Lecture Notes 3: Rootfinding and Optimization

Ivan Rudik

ECON 509: Computational Methods

February 12, 2016

1 Why do we need to numerically optimize?

Virtually all economic problems boil down into a maximization, minimization, or rootfinding problem. With general functional forms in a simple setting, we can perform the necessary analytic operations to characterize the optimum or obtain sufficient conditions for certain behavior. However if the system of equations is highly complex, then we may not be able to achieve a closed form solution or even a clean set of equations to gain intuition about the equilibrium. Indeed, many systems (such as the climate system or electricity markets) display complex non-linear or non-convex behaviors that require many equations to capture, limiting the use of simple analytic approaches. In many of these systems, there exists a trade-off between developing a more complex and accurate model, and being able to get general closed-form results. In cases where accuracy is more important than gaining intuition from closed-form solutions, we need to employ numerical optimization methods to get solutions for the equilibrium outcome or trajectory.

2 Non-linear Equations

Solutions to linear systems can be obtained using linear algebra techniques. However most economic systems are non-linear and require different methods. Many economic problems can be solved by rootfinding. To maximize or minimize a function we must first find where the gradient of that function is zero. In general we are trying to compute solutions to,

$$f(x) = 0,$$

where $f : \mathbb{R}^n \rightarrow \mathbb{R}^n$. Alternatively, in economics we often want to find fixed points,

$$g(x) = x,$$

but this can be re-framed as a root-finding problem by shifting x to the left hand side, $f(x) = g(x) - x$. Lastly, economics problems virtually always involve constraints, and these constraints do not necessarily have to be binding. For example we may wish to solve the first-order conditions of a profit-maximization problem. But the firm's control vector, x, may be constrained to be $x \in [a, b]$, resulting in a *complementarity problem*,[1]

$$x_i > a_i \Rightarrow f_i(x) \geq 0 \;\; \forall i = 1, ..., n$$
$$x_i < b_i \Rightarrow f_i(x) \leq 0 \;\; \forall i = 1, ..., n$$

What do the equations of the complementarity problem mean? If the constraints on control i do not bind, $a_i < x_i < b_i$, then the first-order condition for control i must equal precisely zero. But suppose the upper bound constraint binds in equilibrium: $x_i = b_i$. Since $x_i > a_i$ we have that $f_i(x) \geq 0$. Since the constraint binds, the first order conditions will not necessarily be zero. The objective function may still be increasing in x_i at $x_i = b_i$, but because the constraint limits us from raising x_i any further, we cannot actually ever reach $x_i = 0$.

There exists a vast toolkit for numerically solving non-linear problems. We will study the basic methods and other algorithms that are widely used in standard optimization packages.

3 Basic Non-Linear Rootfinders

3.1 The Bisection Method

The intermediate value theorem tells us that if a continuous, real-valued function on a given interval takes two distinct values, a and b, then it must also must achieve all values in between those two distinct values somewhere in its domain. The intermediate value theorem is the motivation behind one of the most simple optimization algorithm: the bisection method. If we have a continuous one-dimensional function, and we know at one value the function is positive, and at another distinct value the function is negative, then a root of the function must fall somewhere in between these two points. The key question is: starting from these two initial points, how do we find where the root is efficiently? Suppose we have some function, $f(x)$, the following algorithm outlines how we find the root with the bisection method:

[1] Recall the complementarity slackness conditions from Math Camp.

```
% Initialize bounds
lb = a;
ub = b;
% Initialize tolerance
tol = 1e-5;

% Error check
if sign(f(a)) == sign(f(b))
        disp("Error, starting values have the same sign.");
else
        s = sign(f(a));
end

% Bisect the interval at x
x = (a+b)/2;
d = (b-a)/2;

% Iterate until convergence
while d > tol
        d = d/2;
        % If the sign of the new bisection has the same sign as
        % the lower bound, we are below the root, move up. Else,
        % we are above the root and must move down.
        if s == sign(f(x))
                x = x + d;
        else
                x = x - d;
        end
end
```

The bisection method works by continually bisecting the interval until convergence, i.e. until we are bisecting an interval of sufficiently small length. We first select a point exactly at the midpoint of the interval $[a, b]$. The root must lie in either the lower subinterval or upper subinterval. If the midpoint has the same sign as the lower bound, then a root must lie to the right of this midpoint. Knowing this, we have a new subinterval: $[a + \frac{b-a}{4}, b]$. We bisect the new interval and repeat until convergence.

The bisection method is incredibly robust. Given a function meeting the conditions to satisfy the intermediate value theorem, the bisection method is *guaranteed* to converge, and to do so in a specific number of iterations. A root can be computed to some arbitrary precision ϵ in a maximum of $log((b-a)/\epsilon)/log(2)$ iterations. This robustness comes with its drawbacks. The bisection method only works in one dimension and many interesting problems are multidimensional. Moreover, the bisection method is relatively slow because it does not take advantage of information like the curvature of the function.

3.2 Function Iteration

Fixed points of n-dimensional functions can be computed using function iteration.[2] Since we can always recast a fixed point problem, $f(x) = x$ as a rootfinding problem, $f(x) - x = 0$, it can be used for these as well. Function iteration is a simple procedure,

```
% Initial guess
diff = inf;
tol = 1e-5;
x_old = 0;

while diff > tol
        x = f(x_old);
        diff = x - x_old;
        x_old = x;
end
```

This algorithm is illustrated in Figure 1. We begin with an initial guess, x_0 of the fixed point. We plug that guess into the function, $f(x_0)$. Then this becomes our new guess of the fixed point, $x_1 = f(x_0)$. This process is repeated until convergence. This is the function version of the functional problem in solving dynamic programming models.

