

1 Numerical Methods

In this section we focus on three very common computational tasks in applied microeconomics:

- i) calculating derivatives numerically
- ii) calculating integrals numerically
- iii) solving non-linear optimization problems: $\min f(\theta)$

The methods we discuss are developed in far greater detail outside of economics in the numerical analysis literature. For an introduction, see Atkinson (1989).

1.1 Computation in Economics

A couple of brief remarks about computation. See the Judd (1998) and Miranda and Fackler (2002) textbooks for a fuller discussion.

1.1.1 Computers Can Only Approximate

Computers can only do basic arithmetic (+ - x /). Everything else the computer does is an approximation.

Numbers are stored in the form $m2^e$, where m and e are integers defined in some range depending on the precision of the computer.

Most computers represent numbers in double precision and can represent numbers typically up to 12-16 digits. Single precision represents numbers using fewer digits, typically 6-8. Double precision is the default in Matlab.

For example, the number -3210.48 is approximated in double precision as

$$-7059920181484585x2^{-41} = -3210.48000000000001818989\dots$$

Machine Epsilon

Machine epsilon (ϵ) is the smallest quantity that is representable by a computer. Machine ϵ is defined as the smallest ϵ such that the machine knows that $1 + \epsilon > 1 > 1 - \epsilon$. Machine epsilon provides the spacing between machine representable numbers.

In Matlab, the command $eps(x)$ provides the spacing between x and the next machine representable integer.

For $x = 1$,

```
eps(1)
```

```
ans =
```

```
2.220446049250313e-016
```

The computer knows that

$$1 + eps(1) = 1 + 2.220446049250313e - 016 \neq 1$$

For single precision, machine epsilon is smaller

```
eps(single(1))
```

```
ans =
```

```
1.1920929e-007
```

An Example of Rounding Error

Rounding errors occur because numbers must be rounded to the nearest machine representable integer. Try this in Matlab

```
10000000.2 - 10000000.1
```

```
ans =
```

```
0.09999999962747
```

The correct answer is 0.1, but because of rounding we have error.

Machine Zero and Infinity

Machine zero is the smallest number which is representable on a computer.

Machine infinity is the largest number that both it and its negative are machine representable by a computer.

On my (laptop) computer, Matlab reports that machine infinity and machine zero are

```
realmax
```

```
ans =
```

```
1.7977e+308
```

```
realmin
```

```
ans =
```

```
2.2251e-308
```

In single precision,

```
realmax('single')
```

```
ans =
```

```
3.4028e+038
```

```
realmin('single')
```

```
ans =
```

```
1.1755e-038
```

Underflow

An *underflow* error occurs when the output of an operation involving greater than machine zero numbers is a number below machine zero. Typically, Matlab will report this as “-Inf”.

Calculate:

```
1 - 1e1000
```

```
ans =
```

```
-Inf
```

Overflow

An *overflow* error occurs when the output of an operation involving less than machine infinity numbers is a number above machine infinity. Typically, Matlab will report this as “Inf”.

Calculate:

```
5e1000
```

```
ans =
```

```
Inf
```

Scaling

Because of these overflow and underflow errors, it is important to scale numbers appropriately. This is particularly important for optimization algorithms with an unrestricted parameter space. We would have an overflow problem if we were to try a parameter of $\theta = 1000$ for an objective function of $f(\theta) = e^\theta$. In Matlab,

```
exp(1000)
```

```
ans =
```

```
Inf
```

Logical Operations

Machine representation can interfere with logical operations. Consider this from STATA:

```
help datatypes
```

Precision of numeric storage types

floats have about 7 digits of accuracy; the magnitude of the number does not matter. Thus, 1234567 can be stored perfectly as a float, as

can 1234567e+20. The number 123456789, however, would be rounded to 123456792. In general, this rounding does not matter.

If you are storing identification numbers, the rounding could matter. If the identification numbers are integers and take 9 digits or less, store them as longs; otherwise, store them as doubles. doubles have 16 digits of accuracy.

Stata stores numbers in binary, and this has a second effect on numbers less than 1. $1/10$ has no perfect binary representation just as $1/11$ has no perfect decimal representation. In float, .1 is stored as .10000000149011612. Note that there are 7 digits of accuracy, just as with numbers larger than 1. Stata, however, performs all calculations in double precision. If you were to store 0.1 in a float called x and then ask, say, "list if x==.1", there would be nothing in the list. The .1 that you just typed was converted to double, with 16 digits of accuracy (.100000000000000014...), and that number is never equal to 0.1 stored with float accuracy.

One solution is to type "list if x==float(.1)". The float() function rounds its argument to float accuracy; see functions. The other alternative would be store your data as double, but this is probably a waste of memory. Few people have data that is accurate to 1 part in 10 to the 7th. Among the exceptions are banks, who keep records accurate to the penny on amounts of billions of dollars. If you are dealing with such financial data, store your dollar amounts as doubles. See [U] 13.10 Precision and problems therein.

Commands for STATA:

```
input d
0.1
end
gen y = 1 if d == 0.1
tab y, m
```

Result:

no observations

Matlab doesn't have this problem because everything created in Matlab is stored in double precision. Thus,

```
d = 0.1
if d = 0.1
y = 1
end
```

Result:

```
y =
1
```

1.1.2 3 Types of Computations and their Error

1) *Direct Methods*

Direct Methods are when there is a closed form expression: e.g. $x = a/b$. The only error here is rounding error which will depend on the degree of precision in which a and b are stored. The rounding error may actually cause a continuous function to be discontinuous numerically. For example, for multiple a/b values larger than machine infinity, $x = f(a, b) = a/b = \infty$.

2) *Series Methods*

Series Methods are operations involving an infinite or long finite series which must be approximated by truncating the sequence at some point. For example, the exponential function is defined as

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}$$

A computer would approximate this as

$$e^x \approx \sum_{n=0}^N \frac{x^n}{n!}$$

for finite N . The approximation error here is because we use a finite series to approximate an infinite series. These truncation errors can also cause this continuous function $f(x) = e^x$ to be numerically discontinuous in x .

3) *Iterative Methods*

Iterative methods depend on the particular algorithm and starting point.

For example, consider an optimization problem without a closed form solution,

$$\theta^* = \arg \min_{\theta \in \Theta} f(\theta),$$

with continuous parameter space Θ , which needs to be solved using an iterative optimization algorithm. Typically, these iterative algorithms start from a vector of starting parameters, and successively evaluate the objective function at different trial vectors of parameters until a convergence criteria is met.

Unlike series methods, there is no one path through the parameter space to complete this operation. We can only say that with an infinite number of iterations, we can reach the actual minimizing vector. For a number of reasons, the solution from this iterative algorithm, call it $\widehat{\theta}^*$, can be very different from the actual solution we discuss in econometrics, $\theta^* = \arg \min f(\theta)$. Given this, we might say that iterative methods have an “error” of the form $\widehat{\theta}^* - \theta^*$. Characterizing these errors is very difficult. We’ll discuss iterative optimization algorithms in more detail below.

1.1.3 Art vs. Science

Many of the methods to solve problems numerically on a computer do not have very precise justifications. In theoretical econometrics, we can often compare estimators based on verifiable criteria such as consistency and pre-

cision. In applied econometrics, there may be several different ways to implement a problem numerically, and there is no clear cut reason to prefer one method over the other. Instead, the criteria for preferring one numerical method over another is often based on practical and somewhat ad hoc reasoning. This is the “art” of applied econometrics, as opposed to the “science” of theoretical econometrics.

1.1.4 Speed Matters

In addition, a consideration not found in theoretical econometrics, but which is very important to applied methods, is the computation time it takes to conduct an operation. With an infinite amount of computer resources many of the issues we have to deal with here become irrelevant. With finite computer resources, the simpler and more time consuming methods to approaching problems may not be the best.

1.2 Numerical Differentiation

Finite difference methods to calculate derivatives are often used in numerical optimization and for the construction of objects requiring derivatives (e.g. standard errors for parameter estimates) when the analytic derivative does not have a closed form. We might also consider using a numerical derivative if calculating the closed form would be too time-consuming or too prone to human errors.

1.2.1 One-Sided Approximations

The derivative is defined as

$$f'(x) = \lim_{\epsilon \rightarrow 0} \frac{f(x + \epsilon) - f(x)}{\epsilon},$$

where x is a scalar.

This suggests an approximation based on a one-sided finite difference for small $h > 0$:

$$\hat{f}'(x) = \frac{f(x + h) - f(x)}{h}$$

Another way to derive this is using a Taylor expansion. The first order Taylor series approximation to the scalar valued function $f(y)$ at the point $y = a$ is

$$f(y) \approx f(a) + f'(a)(y - a)$$

Define $y = x + h$ and evaluate the function at $a = x$.

$$f(x + h) \approx f(x) + f'(x)h$$

Define the first order remainder as

$$R_1(x, h) = f(x + h) - [f(x) + f'(x)h]$$

$R_1(x, h)$ is declining as h gets smaller. $R_1(x, h)$ is order of magnitude $O(h^2)$ as the remaining terms in the Taylor expansion are multiples of h^2 and higher powers. That is,

$$\lim_{h \rightarrow 0} \frac{|R_1(x, h)|}{|h|^2} < \infty$$

We can then write,

$$f(x + h) = f(x) + f'(x)h + O(h^2)$$

Solving for $f'(x)$

$$f'(x) = \frac{f(x + h) - f(x)}{h} + O(h)$$

1.2.2 Two-Sided Approximations

A more accurate approximation can be found using a two-sided finite difference. Two second order Taylor approximations are

$$f(x + h) = f(x) + f'(x)h + f''(x)\frac{h^2}{2} + R_2(x, h)$$

$$f(x - h) = f(x) + f'(x)(-h) + f''(x)\frac{(-h)^2}{2} + R_2(x, -h),$$

where

$$R_2(x, h) = f(x + h) - [f(x) + f'(x)h + f''(x)\frac{h^2}{2}]$$

$$R_2(x, -h) = f(x - h) - [f(x) + f'(x)(-h) + f''(x)\frac{(-h)^2}{2}]$$

Both $R_2(x, h)$ and $R_2(x, -h)$ are $O(h^3)$.

Subtract,

$$f(x + h) - f(x - h) = f(x) - f(x) + f'(x)h + f'(x)h + O(h^3)$$

$$f'(x) = \frac{f(x + h) - f(x - h)}{2h} + O(h^2)$$

The two-sided finite difference approximation is then

$$\hat{f}'(x) = \frac{f(x + h) - f(x - h)}{2h}$$

It has error of an order of magnitude less than that of the one-side finite difference: $O(h^2)$ rather than $O(h)$.

