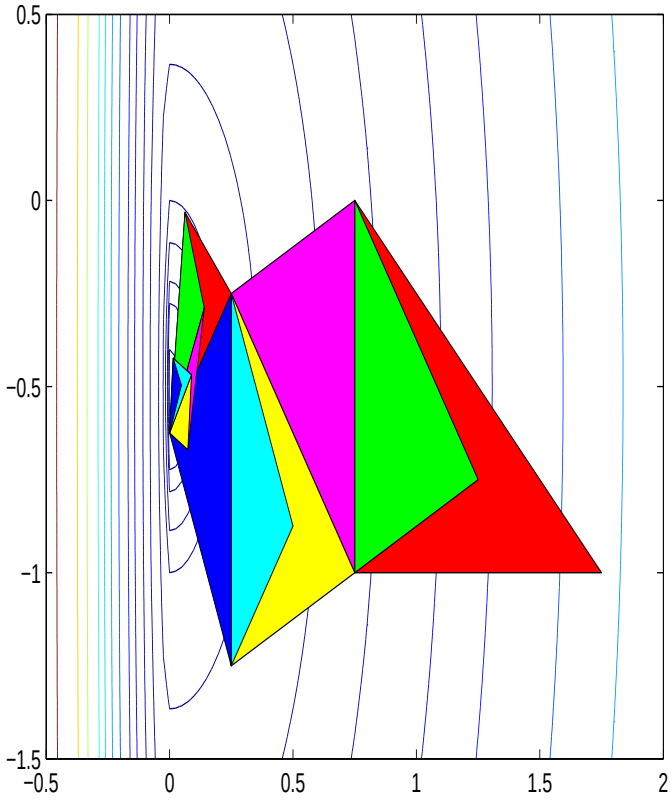


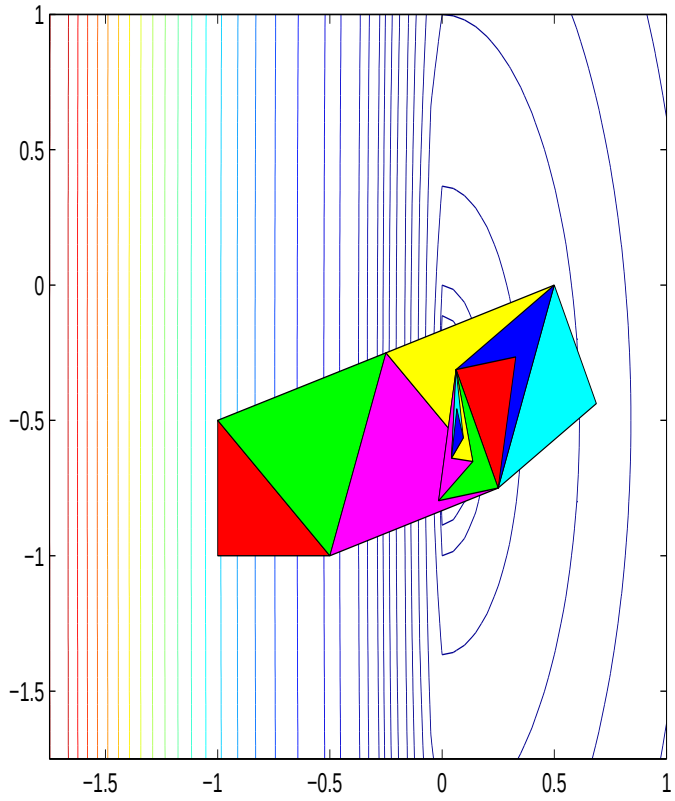
The remarkable McKinnon counterexample to Nelder–Mead.

But how much do practitioners really care (and how much should they care) about a few bad examples, especially if the method works well for their problems?

Sometimes, very little.

And Nelder-Mead works perfectly well on the McKinnon example if the starting simplex is not the pathological one. . .

