Brief Introduction to Mathematical Optimization

Zane Beckwith
UNC-CH CAP-REU Program

Summer 2013

These notes are a very brief introduction to the very not-brief field of mathematical optimization. I want to stress that this tutorial does not pretend to be complete, and its author does not pretend to be an expert on the field. Because I'm no expert, I warn you to take everything I say with a few grains of salt and always check what I say before you use it in an important research context. *Caveat emptor*!

Code examples/exercises will be in Python (Python 2.x). The SciPy package has a module 'scipy.optimization' that has many optimization tools. For further exploration, or for actually using optimization methods in your work, this would be a good place to start looking.

What Is Computational Optimization?

Mathematical optimization seeks to find the 'best' choice among a set of possibilities; of course, we also want to find this best value with a minimum of computational effort. Think of these examples: finding the lowest point on a radar image of the seafloor; determining reaction constants that best describe observed reaction rates; or ordering the cities visited during a trip so as to minimize the gas used.

The set of choices is frequently referred to as the 'search space'; much of the time (particularly, in this tutorial) the search space is simply some subset of Euclidean space. The way we define 'best' is as the minimum of some function f, the 'objective function'. The objective function is a mapping from the search space to the real numbers. Only discussing minima is perfectly general: if your problem actually wants you to find the maximum of some function (e. g. number of users of your website), that's the same as minimizing the negative of the function.

One of the most common examples of optimization in scientific contexts is function fitting: you have some data and want to find a function that describes it. The search space in this case would be the parameter values that characterize the possible functions (e. g. slope and y-intercept for linear functions), and the objective function would be some measure of the distance of your data from the fitted-function's values. Least-squares fitting is the classic example of this type of problem.

There are, however, many many types of questions that may be phrased as optimization problems, and there have thus developed many many methods for finding solutions to these types of problems. Here, I would like to give some examples of the ways the field of mathematical optimization can be divided. The point is that you should get some idea of where your particular problem lies, because each domain probably has methods designed to work well within it, methods that probably don't work well in other domains.

These categories are not mutually-exclusive (an optimization problem can be both linear and convex, for example) and are certainly not exhaustive. In each dichotomy, I've tried to name first the division that is, in general, 'easier' to perform.

You may want to skip the next subsections and go straight to the exercises below if strapped for time. You can come back here if you need to.

Heuristic vs. Algorithmic

An algorithm is a method that, if followed fully and correctly, is known to obtain the best answer; a heuristic should give you a good-enough answer, but maybe not the best, and sometimes it may be a terrible answer. Unfortunately, sometimes a heuristic method is all that can be found for a particular problem. It's always important to know which methods are heuristic and which are algorithmic. If you're using an alleged algorithm, make sure you know for which types of problems the algorithm is in fact known to find the best answer; you may think you can assume the result is the best answer, but if you're giving it the wrong kind of problem you may be wrong.

Local vs. Global

Related to the issue of algorithmic vs. heuristic is the difference between a local optimum and a global one: just like optima for the calculus of single-variable functions, a local optimum means no *nearby* choices are better, but there may be better choices elsewhere in the search space. Methods that find local optima are frequently faster and easier to use; you just have to make sure the result you get is the true global optimum. A quick way to do this is to start a local optimizer in multiple regions of you search space. However, this is clearly tedious, so if you think there's a high likelihood of getting stuck in a false minimum, know your options (a stochastic method may be smart; see below).

Linear vs. Non-linear

Much of the classic work on optimization focuses on optimizing linear functions of the search space: 'linear programming' is what it's called. While many problems can in fact be described, or at least approximated, by linear functions, many cannot. However, if yours can be, the linear methods are probably faster, and maybe algorithmic.

Convex vs. Non-convex

Convex is a property of functions that essentially boils down to "There's only one minimum and you can't get trapped in false-minima". As with linear problems, if yours is a convex problem, the methods available to you are refined and powerful.

Continuous vs. Discrete ('Combinatorial')

Though we're focusing on problems in which the search space is continuous, there are many problems in which it's discrete (e. g. "Which route between these cities takes the least amount of time?"). The methods for these two classes of problem are, not surprisingly, quite different, so make sure you don't try to fit a square peg into a round hole.

Deterministic ('Gradient') vs. Stochastic

Many of the classic optimization methods are deterministic, meaning that if you start the same problem with the same initial conditions, you're going to get the same result with the same intermediate steps. Because these methods frequently rely on calculating gradients to find which direction to move for finding the minimum, they're frequently known as 'gradient' methods.