3.3 Newton's Method

Newton's method and its variants are the workhorse of solving n-dimensional non-linear problems. Newton's method works by taking a difficult non-linear problem, and replacing it with a sequence of easier linear problems. Given certain conditions, the sequence of solutions will converge to the

[2]We will learn how to use this to solve functionals in dynamic programming settings.

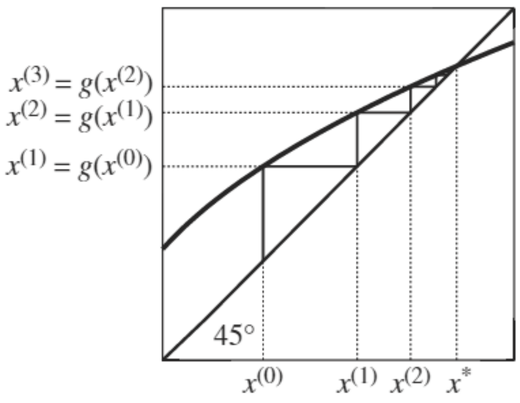

Figure 1: Function iteration.

true solution. Newton's method is displayed in Figure 2. We begin with an initial guess of the root, x_0, of $f(x)$. We then approximate our non-linear problem with its first-order Taylor expansion about x_0. In one dimension this is just the tangent line at x_0, in many dimensions its the tangent surface. We then find the root of the linear approximation of $f(x)$. Call this x_1. We use this as our new and improved guess of the root and repeat the process until convergence, i.e. $x_n - x_{n-1}$ is small.

Newton's method can be applied to a problem of arbitrary dimensions. If we begin with some initial guess of the root vector $\mathbf{x_0}$, our new guess $\mathbf{x_{k+1}}$ given some arbitrary point in the algorithm, $\mathbf{x_k}$ is obtained by approximating $f(\mathbf{x}$ using a first-order Taylor expansion about $\mathbf{x_k}$ and solving for $\mathbf{x}$:

$$f(\mathbf{x}) \approx f(\mathbf{x_k}) + f'(\mathbf{x_k})(\mathbf{x} - \mathbf{x_k}) = 0$$
$$\Rightarrow \mathbf{x_{k+1}} = \mathbf{x_k} - \left[f'(\mathbf{x_k})\right]^{-1} f(\mathbf{x_k})$$

Algorithmically, this process can be written as,

```
% Initial guess
diff = inf;
tol = 1e-5;
x_old = 0;

while diff > tol
        x = f(x_old) - inv(f_prime(x_old))*f(x_old);
```

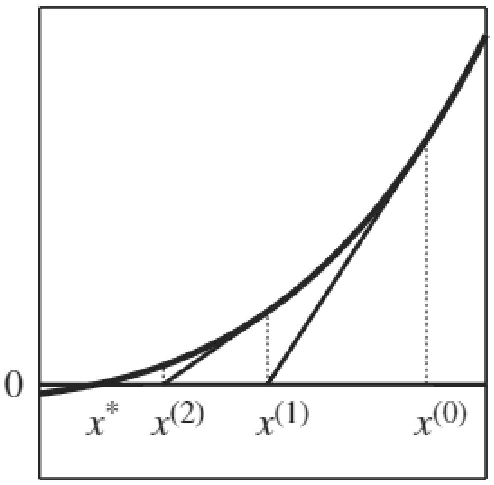

Figure 2: Newton's method.

```
        diff = x - x_old;
        x_old = x;
end
```

Newton's method has relatively nice properties which guarantee convergence. We require f to be continuously differentiable, the initial guess to be "sufficiently close" to a root of f, and f' must be invertible near the root. Unfortunately there is no way to determine what sufficiently close is, but in practice, most reasonable initial guesses for the root of f will lead to convergence. We need f' to be invertible near the root so that the above algorithm is well-defined. In addition to this analytic necessary condition, we also require that f' to be well-conditioned near the root as well. If f' is ill-conditioned, rounding errors will halt convergence of the algorithm.

Problem: Code up a solution to the following problem using Newton's method. Consider a Cournot duopoly where inverse demand is given by $P(q) = q^{-1/1.6}$, and the firm cost functions are $C_1(q_1) = 0.3\, q_1^2, C_2(q_2) = 0.4\, q_2^2$.

Solution tips: What determines the equilibrium? The first-order conditions. First code a function that returns the value and Jacobian of the first-order conditions given an arbitrary quantity vector. Then we can plug this function and an initial guess into the `newton` function in the CompEcon toolbox. The solutions will be $q_1 = 0.8396$ and $q_2 = 0.6888$.

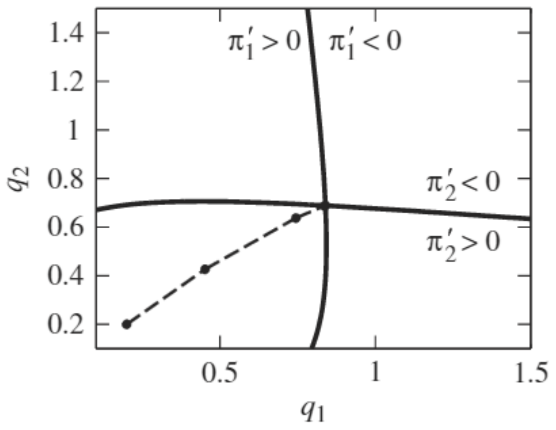

Figure 3: Newton's method example.

Figure 3 graphically displays how Newton's method solves this problem. The two lines are the firms reaction functions. Recall that an equilibrium of a game is when neither player wishes to deviate, this is where the reaction functions intersect.