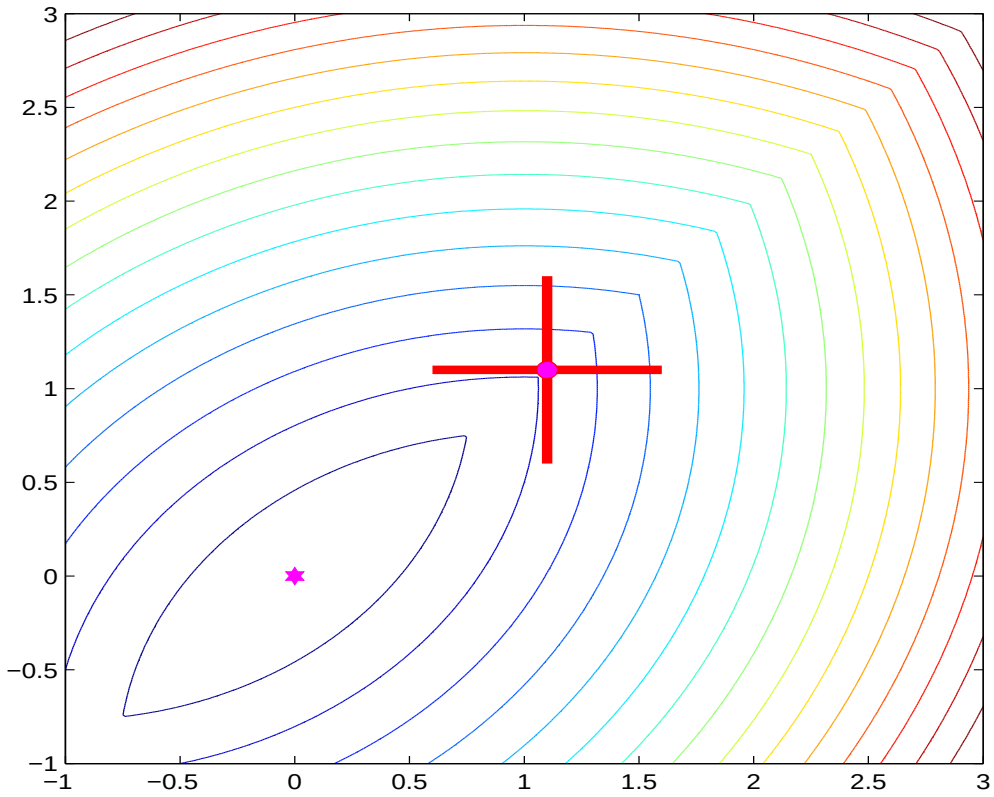

0.5
0
−0.5
−1
−1.5
−0.5
0
0.5
1
1.5
2


1
0.5
0
−0.5
−1
−1.5
−1.5
−1
−0.5
0
0.5
1

Since problems from real-world applications are often inordinately complicated, time-consuming, and/or proprietary, we need "artificial" ("academic") test problems that are agreed to display the important features of the real-world problems.

For example, several real-world applications are known in which there are structural discontinuities in the gradient.

Hence the Dennis-Woods example (next) is extremely useful because it illustrates a generic failure of pattern search methods, including coordinate search, and has suggested generalizations to overcome this form of failure.


3
2.5
2
1.5
1
0.5
0
−0.5
−1
−1
−0.5
0
0.5
1
1.5
2
2.5
3

A further hazard of comparative numerical testing is that the testers can upset their friends and colleagues, based on the standard litany of issues involving software authors:

- "You didn't use the latest version";
- "You didn't use appropriate values for the optional parameters";
- "My software was never intended for problems of the kind being tested".

For those who are sufficiently brave, rigorous comparative software testing invariably exposes room for improvement and/or weaknesses and/or defects in the codes being compared, even if it's not clear that the results generalize.

As the 15-author paper "Comparison of derivative-free optimization methods for groundwater supply and hydraulic capture community problems", by K. Fowler et al., 2008, diplomatically states in its conclusions:

> Algorithmic maturation is expected for all methods and is already underway for some of the methods. These algorithmic changes can reasonably be expected to improve performance for this challenging set of test problems.

Software is a moving target.

A major numerical issue related to reliability of direct search methods: even though far from the optimum, they can become stuck in a rut, stagnate, bog down,. . .

In these cases, something needs to be done to escape stagnation.

But. . . direct search methods can also cease to make progress when they are close to the optimum, in which case they really should stop.

And it is often almost impossible to distinguish these two situations numerically.

This dilemma has been known for a long time.