However, deterministic methods can have issues with getting stuck in false minima. To combat this, you can use a stochastic method, which, in one way or another, uses random number generation to sample larger portions of your search space. In general, though, stochastic methods, with their random number generation requirements, are going to be computationally-costly compared to traditional deterministic methods. Many commercial optimization packages actually use some combination of the two.

Example: Scattering Cross-Section Simulation

So far, this has been pretty general and vague; let's get down to brass tacks with a (contrived) example.

Let's say you work in a nuclear physics lab, scattering a beam of particles off a target to study some nuclear-astrophysics process. The beam apparatus is controlled by two parameters: the energy of the particles, and the projection of the particles' average spin direction along the beam direction (the 'helicity').

When designing the experiment, you want to use the parameter values that give you the highest cross-section, so you can observe the largest-possible number of events.

Luckily, one of your collaborators has provided you with a program that simulates the collision and reports the scattering cross-section, given your two parameters as inputs. Let's call their simulation program 'Scatter'. Unfortunately, we're going to assume that this program is somewhat computationally-intensive, so you can't simply play around with it to find what appears to be best.

What you need, then, is to find the *optimal* energy and helicity that give you the highest cross-section. To do this, we're going to use the Nelder-Mead optimization method.

Nelder-Mead or 'Downhill Simplex' Method

The Nelder-Mead method is a fairly-simple, deterministic, heuristic method for finding optima in non-linear, not-necessarily-convex, continuous problems. However, it's known to have a problem with returning non-optimal results. For that reason, these days more sophisticated methods are usually used; for the purposes of illustration, though, it's fine.

Python's scipy.optimize.fmin function uses this method.

This method uses a simplex to probe the values of the function. A simplex is just the simplest shape you can make in N dimensions by using $N+1$ points; for example, a line in 1-dimension, or a triangle in 2-dimensions.

The Nelder-Mead method samples the function at the $N+1$ points of its simplex, then alters the points in the simplex to feel its way to the minimum of the function. Specifically, it:

Reflects Flipping itself end-over-end

Expands Pushing one point away from itself

Contracts Pulling one point in toward itself

Reduces Shrinking all points inward

The method has a nice intuitive visualization, as in this gif of a two-dimensional example; the grey simplex is the previous position, and the red is the current. The reason for the name 'Downhill Simplex' is pretty clear from watching this.

You can take a look at my source code for 'neldermead.py' to get a better view of how these steps are implemented; it's surprisingly-straightforward.

Give It a Try

Now that we understand our method, it's time to get busy. In the "src" directory, above this one, you're given the python module I've written to implement the Nelder-Mead method ('neldermead.py') as well as the 'simulation code' for Scatter ('scatter.py').

Assume that the energies and helicities you're interested in are located between -5 and 5 (don't worry about units). Now, go and find the best values for your beam!

What did you find?

Did you try multiple tolerance values? What happened?

Did you try multiple starting simplices? What happened?

You have the source code for the Scatter 'simulator', so you can go ahead and peak inside. You may find the Wikipedia page on "Himmelblau's function" interesting. . . .

Also, you can go into 'neldermead.py' and play with some of the parameters, particularly α, etc.

Taste of Stochasticity: Simulated Annealing

As mentioned above, deterministic methods like the Nelder-Mead can sometimes get stuck in false minima: the simplex just sniffs out the decreasing direction, but if there's a lower valley far away, it has no idea.

To combat this, you can let you method jump around all over your search space. Now, rather than falling in love with what *locally* looks like the minimum, the method can sample minima far away.

Such methods are known as stochastic methods. Clearly, they're very powerful for finding global minima. However, this power comes with a price: stochastic methods are generally computationally more expensive than deterministic ones, and they can be fussy to get working properly.

One of the most famous stochastic methods is Simulated Annealing. This method is closely related to Monte Carlo methods (specifically, the Metropolis method), and simulates the kind of 'hopping' between states that systems do when at a finite temperature.

Just as materials are heated then cooled rapidly ('annealed') to trap them in a particular state, your objective function can be sampled by a large number of differing choices (this is high temperature), with the likelihood of a choice jumping to a very different choice decreasing with time (this is the lowering of the temperature, the annealing).

Wikipedia has a nice page on simulated annealing, in particular a nice gif of the method finding the highest point in a 1-dimensional function.

The python package SciPy has, in addition to many nice minimization and optimization modules, a module for simulated annealing: 'scipy.optimize.anneal'. With the remaining time, you may want to check out this module, as well as the Wikipedia page.