3.4 Quasi-Newton Methods

3.4.1 Secant Method

Analytic derivatives are troublesome. Coding up an analytic derivative leads to a high probability of coding error. We wish to avoid this if possible, particularly if we know computation time will not be an issue.[3] Instead of coding an analytic derivative, we can use finite difference methods to approximate the derivative for a one dimensional problem. This is called the Secant method. Using our current root guess x_k and our previous root guess x_{k-1} yields our approximation of the derivative:

$$f'(x_k) \approx \frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}}.$$

Our new iteration rule then becomes,

$$x_{k+1} = x_k - \frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})} f(x_k)$$

[3]Moreover, analytic derivatives to many problems are slow since they require a large number expensive operations (multiplication, exponentiation). Ken Judd's presentation slides demonstrate this well and show a way to avoid these operations by doing the differentiation in a clever way: http://ice.uchicago.edu/2008_presentations/Judd/Curse_in_Dallas.pdf

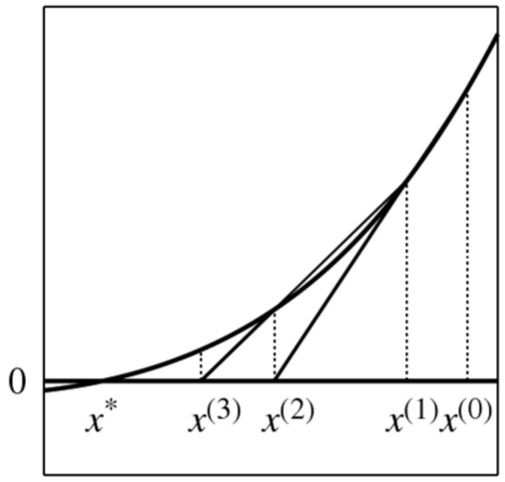

Figure 4: Secant method example.

where now we require two initial guesses so that we have an initial approximation of the derivative.

3.4.2 Broyden's Method

Broyden's method is the most widely used rootfinding method for n-dimensional problems. It is a generalization of the secant method where we have our usual sequence of root guess vectors, but we also now have a sequence of guesses of the Jacobian at the root. Similar to the secant method, we must initially provide a guess of the root, x_0, but also a guess of the Jacobian, A_0.[4] Broyden's method has the same updating procedure as before for our guess of the root, but now with our guess of the Jacobian replacing the analytic Jacobian,

$$\mathbf{x_{k+1}} = \mathbf{x_k} - A_k^{-1} f(\mathbf{x_k}).$$

However, we still need to update A_k. This update is performed by making the smallest change, in terms of the Frobenius matrix norm[5], that satisfies what is called the *secant condition*,

$$f(\mathbf{x_{k+1}}) - f(\mathbf{x_k}) = A_{k+1} \left(\mathbf{x_{k+1}} - \mathbf{x_k}\right).$$

In other words, the updated differences in root guesses, and the function value at those root guesses, should align with our estimate of the Jacobian at that point. This gives us a rule to update the

[4]This first guess is usually the numerical approximation to the Jacobian at x_0.
[5]The square root of the sum of the squared elements. Very similar to the Euclidean vector norm.

Jacobian,

$$A_{k+1} = A_k + [f(\mathbf{x_{k+1}}) - f(\mathbf{x_k}) - A_{k+1}(\mathbf{x_{k+1}} - \mathbf{x_k})] \frac{\mathbf{x_{k+1}} - \mathbf{x_k}}{(\mathbf{x_{k+1}} - \mathbf{x_k})^T(\mathbf{x_{k+1}} - \mathbf{x_k})}$$

How can we accelerate this method? One way is to not update our estimate of the Jacobian, but to update our estimate of the inverse of the Jacobian. this allows us to avoid a matrix inversion in the update of the root estimate. Our Jacobian inverse update rule follows,

$$B_{k+1} = B_k + \frac{[d_k - u_k]d_k^T B_k}{d_k^T u_k}$$

where $d_k = (\mathbf{x_{k+1}} - \mathbf{x_k})$, and $u_k = B_k\,[f(\mathbf{x_{k+1}}) - f(\mathbf{x_k})]$.

Broyden's method converges under a relatively weak set of conditions. f must be continuously differentiable, x_0 must be initially close to the root of f where f' must also be invertible, and A_0 or B_0 must be sufficiently close to the Jacobian. Again, there is no real way to check what sufficiently close means. Also, Broyden's method may fail if the Jacobian is ill-conditioned near the root so A_k is inaccurate, making x_{k+1} not necessarily converge well. Note that A_k and B_k will not necessarily converge to the Jacobian or inverse Jacobian.

3.5 Convergence Speed

Different root finding algorithms will converge at different speeds. A sequence of iterates $x^{(k)}$ is said to converge to x^* at a rate of order p if there is a constant C such that

$$||x^{(k+1)} - x^*|| \leq C||x^{(k)} - x^*||^p,$$

for sufficiently large k.[6] If $C < 1$ and $p = 1$, the rate of convergence is linear. If $1 < p < 2$, convergence is superlinear, and if $p = 2$ convergence is quadratic. The higher order the convergence rate, the faster it converges.

Convergence rates of the methods we use are known. The bisection method converges at a linear rate with $C = 0.5$. Function iteration converges at a linear rate with $C = ||f'(x^*)||$. Secant and Broyden methods converge at a superlinear rate with $p \approx 1.62$, and Newton's method converges quadratically.