1.2.3 An Example

Let's approximate $f(x) = x^3$. The analytic derivative is $f'(x) = 3x^2$. At $x = 10$, $f'(x) = 300$.

With $h = 0.01 * x = 0.1$ (one percent of x at $x = 10$), the one-sided numerical derivative is

$$\hat{f}'(x) = \frac{(10 + 0.1)^3 - f(10)^3}{0.1} = 303.01$$

The two-sided numerical derivative is

$$\hat{f}'(x) = \frac{(10 + 0.1)^3 - f(10 - 0.1)^3}{0.2} = 300.01$$

1.2.4 Using More Points

To see where these approximations come from and how other approximations using more points could be formed, let's examine the basic 3 point formula.

We evaluate the function at three points: x , $x + h_1$, and $x + h_2$. The approximation to the derivative is a weighted sum of these three points, with weights a , b , c :

$$f'(x) \approx af(x) + bf(x + h_1) + cf(x + h_2)$$

To find the weights, we use a second order Taylor series approximation for $f(x + h_1)$ and $f(x + h_2)$ around x

$$bf(x + h_1) = bf(x) + bf'(x)h_1 + bf''(x)\frac{h_1^2}{2} + O(h_1^3)$$

$$cf(x + h_2) = cf(x) + cf'(x)h_2 + cf''(x)\frac{h_2^2}{2} + O(h_2^3)$$

We can add each of the second order Taylor series approximations (plus $af(x)$ as well) to get

$$\begin{aligned} &af(x) + bf(x + h_1) + cf(x + h_2) = \\ &[af(x)] + [bf(x) + bf'(x)h_1 + bf''(x)\frac{h_1^2}{2}] + [cf(x) + cf'(x)h_2 + cf''(x)\frac{h_2^2}{2}] + O(h_1^3) + O(h_2^3) \end{aligned}$$

Simplifying,

$$\begin{aligned} &af(x) + bf(x + h) + cf(x + h) = \\ &(a + b + c)f(x) + (bh_1 + ch_2)f'(x) + \frac{(bh_1^2 + ch_2^2)}{2}f''(x) + O(h_1^3) + O(h_2^3) \end{aligned}$$

We obtain a good approximation if the weights satisfy these three condition (given a h_1 and h_2):

Condition i) Eliminate the function evaluation $f(x)$

$$a + b + c = 0$$

Condition ii) Eliminate the second derivative component

$$\frac{(bh_1^2 + ch_2^2)}{2} = 0$$

Condition iii) The term on $f'(x)$ is 1

$$bh_1 + ch_2 = 1$$

For a given h_1 and h_2 , this is a system of three equations in three unknowns a , b , c . We can solve this for the unique a^* , b^* , and c^* . The three point approximation is then

$$\hat{f}'(x) = a^* f(x) + b^* f(x + h) + c^* f(x + h) + O(h_1^3) + O(h_2^3)$$

1.2.5 Special Cases of Multiple Points

Case i) *One-side derivative approximation*

The one-side derivative approximation is $c^* = 0$ (eliminate third point).

Set $h_1 = h_2 = h$.

$$a^* = \frac{1}{h}$$

$$b^* = -a^*$$

One-side approximation:

$$\hat{f}'(x) \approx \frac{1}{h}f(x) - \frac{1}{h}f(x+h)$$

The one-sided approximation meets conditions i).

Condition iii) is also satisfied:

$$\frac{1}{h}h + 0 = 1$$

But these choices of weights do not eliminate the $f''(x)$ term (condition ii). As we showed above it has higher approximation error than the two-side approximation.

Case ii) *Two-side derivative approximation*

Set $h_1 = h$ and $h_2 = -h$.

$$a^* = 0$$

$$b^* = \frac{1}{2h}$$

$$c^* = -b^*$$

Condition i) is satisfied.

Condition iii) is also satisfied

$$bh_1 + ch_2 = 1$$

$$b^*h - b^*(-h) = \frac{1}{2h}h + \frac{1}{2h}h = 1$$

Condition ii) is also satisfied

$$\frac{(bh_1^2 + ch_2^2)}{2} = 0$$

$$\frac{(b^*h^2 - b^*h^2)}{2} = 0$$

1.2.6 Approximating a Gradient

If X is a $K \times 1$ vector, the gradient can be approximated numerical as

$$\widehat{G}(X) = \left[\frac{\partial \widehat{f}(X)}{\partial X_1}, \dots, \frac{\partial \widehat{f}(X)}{\partial X_K} \right],$$

where

$$\frac{\partial \widehat{f}(X)}{\partial X_k} = \frac{f(X + h_k e_k) - f(X - h_k e_k)}{2h},$$

and $e_k = [0, \dots, 0, 1, 0, \dots, 0]$ is a $K \times 1$ vector with a 1 in the k th position. h_k is the finite difference for the k th variable. In words, we approximate a gradient using finite difference for each parameter k separately, keeping all other parameters at their original values.

Note that approximating a K dimension gradient using a two sided approximation involves $2K$ function evaluations.

1.2.7 Approximating Second Derivatives

We can approximate second derivatives the same way as with first derivatives.

For a scalar x , we can write the third-order Taylor series expansions as

$$f(x + h) = f(x) + f'(x)h + f''(x)\frac{h^2}{2} + f'''(x)\frac{h^3}{6} + O(h^4)$$

$$f(x - h) = f(x) - f'(x)h + f''(x)\frac{h^2}{2} - f'''(x)\frac{h^3}{6} + O(h^4)$$

Add the two equations,

$$f(x+h) + f(x-h) = 2f(x) + f''(x)h^2 + O(h^4)$$

Re-arrange,

$$f''(x) = \frac{f(x+h) + f(x-h) - 2f(x)}{h^2} + O(h^2)$$

The finite difference approximation to the second derivative is then

$$\widehat{f}''(x) = \frac{f(x+h) + f(x-h) - 2f(x)}{h^2}$$

It has an approximation error of order of magnitude $O(h^2)$.

The approximation to the cross-partial for $f(x_i, x_j)$ is

$$\frac{\partial^2 \widehat{f}(x_i, x_j)}{\partial x_i \partial x_j} = \frac{f(x_i + h_i, x_j + h_j) - f(x_i, x_j + h_j) - [f(x_i + h_i, x_j) - f(x_i, x_j)]}{h_i h_j}$$

Approximations for multi-dimensional Hessian matrices involve substantially more calculations. See the Miranda and Fackler (2002) textbook.

1.2.8 Why Calculating Second Derivatives is Avoided

In general, numerically computing Hessian matrices is often avoided for two reasons:

1) Too many function evaluations are required.

2) The numerically calculated Hessian matrix may be ill-behaved (e.g. singular). In fact, a number of refinements to the Newton-Raphson optimization algorithm, which we'll discuss below, were developed in order to avoid computing the Hessian numerically.

1.2.9 How Large Should h Be?

If we are trying to approximate the limit argument in the definition of a derivative, we should prefer a small h . However, the numerical derivative may be highly inaccurate for small h . This is particularly a problem if the $f(\cdot)$ function is approximated or involves a non-smooth simulator (e.g. a crude frequency simulator, see next section). It is possible that for a sufficiently small h , the numerical approximation to a derivative is 0, where in fact it is non-zero. This would cause derivative based optimization algorithms to fail, and be a major problem for the computation of standard errors using numerical derivatives. In general, the more accurately the function $f(\cdot)$ can be represented (either because of machine precision or because $f(\cdot)$ is approximated in some way), the more confident we can be that a small h is not too small.

The Miranda and Fackler (2002) textbook recommend that h for the one-sided derivative be set to

$$h = \max(|x|, 1)\sqrt{(\epsilon)}$$

where ϵ is machine epsilon. This is written to avoid using too small h for small $|x|$.

For two-side derivatives, Miranda and Fackler (2002) recommend

$$h = \max(|x|, 1)\sqrt[3]{\epsilon}$$

In our example above, $f(x) = x^3$ at $x = 10$, actual $f'(x) = 300$, the one-sided h is

$$h = \max(|x|, 1)\sqrt{(\epsilon)} = 10 * \sqrt{(2.220446049250313e - 016)} = 1.4901e - 007$$

The one-sided numerical derivative is

$$\hat{f}'(x) = 300$$

For the two-sided derivative

$$h = \max(|x|, 1)\sqrt[3]{\epsilon} = 6.0555e - 005$$

$$\hat{f}'(x) = 300$$

Although these h values are able to approximate exactly the derivative, these h seem rather small to me for many applications.

1.2.10 Playing Games with Finite Derivatives

It's important to understand that the choice of h can affect the computation of certain objects. Let's consider an example of computing standard errors for parameter estimates in a method of moments estimator (described in more detail below).

Assume we have an iid. sample of $i = 1, \dots, N$ data observations. The MOM estimator is

$$\hat{\theta} = \arg \min g(\theta)'g(\theta),$$

where the $K \times 1$ vector $g(\theta)$ is defined as

$$g(\theta) = \frac{1}{N} \sum_i g_i(\theta)$$

$$E[g_i] = 0_K$$

The asymptotic distribution of the MOM estimator is then

$$\hat{\theta} \sim^a N\left(\theta_0, \frac{D_0^{-1}W_0D_0^{-1}}{N}\right),$$

where $D_0 = E[g'(\theta_0)] = E[\frac{\partial g(\theta_0)}{\partial \theta}]$,

$$\sqrt{N}(g(\theta_0) - 0) \rightarrow^d N(0, W_0),$$

and W_0 is the population variance of $g(\cdot)$ evaluated at the true parameter θ_0 .

We can estimate the K dimensional vector of standard errors for $\hat{\theta}$ as

$$\text{SE}(\hat{\theta}) = (\text{diag}[\frac{1}{N}D(\hat{\theta})^{-1}W(\hat{\theta})D(\hat{\theta})^{-1}])^{1/2}$$

where

$$D(\hat{\theta}) = \frac{\partial g(\hat{\theta})}{\partial \theta}$$

If this derivative is computed numerically using a one-sided finite difference h , then the approximation is

$$\widehat{D}(\hat{\theta}, h) = \frac{g(\hat{\theta} + h) - g(\hat{\theta})}{h}$$

where I include the h to indicate that this derivative depends on h .

