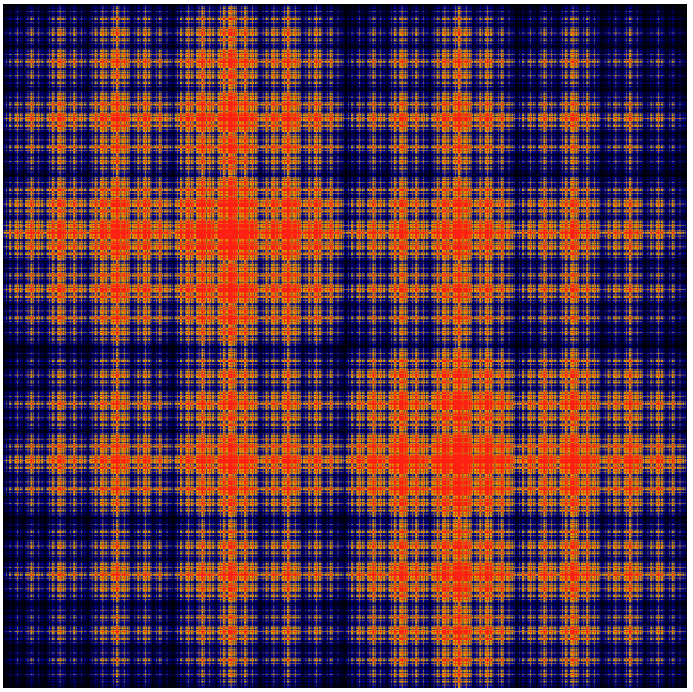
Applied Mathematics
and
Computing

Volume I



List of Contributors

J. Humpherys
Brigham Young University

J. Webb
Brigham Young University

R. Murray
Brigham Young University

J. West
University of Michigan

R. Grout
Brigham Young University

K. Finlinson
Brigham Young University

A. Zaitzeff
Brigham Young University

Preface

This lab manual is designed to accompany the textbook *Foundations of Applied Mathematics* by Dr. J. Humpherys.

Lab 23

Application: Newton's Method

Lesson Objective: *Understand Newton's Method*

One important technique in technical computing is Newton's method. The goal of Newton's method is to find x^* such that $f(x^*) = 0$. This method is especially important in optimization, where our goal is to find minima and maxima of functions. Newton's method is an iterative method (much like eigenvalue finders: remember how those provably have to be iterative?). Newton's method, in one dimension, is defined as follows:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Essentially Newton's method approximates a function by its tangent line, and then uses the zero of the tangent line as the next guess for x_n.

Newton's method is powerful because of the speed of convergence. In many cases Newton's method converges to the actual root quadratically, meaning that the error term is squared at every iteration. This fast convergence makes it a very powerful algorithm.

Newton's method does suffer from the flaw that its convergence is dependent upon an initial guess. If the initial guess is not sufficiently close the convergence can be much slower, or may never occur. There are even certain pathological functions for which newton's method will never converge. However, these functions are very rare, and as a rule Newton's method converges very quickly.

Problem 1 Write a Newton's method function that runs whether or not the user inputs a derivative function. If the user gives a derivative function, use that. Otherwise estimate the derivative numerically within the Newton's method function. In python this can be done by defining the derivative function as a keyword argument.

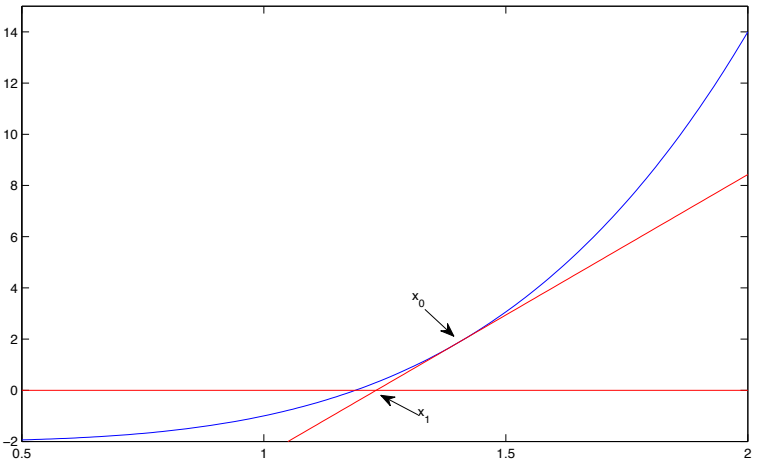


Figure 23.1: An illustration of how one iteration of Newton's method works

Your function definition will look something like this `def new_newton(f, x0, df=None, tol=0.001)`

