Unconstrained Numerical Optimization
An Introduction for Econometricians

Fall 2021, version 2.0

Advanced Microeconometrics

Anders Munk-Nielsen

Abstract

This note introduces unconstrained numerical optimization and its use in Python. The note is structured with the intention that details can be skipped if the reader is not interested in them. The example in Section 4 will illustrate how the optimization theory works in practice in Python and in particular give an example where it does not work and explain why. Section 5 is intended to be used for quick reference when things go awry.

Contents

1 Introduction

In this note, we will consider the problem,

$$\min_x f(x). \tag{1}$$

Note that any maximization problem can be turned into a minimization problem by minimizing -1 times that function, so it is completely general. The formulation nests M-estimators directly, for example the likelihood function can be written as the minimization problem,

$$\min_\theta N^{-1} \sum_{i=1}^{N} -\log l(\theta|y_i, x_i),$$

where (y_i, x_i) is data for observation i and l is the likelihood function. To make notation simpler to read, we will subsume the average in this note and just work with a function f. Note that in most econometric applications, the objective will also be a function of data (like y_i and x_i above), but this is kept fixed throughout estimation, so we only consider the objective as a function of parameters.

Throughout this note, we will be discussing methods for finding *local optima*. In general, there is no way of ensuring that one has found the global optimum (in fact, it can be proven that unless one's algorithm visits every single x with some positive probability, this can never be guaranteed). The only generally accepted method for ensuring that the *global* optimum has been found is using a *multi-start procedure*, where the optimization algorithm is started at many different initial values.

Optimizers fall into two overarching groups;

1. Gradient-based,
2. Gradient-free.

The Nelder-Mead algorithm is the only example of a gradient-free optimizer that will be discussed herein. From personal experience, gradient-based optimizers are faster but require the problem to be "nicer" (i.e. more smooth, look more like a polynomial). The Nelder-Mead algorithm is typically very useful when the starting values are bad or the problem is difficult, e.g. if complicated calculations are involved which might result in roundoff error, producing noise in the evaluation of f. However, be warned that if you have an error in your criterion function, it will often cause trouble for the more fine-tuned gradient-based whereas Nelder-Mead will happily work on your weird error-filled function which is most likely not well-behaved or convex.

We start by discussing gradient-based methods and then turn to a brief discussion of the Nelder-Mead algorithm in the end. Table 1 gives an overview of the optimizers we will cover.

Table 1: Overview of optimizers

	Newton	BFGS	BHHH	Nelder-Mead	Steepest Descent
`method`	User written	`BFGS`	`CG`	`Nelder-Mead`	n.a.
Option	–	[default]	Provide user-written Hessian		
Gradient used	✓	✓	✓	÷	✓
Hessian used	✓	✓	✓	÷	÷
Step	$f'(\cdot)/f''(\cdot)$	$f'(\cdot)/f''(\cdot)$	$f'(\cdot)/f''(\cdot)$	Heuristic	$\gamma f'(\cdot)$
Hessian	Numeric	Iterative updating	Outer product	Not used	Not used
Best for	Nice f but weird Hessian	Nice f	Likelihood estimation	Nasty f	Non-convex or non-quadratic f
Iterations	Medium	Few	Few	Many	Many
Globalization	Line search	Line search	Line search	n.a.	Line search

2 Gradient-based Optimization

Intuitively, one can think of the core step of an optimization problem as: given an initial point x_0, what is our best guess of the minimizing value of x? In other words, what should the next value of x be. Most methods rely both on the *slope* and *curvature* of the function at x_0 (i.e. first and second derivatives). The exception is the steepest descent method, which only relies on the first derivative.

2.1 Steepest Descent Methods

The steepest descent (also called gradient descent) algorithm relies only on the first-derivatives of f at x_o. It looks at the slope of f and moves downhill. The challenge is how far to go. If we just keep taking tiny steps downhill, we will eventually reach a local minimum. However, it may take very long for us to get there. Therefore, the step we take is simply proportional to the derivative, i.e. for $x \in \mathbb{R}^1$,

$$x_1 = -\gamma f'(x_0),$$

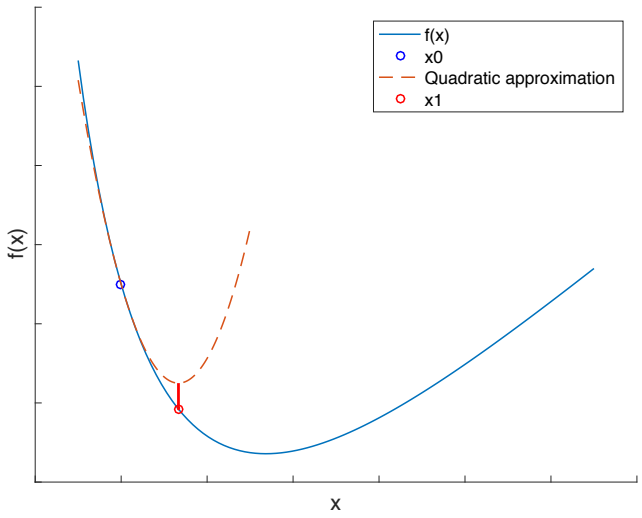
and for vector-valued x,

$$x_1 = -\gamma \nabla f(x_0).$$

The *step size*, γ, is allowed to change with every iteration. It is typically chosen based on *line searching*, outlined in Section 2.5.

The algorithm can be awfully slow. In the example in Section **??**, the steepest descent algorithm requires 56,649 function evaluations to do find the minimum, compared to just 20 function evaluations for the default newton-based algorithm. The gradient descent algorithm is simple to understand but is almost always outperformed by Newton-based algorithms.

Figure 1: Function with Quadratic Approximation



Except in the case of highly non-quadratic or non-convex problems; in those cases, the Hessian is not at all informative about where to go whereas the gradient descent algorithm will just keep going down hill.

2.2 Newton-based Methods

To do this, we will form a 2nd order Taylor approximation to f and set the next x to be the minimizer for that approximation. A 2nd-order Taylor approximation is the best quadratic function that approximates f in a neighborhood of x_0. Once we have taken the step and gone to the x that minimizes the quadratic approximating function, we will form a new approximation at this new value of x and proceed in that manner until we arrive at a stationary point (i.e. where the gradient is the zero vector). In Table 1, it says that the Newton-based optimizers (Newton, BFGS, BHHH) work well when "f is nice". What is meant by this is that the function cannot be too far from a quadratic function and should in particular be convex (most places). If f is very far from a quadratic function, of course the quadratic approximation will be poor and we are in trouble.

Let us start by considering the simplest version of the problem where x is a scalar. Then the approximating quadratic function that we are minimizing is

$$\min_{x \in \mathbb{R}} f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2} f''(x_0)(x - x_0)^2.$$

Figure 1 illustrates this graphically. We are constructing a quadratic function that at the point x_0 has the same slope as f, namely $f'(x_0)$, and the same curvature, $f''(x_0)$, and then

we move to the bottom of this function and assign our new iterate, x_1, to the x-value here.