The standard errors using finite difference derivatives then also depend on h

$$\widehat{\text{SE}}(\hat{\theta}, h) = (\text{diag}[\frac{1}{N}\widehat{D}(\hat{\theta}, h)^{-1}W(\hat{\theta})\widehat{D}(\hat{\theta}, h)^{-1}])^{1/2}$$

We could then think about minimizing the standard errors by choosing h :

$$h^* = \arg \min_h \widehat{\text{SE}}(\hat{\theta}, h)$$

Doing this would seem to be very bad practice, as it would let researchers choose h 's to minimize their standard errors. However, often h has to be chosen specifically for individual problems. Notice that if we were to use too small of an h for a non-smooth objective function, $\widehat{D}(\hat{\theta}, h)$ approaches 0 and $\widehat{D}(\hat{\theta}, h)^{-1}$ and $\widehat{\text{SE}}(\hat{\theta}, h)$ approach ∞ .

1.3 Numerical Integration

Numerical integration or quadrature is a deterministic approach to evaluating integrals. This is in contrast to the stochastic or simulation (Monte Carlo) based approach we'll discuss later.

Here we'll go over some of the basic ideas in numerical integration. If you want more information on numerical integration, see Judd (1998), Atkinson (1989), or Kythe and Schaferkötter (2005).

1.3.1 Definite Integrals

We want to approximate the following definite integral

$$I = \int_a^b f(x)dx,$$

where a and b are finite constants.

There are two motivations for approximating integrals. First, as with numerical derivatives, analytically evaluating the integral may be too messy and time-consuming. Second, some integrals do not have closed form expressions.

The numerical or quadrature approach to approximating integrals is based on approximating the integrand $f(x)$ with a function which has an analytic integral. We approximate the integral as

$$\hat{I} = \int_a^b \hat{f}(x) dx$$

The analytic evaluation of the integral then defines a quadrature rule for that integral

$$\hat{I} = \sum_{j=1}^M w_j f(x_j),$$

where the original function is evaluated at M points or nodes x_j . w_j are weights for the these points.

1.3.2 Indefinite Integrals

For the evaluation of indefinite integrals, we have two options. The first option is use definite integrals with large a and b . This approach would likely require a relatively large number of function evaluations to achieve a good approximation.

The second option is a change of variables. A general change of variables formula transforms is

$$\int_a^b g(y)dy = \int_{h^{-1}(a)}^{h^{-1}(b)} g(h(x))h'(x)dx$$

where $y = h(x)$ is a non-linear relation, and $h'(x)$ is the derivative of $h(x)$ with respect x . For example, if want to evaluate this integral

$$I = \int_0^\infty g(y)dy$$

A change of variables for this integral takes the form

$$I = \int_0^1 g(h(x))h'(x)dx$$

where $h(0) = 0$ and $h(1) = \infty$.

One such transformation is

$$h(x) = \frac{x}{1-x}$$

$$\lim_{x \rightarrow 0} h(x) = 0$$

$$\lim_{x \rightarrow 1} h(x) = \infty$$

Using this change of variables, we can write

$$I = \int_0^\infty g(y)dy = \int_0^1 g\left(\frac{x}{1-x}\right)(1-x)^{-2}dx$$

since by the product rule

$$h'(x) = 1(1-x)^{-1} + x(-1)(-1)(1-x)^{-2} = \frac{1}{1-x}\left[1 + \frac{x}{1-x}\right] = (1-x)^{-2}$$

We can now use quadrature rules to evaluate this integral on the interval $[0, 1]$.

Note that this change of variables only works for integrals where this limit exists

$$\int_0^\infty f(x)dx = \lim_{b \rightarrow \infty} \int_0^b f(x)dx$$

This would not be the case, for example, for $\int_0^\infty 1dx$.

1.3.3 Multi-Dimension Quadrature

Consider a K -dimensional integral

$$I = \int \cdots \int f(x_1, \dots, x_K)dx_1, \dots, dx_K,$$

where $f(x_1, \dots, x_K)$ is the joint PDF of K random variables.

The simplest quadrature method to approximate this multi-dimensional

integral uses the *product rule* based on the product of one-dimensional quadrature rules.

As an example, consider a two-dimensional integral,

$$I = \int \int f(x_1, x_2) dx_1 dx_2,$$

For each dimension of the integral, we have evaluation points and weights

$$x_{1,1}, x_{1,2}, \dots, x_{1,J}$$

$$w_{1,1}, w_{1,2}, \dots, w_{1,J}$$

$$x_{2,1}, x_{2,2}, \dots, x_{2,B}$$

$$w_{2,1}, w_{2,2}, \dots, w_{2,B}$$

The product rule evaluates the function at all combinations of the evaluation points. The weights for these joint evaluation points is the product of the weights for the individual integrals.

The product rule approximation for the two dimension integral takes the form

$$\hat{I} = \sum_{j=1}^J \sum_{b=1}^B w_{1,j} w_{2,b} f(x_{1,j}, x_{2,b})$$

This two-dimensional integral is evaluated at $J \times B$ points.

For a K dimension integral, in which each one-dimensional integral is evaluated at M points, the product rule quadrature requires M^K function evaluations. This “curse of dimensionality” implies that product rule quadrature may be computationally infeasible for large dimension integrals. For this reason, many economic applications have used Monte Carlo methods.

1.3.4 Two Types of Quadrature Formulas

There are two general classes of quadrature formulas:

i) *Newton-Cotes Formulas*

Newton-Cotes formulas for numerical integration are simple and straightforward. They are based on using a fixed number of points which are equally spaced in the domain $[a, b]$. An integrand which has an analytic solution is formed by interpolating the function between these points. Examples of Newton-Cotes rules are the Midpoint, Trapezoid, and Simpson’s rules.

ii) *Gaussian Quadrature*

Gaussian quadrature uses a low order polynomial approximation to the integrand. The Gaussian quadrature rules are based on the analytic integral for the polynomial approximation. The distinguishing feature of the Gaussian quadrature rules are that they use specific weights and nodes for particular kinds of integrals. Example of Gaussian quadrature include Gauss-Hermite, Gauss-Legendre, etc.

1.4 Numerical Integration: Newton-Cotes Rules

1.4.1 Midpoint Rule

The midpoint rule uses a single rectangle centered at the midpoint of $[a, b]$, $\frac{a+b}{2}$, with height equal to $f(\frac{a+b}{2})$. The midpoint rule essentially uses a linear approximation to the integral. The midpoint rule approximation is

$$\hat{I} = (b - a)f\left(\frac{a + b}{2}\right)$$

This approximation will likely be too coarse. Consider approximating the integral using several rectangles. Divide up the range of $[a, b]$ into $M + 1$ points: $x_0 < x_1 < \dots < x_M$. The piecewise or composite midpoint rule approximation is then

$$\hat{I} = \frac{(b - a)}{M} \sum_{j=1}^M f(\bar{x}_j),$$

where $\bar{x}_j = \frac{1}{2}(x_{j-1} + x_j)$ are the midpoints between x_{j-1} and x_j .

As you might expect, approximating the integral using more function evaluations decreases the approximation error.

1.4.2 An Example of the Midpoint Rule

Let's approximate the following integral:

$$I = \int_0^1 x^2 dx$$

The analytic answer is $I = 1/3$.

First, we use the one point midpoint rule:

$$\hat{I} = (1 - 0) * (1/2)^2 = 1/4$$

Next, let's evaluate the function at 3 points ($m = 3$). Split the range of $[1, 0]$ into 4 points: 0, 1/3, 2/3, 1. The 3 midpoints are 1/6, 1/2, 5/6.

$$\hat{I} = \frac{(1 - 0)}{3}((1/6)^2 + (1/2)^2 + (5/6)^2) = 0.32407$$

This is much closer to the analytic answer than 1/4.

1.4.3 Trapezoid Rule

The trapezoid rule approximates the integral using the integrand evaluated at the endpoints.

$$\hat{I} = \frac{b - a}{2}[f(a) + f(b)]$$

The height of the trapezoid is $b - a$. The length of the two sides is $f(a)$ and $f(b)$.

Notice that the trapezoid rule involves twice as many function evaluations as the simple one point midpoint rule. It therefore has less approximation error.

A multiple point piecewise trapezoid rule using $M + 1$ evaluation points

$$\hat{I} = \frac{b-a}{2M} [f(x_0) + 2f(x_1) + 2f(x_2) + \cdots + 2f(x_{M-1}) + f(x_M)],$$

where the evaluation points are $x_j = a + j\frac{b-a}{M}$.

Notice that all of the interior points (x_1, \dots, x_{M-1}) are multiplied by 2 because they are the endpoints for 2 contiguous trapezoids.

1.4.4 Simpson's Rule

The midpoint and trapezoid rules use linear approximations. Simpson's rule uses a quadratic approximation by evaluating the integrand at three points: the endpoints a and b , and the midpoint $m = 1/2(a+b)$. Simpson's rule uses the unique quadratic function which passes through the three graph points:

$$[a, f(a)], [m, f(m)], [b, f(b)]$$

This quadratic approximation function at these three points, call it $P(x)$, is defined as

$$P(x) = f(a) \frac{(x-m)(x-b)}{(a-m)(a-b)} + f(m) \frac{(x-a)(x-b)}{(m-a)(m-b)} + f(b) \frac{(x-a)(x-m)}{(b-m)(b-a)}$$

The integral is then approximated as the area under this quadratic function:

$$\hat{I} = \int_a^b P(x)dx = \frac{b-a}{6}[f(a) + 4f(m) + f(b)]$$

A $M + 1$ points composite Simpson's rule is the piecewise quadratic approximation for three consecutive points:

$$\hat{I} = \frac{b-a}{3M}[f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + \cdots + 4f(x_{M-1}) + f(x_M)],$$

where the evaluation points are $x_j = a + j\frac{b-a}{M}$

1.5 Numerical Integration: Gaussian Quadrature

Like Newton-Cotes rules, Gaussian quadrature approximates integrands by evaluating the integrand at specific points. The Gaussian quadrature approximation to the integral I is

$$\hat{I} = \sum_{j=1}^m w_j f(x_j)$$

However, the Gaussian quadrature rules choose both the evaluation points or nodes x_j and the weights for each point w_j in order to approximate specific integrals well. In contrast, the Newton-Cotes rules uses evenly spaced evaluation points across the $[a, b]$ domain.

The evaluation points and weights are chosen so that the Gaussian quadra-

ture approximation holds exactly for a polynomial approximation to the integral. In general, Gaussian quadrature using M evaluation points is exact up to $2M - 1$ order polynomial approximation to the integrand.

1.5.1 Derivation of Gaussian Formulas

To see how this works, let's consider an example. We use $M = 2$ evaluation points.