Keep in mind that these convergence rates only account for the number of *iterations* of the method. The steps taken in a given iteration of each solution method may also vary in computational cost because of differences in the number of arithmetic operations. Although an algorithm

[6]This is a very similar definition to that of Big-O notation, but for sequences instead of functions.

may take more iterations to solve, each iteration may be solved faster and the overall algorithm takes less time. For example, function iteration only requires a function evaluation during each iteration. Broyden's method requires both a function evaluation and matrix multiplication, while Newton's method requires a function evaluation, a derivative evaluation, and solving a linear system. Function iteration is usually slow. Broyden's method can be faster than Newton's method if derivatives are costly to compute.

4 Complementarity Problems

Most interesting economic settings are complementarity problems. For example, returning to the notation in the second section of the lecture note, we can show that economic equilibria are subject to complementarity. Let x is an n-dimensional vector of some economic action, a_i denotes a lower bound on action i, and b_i denotes the upper bound on action i, and $f_i(x)$ denotes the marginal arbitrage profit of action i. There are disequilibrium profit opportunities if either $x_i < b_i$ and $f_i(x) > 0$ (here we can increase profits by raising x_i) or if $x_i > a_i$ and $f_i(x) < 0$ (we can increase profits by decreasing x_i). We obtain a no-arbitrage equilibrium if and only if x solves the complementary problem $CP(f, a, b)$. We can write out the problem as finding a vector $x \in [a, b]$ that solves,

$$x_i > a_i \Rightarrow f_i(x) \geq 0 \;\; \forall i = 1, ..., n$$
$$x_i < b_i \Rightarrow f_i(x) \leq 0 \;\; \forall i = 1, ..., n$$

If a component of x is strictly above the lower bound, then the function must be weakly increasing in that direction. If a component of x is strictly below the upper bound then the function must be weakly decreasing in that direction. Therefore if there is an interior solution, the function be precisely be zero. If there is a corner solution at the upper bound b_i for x_i, the function f must be increasing in direction i. The opposite is true if we were at the lower bound.

Economic optimization problems are also typically complementarity problems where we are maximizing a function (e.g. profit) subject to some constraint (e.g. price floors). The Karush-Kuhn-Tucker theorem shows that x solves the constrained optimization problem if and only if it solves the complementarity problem. Consider a simple example Marshallian competitive equilibrium. We have a competitive equilibrium if and only excess demand, $E(p)$, is zero. If the government imposes a price floor, then we can actually have excess demand be non-zero in equilibrium, but only if the price floor is binding.

Consider another example: the single commodity competitive spatial price equilibrium model.

Suppose there are n regions of the world, and excess demand for the commodity in region i is $E_i(p_i)$. If there's no trade, then our equilibrium condition is that $E_i(p_i) = 0$ in all regions of the world: a simple rootfinding problem. Now suppose that there is trade between regions with marginal transportation cost between regions i and j of c_{ij}. Let x_{ij} be the amount of the good traded from region i to region j and that there is a capacity constraint on the shipping vessel of b_{ij}.

The marginal arbitrage profit from shipping a unit of the good from i to j is $p_j - p_i - c_{ij}$. If this is positive, there's an incentive to ship more goods to region j from region i. If it's negative, then there's an incentive to decrease shipments. We have an equilibrium only if all the arbitrage opportunities are gone. This condition is, for all region pairs i and j:

$$\begin{gathered}0 \leq x_{ij} \leq b_{ij}\\ x_{ij} > 0 \Rightarrow p_j - p_i - c_{ij} \geq 0\\ x_{ij} < b_{ij} \Rightarrow p_j - p_i - c_{ij} \leq 0\end{gathered}$$

How do we formulate this as a complementarity problem? Market clearing in each region requires:

$$\sum_k [x_{ki} - x_{ik}] = E_i(p_i).$$

This implies that we can solve for the price in region i,

$$p_i = E_i^{-1}\left(\sum_k [x_{ki} - x_{ik}]\right),$$

if,

$$f_{ij}(x) = E_j^{-1}\left(\sum_k [x_{kj} - x_{jk}]\right) - E_i^{-1}\left(\sum_k [x_{ki} - x_{ik}]\right) - c_{ij},$$

then x is an equilibrium vector of trade flows if and only if x solves CP(f,0,b) and x, f, and b are n^2x1 vectors. Even this highly complex trade-equilibrium model can be reduced to a simple complementarity problem.

The structure of a complementary problem is made clear with plots of the uni-dimensional case. Figure 5 displays four plots of f_i in different circumstances. In panel a, f_i is negative so f is decreasing in direction i, so the unique solution is at $x_i = a_i$. In panel b, f_i is positive so the function is still increasing and our solution is $x_i = b_i$. In panel c, $f_i = 0$ at some point $x^* \in [a, b]$ so we have an interior solution $x_i = x^*$. Panel d displays a troublesome case, when f_i is positively sloped. f_i does intersect 0 at x^*, but if we began in disequilibrium at $< x^*\bar{x} < b_i >$, we have that $f_i(\bar{x}) > 0$, so we would move towards $x_i = b_i$ as our equilibrium! The same would happen if we

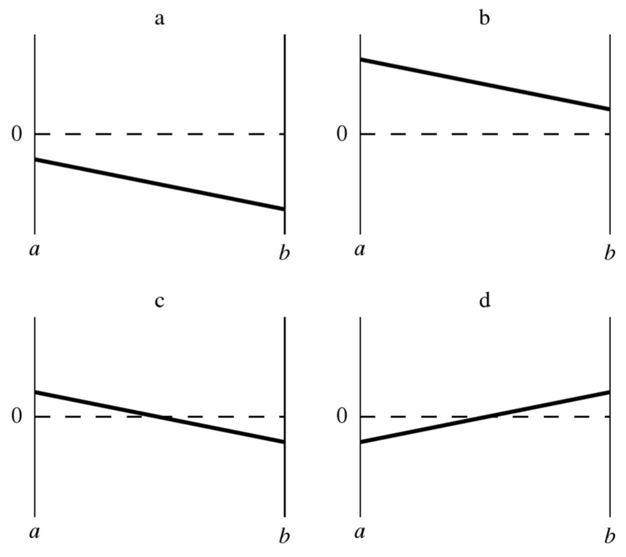

Figure 5: Univariate complementary problem.

began below x_i. This is a case of multiple equilibria.