In the more general, x is a $P \times 1$ vector, and the problem takes the form

$$\min_{x \in \mathbb{R}^P} f(x_0) + \nabla f(x_0)'(x - x_0) + \frac{1}{2}(x - x_0)'\nabla^2 f(x_0)(x - x_0).$$

Here, $\nabla f(x_0)$ is the $P \times 1$ vector of first derivatives (called the *gradient*), so that the term $\nabla f(x_0)'(x - x_0)$ is scalar. The Hessian matrix, $\nabla^2 f(x_0)$, is the $P \times P$ matrix of 2nd derivatives, so the last term is a quadratic form and therefore also scalar. Alternatively, we may formulate the model in terms of the *step*, which we can write as $s \equiv x - x_0$, and the problem becomes

$$\min_{s \in \mathbb{R}^P} f(x_0) + \nabla f(x_0)'s + \frac{1}{2}s'\nabla^2 f(x_0)s. \tag{2}$$

Intuitively, when we replace (1) with (2), we are finding the quadratic function (2nd-order polynomial) that is the best approximation to f at the point x_0. Then we minimize that function rather than the original f. Therefore, if f is locally concave, we will go in the wrong direction. Similarly, if f is highly "non-quadratic" (so that the third term in the Taylor expansion is very important), the approximation is very poor and we may for example "over-shoot" and take a too long step. To avoid these problems, the quadratic optimization problem (2) should be accompanied by a *globalization strategy*; this is a way of trying to avoid overshooting (and sometimes also undershooting; then it is called a *greedy* globalization strategy). See Figure 2 for an example where the function is locally very non-quadratic at x_0, resulting in the quadratic approximation overshooting the minimum. Intuitively, the gradient and Hessian of f will give us a *direction* along which we will search for candidate improvements. We will first try the most obvious candidate, but if that turns out not to result in an improvement, the globalization strategy will indicate how we should proceed.

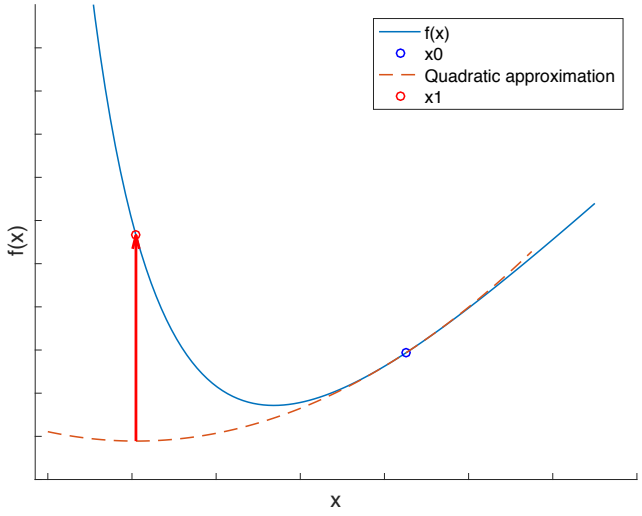
You may be wondering why we only choose a quadratic approximation to the true function and do not proceed to a cubic (3rd-order Taylor approximation) or even higher. The reason is that minimizing a quadratic function is very simple and requires only linear algebra, which is extremely fast. Going to a higher-order polynomial would require us to do a more complex optimization problem to minimize the approximating function, which defeats the purpose of minimizing a local approximating function.

In order to come up with an algorithm, we will now need to do decide on three components;

1. How to choose the step, s,
2. A globalization strategy,
3. A method for minimizing the quadratic function.

Let us postpone the globalization strategy for a while and focus on the other two. We will first

Figure 2: Local Non-quadraticness Leading to Overshooting



discuss the scalar case, then consider the vector case and finally we will add the globalization strategy.

2.3 Scalar x, no globalization strategy

In the simplest case, we are solving the problem

$$\min_{s\in\mathbb{R}} f(x_0) + f'(x_0)s + \frac{1}{2}f''(x_0)s^2.$$

Interior solutions will satisfy the first-order conditions,

$$\begin{aligned} \Rightarrow \text{FOC} : f'(x_0) &= -f''(x_0)s \\ \Leftrightarrow s &= -\frac{f'(x_0)}{f''(x_0)}, \end{aligned} \tag{3}$$

assuming that $f''(x_0) \neq 0$. The intuition is that $f'(x_0)$ gives the direction while $f''(x_0)$ tells us how large of a step, we should take; when $f'(x_0) < 0$, the function slopes downwards and we should take a step in the positive direction in order to reduce the function value (and vice versa). When $f''(x_0)$ is small (but positive), it means that the slope of f only changes very slowly, indicating that we should be taking a larger step (and vice versa).

2.4 Vector x, no globalization strategy

Consider the case where $x \in \mathbb{R}^P$. Here, the first-order conditions necessary for interior solutions to (2) yield

$$\text{FOC} : \nabla f(x_0) = -\nabla^2 f(x_0) s.$$

If f is strictly convex at x_0, then $\nabla^2 f(x_0)$ is positive definite and we can invert the equation no obtain

$$s = -[\nabla^2 f(x_0)]^{-1} \nabla f(x_0). \tag{4}$$

This is simply the vector-generalization of (3) and the intuition is the same. A very simple minimal working minimizer might thus look like below.

```
def min_newton(f, grad, hess, x0, maxit):
        ''' minimize function using quadratic approximations '''
        x = x0 # initialize
        for it in range(maxit):
                g = grad(x)
                H = hess(x)

                # compute step
                step = -np.linalg.solve(H,g)
                x = x + step

                # check convergence
                if np.max(np.abs(g)) < tol:
                        break

        return x
```

Note that the minimizer never actually uses the function itself! Therefore, it does not check if the new updated value of `x` results in an improvement or not. We will get back to this later when we discuss how we might expand the algorithm to check this and what to do if the new candidate results in a *worse* function value (this is called the *globalization strategy*).

Next, note that the following two lines are equivalent

```
# two equivalent ways of computing the step
step = np.linalg.solve(H,g) # computationally fast
step = np.linalg.inv(H) @ g # computationally slow
```

One should avoid calculating the inverse of the Hessian unless needed – it is a waste of computational resources.[1] The "`solve`" function solves linear systems of the form $As = b$ for

[1] In Matlab, you would write `s = -hessian\gradient`.

s, where A is $P \times P$ and s, b are $P \times 1$. It turns out, that this system can be solved without ever computing A^{-1} and thus saving computational time and memory resources.[2]

In spite of using `solve`, the $P \times P$ system may become unwieldy if P is too large. Consequently, there are alternative ways to `solve` that can be even smarter:

- Cholesky method,
- Conjugate gradient method.

Both methods are computationally superior to finding the inverse of the Hessian as in (4) and yield almost identical solutions. The Cholesky method is a more direct method whereas the conjugate gradient method iteratively computes candidate solutions that become better successively. When P is large, the Cholesky method may be slow and the conjugate gradient method will speed things up. This is applicable if the evaluation of f is generally very fast (several evaluations per second). For most econometric problems, it is not important.

If f is non-convex, then we may not even be able to solve the equation. Similarly, if our numerical calculation of the Hessian is poor, we may get trouble with finding the step size in equation (4). In these cases, it is generally advisable to check for errors in the code that calculates f and otherwise maybe try to see if a gradient-free optimizer can break free of the locally non-convex area. There are alternative methods for dealing with the inversion in the case of non-positive definite Hessian matrices, but the general advice is to hope that the problem disappears as we get towards the minimum.