We want to evaluate this definite integral

$$I = \int_{-1}^1 f(x) dx$$

Assume the integrand is exactly a third order polynomial

$$f(x) = c_0 + c_1x + c_2x^2 + c_3x^3$$

The Gaussian approximation is

$$\hat{I} = \sum_{j=1}^2 w_j (c_0 + c_1x_j + c_2x_j^2 + c_3x_j^3)$$

If we set the approximation equal to the actual integral, we have

$$I = \hat{I}$$

$$\int_{-1}^1 (c_0 + c_1x + c_2x^2 + c_3x^3)dx$$

$$= \sum_{j=1}^2 w_j (c_0 + c_1x_j + c_2x_j^2 + c_3x_j^3)$$

Re-writing both sides,

$$\int_{-1}^1 c_0 dx + \int_{-1}^1 c_1 x dx + \int_{-1}^1 c_2 x^2 dx + \int_{-1}^1 c_3 x^3 dx$$

$$= c_0(w_1 + w_2) + c_1(w_1x_1 + w_2x_2) + c_2(w_1x_1^2 + w_2x_2^2) + c_3(w_1x_1^3 + w_2x_2^3)$$

Evaluating the integral on the left-hand side,

$$2c_0 + 0c_1 + \frac{2}{3}c_2 + 0c_3$$

$$= c_0(w_1 + w_2) + c_1(w_1x_1 + w_2x_2) + c_2(w_1x_1^2 + w_2x_2^2) + c_3(w_1x_1^3 + w_2x_2^3)$$

This equation is satisfied if

$$2 = w_1 + w_2$$

$$0 = w_1x_1 + w_2x_2$$

$$\frac{2}{3} = w_1x_1^2 + w_2x_2^2$$

$$0 = w_1 x_1^3 + w_2 x_2^3$$

This is a four equation system with four unknowns: w_1, w_2, x_1, x_2 .

The unique solution is

$$x_1 = -\frac{1}{\sqrt{3}}, x_2 = \frac{1}{\sqrt{3}}, w_1 = 1, w_2 = 1.$$

Using these $M = 2$ nodes and associate weights, our Gaussian approximation is exact for integrands of polynomial order equal to or less than $2M - 1 = 3$.

1.5.2 Examples

Let's try this for an integrand that is lower than a 3rd order polynomial.

$$I = \int_{-1}^1 x^2 dx = \frac{2}{3}$$

Here $f(x) = x^2$.

The Gaussian approximation using $M = 2$ nodes is

$$\hat{I} = 1 * f\left(-\frac{1}{\sqrt{3}}\right) + 1 * f\left(\frac{1}{\sqrt{3}}\right) = 1/3 + 1/3 = 2/3$$

Next, let's try this for an integrand that is a higher than 3rd order polynomial.

$$I = \int_{-1}^1 x^4 dx = \frac{x^5}{5} \Big|_{-1}^1 = \frac{1}{5} - \frac{-1}{5} = \frac{2}{5}$$

The Gaussian approximation using $M = 2$ nodes is

$$\begin{aligned}\hat{I} &= 1 * f\left(-\frac{1}{\sqrt{3}}\right) + 1 * f\left(\frac{1}{\sqrt{3}}\right) = \left(\frac{1}{\sqrt{3}}\right)^4 + \left(-\frac{1}{\sqrt{3}}\right)^4 \\ &= \frac{2}{9}\end{aligned}$$

1.5.3 Specific Gaussian Quadrature Formulas

In practice, we wouldn't derive the nodes and weights, as in the simple example above. Instead, we use already developed formulas and tables of nodes and weights for specific integrals.

i) *Gauss-Legendre Quadrature*

Gauss-Legendre quadrature approximates integrals of this form

$$\int_{-1}^1 f(x)dx \approx \sum_{j=1}^M w_j f(x_j)$$

This is the same as the integral we looked at above. The weights and nodes for $M > 2$ can be found in Judd (1998).

Or more generally, for domain $[a, b]$, we use the change of variables:

$$\int_a^b f(x)dx \approx \sum_{j=1}^M \frac{b-a}{2} w_j f\left(\frac{(x_j + 1)(b-a)}{2} + a\right)$$

The multiplicative factor $\frac{b-a}{2}$ is taken out of the weights for convenience and numerical precision.

ii) *Gauss-Laguerre Quadrature*

Gauss-Laguerre quadrature is for integrals of the form

$$\int_0^{\infty} f(x)e^{-x}dx \approx \sum_{j=1}^M w_j f(x_j)$$

The nodes and weights can be found in Judd (1998).

iii) *Gauss-Hermite Quadrature*

Gauss-Hermite quadrature is for integrals of the form

$$\int_{-\infty}^{\infty} f(x)e^{-x^2}dx \approx \sum_{j=1}^M w_j f(x_j)$$

The nodes and weights can be found in Judd (1998).

1.6 Optimization Algorithms

A large number of estimators do not have closed form representations and require numerical optimization. We first consider the leading types of optimization algorithms for unconstrained optimization problems of the form

$$\theta^* = \arg \min_{\theta} f(\theta)$$

For a discrete and small parameter space, a simple grid search method could be used to find θ^* . For a continuous and/or high dimension parameter

space, more complex algorithms are required.

In order to emphasize that without a closed form for θ^* , the solution we obtain using iterative methods is only approximate, we write the solution obtained as $\hat{\theta}^*$.

1.6.1 Basic Structure of a Non-Linear Optimization Algorithm

Step 1) *Choose Starting Parameters*

The researcher first chooses a vector of starting parameters, θ_0 .

Step 2) *Calculate Next Candidate Parameter*

Given the candidate parameter from the last step θ_t , the algorithm defines the next candidate parameter θ_{t+1} . This typically requires computationally expensive objective function evaluations. The specific algorithm determines the particular sequential *search path* of trial parameters to consider. The search path for a given starting vector θ_0 is

$$\{\theta_0, \theta_1, \theta_2, \dots\}$$

Step 3) *Stop When Stopping or Convergence Criteria is Met*

If the current vector of trial parameters satisfies a stopping or convergence criterion, the algorithm stops and the current trial parameters are accepted

as the parameter estimates. Typically, these stopping criteria are based on convergence in the sequence of trial parameters or the evaluated objective function at the trial parameters.

1.6.2 Convergence/Stopping Criterias

i) *Scalar Parameters*

1) A stopping criteria for a scalar trial parameter θ_{t+1} might take the form:

θ_{t+1} is the accepted parameter (i.e. $\theta_{t+1} = \hat{\theta}^*$) if

$$\frac{|\theta_{t+1} - \theta_t|}{|\theta_t|} < \epsilon,$$

where $\epsilon > 0$ is stopping criteria parameter.

If θ_t converges to zero, we might never obtain convergence.

2) An alternative is

$$\frac{|\theta_{t+1} - \theta_t|}{1 + |\theta_t|} < \epsilon$$

Or we could simply have a criteria of

$$|\theta_{t+1} - \theta_t| < \epsilon$$

and adjust ϵ appropriately for the scale of θ .

ii) *Multivariate Parameters*

1) For $K > 1$ dimension θ , we could use some multivariate measure of distance, such as the Euclidian norm.

The parameter vector $\theta_{t+1} = [\theta_{1,t+1}, \theta_{2,t+1}, \dots, \theta_{K,t+1}]$ is the accepted vector of parameters if

$$\|\theta_{t+1} - \theta_t\| = \sqrt{(\theta_{1,t+1} - \theta_{1,t})^2 + (\theta_{2,t+1} - \theta_{2,t})^2 + \dots + (\theta_{K,t+1} - \theta_{K,t})^2} < \epsilon,$$

2) Or, our stopping criteria could be that a scalar stopping criteria for each parameter must be met.

$\theta_{t+1} = [\theta_{1,t+1}, \theta_{2,t+1}, \dots, \theta_{K,t+1}]$ is the accepted vector of parameters if for all k

$$|\theta_{k,t+1} - \theta_{k,t}| < \epsilon_k$$

iii) *Criteria using the Objective Function*

We could also include a stopping criteria for the objective function written in the same way as that for the parameters. θ_{t+1} is the accepted parameter if

$$|f(\theta_{t+1}) - f(\theta_t)| < \epsilon,$$

where $\epsilon > 0$ is a stopping parameter.

iv) *Criteria using Derivatives*

The convergence criteria could involve checking the first and second order conditions at the current trial parameters. θ_{t+1} is the accepted parameters if

$$|f'(\theta_{t+1})| < \epsilon,$$

where $f'(\cdot)$ is the first derivative evaluated at the candidate parameter θ_{t+1} , and $\epsilon > 0$ is a stopping parameter. Derivative based criteria are typically only used for derivative based optimization algorithms, but there is no reason it couldn't be used for other algorithms.

1.6.3 3 Types of Optimization Algorithms

1) *Gradient based algorithms* use derivatives of $f(\theta)$ to pick the succession of parameter points to evaluate. These are more called Newton algorithms.

2) *Non-gradient based algorithms* do not require the computation of derivatives. The leading example here is the Nelder-Mead simplex method.

3) *Stochastic search algorithms* use random draws to pick points on the parameter space to evaluate. These methods are relatively recent. The two major examples are simulated annealing and genetic algorithms.

1.6.4 Choosing an Algorithm

Which type of algorithm works best depends on the type of problem. There is less “science” than “art” involved in choosing an optimization algorithm. In general, without a closed form representation of the estimator, we cannot say that evaluating a finite number of parameters within the parameter space guarantees we find the global minimum. Therefore, choosing among optimization algorithms is typically based on practical considerations, which have no formal justifications.

There are four basic issues one needs to consider when choosing an optimization algorithm.

1) *Speed*

One issue is the time it takes the algorithm to find a set of parameters which satisfy the stopping criteria. Often the major determinant of computation time is the number of function evaluations required by the algorithm to find a local minimum. Gradient based methods generally involve fewer function evaluations than the other methods.

2) *Robustness to Non-Smoothness*

A second issue is the robustness of the algorithm to non-smoothness in the objective function. Gradient based methods typically perform poorly if the objective function is not smooth. Non-smoothness may lead the algorithm to stop prematurely as it gets stuck in a kink in the objective function.

Non-gradient based methods and stochastic search methods perform better with non-smooth objective functions.

3) *Robustness to Starting Parameters*

Gradient based methods are typically faster than the other two methods in leading to a local minimum. However, generally, it is believed that these methods are less robust to starting parameters. That is, a gradient based algorithm started at different starting parameters are more likely to lead to a different local minimum. One simple way to make any algorithm somewhat robust to starting parameters is to start the algorithm at several non-trivially different sets of parameters. Stochastic search algorithms are intended to be more robust to starting parameters since there is some probability that the algorithm searches over all points on the parameter space.

4) *Ease of Programming*

An often overlooked practical consideration is the amount of the programming or code writing one has to do for particular algorithms. For the more common Newton and simplex algorithms, code already exists. One has to balance the costs to creating specialized algorithms tailored to specific objective functions versus the gains from this specialization.