4.1 Numerical Solution Methods

A complementarity problem can be re-framed as a rootfinding problem easily by writing,

$$\hat{f}(x) = min(max(f(x), a - x), b - x) = 0.$$

We can then solve the problem with conventional rootfinding techniques, and others more geared towards problems with kinks that arise in complementarity problems. See Miranda and Fackler (2002).

5 Optimization Methods

Maximization problems are central to economics.[7] How we solve maximization problems has many similarities to rootfinding and complementarity problems. Indeed, the first-order conditions of an unconstrained maximization problem are just a rootfinding problem. Moreover, the first-order conditions of a constrained optimization problem is just a complementarity problem (complementary

[7]For those of you working in empirical fields, even simple OLS is an optimization problem. We are trying to minimize the sum of squared errors. This can be solved with some simple linear algebra, but more complex estimation strategies like maximum likelihood may require algorithms we will discuss in class.

slackness conditions). We will analyze a set of common optimization routines which can be used to solve very simple problems, or problems that do not display the nice features we typically assume (strictly increasing/decreasing, strictly convex/concave) to guarantee a unique global maximum or minimum.

5.1 Derivative-Free Methods

Similar to rootfinding, we have a set of optimization tools that search for a solution of a one-dimensional problem over smaller and smaller brackets.

5.1.1 Golden Search

One of the most common derivative-free methods is called the *golden search method.* If we have a continuous one dimensional function, $f(x)$, and we want to find a local maximum in some interval $[a, b]$ we can use a routine similar to the bisection method,

1. Select points $x_1, x_2 \in (a, b)$ where $x_2 > x_1$
2. If $f(x_1) > f(x_2)$ replace $[a, b]$ with $[a, x_2]$, else replace $[a, b]$ with $[x_1, b]$
3. Repeat until convergence criterion is met

By replacing the endpoint of the interval next to the evaluated point with the lowest value, we are ensuring that the higher valued point remains in the interval. This guarantees that a local maximum of $[a, b]$ still remains (and may actually be either a or b). We can repeat this process until the interval is at some predefined length where we say the algorithm has converged and we have gotten close enough to the answer.

A critical decision in this algorithm is the selection of the evaluation points x_1 and x_2. The most common strategy ensures that the new interval is independent of whether the upper or lower bound is replaced, and only requires one function evaluation per iteration. This saves us evaluating an if statement, and a function evaluation in the algorithm. There is precisely one algorithm that satifies this:

$$\begin{gathered} x_i = a + \alpha_i(b - a) \\ \alpha_1 = \frac{3 - \sqrt{5}}{2} \qquad \alpha_2 = \frac{\sqrt{5} - 1}{2} \end{gathered} \tag{1}$$

The value of α_2 is called the golden ratio and is where the algorithm gets its name. If on our first iteration, $x_1 > x_2$, we replace b with x_2 and calculate a new x_2 with the above rule.

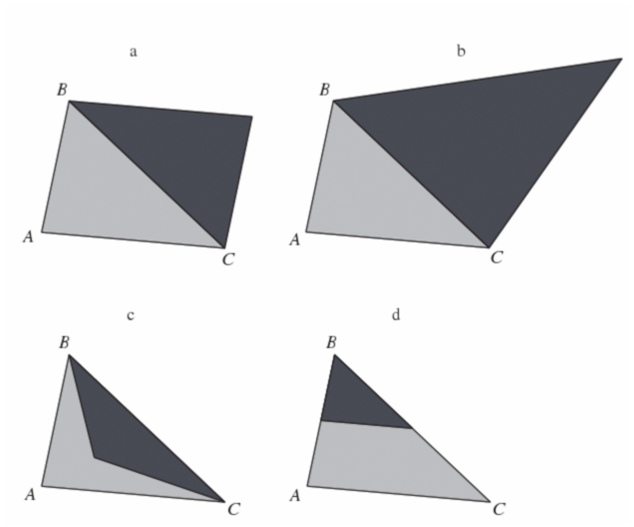

Figure 6: The Nelder-Mead algorithm.

5.1.2 Nelder-Mead: Simplex

The golden search method only works for a one dimensional function. There do exist derivative-free optimization routines for problems of arbitrary dimensions. The benchmark algorithm is *Nelder-Mead* (NM), or the simplex method. NM works by evaluating the function at $n+1$ points in a n dimensional problem. This forms a simplex, for example, in a 2 dimensional problem this forms a triangle. Similar to golden search, we will manipulate the point with the lowest value. This process is shown in Figure 6.

First, we *reflect* the point (a) with the lower value through the opposite face of the subplex and evaluate the new point: if we move away from the lowest value point through our two highest value points do we improve?[8] If we find a point that is the new highest value point, we can *stretch* the simplex further in that direction (b). However, if the new point does not improve much upon the old point then we are likely in a "valley" of the function so we can *contract* the simplex starting from the initial points (c). If this new point is still the worst point, we *shrink* the entire simplex towards the best point (d).