2.5 The Line Search Algorithm

The intuition in the line search algorithm is that we search for the optimal step along the one-dimensional line indicated by (4). That is, we choose the step

$$s = -\lambda[\nabla^2 f(x_0)]^{-1}\nabla f(x_0),$$

where $\lambda > 0$. Typically, we start out with $\lambda = 1$ and if that fails, we try $\lambda = 0.5$, then $\lambda = 0.25$, etc. If the problem is highly non-convex and has a very small Hessian locally, then it may result in s being too large and dampening it with $\lambda < 1$ can then help. This approach is known as the "backtracking Armijo line search". If one implements a "greedy" version of the line search, it could also be possible to experiment with $\lambda > 1$ but that is not the standard case; the primary purpose of the line search is to avoid situations where we overshoot the minimum and end uphill anyway. Figure 2 illustrates such a case of overshooting; note how the objective function does not change much at the initial point (low Hessian). This is the non-quadraticness that results in the too large step.

[2] When A is positive definite, one can find a lower triangular matrix L such that $A = LL'$ and find $s = (L')^{-1}L^{-1}b$, which is computationally far simpler.

If it is not possible to find an improvement this way, the optimizer gives up eventually. It the returned dictionary will have `success` set to `False` and have the following `message`:

```
message: 'Desired error not necessarily achieved due to precision loss.'
```

This problem can happen for many reasons. If you are in a local non-convex region, you can switch to Nelder-Mead to see if it can break free. Otherwise, it can be a sign of errors in your function, outliers in your dataset resulting in loss of numerical precision, etc.

There are alternative implementations of the line search where one requires not only a decrease in f, but a *sufficiently large* decrease in f according to some inequality. This results in a rule for whether or not to *accept* any proposed step by the algorithm.

2.6 The Trust-region Method

The idea is similar to the line search algorithm in that we want to avoid the optimizer taking a huge step and overshooting the true minimum. Instead of searching for the optimal step size, the trust-region restricts the range of s-values that we search for in the quadratic problem, replacing (2) with the *constrained* quadratic problem

$$\begin{aligned} &\min_s f(x_0) + \nabla f(x_0)'s + \frac{1}{2}s'\nabla^2 f(x_0)s, \\ &\text{s.t.} \quad \|s\| \leq \Delta, \end{aligned} \tag{5}$$

where Δ is the trust region's *radius*. Note that the constraint defines a ball in $\mathbb{R}^P$ if the Euclidean norm (the typical distance), $\|\cdot\|_2$ is used. A trust-region algorithm now requires two more ingredients,

1. A method for solving the constrained quadratic program (5),
2. A rule for updating Δ.

There are many linear algebra methods for solving such problems and since they will not be the computational bottleneck in typical econometric applications, we will not worry about them here. The rule for updating the radius will be based on a comparison of the realized decrease achieved in f after taking the step compared to the "expected decrease in f" (this latter bit of course requires some model). If the decrease is smaller than expected, the radius is increased and vice versa.

2.7 Gradients

No matter the choice of globalization strategy, both the line search and the trust region relies on a gradient vector, $\nabla f(x_0)$, and a Hessian matrix, $\nabla^2 f(x_0)$. In this section, we cover the gradient and in the next section the Hessian.

There are generally two methods for obtaining the gradient;

1. Analytic gradient (more precise, but costly in human computational time),
2. Numerical gradient (using some finite-difference approximation).

Generally, the analytic gradients are preferable if they are available but it can take time to code them up and verify their correctness. Using analytic gradients tends to result in more precise and faster optimization. Additionally, optimization will be more robust to numerical error in the evaluation of f (so long as the same error is not present in the evaluation of ∇f). Below is an example where the Rosenbrock function has been coded in Python (the input, `x`, should have two elements).

```
def rosen(x):
        ''' evaluates the Rosenbrock function '''
        return 100 * (x[1] - x[0]**2)**2 + (1.0 - x[0])**2
```

When gradients are not supplied analytically in Matlab, the optimizer will obtain them using numerical differentiation. That is, it will approximate it by

$$\left.\frac{\partial f}{\partial x_k}\right|_{x=x_0} \cong \frac{f(x_1) - f(x_0)}{h}, \quad \text{where } x_{1j} = \begin{cases} (1+h)x_{0j} & \text{if } j = k \\ x_{0j} & \text{otherwise.} \end{cases} \tag{6}$$

That is, x_1 is the same as x_0 except at the coordinate to the entry, which we are taking the derivative with respect to. Note that the step is *relative* — this is by far superior to using the same absolute step regardless of the magnitude of each x_{0k}. The finite difference in (6) is a *forward difference*; alternatively, one can use a *centered difference*, which is more precise but requires two evaluations of the objective function for each parameter (so $2P$ evaluations), which is often too expensive to be preferable. Instead, (6) re-uses $f(x_0)$ (which can also be supplied to the gradient calculator) and thus only requires $P + 1$ evaluations.

Generally speaking, the step size should be set to

$$h^* = \sqrt{\epsilon},$$

where ϵ is *machine precision*. In Python, this is obtained from `np.finfo(float).eps`:[3]

$$\epsilon = 2.2204 \cdot 10^{-16}.$$

This is the distance in floating point arithmetic (how the computer represents numbers) between 1 and the closest number on the real line in the positive direction. That is, if you

[3] In Matlab, it is the builtin variable, `eps`.

type `1 + eps/2`, you will get precisely 1 (in fact, `1 + eps/x` for any `x>1` will return just `1`). Since there are 52 bits available to represent the number, $\epsilon = 2^{-52}$.

Why is it then optimal to choose $h = h^*$? This is because with finite differences, there are two sources of error:

1. Approximation error,
2. Roundoff error.

The first error comes from the fact that we are using a finite difference approximation. As you may recall from basic calculus, the true derivative at the point x_0 is the *limit* of the forward differences in (6) when $h \to 0$. The second source of error is perhaps new to many economists and comes from the fact that a computer cannot represent all the real numbers and therefore makes roundoff error. This problem becomes particularly bad when the numbers we operate on are extremely small (such as subtractions between numbers that are very close or division by very dis-similar numbers). In the subsection below, we will discuss this more and consider what h will optimally balance these two errors off. The general rule of thumb is to choose $h = 10^{-8}$, which is also (close to) the default in most software packages. If the function is very complicated and involves many steps that might lead to roundoff error, it may be advisable to work with a larger step size, e.g. $h = 10^{-6}$, because the approximation error may be much larger in that case.