1.7 Gradient Based Optimization Algorithms

The Train (2001) textbook has the clearest description of gradient based methods.

1.7.1 Basic Form

Gradient based algorithms have this basic form for choosing the next trial parameter vector:

$$\theta_{t+1} = \theta_t - \gamma_t d_t$$

where θ_t is $K \times 1$, γ_t is the $K \times 1$ step length, and d_t is the $K \times 1$ step direction. d_t is some function of first and second derivatives. Different methods have different values for γ_t and d_t .

1.7.2 Steepest Descent

The simplest gradient based method sets $\gamma_t = 1$ and uses the gradient as the search direction $d_t = G(\theta_t)$. Each new candidate parameter vector θ_{t+1} is defined as

$$\theta_{t+1} = \theta_t - G(\theta_t)$$

where the $K \times 1$ gradient is $G(\theta_t)$.

$$G(\theta_t) = \left[\frac{\partial f(\theta)}{\partial \theta_1} \Big|_{\theta_{1t}}, \frac{\partial f(\theta)}{\partial \theta_2} \Big|_{\theta_{2t}}, \dots, \frac{\partial f(\theta)}{\partial \theta_K} \Big|_{\theta_{Kt}} \right],$$

where each element is the partial derivative of the objective function with respect to each parameter θ_k evaluated at the current trial parameter θ_{tk} .

For scalar θ , this is simply

$$\theta_{t+1} = \theta_t - f'(\theta_t)$$

Although the name “steepest descent” suggests some sort of optimality of this method, this method is generally slow to converge. See the discussion in the Train (2003) textbook.

1.7.3 The gradient provides the following information

i) The sign of $f'(\theta_t)$ indicates whether the θ should be increased or decreased in order to decrease the objective function.

ii) If we are near a local minima, $f'(\theta_t)$ should be close to zero, and a small step is taken. If we are far from a local minima, $f'(\theta_t)$ is large, and a large step is taken.

1.7.4 Steps for Steepest Descent

For the scalar case, the basic steepest descent algorithm uses these steps:

Step 1: Choose starting parameter θ_0 .

Step 2: Evaluate gradient at θ_0 . $G(\theta_0)$ can be calculated analytically if a closed form for $G(\theta)$ is available, or calculated using finite difference numerical derivative methods.

Step 3: Calculate next candidate parameter.

$$\theta_1 = \theta_0 - G(\theta_0)$$

Step 4: Check convergence criteria.

At this point, we evaluate whatever convergence or stopping criteria we have specified (see above). For example, a convergence criteria could be to stop the algorithm and accept candidate parameter θ_{t+1} if

$$\frac{|\theta_{t+1} - \theta_t|}{|\theta_t|} < \epsilon,$$

where $\epsilon > 0$ is some small number.

We calculate

$$\frac{|\theta_1 - \theta_0|}{|\theta_1|} < \epsilon,$$

If θ_1 satisfies the convergence criteria, we stop. If not, the algorithm continues and repeats Step 2, calculating $G(\theta_1)$ and $\theta_2 = \theta_1 - G(\theta_1)$, and so on.

Notice that as we approach the solution $\theta^* = \arg \min f(\theta)$, $G(\theta)$ approaches zero. At the solution, the convergence criteria for any positive ϵ is satisfied.

If $\theta_t = \theta^*$, then

$$\theta_{t+1} = \theta^t - G(\theta^*) = \theta^t = \theta^*$$

and

$$\frac{|\theta_{t+1} - \theta_t|}{|\theta_t|} = 0 < \epsilon,$$

1.7.5 Newton-Raphson

The Newton-Raphson algorithm is based on a second order polynomial approximation to the objective function. For this method, $\gamma_t = 1$ and

$$d_t = H(\theta_t)^{-1}G(\theta_t),$$

where $H(\theta_t)$ is the $K \times K$ Hessian matrix evaluated at the current trial parameters and $G(\theta_t)$ is the $K \times 1$ gradient vector defined above.

For scalar θ ,

$$d_t = \frac{f'(\theta)}{f''(\theta)}$$

As with the steepest descent method, Newton-Raphson uses gradient in-

formation to indicate the direction of search. The Newton-Raphson algorithm also uses second derivative information. The second derivative helps determine how big the next step should be. If the curvature is large ($f''(\theta)$ is large), we are near a local minima, and a small step ($\frac{1}{f''(\theta)}$) is taken. If the curvature is small, we are far away from a local minima, and a large step is taken.

1.7.6 Deriving Newton-Raphson

Let's look at the derivation of the Newton-Raphson algorithm. Assume θ is scalar and the objective function is a second order polynomial:

$$f(\theta) = a - b\theta + c\theta^2$$

The minimum is

$$-b + 2c\theta = 0$$

$$\theta^* = \frac{b}{2c}$$

We know that this point is the global minimum by checking the second order condition: $2c(\frac{b^2}{2c}) = b^2 > 0$.

Now let's use the Newton-Raphson algorithm to solve the problem.

Step 1: Choose any starting parameter θ_0 .

Step 2: Choose Next Candidate Parameter

The Newton-Raphson algorithm says that the next candidate parameter is

$$\theta_1 = \theta_0 - H(\theta_0)^{-1}G(\theta_0)$$

$$H(\theta_0) = 2c$$

$$G(\theta_0) = -b + 2c\theta_0$$

Substituting,

$$\theta_1 = \theta_0 - \frac{-b + 2c\theta_0}{2c}$$

$$\theta_1 = \frac{b}{2c}$$

This is the minimum of the function we found above. However if we don't necessarily know this is the minimum, we would check the convergence algorithm.

Step 3: Check Convergence Criteria

Depending on our θ_0 value and our convergence criteria, we might stop at this point. Let's assume θ_1 does not satisfy our stopping criteria, and we repeat Step 2 again.

Step 2: Choose Next Candidate Parameter

$$\theta_2 = \theta_1 - H(\theta_1)^{-1}G(\theta_1)$$

$$G(\theta_1) = -b + 2c\theta_1 = -b + 2c\frac{b}{2c} = 0$$

$$\theta_2 = \theta_1$$

Step 3: Check Convergence Criteria

Since the gradient is zero at θ_1 , θ_2 would satisfy whatever reasonable convergence criteria you would choose. Therefore $\theta^* = \theta_1 = \frac{b}{2c}$, which is the global minimum.

The Newton-Raphson algorithm finds the minimum point of a quadratic function in one step. For other types of objective functions, the Newton-Raphson algorithm may take more steps or not find the global minimum at all.

1.7.7 An Example of Newton-Raphson

Assume our objective function is

$$\theta^* = \arg \min_{\theta \in [0, \infty]} -2\theta^2 + \frac{1}{3}\theta^3$$

(Graph: $f(0) = 0$; in $0 < \theta < 6$, $f(\theta) < 0$; $f(6) = 0$; $\theta > 6$, $f(\theta) > 0$)

Note: The objective function has a global minimum at $\theta^* = -\infty$. We have constrained the parameter space to give us a local minimum.

We can analytically solve for the estimator

$$-4\theta + \theta^2 = 0$$

$$-4 + \theta = 0$$

$$\theta^* = 4$$

Now let's use the Newton-Raphson algorithm to solve the optimization problem.

Our stopping criteria is the following: $\hat{\theta}^* = \theta_t$ if

$$\left| \frac{\theta_t - \theta_{t-1}}{\theta_{t-1}} \right| < 0.001$$

Otherwise, we continue the algorithm.

To compute the algorithm, we'll need these first and second derivatives:

$$G(\theta) = f'(\theta) = -4\theta + \theta^2$$

$$H(\theta) = f''(\theta) = -4 + 2\theta$$

Iteration 1

Our starting value is $\theta_0 = 3$.

$$G(3) = -3$$

$$H(3) = 2$$

$$\theta_1 = 3 - (-3) * \left(\frac{1}{2}\right) = 3 + 3/2 = 4.5$$

Check stopping criteria:

$$\left|\frac{4.5 - 3}{3}\right| = 0.5 > 0.001$$

We continue the algorithm.

Notice that we actually overshot the minimum point of 4. A line search step, discussed below, might help us get closer to the minimum in fewer steps.

Iteration 2

$$G(9/2) = 2.25$$

$$H(9/2) = 5$$

$$\theta_2 = 4.5 - (2.25) * \frac{1}{5} = 4.05$$

Since we overshot the local minimum point in the last step, the sign of $G(\theta_1)$ is now positive, whereas in the last step it was negative.

Check stopping criteria:

$$\left| \frac{4.05 - 4.5}{4.5} \right| = 0.1 > 0.001$$

We continue the algorithm.

Iteration 3

$$G(4.05) = 0.2025$$

$$H(4.05) = 4.1$$

$$\theta_3 = 4.05 - (0.2025) * \frac{1}{4.1} = 4.0006$$

Check stopping criteria:

$$\left| \frac{4.0006 - 4.05}{4.05} \right| = 0.012195 > 0.001$$

We continue the algorithm.

Iteration 4

$$G(4.0006) = 0.0024394$$

$$H(4.0006) = 4.0012$$

$$\theta_4 = 4.0006 - (0.0024394) * \frac{1}{4.0012} = 4$$

Check stopping criteria:

$$\left| \frac{4 - 4.0006}{4.0006} \right| = 0.00015239 < 0.001$$

The stopping criteria has been met. $\hat{\theta}^* = \theta_4 = 4$.

1.7.8 Refinements on Newton-Raphson: BHHH

With the Newton-Raphson method, we need to calculate a Hessian matrix at each vector of trial parameters. This can be computationally expensive, especially (as is common) if we are calculating these matrices using numerical methods. In addition, the computed Hessian is often ill-behaved (e.g. it

might be singular).

The BHHH method (Berndt, Hall, Hall, and Hausman) use the outer-product of the gradient vectors to replace the computation of the Hessian in the Newton-Raphson:

$$H(\theta_t) = - \sum_{i=1}^N g_i(\theta_t) g_i(\theta_t)',$$

where the objective function is

$$f(\theta) = \sum_{i=1}^N f_i(\theta)$$

and

$$g_i(\theta) = \frac{\partial f_i(\theta)}{\partial \theta}$$

This is based on information matrix equality for MLE, where at the true parameter θ_0 , the following holds for an i.i.d. sample of size N :

$$E\left[\frac{\partial^2 \ln L(\theta_0)}{\partial \theta^2}\right] = - \sum_{i=1}^N E\left[\frac{\partial \ln L_i(\theta_0)}{\partial \theta} \frac{\partial \ln L_i(\theta_0)}{\partial \theta'}\right]$$

where $\ln L_i(\theta)$ is the log likelihood function evaluated at θ for some i , and $\ln L(\theta) = \sum_{i=1}^N \ln L_i(\theta)$.