NM is a simplex algorithm, but it is both slow and unreliable. It may be worth trying if you know derivative evaluations will be costly for other algorithms, which make NM appear more attractive. It works poorly when n is large, but tends to have advantages if the objective function is noisy or discontinuous.

[8]Reflections maintain non-degeneracy of the simplex which has nice properties for convergence.

Subplex The subplex method was introduced to improve upon NM. Subplex decomposes the problem into lower-dimensional subspaces of the original problem that the NM algorithm can search more efficiently. See Rowan (1990) for more details and Hollingsworth (2015) for an application in a structural IO model.

5.2 Two Large Classes of Algorithms for Unconstrained Optimization

All algorithms require a starting point x_0. The algorithm proceeds to the next point in some systematic approach. Eventually, the algorithm works through a series of iterates, $\{x_k\}_{k=0}^{\infty}$ until it has "converged" with sufficient accuracy. All algorithms take these steps, but may do it in different ways and using different information. Most unconstrained optimization algorithms fall into one of two classes: line search and trust region.[9] We will discuss how these classes work and touch on the commonly used algorithms within each class. But before we go any further we will revisit some basic optimization math which should help us gain intuition for how these algorithms work.

5.2.1 What is a solution?

In economics we typically wish to find a global maximum to our objective f. In most other disciplines we are minimizing. Most texts follow this minimization assumption and we will go along with this terminology since it matches up to the names of the methods more closely. Some point x^* is a global minimizer of f if $f(x^*) \leq f(x)$ for all x, where x ranges over the entire domain of the function. However, most algorithms are *local* minimizers. We are finding points that satisfy the following statement: A point x^* is a local minimizer if there exists a neighborhood N of x^* such that $f(x^*) \leq f(x)$ for all $x \in N$. Typically, analytical problems in economics are formulated so there exists only one unique maximum, so that any local maximum found by an algorithm is also the global maximum. However many problems do not satisfy these convenient assumptions (e.g. strictly increasing and strictly concave objectives).[10]

How do we actually find a local minimum? Must we evaluate every single possible point? Clearly the answer is no since the Nelder-Mead and Golden Search algorithms do not need to search every single point. If our function is smooth, we can take advantage of higher order derivatives of the function to inform us about the functions shape and reduce the work we must do. If f is twice continuously differentiable, we can distinguish whether x^* is a local minimizer by analyzing the gradient ∇f and the Hessian $\nabla^2 f$. We study and numerically solve these smooth problems using Taylors theorem which states that if f is 2 (n times more generally) times differentiable, then there

[9]There are also conjugate gradient and quasi-Newton methods that fall within these broader classes.

[10]Issues like this may come up in IO where there exist multiple equilibria. If we solve our problem and find one of the equilibria, it may not be the dominant equilibrium in a pareto sense.

exists a $t \in (0,1)$ such that,

$$f(x^* + p) = f(x^*) + \nabla f(x^*)^T p + \frac{1}{2!} p^T \nabla^2 f(x^* + tp)\, p.$$

Note this is an exact equality, not an approximation. Using this definition we can prove necessary and sufficient conditions for a maximum. These can be found in Nocedal and Wright (2006), but we will not being going over the proofs here. Our first-order necessary condition is that $\nabla f(x^*) = 0$, the function cannot be increasing or decreasing at our local maximum. A second-order necessary condition is that $\nabla^2 f(x^*)$ is positive semi-definite. E.g. in one dimension the second derivative is weakly positive. A sufficient condition for a strict local minimum is that our first-order condition is satisfied and that our Hessian is positive definite.

5.2.2 Line Search Methods

In line search algorithms, we start from some current iterate x_k, and begin by selecting some direction to move in, p_k to find our next iterate with a higher function value. The question is, how far along the direction p_k do we move? We answer this question by "approximately solving" the following minimization problem to find what is called our *step length* α,

$$\min_{\alpha > 0} f(x_k + \alpha p_k)$$

. We are finding the distance to move, α in direction p_k that minimizes our objective f. We typically do not perform the full minimization problem since it is costly. Instead, we only trial a limited number of step lengths α before picking the best one and moving onto our next iterate x_{k+1}. But we still haven't answered, what direction p_k do we decide to move in? An obvious choice is the direction that yields the *steepest descent*, $-\nabla f_k$, the direction that makes f decrease most rapidly (k indicates we are evaluating f at iteration k). Figure 7 displays a figure of the steepest descent direction for some function and also plots the functions contour lines.

We can verify this is the direction of steepest descent by referring to Taylor's theorem. For any direction p and step length α, we have that,

$$f(x_k + \alpha p) = f(x_k) + \alpha\, p^T \nabla f_k + \frac{1}{2!} \alpha^2 p^T \nabla^2 f(x_k + tp)\, p.$$

What is the rate of change in f along p at x_k? It is simply the coefficient on α in the second term: $p^T \nabla f_k$. Therefore, the unit vector of quickest descent solves,

$$\min_p p^T \nabla f_k,$$

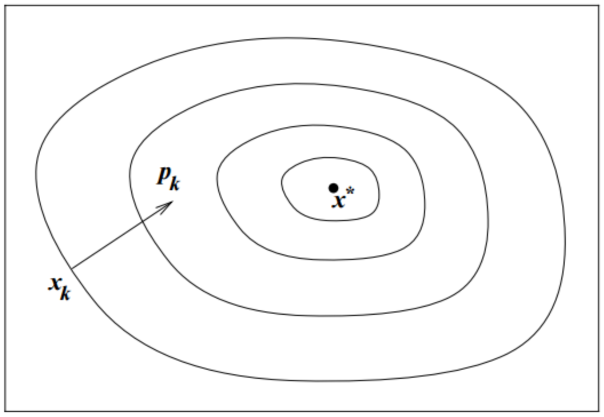

Figure 7: The steepest descent direction.