2.7.1 Optimal Step Size h

Let us first try to understand roundoff error better. It comes from the fact that the computer works with finite precision. Not every single real number can be represented in the computer's memory. Instead, it will constantly be choosing the closest number to the ones it can to the ones we ask it to store (*some* numbers of course coincide and luckily, for example all the natural numbers are among such). For example, in Python, with `eps = np.finfo(float).eps`,

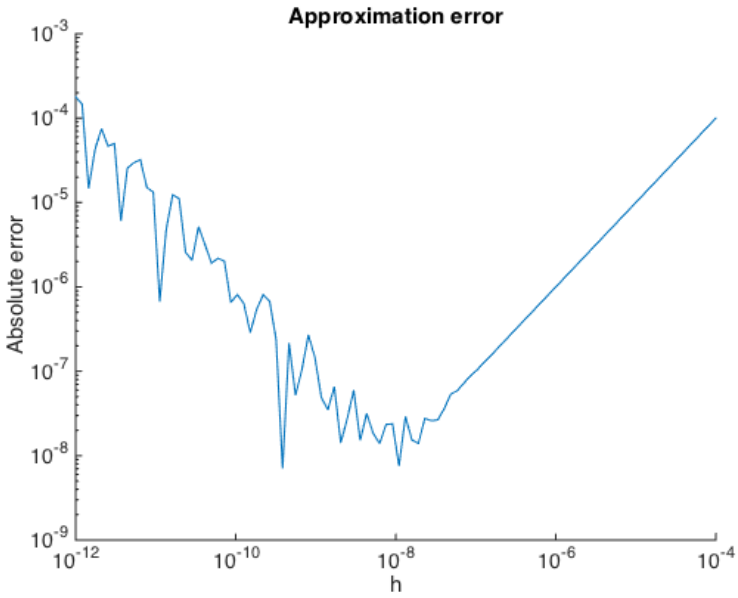
$$\texttt{10} + \texttt{eps} - \texttt{10} = \texttt{0},$$

because the number `10 + eps` cannot be represented in machine precision. However,

$$\texttt{10} - \texttt{10} + \texttt{eps} = \texttt{2.2204e-16},$$

so the sequence of operations matters when working with numbers on a computer. It turns out that in particular subtraction and division can cause problems on a computer. So *the smaller the number we are dividing by and the closer the numbers we are subtracting, the larger the roundoff error*. This is why we should not set h too small. Figure 3 shows the approximation error in a finite difference approximation to the derivative of the function $f(x) = x^2$ at $x_0 = 1$

Figure 3: Approximation Error and the Choice of h



for different choices of h. On the right hand side, the error is dominated by the *approximation error* (and it appears to be a smooth function in h). On the left hand side, it is dominated by *roundoff error* (and that error looks much more jittery and unexpectable, as one might suspect).

2.8 Hessian

There are three ways in which to obtain the Hessian:

1. Analytic Hessian,
2. Clever updating schemes (e.g. the BFGS),
3. Using an approximation derived from theory (e.g. the BHHH for Maximum Likelihood),
4. Numerical Hessian.

The analytic Hessian is obtained by doing derivatives with pen and paper (or with some symbolic mathematics software, like Maple), and coding them up as a part of the criterion function. From personal experience, it is not simply the case that the the analytic option is always better. Mainly because it is extremely time-consuming to derive the Hessian analytically and very hard to verify that it has been done correctly.

In the other extreme, the fully numerical Hessian is easy to use because it is always available; we just take finite differences of the finite differences (essentially using equation (3)

P times for each of the P entries in the gradient). However, it is costly to evaluate using finite differences, requiring P^2 evaluations of the f function. This is why in Table 1, it says that the pure Newton method is medium-costly; it might get to the minimum in a low number of *iterations*, but at each step, it will have to evaluate the objective function P^2 times in order to obtain the Hessian.

In between these two extremes are the BFGS updating scheme and the BHHH approximation. The BHHH algorithm uses an approximation to the Hessian that relies on the gradient only. The approximation is only valid in certain models, most notably maximum likelihood. The BFGS algorithm is the default in `scipy.optimize.minimize` (and in Matlab in `fminunc`). It uses information on the current and previous x and ∇f to make an informed guess as to what the Hessian is. The BFGS algorithm turns out to perform extremely well in practice for a surprisingly wide class of optimization problems. In the following two subsections, we will discuss the BFGS updating scheme and the BHHH Hessian approximation. The finer mathematical details of the BFGS algorithm are not required exam knowledge; only an intuitive understanding is required.

2.8.1 BFGS

The standard implementation of the unconstrained optimizer in `scipy`, `minimize`, uses the BFGS updating scheme (named after Broyden, Fletcher, Goldfrab and Shanno). This, and other updating schemes like it, consists of using the gradient from the current and the previous iterations to construct an informed guess as to what the Hessian might look like. Specifically, the BFGS approximates $\nabla f(x_0)$ by the matrix H_0. This matrix can be initialized to be the identity matrix. Then when the step, s, has been found (subject to the globalization rule, line search or trust region), the updated Hessian, H_1, is constructed as

$$H_1 = H_0 + \frac{1}{y's} yy' - \frac{1}{s'H_0 s} H_0 ss' H_0,$$

where $y \equiv \nabla f(x_0 + s) - \nabla f(x_0)$.[4] The main advantage of the BFGS approximation of the Hessian is that it requires no additional evaluations of f in excess of what is already being done, it merely requires us to store the previous gradient's value. The update step for the Hessian just requires linear algebra, which will typically be so fast that the total computational time is unaffected since in general in econometric applications, the dimension of P is so low that it is almost only the evaluation of f that is computationally costly.

In Table 1 it says that the BFGS works well when the function f is nice, whereas the pure Newton algorithm also works well when the Hessian is weird. What I mean here is that it might for example be the case that the Hessian of the objective function changes a lot when

[4] As a side-note, both the matrices yy' and $H_0 ss' H_0$ have rank 1 and are symmetric. Together, they form a rank 2 update of the Hessian.

x changes; in that case the updating scheme used by BFGS might not work that well and it might be better to actually calculate the numerical hessian at each new point, x.

2.8.2 BHHH

The BHHH is an important algorithm for estimating likelihood models. It was first proposed by Berndt, Hall, Hall and Hausman (1974), and it uses the average outer product of the scores as an approximation to the expected Hessian. This approximation is only applicable when f takes the form of a sum (or equivalently, an average), i.e.

$$f(x) = N^{-1} \sum_{i=1}^{N} q(w_i, x),$$

(recall that x plays the role of parameters, which we otherwise label θ, and w_i is individual i's data) and when either q is the likelihood function so that the *information matrix equality* is satisfied (which it always is for maximum likelihood) or q and the model structure is such that the *generalized information matrix equality* is satisfied. In that case, the Hessian can be approximated by

$$\nabla_x^2 f(x) \cong N^{-1} \sum_{i=1}^{N} \nabla_x q(w_i, x)' \nabla_x q(w_i, x),$$

where the inner term is the outer product of the scores.[5] Thus, the BHHH step becomes,

$$s = - \left[N^{-1} \textstyle\sum_{i=1}^{N} \nabla_x q(w_i, x)' \nabla_x q(w_i, x) \right]^{-1} \left[N^{-1} \textstyle\sum_{i=1}^{N} q(w_i, x) \right].$$

The approximation has the clear advantage that since we aldready calculated gradients, no additional evaluations of the criterion function are required. However, it does require us to store the criterion value *for each observation*, which we might not otherwise be doing. In other words, we cannot use the gradient that comes out of `fminunc` to construct this outer product of the scores.

In the context of maximum likelihood, this approximation works when *i)* we are close to the true parameters, *ii)* the sample size, N, is large, and *iii)* the model is correctly specified. For example, if the starting values are very bad, this may result in poor performance.