The advantage to BHHH is that only first derivatives need to be calculated at each step in the estimation algorithm. Although the justification for this

method is the information matrix equality for MLE, it could be argued that it is a suitable approximation for other extremum estimators.

1.7.9 Refinements on Newton-Raphson: DFP

The DFP method (Davidson, Fletcher, and Powell) uses a more complex algorithm to avoid computing the Hessian matrix.

Their algorithm uses the iteration

$$\theta_{t+1} = \theta_t + A_t G_t,$$

where A_t and G_t is shorthand for $A(\theta_t)$ and $G(\theta_t)$,

$$A_t = A_{t-1} + \frac{p_{t-1} p'_{t-1}}{p'_{t-1} q_t} - \frac{A_{t-1} q_t q'_t A_{t-1}}{q'_t A_{t-1} q_t},$$

$$q_t = G_t - G_{t-1},$$

$$p_{t-1} = -\gamma_{t-1} A_{t-1} G_{t-1},$$

and A_0 is positive definite (e.g. $A_0 = I$).

This algorithm is motivated by the fact that after successive iterations ($t \rightarrow \infty$), $q_\infty = p_\infty = 0$, and $A_\infty = -H_\infty^{-1}$. That is, the algorithm converges

to the Newton-Raphson algorithm where the Hessian is explicitly calculated.

1.7.10 Refinements on Newton-Raphson: Line Search

An additional refinement on Newton-Raphson is to include a line search at each step in the algorithm. A line search chooses the step length γ_t optimally. In Newton-Raphson, $\gamma_t = 1$ at each step. This step length may be too large or too small. For example, we might over-shoot a local minima if the step length is too large.

To use a line search, we first compute the step direction $d_t = H(\theta_t)^{-1}G(\theta_t)$. Then we find the optimal step length for each parameter as

$$\gamma_t^* = \min_{\gamma_t > 0} f(\theta_t + \gamma_t d_t)$$

Notice that this involves another optimization problem. However, this problem is only one-dimensional (for each parameter), and does not require re-calculating the d_t , which is the more computationally intensive step. The line search step is usually thought to decrease the number of function evaluations needed relative to the fixed step length algorithm. In addition, the line search forces each successive vector of trial parameters to non-trivially decrease the objective function, which may not be the case with Newton-Raphson with $\gamma_t = 1$.

One simple way to solve the line search problem would be to use a simple grid search over values of γ_t . More complicated solutions involve using the

Newton-Raphson algorithm to find the optimal step length. At each step t , we start with an initial guess of the step length (γ_0), and then iterate on the one-dimensional step length until a convergence criteria is met. The iteration algorithm is

$$\gamma_{k+1} = \gamma_k - \frac{f'(\gamma_k)}{f''(\gamma_k)}$$

1.8 Simplex Method

The simplex method (Nelder-Mead 1967), also known as the “downhill simplex” or a “polytope” method, is a non-gradient based optimization algorithm which has the advantage that no derivatives are required to be calculated. Assume the parameter vector θ is of dimension J . To make the notation easier, denote the $J \times 1$ vector of starting values as A_0 .

$$A_0 = [\theta_1, \theta_2, \dots, \theta_J]$$

1.8.1 NM Simplex Steps

Step 1a: *Create the Initial Simplex*

An initial simplex is denoted by its $J + 1$ vertices: A_0, A_1, \dots, A_J .

$$A_0 = [\theta_1, \theta_2, \dots, \theta_J]$$

$$\begin{aligned}A_1 &= [\theta_1 + s_1, \theta_2, \dots, \theta_J] \\A_2 &= [\theta_1, \theta_2 + s_2, \dots, \theta_J] \\&\vdots \\A_J &= [\theta_1, \theta_2, \dots, \theta_J + s_J],\end{aligned}$$

where s_1, s_2, \dots, s_J are the initial step sizes.

In Matlab (*fminsearch* command), the step sizes in the initial simplex are in terms of percentages of the parameter. That is, $s_k = \delta\theta_k$ for all k . $\delta \neq 0$ is a parameter that controls the deviation of the other simplex vertices from the starting values. In Matlab, the *fminsearch* simplex algorithm sets $\delta = 0.05$ (5 percent deviation) for non-zero starting parameters. If $\theta_k = 0$, Matlab sets the vertex point to 0.00025. There is no clear reason why other values could not be used.

For one parameter (θ_1), the simplex is a line segment. The two vertices are

$$[\theta_1, s_1 + \theta_1]$$

For two parameters (θ_1, θ_2), the simplex is a triangle. The three vertices of the triangle are

$$[(\theta_1, \theta_2), (s_1 + \theta_1, \theta_2), (\theta_1, s_2 + \theta_2)]$$

Step 1b: *Order the Simplex Vertices*

The NM simplex calculates the objective function at each of the $J + 1$ simplex points: $f(A_0), f(A_1), \dots, f(A_J)$. Without loss of generality, re-order the points on the initial simplex as

$$f(A_0) < f(A_1) < \dots < f(A_J).$$

A_0 is the best point, and A_J is the worst point.

Step 2: *Calculate the Reflection Point*

The NM algorithm replaces the worst point on the simplex A_J with another point which has a lower objective function evaluation. For ease of exposition, we say that the point A_j is an “improvement” over or “better” than the point A_k if $f(A_j) < f(A_k)$. The first candidate improvement point is the reflection point. The NM simplex calculates the reflection point A_J^R as the reflection of the worst point, A_J , through the centroid M of the remaining points, A_0, A_1, \dots, A_{J-1} . The centroid is

$$M = \frac{1}{J} \sum_{j=0}^{J-1} A_j$$

The reflection point is then

$$A_J^R = M + \alpha(M - A_J),$$

where $\alpha > 0$ is an algorithm parameter. The default value is typically $\alpha = 1$.

Step 3: Update the Simplex

We next evaluate the objective function at the reflection point to obtain $f(A_J^R)$. There are three cases depending on the objective function value at the reflection point relative to the previously calculated values of the objective function at the other points on the existing simplex.

Case 1:

If A_J^R is an improvement over the initial best point A_0 , then we continue to move in the same direction by calculating the expansion point A_J^E

$$A_J^E = A_J^R + \gamma(A_J^R - M),$$

where $\gamma > 0$ is an algorithm parameter, typically $\gamma = 1$.

If A_J^E is an improvement over A_0 , then A_J is replaced by A_J^E . The new simplex, including the expansion point, is re-ordered, and we return to Step 2. If A_J^E is not an improvement over A_0 , then A_J is replaced by A_J^R , and we re-order the simplex and return to Step 2.

Case 2:

If A_J^R is not an improvement over A_0 , but A_J^R is better than the next worst point A_{J-1} , then A_J is replaced by A_J^R , and we re-order the simplex

and return to Step 2.

Case 3:

If A_J^R is not an improvement over A_0 , and worse than the next worst point A_{J-1} , then we calculate the contraction point A_J^C as

$$A_J^C = M + \beta(\tilde{A}_J - M),$$

where $\tilde{A}_J = A_J^R$ if $f(A_J^R) < f(A_J)$, and $\tilde{A}_J = A_J$ otherwise. $0 < \beta < 1$ is an algorithm parameter, typically $\beta = 1/2$.

If A_J^C is an improvement over A_J , then A_J is replaced by A_J^C , and we re-order the simplex and return to Step 2.

Shrink

If A_J^C is not an improvement over A_J , then we shrink the entire simplex toward the best point A_0 . The new simplex is defined by these $J + 1$ points

$$[A_0, (\tau A_0 + (1-\tau)A_1), (\tau A_0 + (1-\tau)A_2), \dots, (\tau A_0 + (1-\tau)A_{J-1}), (\tau A_0 + (1-\tau)\tilde{A}_J)],$$

where $0 < \tau < 1$ is an algorithm parameter, typically $\tau = 1/2$. Using this new simplex, we return to Step 2.

1.8.2 Alternative Simplex Algorithms

There is no one type of simplex algorithm. The simplex algorithm described above is modeled on the original Nelder-Meade (1965). The Judd (1998) textbook describes a simplex algorithm in which only the reflection point is calculated. If the reflection point is an improvement, we form the simplex using this point in place of the previous worst point. We then re-order the points and calculate a new reflection point. If the reflection point is not an improvement, then we conduct the shrink procedure.

Matlab

The Matlab command *fminsearch* has a different sequence of steps. Steps 1 and 2 are basically the same. For Step 3, Cases 1 and 2 are basically the same. For Case 3, Matlab computes potentially two contraction points: the contract inside point, which is like the contraction point described above, and a contract outside point.

If $f(A_J^R) > f(A_J)$, then we compute the Contract Inside point as

$$A_J^{CI} = M + \psi(A_J - M),$$

where $0 < \psi < 1$ is an algorithm parameter, $\psi = 0.5$ in Matlab.

If $f(A_J^R) < f(A_J)$, then we compute the Contract Outside point as

$$A_J^{C0} = M + \psi\alpha(M - A_J),$$

We can see these different points (reflection, expansion, contract inside, contract outside) as a type of line search along a line through the worst point on the simplex. There is no reason why additional points other than these points could not be calculated along this line or along lines through other points.

1.8.3 Simplex Example

Let's consider a simple example. $J = 2$. We want to solve this problem:

$$\min_{\theta_1, \theta_2} \theta_1^2 + \theta_2^2$$

The true minimum is $\theta_1^* = 0$ and $\theta_2^* = 0$.

Our starting values are $\theta_0 = [1/2, 1/2]$.

As a convergence criteria, I use the criteria from Matlab's `fminsearch` command.

There are two criteria which need to be satisfied:

Criterion 1: The "diameter" of the simplex must be less than $\epsilon_1 > 0$

Stop and accept best parameter vector on simplex as $\hat{\theta}^*$ if

$$\epsilon_1 > \max(\max(| [A_0, A_0] - [A_1, A_2] |))$$

Criterion 2: The absolute value of the function evaluations across all points on the simplex must be less than $\epsilon_2 > 0$.

Stop and accept best parameter vector on simplex $\hat{\theta}^*$ if

$$\epsilon_2 > \max(| [f(A_0), f(A_0)] - [f(A_1), f(A_2)] |)$$

I set $\epsilon_1 = 0.001$ and $\epsilon_2 = 0.001$.