Powell (1964)[a]: "... a compromise has to be made between stopping the iterative procedure too soon and calculating f an unnecessarily large number of times".

Swann (1974)[b]: "... [direct search methods] are more liable [than gradient-based methods] to end prematurely or to prolong the search unnecessarily".

---

[a] An efficient method for finding the minimum of a function of several variables without calculating derivatives, *Computer Journal*.

[b] Constrained optimization by direct search, in *Numerical Methods for Constrained Optimization*.

To escape stagnation, Brent (1973)[a] proposes adding a random step when recent steps have not improved the current approximation to the minimizer.

The idea is based on PARTAN (parallel tangents),[b] a strategy for accelerating the slow convergence of steepest descent.

---

[a] *Algorithms for Minimization Without Derivatives*, Prentice–Hall.

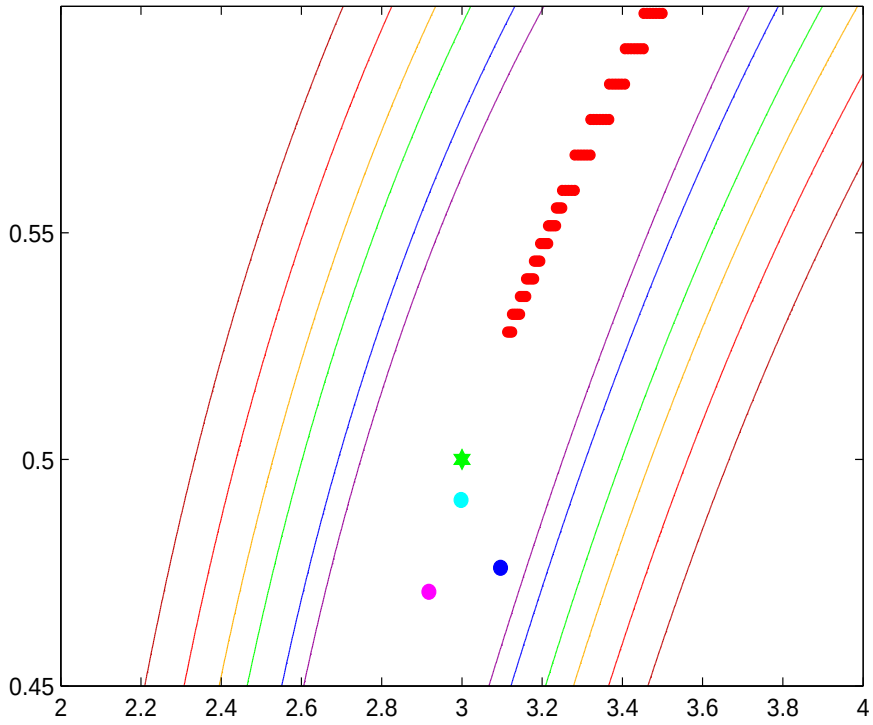
[b] Shah, Buehler, and Kempthorne (1964), Some algorithms for minimizing a function of several variables, *SIAM Journal on Applied Mathematics*.

Recent work in this domain:

A very interesting May 2011 Argonne report by Larson and Wild, "Non-intrusive termination of noisy optimization", proposes several families of parameterized termination criteria, along with extensive numerical tests that suggest guidelines for use in practice.

MHW: trying to update the idea of adding randomness to avoid stagnation and/or detect when a direct search method should terminate.

A typical picture from many experiments, where the red blobs show stagnation, the optimal point is magenta, and the cerulean point is the "breakaway" point.



(Anodyne) concluding thoughts on numerical advances in direct search methods: much remains to be done about

1. Adapting direct search methods to take advantage of evolving high-end computing hardware, and
2. Defining procedures and criteria, based on user-provided information as well as careful sampling, that will increase the probability of reasonable termination while avoiding stagnation.

A much more difficult task:

Concluding thoughts about Sven. . . how to convey his many contributions as a scientist and a person??

The only feasible solution—we can take his scientific contributions as a **self-evident hypothesis** with no need of proof.

And to represent Sven's *joie de vivre*, we can turn to one of the favorite images in Beatrix Potter, a **lively and lovable** mouse dancing a jig.

Happy birthday, dear Sven!!