[5] Essentially, the information matrix equality states that the expected value of the Hessian matrix (times a scalar) is equal to the expected value of the outer product of the scores, i.e.

$$\sigma_o^2 \mathbb{E} \left[\nabla_\theta^2 q(w, \theta_o) \right] = \mathbb{E} \left[\nabla_\theta q(w, \theta_o)' \nabla_\theta q(w, \theta_o) \right],$$

where θ_o denotes the true minimizing parameters (x throughout the rest of this note). The equation is discussed in Wooldridge (2010; p. 417).

2.9 Termination

The optimization algorithm can terminate for a number of reasons;

1. Gradients are sufficiently close to zero,
2. Change in x is too small,
3. Change in f is too small,
4. Step size too small (line search or trust region radius failure),
5. Maximum iterations reached.

Only termination due to criterion 1 is truly satisfactory, since the necessary condition for interior optimum is gradients being equal to zero. From that point of view, it seems strange that it should be possible to converge to a point where gradients indicate a slope of the function, but where the optimizer fails to decrease the function value along the direction (termination due to 2, 3 or 4). One possible explanation in this case is that there is a problem with the gradients. If there is numerical noise (roundoff error) in the function, then perhaps increasing the step size for the numerical gradients might yield a better gradient approximation. It might also be that the optimizer has gotten stuck in a local minimum or a local non-convexity. In that case, try using a gradient-free optimizer for a few iterations to see if it can break out of such a local problematic area.

The default maximum iterations in Python is $200P$. If the optimizer terminates due to criterion 5, the obvious solution is to just provide more iterations. Generally, econometric problems can be much less well-behaved than the simple analytic functions studied in the optimization literature. In particular when the starting values are bad.

3 Gradient-free Optimization

The Nelder-Mead method or the Simplex method is an optimization algorithm that does not rely on gradients. Its theoretical properties are much poorer than those of gradient-based optimizers; for example, once a gradient-based optimizer gets within the "region of attraction" of the true minimizer, quadratic convergence kicks in. This means that the distance to the truth will be doubling the exponent, e.g. $10^{-1}, 10^{-2}, 10^{-4}, 10^{-8}$. No similar theoretical results hold for the simplex algorithm, so it tends to use a lot of steps to get the final distance covered. This is why it says in Table 1 that the algorithm is computationally costly. Each individual iteration it takes requires only a single evaluation of the objective function, but it typically takes many many more iterations than the Newton-based solvers.

The simplex algorithm works by always keeping track of $P+1$ points, $\{(x_p, f(x_p))\}_{p=1}^{P+1}$; such a structure is called a simplex, hence the name. Intuitively, the algorithm then keeps

moving away from the point that is the highest (i.e. the worst point). If the algorithm sense that it's moving in a good direction, it will be "expanding" the simplex, taking bigger and bigger steps. When it encounters non-improvements in the direction of the lowest point, it will instead be shrinking the simplex points towards the lowest one. If you google Nelder-Mead, there are plenty of youtube videos and GIF illustrations to show examples of the optimizer at work to aid the intuition of how it functions.

The termination criteria will naturally not involve gradients; the algorithm only terminates if

1. The change in f is sufficiently small (default 10^{-4}),
2. The change in x is sufficiently small (default 10^{-4}),
3. Maximum iterations have been reached (default is $200P$).

Naturally, the third is not a satisfactory termination message. However, since the simplex algorithm can be used very effectively to break free of local minima or areas of non-convexity, you may find it useful to be switch to Nelder-Mead for a few hundred iterations before switching back to a gradient-based method. In the following section, the specific details of the algorithm is presented.

3.1 More Details

The precise details of the Nelder-Mead algorithm are shown in Algorithm 1. These details are not required knowledge but to satisfy the curious reader.

The algorithm will terminate when either the change in x or $f(x)$ becomes sufficiently small. This is computed as

$$\begin{aligned} &\text{tolerance measure in } x: \quad \|\mathbf{x}_{g+1} - \mathbf{x}_g\|_\infty < \texttt{xatol}, \\ &\text{tolerance measure in } f: \quad |f(x_{g+1}) - f(x_g)| < \texttt{fatol}, \end{aligned}$$

where the "infinity norm" for vectors is $\|\mathbf{x}\|_\infty \equiv \max_k |x_k|$.

The Nelder-Mead algorithm takes a lot of heat from mathematicians for not having nice convergence properties, but it is surprisingly robust in practice. In particular, it will always result in improvements, even with a locally non-convex criterion function. However, it can spend an inordinately large number of iterations to achieve final convergence, whereas Newton-based optimizers are very fast once they get close to the optimum (known as the "domain of attraction", where quadratic convergence kicks in, assuming the function is well-behaved).

Algorithm 1 Nelder-Mead

1. Order the points in order of function value, $f(x_{(1)}) \leq \cdots \leq f(x_{(P+1)})$ ($x_{(P+1)}$ being the worst).

2. Compute the *centroid*, $\overline{x}$, as the average of all points except the worst,

$$\overline{x} = P^{-1} \sum_{p=1}^{P} x_{(p)}.$$

3 (reflexion). Compute the *reflected point*,

$$x^r = \overline{x} + \lambda(\overline{x} - x_{(P+1)}),$$

where λ is a tuning parameter kept fixed, typically $\lambda = 1$.

If x^r is better than the second-worst but not the best one we have at this stage — i.e. $f(x_{(1)}) \leq f(x^r) \leq f(x_{(P)})$ — then we replace$x_{(P+1)}$ with x^r and go to step 1.

If x^r is the best we have seen so far, go to step 4.

If x^r is not even better than the second-worst, go to step 5.

If x^r is worse than the worst point, go to step 6.

4 (expansion). If the reflected point is better than anything we currently have — $f(x^r) < f(x_{(1)})$ — then we compute the expanded point,

$$x^e = \overline{x} + \gamma(\overline{x} - x_{(P+1)}).$$

Typically, $\gamma = 2$. Note that both x^r and x^e are on the 1-dimensional line through $x_{(P+1)}$ and $\overline{x}$. Replace the worst point ($x_{(P+1)}$) by whichever of x^r and x^e gives the best function value. This is a so-called "greedy" way of obtaining a faster convergence, by speeding up in the direction of an indicated improvement.

5 (contraction). In this case $f(x^r) \geq f(x_{(P)})$ so that the reflected point is no better than the previous second-worst. The contraction step is performed using x^r if this is better than $x_{(P+1)}$ (*outside contraction*) or $x_{(P+1)}$ if this is better than x^r (*inside contraction*):

(outside contraction) If $f(x_{(P)}) \leq f(x^r) < f(x_{(P+1)})$, compute

$$x^c = \overline{x} + \beta(x^r - \overline{x}),$$

where e.g. $\beta = \frac{1}{2}$. If x^c is better than x^r, replace $x_{(P+1)}$ with x^c and go to step 1. If x^c is not better than x^r, go to step 6.