Iteration 1

Step 1

Using step sizes of $1/2$, the initial simplex is given by these three vertices:

$$[A_0, A_1, A_2] = [(1/2, 1/2), (1, 1/2), (1/2, 1)]$$

This forms a triangle. Let's evaluate the function at each of the points on the simplex.

$$f(A_0) = 1/2, f(A_1) = 1.25, f(A_2) = 1.25$$

These points are ordered from best to worst. Ignore the tie and let A_2 be the worst point.

Step 2

The centroid is

$$M = 1/2(A_0 + A_1) = (3/4, 1/2)$$

The reflection point is

$$\begin{aligned} A_2^R &= M + \alpha(M - A_2) \\ &= (3/4, 1/2) + 1[(3/4, 1/2) - (1/2, 1)] = (3/4, 1/2) + (1/4, -1/2) \\ &= (1, 0) \end{aligned}$$

The function evaluation at the reflection point is $f(A_2^R) = 1$.

Step 3

Case 1 doesn't apply, since the reflection point is not an improvement over the best point.

Case 2 applies since the reflection point is better than the worst point, but better than the next to worst point A_1 .

We therefore replace A_2 with A_2^R .

New simplex is

$$[(1/2, 1/2), (1, 1/2), (1, 0)]$$

Re-order:

$$[A_0, A_1, A_2] = [(1/2, 1/2), (1, 0), (1, 1/2)]$$

Check Convergence Criteria

Criterion 1: The “diameter” of the simplex must be less than $\epsilon_1 > 0$

$$\max(\max(|[A_0, A_0] - [A_1, A_2]|))$$

$$A_0 - A_1 = (1/2, 1/2) - (1, 0) = (-3/4, 1/2)$$

$$A_0 - A_2 = (1/2, 1/2) - (1, 1/2) = (-1/2, 0)$$

$$\max(\max(|A_0 - [A_1, A_2]|)) = 3/4 > 0.001$$

Convergence criterion 1 has not been met. We continue the algorithm.

Criterion 2: The absolute value of the function evaluations across all points on the simplex must be less than $\epsilon_2 > 0$.

$$\max(|f(A_0) - [f(A_1), f(A_2)]|)$$

$$f(A_0) = 1/2, f(A_1) = 1, f(A_2) = 1.25$$

$$f(A_0) - f(A_1) = -1/2$$

$$f(A_0) - f(A_2) = -3/4$$

$$\max(|f(A_0) - [f(A_1), f(A_2)]|) = 3/4 > 0.001$$

Convergence criterion 2 has not been met. We continue the algorithm.

Iteration 2

Step 1

New simplex

$$[A_0, A_1, A_2] = [(1/2, 1/2), (1, 0), (1, 1/2)]$$

Step 2

Centroid is

$$M = 1/2(A_0 + A_1) = (3/4, 1/4)$$

The reflection point is

$$\begin{aligned}A_2^R &= M + \alpha(M - A_2) \\ &= (3/4, 1/2) + 1[(3/4, 1/2) - (1, 1/2)] = (0.5, 0)\end{aligned}$$

The function evaluation at the reflection point is $f(A_2^R) = 0.25$.

Step 3

Case 1 applies since this the reflection point is better than the best point.

Calculate expansion point:

$$\begin{aligned}A_2^E &= A_2^R + \gamma(A_2^R - M), \\ &= (0.5, 0) + 1[(0.5, 0) - (3/4, 1/2)] \\ &= (0.25, -0.25)\end{aligned}$$

$$f(A_2^E) = 0.125$$

Since the function evaluation is lower at the expansion point than at the reflection point, we replace A_2 with this point.

New simplex is

$$[(1/2, 1/2), (1, 0), (0.25, -0.25)]$$

As above, we check the convergence criteria.

Remaining Iterations

In my Matlab code, it took 24 iterations to satisfy the convergence criteria.

The simplex on the last iteration is

$$[A_0, A_1, A_2] = [(-0.0002, -0.0005), (-0.0012, 0.0003), (-0.0001, 0.0014)]$$

Function evaluation at these points:

$$[f(A_0), f(A_1), f(A_2)] = 0.00001 * [0.0270, 0.1519, 0.1880]$$

Algorithm approximation to minimum θ^{**}

$$\hat{\theta}^* = (-0.0002, -0.0005)$$

1.9 Simulated Annealing

Another non-gradient based optimization algorithm is called simulated annealing. This approach is referred to as a *stochastic search* algorithm.

1.9.1 Simulated Annealing Steps

Simulated annealing is an iterative algorithm. For simplicity, let's look at the procedure for an optimization problem with a scalar parameter θ . Extending this procedure to higher dimension θ is straight-forward.

STEP 1: *Choose Starting Parameters*

Starting parameters are θ_0 .

STEP 2: *Perturb the Parameters*

We first draw z_{11} from a standard normal distribution. See the next section for information on how to produce random numbers. Our first possible parameter is

$$\theta_{11} = \theta_0 + z_{11}\lambda,$$

where $\lambda > 0$ is a step length parameter defined by the researcher.

STEP 3: *Two Cases*

Case i) *Accept the Parameter*

There are two possibilities for this candidate parameter θ_{11}

We accept θ_{11} and move to Step 4 if

a) $f(\theta_{11}) < f(\theta_0)$

OR

b) $f(\theta_{11}) \geq f(\theta_0)$ but the percent increase in the objective function from $f(\theta_0)$ to $f(\theta_{11})$ is sufficiently small:

$$\frac{f(\theta_{11}) - f(\theta_0)}{|f(\theta_0)|} < Tc_{11}$$

where $T > 0$ is a “temperature” parameter, and c_{11} is a random draw from $U[0, 1]$. This criteria implies that we stochastically accept candidate parameters which increase the objective function with some probability determined by T .

Call the accepted parameter θ_1

Case ii) *Reject the Parameter*

If the parameter is not accepted, we return to Step 2, perturb the parameter again using a new draw, called z_{12} , form θ_{12} , and move to Step 3. We continue forming new parameters $\theta_{11}, \theta_{12}, \dots, \theta_{1R}$ based on draws $z_{11}, z_{12}, \dots, z_{1R}$ until we find an acceptable parameter.

STEP 4: *Evaluate a Stopping Criteria*

For each accepted candidate draw, $\theta_1, \theta_2, \dots$, we evaluate a stopping criteria. If the stopping criteria is met, we stop. If the stopping parameter is not met, we return to Step 2 and perturb the parameter θ_t to create θ_{t+1} .

1.9.2 Stochastic vs. Deterministic Algorithms

Notice that the simulated annealing algorithm has two stochastic elements. First, the next candidate parameters are chosen stochastically in Step 2. This is in contrast to the deterministic way sequences of parameters are chosen in the gradient based and simplex algorithms. Second, within the iterative algorithm itself, we stochastically accept some parameters and not others. Given these stochastic elements, this algorithm may be particularly slow to converge to an acceptable solution. However, the advantage is that this algorithm is somewhat more robust to starting parameters. This type of algorithm is particularly useful for objective functions with many local minima.

1.9.3 An Example of Simulated Annealing

Let's consider the same problem as we used for the Newton-Raphson example.

$$\theta^* = \arg \min_{\theta \geq 0} -2\theta^2 + \frac{1}{3}\theta^3,$$

where θ is a scalar.

(Graph: $f(0) = 0$; in $0 < \theta < 6$, $f(\theta) < 0$; $f(6) = 0$; $\theta > 6$, $f(\theta) > 0$)

The analytic solution is

$$\theta^* = 4$$

Unlike with the Newton-Raphson problem, I used a parameter constraint, restricting $\theta > 0$ (see below).

The restricted problem is

$$\min_{\alpha} f(\alpha)$$

$$\text{s.t. } \alpha > 0$$

The unrestricted problem is

$$\min_{\theta} f(\exp(\theta))$$

where $\alpha = \exp(\theta)$.

The solution to the constrained problem is then $\alpha^* = \ln \theta^*$.

I set $T = 0.01$ and $\lambda = 1$. My starting parameter is the same as with Newton-Raphson, $\theta_0 = 3$. My convergence criteria is $\hat{\theta}^* = \theta_t$ if

$$\frac{|(\theta_{t-1} - \theta_t)|}{|\theta_t|} < \epsilon$$

where $\epsilon = 0.00001$.

My Matlab code contains the algorithm and results. The algorithm approximation to the minimizing parameter is $\hat{\theta}^* = 3.975$. It took 47 total iterations to reach this point, and 1,298 total objective function evaluations.

1.10 Parameter Constraints

Typically, the optimization algorithm searches over an unrestricted parameter space $\theta \in (-\infty, \infty)$ for each parameter. (Although in practice, with good starting values, the algorithm should be searching over a much smaller range.) If the model we are using imposes restrictions on parameters, we can use transformations of the following forms.

1.10.1 General Form

Let α be the scalar model parameter which is restricted to the range $[\alpha_l, \alpha_c]$. θ is the optimization parameter.

The constrained objective function for the econometric model is

$$\hat{\alpha} = \arg \min f(\alpha)$$

s.t.

$$\alpha \in [\alpha_l, \alpha_u]$$

Using the mapping $\alpha = h(\theta)$, we can transform the constrained problem into an unconstrained problem as

$$\min_{\theta} f(h(\theta))$$

1.10.2 Types of Parameter Constraints

We next consider three different constraints and the appropriate $\alpha = h(\theta)$ mappings.

i) $\alpha \in [0, \infty]$

$$\alpha = \exp(\theta)$$

$$\theta = \log(\alpha)$$

Note that we can always write a model in terms of $-\alpha$, so we do not need a separate transformation for $\alpha \in [-\infty, 0]$.

Be careful not to search over very large values of θ (i.e. re-scale the model so θ is small). On my computer, $\exp(1000) = \exp(1001) = \infty$. The transformation is numerically accurate (on my computer) for about $\theta \in [-500, 500]$.

ii) $\alpha \in (-1, 1)$

$$\alpha = \tanh(\theta)$$

$$\theta = \tan(\alpha)$$

This is useful if one of the parameters is a correlation coefficient.

Be careful not to search over very large values of θ (i.e. re-scale the model so θ is small). On my computer, $\tanh(10) = \tanh(11) = 1$. This transformation is numerically accurate (on my computer) for about $\theta \in [-5, 5]$.

iii) $\alpha \in [0, 1]$

$$\alpha = \frac{(\pi/2) + \arctan(\theta)}{\pi}$$

$$\theta = \tan(\alpha\pi - \pi/2)$$

This is useful for parameters which represent probabilities.

1.11 Function Constraints

We next consider numerical methods to solve constrained optimization problems of the form:

$$\min_{\theta} f(\theta)$$

s.t.

$$g(\theta) = 0$$

$$h(\theta) \leq 0$$

where $f : R^K \rightarrow R$, $g : R^K \rightarrow R^M$, $h : R^K \rightarrow R^S$.