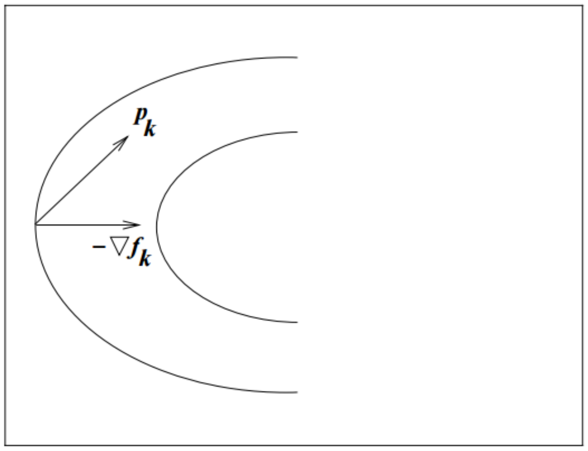

Figure 8: A downhill direction p_k.

subject to $||p|| = 1$. We can re-express the objective as $||p||||\nabla f_k||cos\,\theta$, where θ is the angle between p and ∇f_k. The minimum is clearly attained when $cos\,\theta = -1$ and $p = -\frac{\nabla f_k}{||\nabla f_k||}$, so the direction of steepest descent is simply $-\nabla f_k$. The *steepest descent method* searches along this direction at every iteration k, but may select the step length α_k in several different ways. A benefit of the algorithm is that we only require the gradient of the function, and no Hessian. However it can be very slow.

We can always use search directions other than the steepest descent. In fact, any descent direction, i.e. one that is within 45° of $-\nabla f_k$, is *guaranteed* to produce a decrease in f as long as the step size is sufficiently small. E.g. Figure 8. We can actually verify this with Taylor's theorem.

Recall,

$$f(x_k + \epsilon p_k) = f(x_k) + \epsilon\, p_k^T \,\nabla f_k + O(\epsilon^2).$$

If p_k is in a descending direction, θ_k will be of an angle such that $cos\,\theta_k < 0$. This yields,

$$p_k^T\, f_k = ||p_k||\,||\nabla f_k|| cos\,\theta_k < 0.$$

Therefore $f(x_k + \epsilon p_k) < f(x_k)$ for positive but sufficiently small ϵ.

The most important search direction is not steepest descent but *Newton's direction.* Newton's direction comes out of the second order Taylor series approximation to $f(x_k + p)$,

$$f(x_k + p) \approx f_k + p^T\,\nabla f_k + \frac{1}{2!}\,p^T\,\nabla^2 f_k\, p.$$

Let us define this as $m_k(p)$. We find the Newton direction by selecting the vector p that minimizes $f(x_k + p)$. This ends up being,

$$p_k^N = -\frac{\nabla f_k}{\nabla^2 f_k}.$$

This approximation to the function we are trying to solve has error of $O(||p||^3)$, so if p is small, the quadratic approximation is very accurate. One drawback to the Newton direction is that it requires explicit computation of the Hessian, $\nabla^2 f(x)$. Quasi-Newton search direction, like Quasi-Newton rootfinding, provide alternatives to calculating the Hessian by using an approximation matrix. The Quasi-Newton optimization algorithm is called BFGS after its developers: Broyden, Fletcher, Goldfarb, and Shanno.

5.2.3 Trust Region Methods

Line search methods pick a direction and search along that direction attempting to find a lower function value for a minimization problem. Trust region methods take a different approach. They construct an approximating model, m_k, whose behavior near the current iterate x_k is close to that of the actual function f. We then search for a minimizer of m_k. However these approximating models m_k may not represent f well when far away from the current iterate x_k, so we restrict the search for a minimizer to be within some region of x_k, called a *trust region.* Trust region problems can be formulated as,

$$\min_p m_k(x_k + p),$$

where $x_k + p \in \Gamma$ where Γ is a ball defined by $||p||_2 \leq \Delta$, where Δ is called the trust region radius. Typically the approximating model m_k is a quadratic function (i.e. a second-order Taylor

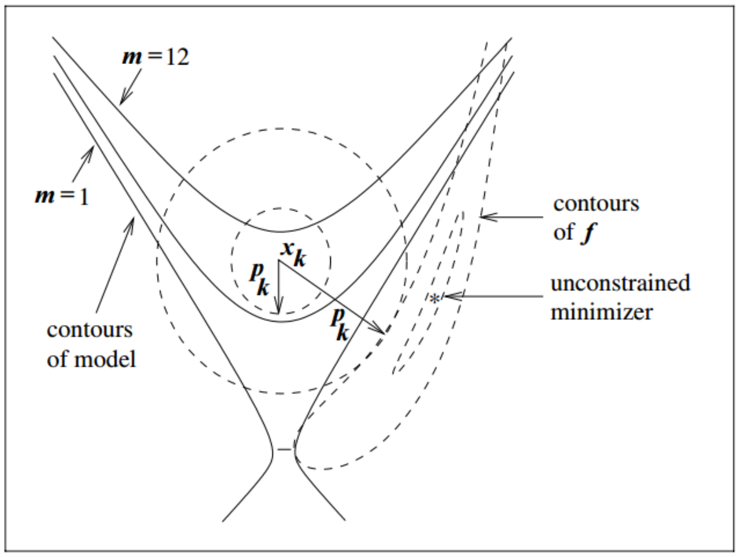

Figure 9: How a trust region algorithm works with an approximating model and the true function.

approximation),

$$m_k(x_k + p) = f_k + p^T \nabla f_k + \frac{1}{2!} p^T B_k p$$

where B_k is the Hessian or an approximation to the Hessian.