(inside contraction) If $f(x^r) \geq f(x_{(P+1)})$, compute

$$x^c = \overline{x} + \beta(x_{(P+1)} - \overline{x}).$$

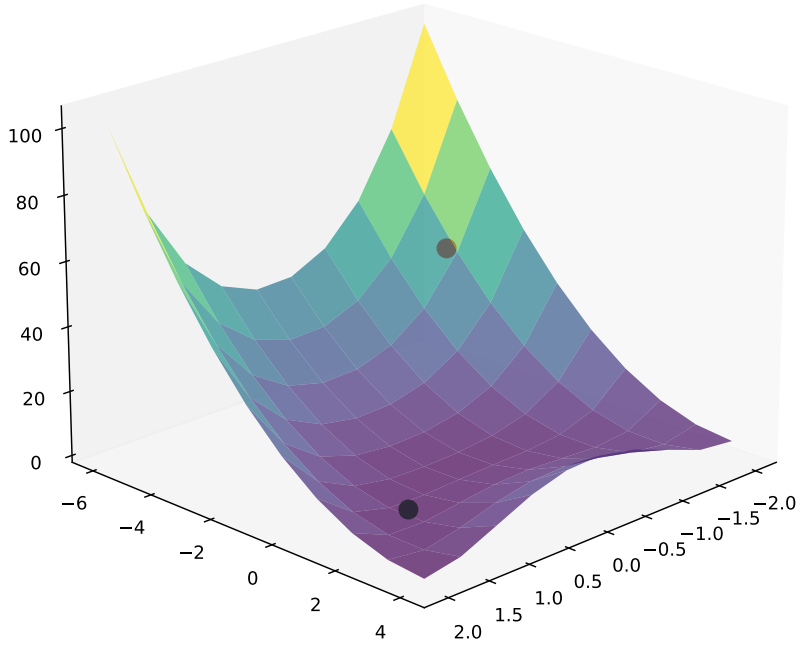
If x^c is better than $x_{(P+1)}$, accept it and go to step 1. Otherwise go to step 6.

6 (shrink). This steps shrinks all the sides of the simplex in the direction of the best point so far by setting

$$x_{(p)} := x_{(1)} + \delta(x_{(p)} - x_{(1)}), \quad p \neq 1.$$

Typically, $\delta = \frac{1}{2}$. After this step, go to step 1.

Figure 4: The Rosenbrock Function



4 Example: The Rosenbrock Function

In this section, we will consider the minimization of the Rosenbrock function and compare how well gradient-based and gradient-free methods do at finding the global minimum.

The function is defined by

$$f(x_1, x_2) = (x_2 - x_1^2)^2 + \frac{1}{2}(1 - x_1)^2.$$

The function is shown in Figure 4 along with the global minimum at $x = (1, 1)$ and the starting value we will be using at $x_0 = (-1.5, -4)$. In Python, the function along with it's gradient vector and Hessian matrices are computed as follows

```
def rosen(x):
    return .5*(1 - x[0])**2 + (x[1] - x[0]**2)**2

def rosen_grad(x):
    ''' returns the vector of 2 partial derivatives '''
    return np.array((-2*.5*(1 - x[0]) - 4*x[0]*(x[1] - x[0]**2), 2*(x[1] - x[0]**2))
```

```
def rosen_hess(x):
    ''' returns the 2*2 matrix of second derivatives '''
    return np.array(((1 - 4*x[1] + 12*x[0]**2, -4*x[0]), (-4*x[0], 2)))
```

4.1 Newton without Globalization

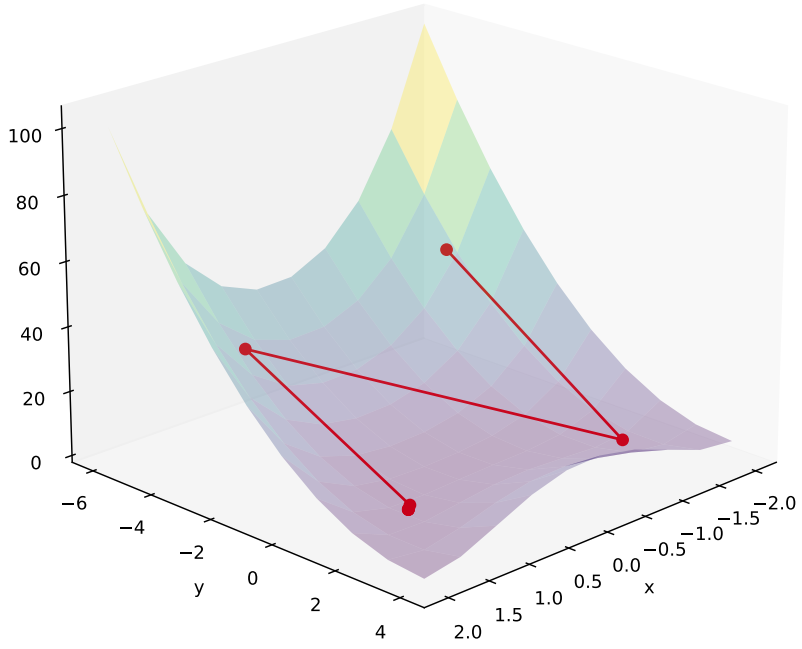
Let us consider the simplest possible implementation of a Newton-based minimizer without a globalization strategy.

```
x = np.array([-1.5, -4.0])

# print initial info
norm = np.max(np.abs(rosen_grad(x)))
print(f'it=0, {x}, {norm}, {rosen(x)}')

for it in range(1,maxit):

        # compute next step
    g = rosen_grad(x)
    h = rosen_hess(x)
    step = np.linalg.solve(h,g)
    x = x - step

    # check convergence
    norm = np.max(np.abs(g))
    print(f'{it=}, x=({x[0]}, {x[1]}),
                norm={norm} f(x)={rosen(x)}')
    if norm < tol:
        print(f'Convergence achieved!')
        break
```

This results in the following output, which is visualized in Figure The progress of the optimizer is shown in Figure 5.

```
it=0, x=( -1.5000,   -4.0000), norm=         40 f(x)=      42.19
it=1, x=( -1.4038,    1.9615), norm=      2.456 f(x)=      2.889
it=2, x=(  0.9143,   -4.5378), norm=      19.57 f(x)=      28.88
it=3, x=(  0.9181,    0.8429), norm=    0.08186 f(x)=   0.003355
it=4, x=(  1.0000,    0.9933), norm=    0.02683 f(x)= 4.502e-05
it=5, x=(  1.0000,    1.0000), norm= 1.243e-07 f(x)= 7.736e-15
it=6, x=(  1.0000,    1.0000), norm= 6.217e-14 f(x)= 2.416e-28
Convergence achieved!
```

Figure 5: Vanilla Newton method without globalization strategy



We note that at `it = 2`, we observe a *higher* value of the function. This step is taken anyway because there is no globalization strategy implemented to reject such a (seemingly) bad step. However, it turns out that from that point, the optimizer quickly reaches the domain of attraction, and from `it`=4, the norm of the gradient starts to decrease very rapidly, on the orders of $10^{-2}, 10^{-7}, 10^{-14}$. Exponents doubling is consistent with quadratic convergence.