1.11.1 Common Sense Approach to Constrained Optimization

i) Since constrained optimization is more difficult, it should always be avoided if possible. Substitute constraints into the objective function if possible. If it is clear that some constraints would not bind for reasonable parameters, restrict the parameter space to include only those values which do not violate the constraints.

ii) First solve the unconstrained problem by ignoring the constraints. If the solution doesn't violate any of the constraints, you have found the constrained minimum.

iii) One approach to function constraints would be to adapt particular optimization algorithms for function constraints. For example, in the simulated annealing algorithm, we could reject parameter random draws which violate the constraint. For a simplex algorithm, we could force the algorithm to always replace any point on a simplex which violated a constraint.

1.11.2 Kuhn-Tucker Approach

Write the Lagrangian as

$$L(\theta, \mu, \lambda) = f(\theta) + \gamma'g(\theta) + \mu'h(\theta)$$

where γ is $M \times 1$, $g(\theta)$ is $M \times 1$, μ is $S \times 1$, $h(\theta)$ is $S \times 1$.

The Kuhn-Tucker theorem says that there is a constrained optimum θ^* , and multipliers γ^* and μ^* which solves this system:

1)

$$\frac{\partial L(\theta, \mu, \lambda)}{\partial \theta} = \frac{\partial f(\theta^*)}{\partial \theta} + \gamma^{*'} \frac{\partial g(\theta^*)}{\partial \theta} + \mu^{*'} \frac{\partial h(\theta^*)}{\partial \theta} = 0,$$

2)

$$\mu_i^* h_i(\theta^*) = 0,$$

where $i = 1, 2, \dots, S$, and μ_i^* and $h_i(\theta)$ are individual elements from their respective vectors.

3)

$$g(\theta^*) = 0_M$$

4)

$$h(\theta^*) \leq 0_S$$

5)

$$\mu^* \leq 0_M$$

The Kuhn-Tucker approach to solving a constrained optimization problem re-casts the problem as a system of non-linear equations. There are specialized methods for solving systems of non-linear equations (see Judd 1998). In this approach, we find μ , λ , and θ vectors which satisfy the system of equations. There may be several such θ vectors. In this case, we choose the θ vector with the smallest $f(\theta)$.

The basic drawback of this approach is that it requires calculating analytically the gradient of the objective function with respect to θ . This is not possible in general.

1.11.3 Penalty Function Approach

A far more commonly used and simpler approach to constrained optimization is based on transforming the constrained problem into an unconstrained one. The *penalty function* approach specifies a penalty for violating the constraint and uses this penalty function to specify an unconstrained problem.

1.11.4 Penalty Function Approach 1

The simplest penalty function problem replaces the constrained problem with the following unconstrained problem:

$$\min_{\theta} f(\theta)$$

where

$$f(\theta) = \begin{cases} P_1 & \text{if } g(\theta) \neq 0 \text{ or } h(\theta) > 0 \\ f(\theta) & \text{otherwise} \end{cases}$$

where $P_1 > 0$ is a penalty parameter, which indicate the “cost” to violating the objective function. For this approach to work, the penalty must be larger than the minimized value of the objective function: $P_0 > f(\theta^*)$.

1.11.5 Penalty Function Approach 2

Another penalty function approach is to write the constrained objective function as

$$\min_{\theta} f(\theta) + \frac{1}{2} P_2 \left[\sum_{i=1}^M (g_i(\theta))^2 + \sum_{j=1}^S (\max\{0, h_j - 0\})^2 \right]$$

where $P_2 > 0$ is a penalty function parameter.

The advantage of this second specification is that it is smooth in the parameter θ , whereas the first penalty function specification is non-smooth for values of θ which violate a constraint. In this specification, the penalty function specification approaches the constrained optimization problem when P_2 becomes large. At $P_2 = \infty$, the constrained problem and the penalty function problem yield the same solution. For small P_2 the penalty function problem and the constrained problem may not yield the same solution.

1.11.6 An Example of Penalty Functions

Let's consider the same problem as we used for the Newton-Raphson example.

$$\min_{\theta > 0} -2\theta^2 + \frac{1}{3}\theta^3,$$

Recall that the global minimum is $\theta = -\infty$. The local minimum for $\theta \in [0, \infty]$ is $\theta^* = 4$. This is the minimum point we found using the Newton-Raphson algorithm starting at $\theta_0 = 3$.

My Matlab code writes the problem in two ways:

i) Unconstrained (no penalty function)

$$\min_{\theta} -2\theta^2 + \frac{1}{3}\theta^3,$$

ii) Constrained (with penalty function)

$$\min_{\theta} -2\theta^2 + \frac{1}{3}\theta^3,$$

where

$$f(\theta) = \begin{cases} P_1 & \text{if } \theta < 0 \\ f(\theta) & \text{otherwise} \end{cases}$$

I set $P_1 = 100$. My code solves the model 8 ways:

1) $\theta_0 = 1$, no penalty function, NM simplex

Result: Converged after 40 objective function evaluations. Parameter estimate is $\hat{\theta}^* \approx 4$.

2) $\theta_0 = 1$, no penalty function, Newton-Raphson

Result: Converge after 6 objective function evaluations. Parameter estimate is $\hat{\theta}^* \approx 4$.

3) $\theta_0 = 1$, with penalty function, NM simplex

Result: Same as above. Parameter estimate is $\hat{\theta}^* \approx 4$.

4) $\theta_0 = 1$, with penalty function, Newton-Raphson

Result: Same as above. Parameter estimate is $\hat{\theta}^* \approx 4$.

Let's also look what would happen if we didn't start on the positive part of the parameter space: $\theta_0 = -1$.

5) $\theta_0 = -1$, no penalty function, NM simplex

Result: Failed to converge after 1000 objective function evaluations. Algorithm is moving toward $-\infty$.

6) $\theta_0 = -1$, no penalty function, Newton-Raphson

Result: Converged after 16 objective function evaluations. Parameter estimate is $\hat{\theta}^* = -2.973612273308435e + 015$

7) $\theta_0 = -1$, with penalty function, NM simplex

Result: Converged after 42 objective function evaluations. Parameter estimate is $\hat{\theta}^* \approx 4$.

8) $\theta_0 = -1$, with penalty function, Newton-Raphson

Result: Converged after 4 objective function evaluations. Parameter estimate is $\hat{\theta}^* = 0$.

1.12 Parallel Computing

Parallel computing uses a number of processors to solve a computation task by having each processor work simultaneously or in parallel on independent tasks. Typically parallel computing works in an iterative fashion by first having a master processor assign some task to several worker or slave processors (or nodes). Each worker processor then completes its assigned task and passes the results back to the master processor. The master processor then sends a second set of tasks to all of the processors, and so on.

1.12.1 A Simple Example

A simple way to use parallelization is the following. Assume we want to sum the elements of a $N \times 1$ vector, i.e. we want to calculate

$$x^* = \sum_i^N x_i$$

If we have $P < N$ processors, we could divide up the vector of x s into

P groups. Each processor simultaneously sums its N/P x s, and reports this group sum to the master processor. The master processor then sums the N/P group sums.

Assume we have have $P = 2$ processors and $N = 10$.

Processor 1 calculates

$$x_1^* = \sum_1^5 x_i$$

Simultaneously, Processor 2 calculates

$$x_2^* = \sum_6^{10} x_i$$

The master processor then calculates:

$$x^* = x_1^* + x_2^*$$

The advantage to parallelization is that it may take less time to sum five numbers (by worker processors) and then 2 numbers (by master processor) than having one processor sum 10 numbers.

1.12.2 Issues in Parallel Computing

Notice in our simple example, one has to decide how finely to divide up the x vector for each processor. This is a common issue in parallel computing: fine grained vs. course grained parallelization. The benefit of fine grained

parallelization is that more processor are working simultaneously. The disadvantage to fine grained parallelization is that more communication is typically required which may take more computer time. In addition, if were to use more processors, the master processor typically would have to complete more calculations in using all of the output of the various processors.

1.12.3 Where Parallelization Can Be Useful in Economics

1) *Numerical Derivatives and Integration*

We could have each processor evaluate a function at a separate point in order to construct numerical derivatives. The master processor would perform the final calculation for the numerical derivative.

Similarly, numerical integration (quadrature) could use parallel processing by having each processor evaluate the integrand at a separate quadrature point or node. The master processor would add up and weight each of these integrand evaluations.

2) *Simulation Methods or Bootstrapping*

We could use parallel processing for simulation methods in the following fashion. Say we want to bootstrap an estimator B times. We could have each $P < B$ processor compute B/P of the estimators. The master processor then would compute the bootstrapping statistic from the full sample of bootstrapped estimators.

3) *Objective Function Parallelization*

Many common objective functions could be parallelized. For example, if we are using maximum likelihood, the estimator is defined as

$$\hat{\theta} = \arg \min - \sum_i^N \log L_i(\theta, X_i, y_i)$$

where X_i and y_i are data, and sample observations are indexed by $i = 1, 2, \dots, N$.

For each trial parameter vector θ , each of the $P < N$ processors calculate the log-likelihood function at θ for an assigned subset of N/P observations. The master processor would then sum these log-likelihood values to compute the full sample likelihood.

1.12.4 Parameter Level Parallelization

Another way to use parallelization for optimization problems is to parallelize at the parameter level. Lee and Wiswall (2006) develop such an algorithm for the Nelder-Mead simplex method. Our parallel simplex algorithm assigns a separate vector of parameters to each processor corresponding to a point on a simplex. That is, instead of attempting to replace the one worst point on the simplex, we replace the P worst points. Each of the P processors then conduct the simplex search steps for an improved point (reflection, expansion, etc.), communicate the results, and a new simplex is formed.

The main advantage of parallelization at the parameter level is that the

objective function does not need to be broken down into independent tasks in order to take advantage of parallel computing. This advantage is two-fold.

i) First, for objective functions which cannot be broken down into independent tasks, and therefore parallelization is not possible at the objective function level, the parallel NM algorithm is still applicable. For example, objective functions based on models of interacting agents may not allow the objective function to be divided into separate tasks. Even if some parallelization of the objective function is possible, it may not be possible to break down the objective function into enough independent tasks to fully use all of the available processors. This is the case, for example, if the objective function is broken down by agent types, but the researcher has more processors available than agent types.

ii) Second, even in cases where parallelization at the objective function level is possible, our parallel NM simplex algorithm avoids the need to rewrite computer code for each version of a model and objective function or as more processors become available.