Compare the performance of Newton's method when you input the derivative and when you don't. How well does each converge? Which runs faster? Try the following functions:

- $cos(x)$
- $sin(1/x) * x^2$
- $(sin(x)/x) - x$

Problem 2 Test your newton's method function on $x^{1/3}$. Use random starting points around zero. What do you see? Prove that for any non-zero starting point that newton's method will diverge.

Problem 3 A basin of attraction can be loosely defined as a set that will eventually converge to a specific root. Pick random points on the interval $[-2, 2]$ as starting

points and apply newton's method to the function $x^3 - 2x + 1/2$. Display the basins of attraction for this particular function. What do you observe?

Problem 4 Extend your Newton's method even further so that it will work on systems of equations. Suppose that $F : \mathbb{R}^n \to \mathbb{R}^n$. The relevant equation is

$$x_{i+1} = x_i - J^{-1}F(x_i)$$

Note that you **should not** calculate the inverse Jacobian. `sp.solve(A,b)` gives you the solution x to the equation $Ax = b$. Use this fact to calculate $J^{-1}F$ from J and F. You should be able to make this function work whether or not the user inputs a Jacobian. This also means that you will have to implement your own `jacobian` function.

BYU-MCL Boot Camp
Optimization Labs
Professor Richard W. Evans

1 Introduction

Well defined systems of equations often do not have analytical solutions. However, numerical solutions can also be difficult to find if the system is highly nonlinear, involves many equations, has singularities, involves inequality constraints, or some combination of these. The objective of these exercises is to familiarize you with some of the different root finding and optimization routines available through Python and specifically in the `scipy.optimize` library. You will learn some of the advantages and limitations of each one. You will come away from this understanding that numerical optimization involves a little bit of artistry and an intimate understanding of the theory behind the equations you are optimizing.

2 Root finding

Let $\mathbf{F}(\mathbf{x})$ be a system of m functions of the vector $\mathbf{x}$ of length n, and the function $\mathbf{F}(\mathbf{x})$ returns a vector of length m. The root of the function is the particular vector $\mathbf{x}$ such that $\mathbf{F}(\mathbf{x}) = \mathbf{0}$, where $\mathbf{0}$ is an m-length vector of zeros. Because many economic models are characterized by systems of equations, the roots of those systems are often the solutions to economic models.

If a system of equations $\mathbf{F}(\mathbf{x}) = \mathbf{0}$ is linear, it can be represented as $\mathbf{Ax} - \mathbf{b} = \mathbf{0}$ or $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ is an $m \times n$ matrix, $\mathbf{x}$ is an $n \times 1$ vector, $\mathbf{b}$ is an $m \times 1$ vector, and $\mathbf{0}$ is an $m \times 1$ vector of zeros. This is the classical linear algebra problem in which there may be many, exactly one, or no solution to the linear system $\mathbf{Ax} = \mathbf{b}$.

1. There is exactly one solution to $\mathbf{Ax} = \mathbf{b}$ if $\mathbf{A}$ is square ($n \times n$ or $m = n$) and has full rank, $\text{rank}(\mathbf{A}) = n$. This means $\mathbf{A}$ has n linearly independent rows and $\mathbf{A}$ is invertible.

2. There are multiple solutions to $\mathbf{Ax} = \mathbf{b}$ if $\text{rank}(\mathbf{A}) < n$.

3. There is no solution to $\mathbf{Ax} = \mathbf{b}$ if $\text{rank}(\mathbf{A}) > n$.

If the system $\mathbf{F}(\mathbf{x}) = \mathbf{0}$ is nonlinear, the process of finding the roots is much less straightforward. It often involves a Newton method and the evaluation of a gradient or matrix of derivatives (Jacobian matrix).