4.2 BFGS

We will first minimize the function using a gradient-based solver. To do this, we must first set the optimization options. We will specify a quite strict stopping criteria, namely a tolerance of 10^{-8}, requiring the norm of the gradient vector to be at least this small. It may not always be possible to get the gradients to be so small, in particular if there is numerical noise in the function evaluation and numerical gradients are being used.

```
x0 = np.array([-1,-4])
res = minimize(rosen, x0, method='BFGS', tol=1e-8)
```

This tells Python to start the optimization at the point $x_0 = (-1.5, -4)$ using the specified options. The output from running this is:

```
      fun: 2.005550405057142e-11
 hess_inv: array( ... )
      jac: array( ... )
  message: 'Optimization terminated successfully.'
     nfev: 156
      nit: 41
     njev: 52
   status: 0
  success: True
        x: array([0.99999552, 0.99999104])
```

The first thing we note is that `success` is set to `True`, indicating that `minimize` terminated due to the gradients being smaller than the tolerance; this indicates that we have indeed found a stationary point. It took 41 iterations and 156 function evaluations to get to the stationary point. Python also indicates this to us by saying "`Optimization terminated successfully`". The true global minimum of the Rosenbrock function is $(1, 1)$, so we are very close indeed.

The output furthermore informs us that the function was called 156 times, the algorithm took 41 iterations to complete, and that the gradient was evaluated 52 times (`njev`, with `j` denoting Jacobian). The library `scipy.optimize` uses the word "Jacobian", which generally is a *matrix* of partial derivatives for a *vector*-valued function. Since we only consider optimization for *scalar*-valued functions, we will stick with the word "gradient." This, `jac` shows us the gradient vector, which we can see to have the values $(5.26 \cdot 10^{-9}, -2.58 \cdot 10^{-9})$: we note that both elements have absolute values below the requested tolerance of 10^{-8}, which is our convergence criterion.

Figure6 shows the steps taken by the BFGS algorithm in minimizing the function. The convergence to the global minimum is quite fast and occurs very smoothly.

4.3 Gradient-free Optimization

If we instead want to minimize the function using Nelder-Mead (`fminsearch`), the required code is

```
res = minimize(rosen, x0, method='Nelder-Mead')
print(res)
```

which results in the following output

```
 final_simplex: (...)
           fun: 1.7748487192891172e-10
       message: 'Optimization terminated successfully.'
          nfev: 93
```

Figure 6: BFGS

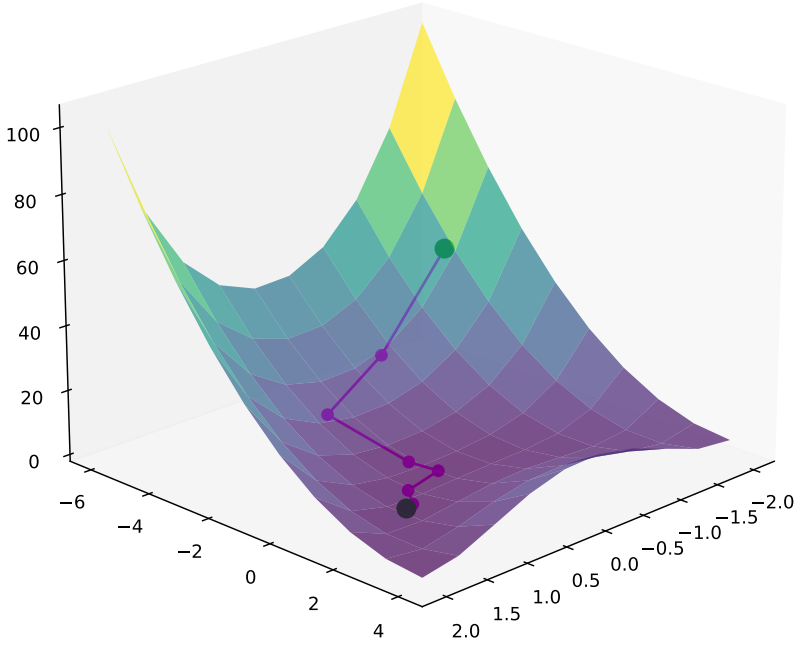
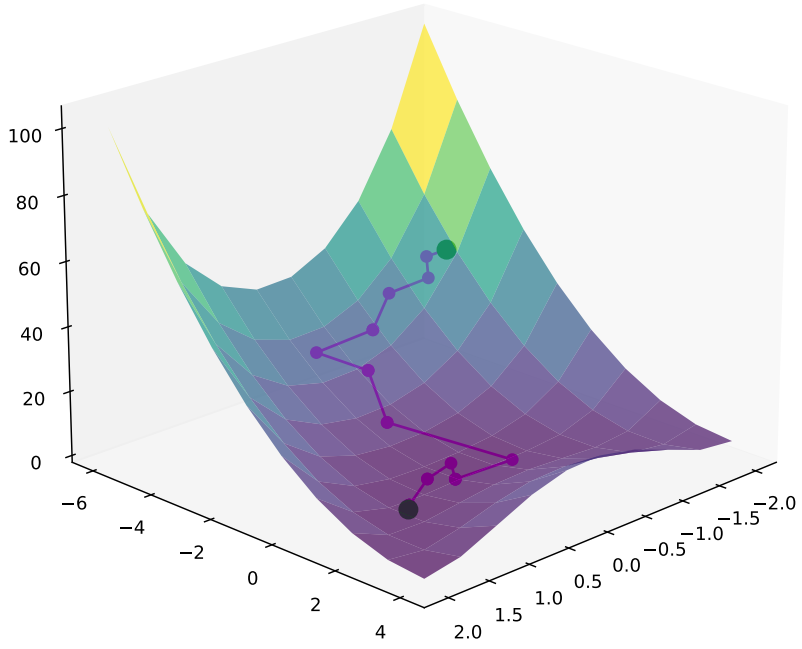


Figure 7: Nelder-Mead




```
           nit: 49
        status: 0
       success: True
             x: array([1.00000183, 1.00001693])
```

We see that the algorithm managed to get satisfactorily close to the optimum in 49 iterations, which required 93 function evaluations. Figure 7 shows the steps taken towards the minimum point. We see that the optimizer is taking many more steps but eventually gets to the true minimum. The path taken by the algorithm looks more like what a ball rolling down a hill would be doing, which makes sense given the intuitive heuristic of the algorithm.

4.4 A More Difficult Example: Illustrating the Effect of Noise

In this section, we will consider an example where we add noise to the function evaluation. This results in the gradient-based optimizer failing miserably when it uses numerical gradients and not working very well even when analytic gradients are used. The gradient-free optimizer is much more robust to this type of noise.