Whats the fundamental difference between line search and trust region? Line search first picks a direction then searches along that direction for the optimal step length. Trust region does the opposite, we first define our step length, via the trust region radius, then search for the optimal direction. There is a special case of the trust region where if we set B_k to zero, the solution to the problem is $p_k = -\frac{\Delta_k \nabla f_k}{||\nabla f_k||}$. This is just the steepest descent solution for the line search problem.

6 Scaling

6.1 Constrained Optimization

Constrained optimization problems are the most prevalent form in interesting economic problems. In the interest of time we will briefly cover them as they are typically just variants on unconstrained problems that handle the constraints in different ways. We will discuss three types of constrained optimization algorithms: penalty methods, active set methods, and interior point methods.

6.1.1 Penalty Methods

Suppose we wish to minimize some function subject to some function of the control variables equaling some value,[11]

$$\min_x f(x) \text{ subject to: } c_i(x) = 0.$$

How does the algorithm know to not violate the constraint. One way is to introduce a *penalty function* into our objective and remove the constraint. We would then have the following optimization problem,

$$Q(x;\mu) = f(x) + \frac{\mu}{2}\sum_i c_i^2(x),$$

where μ is the penalty parameter. The second term increases the value of the function (works against minimization), the larger the violation of the constraint. Larger values of μ increasingly penalize any violation. Notice that the penalty terms are smooth so we can just use unconstrained optimization techniques to solve the problem by searching for iterates of x_k. We also may want to consider sequences of $\mu_k \to \infty$ as $k \to \infty$. I.e. we may want to allow larger constraint violations early on so the search problem is easier. As we close into the solution, we require increasing strictness in satisfying the constraint.[12]

6.1.2 Active Set Methods

Active set methods are alternatively called sequential quadratic programming methods: they replace the large non-linear constrained problem with a constrained quadratic programming problem and use Newton's method to solve the sequence of simpler quadratic problems. Using the above constrained optimization problem, we can formulate the Lagrangian as,

$$L(x, \lambda) = f(x) - \lambda^T c(x).$$

Denote $A(x)^T$ as the Jacobian of the constraints,

$$A(x)^T = [\nabla c_1(x), ..., \nabla c_m(x)].$$

[11]This can be generalized to inequality constraints as well.

[12]There are also augmented Lagrangian methods that take the quadratic penalty method and add in explicit estimates of Lagrange multipliers to help force binding constraints to bind precisely. See Nocedal and Wright (2006) for more information.

The first-order conditions can be written as,

$$\nabla f(x) - A(x)^T \lambda = 0$$
$$c(x) = 0$$

Define this system of equations as $F(x, \lambda)$. Any solution to the equality constrained problem, where $A(x^*)$ has full rank also satisfies the first-order necessary conditions. What an active set does is to then use Newton's method to find the solution (x^*, λ^*) of $F(x, \lambda)$. However if we have many constraints, keeping track of all of them can be expensive. Active set methods recognize that if an inequality constraint is not binding, or *active*, then it has no influence on the solution. In the iteration procedure we can effectively ignore it! Active set methods find ways to reduce the complexity of the optimization routine by selectively ignoring constraints that are not active (i.e. non-positive Lagrange multipliers) or close to being active.

6.1.3 Interior Point Methods

Interior point methods are also called barrier methods. These methods are typically used for inequality constrained problems.[13] They get the name interior point from traversing the domain along the interior of the inequality constraints. The main question is how do we ensure that we are on the interior, feasible region? Consider the following constrained optimization problem,

$$\begin{gathered}\min_x f(x) \\ \text{subject to: } c_E(x) = 0, c_I(x) \geq 0.\end{gathered} \tag{2}$$

Interior point methods reformulate this problem as,

$$\begin{gathered}\min_{x,s} f(x) \\ \text{subject to: } c_E(x) = 0, c_I(x) - s = 0, s \geq 0,\end{gathered} \tag{3}$$

where s is a vector of slack variables for the constraints. Interior point methods then introduce what is know as a barrier function, in this case a logarithmic function, to eliminate the inequality

[13]These methods tend to be faster when there are a large number of variables, however the algorithms are often not as robust.

constraint,

$$\begin{aligned} &\min_{x,s} f(x) - \mu \sum_{i=1}^{m} log(s_i) \\ &\text{subject to: } c_E(x) = 0, c_I(x) - s = 0, \end{aligned} \tag{4}$$

where μ is our positive barrier parameter. The barrier function prevents the components of s from approaching zero by imposing a logarithmic barrier. This maintains slack in the constraints. With interior point methods, we solve a sequence of barrier problems, until the sequence of barrier parameters, $\{\mu_k\}$ converges to zero. Then the solution to the barrier problem converges to that of the original constrained optimization problem.

References

Hollingsworth, Alex (2015) "Retail Health Clinics: Endogenous Location CHoice and Emergency Department Diversion."

Judd, Kenneth L. (1998) *Numerical Methods in Economics*, Cambridge, MA: MIT Press.

Miranda, Mario J. and Paul L. Fackler (2002) *Applied Computational Economics and Finance*, Cambridge, MA: MIT Press.

Nocedal, J. and S. J. Wright (2006) *Numerical Optimization*, New York: Springer, 2nd edition.

Rowan, Thomas Harvey (1990) "Functional stability analysis of numerical algorithms."