Use the following description of the 3-period-lived agent, perfect foresight OLG model from the first week of the course in order to complete Exercises 1 through 4 below. The decisions of the households in the economy can be summarized by the following equations.

$$(c_{1,t})^{-\gamma} - \beta(1 + r_{t+1} - \delta)(c_{2,t+1})^{-\gamma} = 0 \tag{1}$$
$$(c_{2,t})^{-\gamma} - \beta(1 + r_{t+1} - \delta)(c_{3,t+1})^{-\gamma} = 0 \tag{2}$$
$$c_{1,t} + k_{2,t+1} - w_t = 0 \tag{3}$$
$$c_{2,t} + k_{3,t+1} - w_t - (1 + r_t - \delta)k_{2,t} = 0 \tag{4}$$
$$c_{3,t} - (1 + r_t - \delta)k_{3,t} = 0 \tag{5}$$
$$w_t - (1 - \alpha)A\left(\frac{K_t}{L_t}\right)^{\alpha} = 0 \tag{6}$$
$$r_t - \alpha A\left(\frac{L_t}{K_t}\right)^{1-\alpha} = 0 \tag{7}$$
$$K_t - k_{2,t} - k_{3,t} = 0 \tag{8}$$
$$L_t - 2 = 0 \tag{9}$$

If we substitute equations (3) through (9) into (1) and (2) and look only at the steady-state, the steady-state values $\left(\bar{k}_2, \bar{k}_3\right)$ are characterized by the following two equations.

$$\begin{aligned} u'\Big(w(\bar{k}_2, \bar{k}_3) - \bar{k}_2\Big) - \beta\Big(1 + r(\bar{k}_2, \bar{k}_3) - \delta\Big) \times ... \\ u'\Big(w(\bar{k}_2, \bar{k}_3) + [1 + r(\bar{k}_2, \bar{k}_3) - \delta]\bar{k}_2 - \bar{k}_3\Big) = 0 \end{aligned} \tag{10}$$

$$\begin{aligned} u'\Big(w(\bar{k}_2, \bar{k}_3) + [1 + r(\bar{k}_2, \bar{k}_3) - \delta]\bar{k}_2 - \bar{k}_3\Big) - ... \\ \beta\Big(1 + r(\bar{k}_2, \bar{k}_3) - \delta\Big)u'\Big([1 + r(\bar{k}_2, \bar{k}_3) - \delta]\bar{k}_3\Big) = 0 \end{aligned} \tag{11}$$

Let the parameter vector values of the model be given by $\theta = [\beta, \gamma, \alpha, \delta, A] = [0.442, 3, 0.35, 0.6415, 1]$. In Exercises 1 and 2, set the tolerance in the solver to `xtol=1e-10`. Also, because some of the root finding methods in Exercise 2 do not allow you to pass extra arguments into the root finding function, you will need to define an anonymous function in addition to your steady-state distribution of capital function. `ksssolve` is the function that I have defined for my root finding algorithm to call to find the steady-state distribution of capital.

```
def ksssolve(kvec, params):
    ... # Define function to solve for steady-state distribution
    ... # of capital "kvec" given parameters vector "params"
```

For some of the root finders, like `fsolve` in Exercise 1, you can pass in many extra arguments through the `args=(params)` syntax. However, two of the root finders in Exercise 2 do not allow for extra arguments to be passed. So you'll have to write an anonymous function in Python to be an intermediate step between the root finding command and the `ksssolve` function.

```
zero_func = lambda x: ksssolve(x, params)
```

For Exercises 1 and 2, you can write a root finding function syntax in the following form,

```
import scipy.optimize as opt
...
kssvec = opt.[RootFinder](zero_func, kinit, [meth='method'], xtol=1e-10)
```

that calls the anonymous function `zero_func`, which passes both the guess for the distribution of capital `kinit` and the parameters `params` to the `ksssolve` function, regardless of whether `fsolve` or the particular method of `root` allows extra argument passing.