Suppose that we add some noise to the Rosenbrock function in the form of a random

normal variable with a standard deviation of 10^{-4}. That is

$$\tilde{f}(x_1, x_2) = f(x_1, x_2) + 10^{-4}\eta, \quad \eta \sim \mathcal{N}(0, 1).$$

In code, we write this as

```
def rosen_noisy(x):
    sigma = 0.0001
    f = 100 * (x[1] - x[0]**2)**2 + (1.0 - x[0])**2
    u = np.random.normal(0,sigma)
    return f+u
```

If we use the default **minimize** algorithm, BFGS, on this problem, we get the following failed output results:

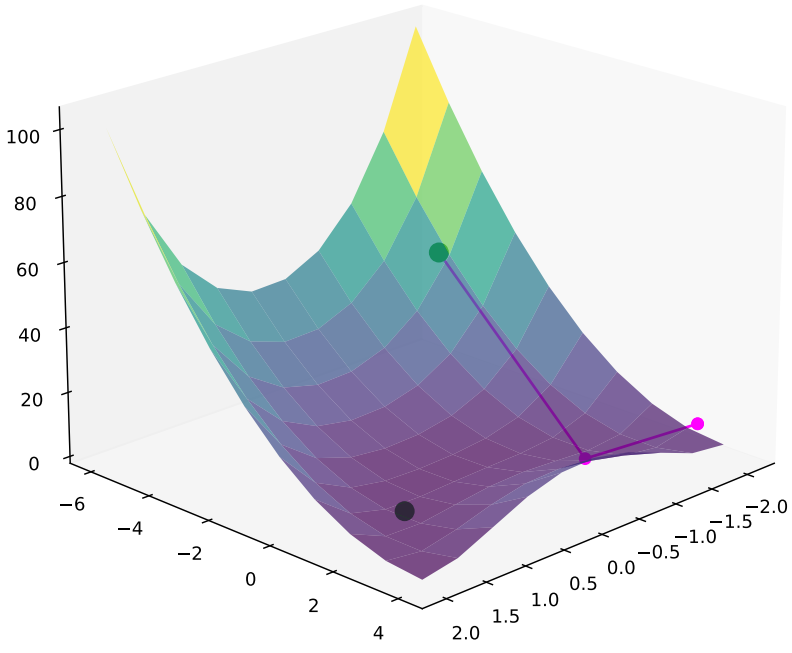
```
Iter.: x0 x1 f(x)
0: -1.5000 -4.0000 42.19
1: -0.8426 2.2941 4.207
2: -2.0895 3.0021 6.633
3: -2.0895 3.0021 6.633
fun: 195.6172635047403
hess_inv: array(...)
jac: array(...)
message: 'Desired error not necessarily achieved due to precision loss.'
nfev: 102
nit: 3
njev: 31
status: 2
success: False
x: array([-2.0895456 , 3.00211868])
```

We see that the optimizer only took 3 iterations but called our criterion function 102 times in its confusion over the strange behavior of the function it was detecting. Poor, tormented optimizer. Instead, Nelder-Mead with default settings will happily solve the optimization problem and find the true parameter values.

What went wrong? The reason is that the numerical gradients will be way off. When the gradient calculator takes a small step of $h = \sqrt{\epsilon} \cong 10^{-8}$, it gets a new random shock on the order of 10^{-4}, which means that it will get a completely wrong indication of the local slope. If instead we provide analytic gradients (for the noiseless Rosenbrock function), it works again. Furthermore, we can actually get BFGS to work by making the step size 10^{-3}, which is done like so:

```
res = minimize(rosen_noisy, x0, method='BFGS', options={'eps':1e-3})
```

Figure 8: Gradient-based Optimizer Failing Due to Stochastic Noise



If on the other hand, we make the function more noisy with $\sigma = 10^{-3}$, Nelder-Mead fails to converge with the default number of iterations, and will in fact never converge, even with 100,000 iterations at default tolerance requirements, although it does get very close to the optimum relatively quickly; it just is never able to satisfy convergence.

5 Troubleshooting

In this section, we will consider a number of symptoms, some likely causes and ideas for solutions.

Problem	**Possible solutions**
[anything]	Try debugging your code. See if you can simplify your problem to something where the solution is trivial/easier; too many bugs are simpler than you might imagine. Shut down parts of the model you do not need. Work on simulated data if at all possible. See Section 5.1.
Noise in the criterion function	**Roundoff errors:** Look for things in your code that might become really small or large. For example, if you could be taking `log` of something too close to zero or `exp` of a large number (e.g., `exp(800) = inf`). Also functions like `normpdf` can cause roundoff error for inputs far from zero. **Simulation/stochastic problems:** If you are using *simulation* methods (if there is any random draws in your evaluation), make sure your seed is fixed and does not change every time you evaluate the criterion function. Try
Unable to find good starting values / function undefined at initial point	If you cannot think of a good set of starting values, it will typically mainly be a few of the parameters. In that case, try doing a *grid search*. I.e. define a range of points (e.g. 10 points from 0 to 1000 below) and then loop through the points and print the function value of your function.

Problem	Possible solutions
Function evaluation is very slow	There are many potential reasons why evaluation might be slow. Some general advise for speeding up code is given in Section 5.2
Max iterations reached without convergence	If possible try to increase `maxiter` to see if this helps. If the function value is unchanged for many many iterations, you could have numerical noise in your criterion function (see previous table).
Step size too small (line search fails)	The gradients or the Hessian could be wrong or you could be in a local area of non-convexity. Try using `Nelder-Mead`.
Optimizer moves into "illegal" values	For example, it can happen that the optimizer tries to insert negative values for a variance parameter (which does not make sense). You can solve this by having your function return `np.inf` if illegal values are inserted. Alternatively, for variance parameters, you can take the absolute value of the parameter.

5.1 Debugging Code

Simplifying: The best advice for debugging is to simplify the code; try to see how simple you can make the setting and still replicate the bug. Shut off as many parts of the model as you possibly can and comment out parts of the code. Then, when you get to a working version, add them back in one at the time.

Narrowing down the cause: Try to be systematic about how you find the cause of your bug; think of a possible problem that could be causing the symptoms you are seeing and then think of a way of testing if that is correct.

Data read incorrectly: Try to work on simulated data and create a very simple data generating process where you can narrow this down. Maybe work only with $N = 3$ and $T = 2$ and use `keyboard` to see what is in the data matrices in the very core of the function where your criterion function is computed.

Parameters enter incorrectly: If you suspect a bug where, for example, the variance term enters incorrectly. Then think about intuitively what the effect should be of increasing the variance term; in most cases, it should mean that the predicted values of the model should get closer to each other (the variance washes out all the differences). Then check if this happens.

Plotting over a grid: Often, it might make sense to form a grid over one or more parameters and calculate the objective value at each of these points and then plot it. This can sometimes tell a lot about what can be wrong.

Randomness: If you think there is randomness in your function (e.g. due to random draws used to compute the function), evaluate the function twice at the same point to see if it returns precisely the same value.

Numerical noise: If you think there is numerical noise in your function, try doing a forward and a backwards finite difference (derivative-approximation). If the function is well-behaved, the two should give approximately the same slope.

5.2 Speeding up Python Code

The best practice for code development is

1. Get the code to work,
2. Make the code fast [if needed].

In step 1, speed does not matter, so develop the code however it is simplest to ensure correctness. Next, you can make the code faster while verifying that you get identical results. A few tips:

Use vectors, don't loop: Loops are easy to understand but slow. So use vectorized operations in `numpy`.

Preallocate: If you are filling out an array iteratively (e.g. in a loop), at least make sure that you have preallocated the output. Otherwise, Python will be iteratively destroying the previous array and allocating a new array with a larger size.

Profiling code: There are various ways of "profiling" your code, which is when you ask Python to find out which specific function calls are taking up too much time. It is crucial to know in advance that you are not optimizing on a part of the code that is not in fact the true bottleneck. Google is your friend.