The last little examples of Python code I want you to use is the time library for clocking computation speeds of different solution methods as well as some output printing commands.

```
import scipy.optimize as opt
import time
...
# Put all parameter and function definition code outside of timer
...
start_time = time.time()
kssvec = opt.[RootFinder](zero_func, kinit, [meth='method'], xtol=1e-10)
elapsed = time.time() - start_time
```

In order to have your script print the output that you want, you can use the following code.

```
print('The SS capital levels and comp. times for fsolve are:')
print("k2ss_f = %.8f" % k2ss_f)
print("k3ss_f = %.8f" % k3ss_f)
print("Time Elapsed: %.6f seconds" % elapsed_f)
```

The `%` operator tells the `print` command that a string is going to be formatted. The `.8f` and `.6f` commands tell the `print` command that the string following the second `%` character is to be displayed as a floating point number with 8 and 6 decimal places, respectively, represented after the decimal.

Exercise 1. Solve for the steady-state distribution of capital savings $\left(\bar{k}_2, \bar{k}_3\right)$ from the three-period lived agent perfect foresight model described in (10) and (11) above using the `scipy.optimize.fsolve` command. Report the computed steady-state distribution of capital $\left(\bar{k}_2, \bar{k}_3\right)$ and how long (in seconds) it took to compute.

Exercise 2. Solve for the same OLG steady-state distribution of capital savings $\left(\bar{k}_2, \bar{k}_3\right)$ using the alternative root finding command `scipy.optimize.root` with each of the following methods options: `hybr`, `broyden1`, and `krylov`. One of these methods will not work. Report the computed steady-state distribution of capital $\left(\bar{k}_2, \bar{k}_3\right)$ for each method and how long (in seconds) each one took to compute.

Exercise 3. What is the biggest difference between the solution from Exercise 1 and different solution methods in Exercise 2? That is, what is the biggest $(\bar{k}_2, \bar{k}_3)_{\text{ex1}} - (\bar{k}_2, \bar{k}_3)_{\text{ex2i}}$ where $i = \{\texttt{hybr}, \texttt{broyden1}, \texttt{krylov}\}$ among methods that worked?

Exercise 4. Which method had the minimum computation time and which method had the maximum computation time among the method in Exercise 1 and the two methods that worked in Exercise 2? [NOTE: You will get different computation times each time you run this. But the relative computation speeds ordering will remain the same.]

Each solver uses different methods and different coding approaches and efficiencies to arrive at the solution. So it can sometimes be a significant task to find a solver that works. Once you find a class of solvers that works, you may face a tradeoff between robustness and computational speed. Another library of convex optimization functions for Python that wraps the BLAS, LAPACK, FFTW, UMFPACK, CHOLMOD, GLPK, DSDP5, and MOSEK routines is `cvxopt`.

3 Unconstrained optimization (minimization)

The characterizing equations or data generating process (DGP) in an economic model often come from some set of optimization problems. In the case of the OLG problem from the first chapter of the course and the infinite horizon, representative agent DSGE model from the second chapter, both households and firms are optimizing. Although some economic problems do involve minimization decisions (e.g., cost minimization, loss function minimization), most focus on some type of maximization. This begs the question of why the code libraries of all major programming languages have only minimizers and not maximizers. First, any maximization problem can be rewritten as a minimization problem. Second, a rewritten minimization problem that must converge to a finite number like zero is easier than a problem that can go to ∞ or $-\infty$.

The difference between a root finder and a minimizer is subtle but important. And the take away should be that a minimizer is more complex and less exact than a root finder, but more flexible. Let's use the general system of equations $\mathbf{F}(\mathbf{x}) = \mathbf{0}$ discussed in Section 2. A root finder is finding a solution $\mathbf{x}$ that delivers the zero vector $\mathbf{0}$. But a minimizer is finding the vector $\mathbf{x}$ that minimizes the scalar valued function $\mathbf{g}(\mathbf{x})$. Note that you often want $\mathbf{g}(\mathbf{x})$ to map $\mathbf{x}$ into the nonnegative real line so that the minimizing solution $\mathbf{x}$ sets $\mathbf{g}(\mathbf{x})$ as close to the scalar 0 as possible. One very common example for the nonnegative scalar valued function $\mathbf{g}(\mathbf{x})$ is the sum of the squares of the value of each individual equation in $\mathbf{F}(\mathbf{x})$: $\mathbf{g}(\mathbf{x}) = \sum_{i=1}^{m} \left[F_i(\mathbf{x})\right]^2$. This minimizer is a least squares solution technique.

Exercise 5. Solve for the steady-state distribution of capital savings $\left(\bar{k}_2, \bar{k}_3\right)$ from the three-period lived agent perfect foresight OLG model described in (10) and (11) in Section 2 above using the `scipy.optimize.fmin` minimizer command. Report your solution and computation time.

[TODO: Insert exercise that changes the initial values.]

Unconstrained optimization (minimization) computational routines obviously are the right choice when the range of the vector being chosen is unbounded. But unconstrained minimizers can also work well, as in Exercise 5, when the bounds are well known and theoretical conditions (Inada conditions) push the solution away from the boundary. Care must simply be taken to input good starting values.

Now we use the representative infinitely lived DSGE Brock and Mirman (1972) model with inelatically supplied labor of $l_t = 1$ in every period and known closed form solution for the equilibrium policy function from the second chapter to illustrate the value of a minimizer. I have characterized a very simple form of i.i.d. uncertainty for the firm's productivity shock in (19) to make computing the expectation in the household's problem easier, and the Brock and Mirman (1972) policy function is in equation (18). The decisions of the representative household and the representative firm in the economy can be summarized by the following equations.

$$(c_t)^{-\gamma} = \beta E\Big[(1 + r_{t+1} - \delta)(c_{t+1})^{-\gamma}\Big] \tag{12}$$

$$c_t = w_t + (1 + r_t - \delta)k_t - k_{t+1} \tag{13}$$

$$w_t = (1 - \alpha)e^{z_t}\left(\frac{K_t}{L_t}\right)^{\alpha} \tag{14}$$

$$r_t = \alpha e^{z_t}\left(\frac{L_t}{K_t}\right)^{1-\alpha} \tag{15}$$

$$K_t = k_t \tag{16}$$

$$L_t = 1 \tag{17}$$

$$k_{t+1} = \psi(k_t, z_t) = \alpha\beta e^{z_t} k_t^{\alpha} \tag{18}$$

$$\Pr(z_t = -0.5) = \frac{1}{2} \quad \text{and} \quad \Pr(z_t = 0.5) = \frac{1}{2} \tag{19}$$

If we substitute equations (13) through (18) into (12), then the equilibrium is characterized by a sequence of Euler equations holding in every period of time in which the expectations on the right-hand-side are formed using knowledge of the distribution of the productivity shock z_{t+1} as described in (19).

$$\Big[w(k_t) + \big(1 + r(k_t) - \delta\big)k_t - k_{t+1}\Big]^{-\gamma} = ...$$
$$\beta E\Big[\big(1 + r(k_{t+1}) - \delta\big)\Big(w(k_{t+1}) + \big[1 + r(k_{t+1}) - \delta\big]k_{t+1} - \psi(k_{t+1}, z_{t+1})\Big)^{-\gamma}\Big] \tag{20}$$

Now suppose that each period is a year, and you have 41 years of data on the capital stock in your economy $\{k_t\}_{t=1}^{41}$. Suppose that you thought that the data $\{k_t\}_{t=1}^{41}$

were generated by a process described in equations (12) through (19). An equivalent way of saying that is to suppose that you though the data were generated by the intertemporal Euler equation (20). But you do not know the value of the parameters of the model $[\beta, \delta, \gamma, \alpha]$. How might you estimate those parameters $[\beta, \delta, \gamma, \alpha]$ to make the model (20) best match the data $\{k_t\}_{t=1}^{41}$?

[TODO: Add GMM exercise.]

4 Constrained optimization (minimization)

[TODO: Add constrained optimization problem.]

References

Brock, W. A., and L. Mirman (1972): "Optimal Economic Growth and Uncertainty: the Discounted Case," *Journal of Economic Theory*, 4(3), 479–513.