Lecture4_RootFinding

November 21, 2017

1 Computational Methods in Economics

1.1 Lecture 4 - Root Finding

```
In [1]: # Author: Alex Schmitt (schmitt@ifo.de)

        import datetime
        print('Last update: ' + str(datetime.datetime.today()))
```

```
Last update: 2017-11-20 13:47:15.415443
```

1.2 Preliminaries

Import Modules

```
In [2]: import numpy as np

        import matplotlib.pyplot as plt
        %matplotlib inline
        import seaborn

        import scipy.optimize

        # import sys
        from importlib import reload
```

1.3 This Lecture

1.4 Introduction

A function $f(x)$ has a *root* (also called a *zero*) at x^* if $f(x^*) = 0$. Two cases are relevant:

- f can be a univariate scalar/real-valued function $f : \mathbb{R} \to \mathbb{R}$, i.e. both input and output are scalars, or both its range and its domain have a dimension of 1
- f can be a vector-valued function $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$, i.e. both its range and its domain have a dimension greater than 1. In this case, finding the roots of a vector-valued function is equivalent to *solving a system of nonlinear equations*.

The intermediate case of a multivariate scalar function $f : \mathbb{R}^n \to \mathbb{R}$ - its inputs are vectors, hence its domain has a dimension greater than 1 - is going to be important in the next lecture.

Finding the root(s) of a function is one of the most common computational problems in economics, often applied when looking for an equilibrium. In other words, an equilibrium is usually defined by a set of equations, as illustrated by the following example.

1.4.1 Example: Neoclassical Growth Model

We have seen the standard NGM already in the first lecture. Recall:

- Utility function:

$$u(c, h) = \frac{c^{1-\nu}}{1-\nu} - B\frac{h^{1+\eta}}{1+\eta} \tag{1}$$

with c denoting consumption and h labor supply.

- Production function:

$$f(k, h) = Ak^{\alpha}h^{1-\alpha} \tag{2}$$

with k denoting the capital stock, and A the productivity level.

- Resource Constraint:

$$k_{t+1} + c_t = f(k_t, h_t) + (1-\delta)k_t = Ak_t^{\alpha}h_t^{1-\alpha} + (1-\delta)k_t \tag{3}$$

- Planner's Problem:

$$\max_{\{c_t, k_{t+1}, h_t\}} \sum_{t=0}^{\infty} \beta^t u(c_t, h_t) \tag{4}$$

s.t. the resource constraint.

First-order conditions

(1) Euler equation

$$c^{-\nu} = \beta \left[(c')^{-\nu} (f_k(k', h') + 1 - \delta) \right] \tag{5}$$

(2) intratemporal optimality condition

$$Bh^{\eta} = c^{-\nu} f_h(k, h) \tag{6}$$

where I have used the notation $c = c_t$ and $c' = c_{t+1}$ (and analogous for k and h) for brevity.

Steady State In an equilibrium, the two first-order conditions, combined with the resource constraint, must hold in every period. We will get to how to solve for the full dynamic allocation later in this course. For now, let's consider the *steady state*, where all variables are constant over time, i.e. $c_t = c_{t+1} = c_s$ and so on. The Euler equation then can be simplied to:

$$1 = \beta \left[f_k(k_s, h_s) + 1 - \delta \right] \tag{7}$$

For the intratemporal optimality condition, use the resource constraint to substitute for consumption:

$$Bh_s^{\eta} = \left[f(k_s, h_s) - \delta k_s \right]^{-\nu} f_h(k_s, h_s) \tag{8}$$

This is a nonlinear system of two equations, with two unknown variables, k_s and h_s, which can be solved using the methods introduced below. We can define a vector-valued function $\mathbf{S}$ with

$$\mathbf{S}(k, h) = \begin{bmatrix} \beta \left[f_k(k, h) + 1 - \delta \right] - 1 \\ \left[f(k, h) - \delta k \right]^{-\nu} f_h(k, h) - Bh^{\eta} \end{bmatrix} \tag{9}$$

Finding the steady state of the model then requires finding a root of function $\mathbf{S}$, i.e. a vector (k_s, h_s) such that

$$\mathbf{S}(k_s, h_s) = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{10}$$

1.4.2 Some Definitions

- For a multivariate real-valued function $f : \mathbb{R}^n \to \mathbb{R}$, the vector consisting of the first derivatives is called the *gradient* (vector):

$$\nabla f(\mathbf{x}) = \begin{bmatrix} \partial f / \partial x_1 \\ \vdots \\ \partial f / \partial x_n \end{bmatrix} \tag{11}$$

where $\mathbf{x}$ is an n-by-1 vector.

- For a vector-valued function $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$, the *Jacobian* (i.e., the matrix of the first derivatives), is defined as

$$J(\mathbf{x}) = \begin{bmatrix} \partial f_1/\partial x_1 & \dots & \partial f_1/\partial x_n \\ \vdots & \ddots & \vdots \\ \partial f_n/\partial x_1 & \dots & \partial f_n/\partial x_n \end{bmatrix} \tag{12}$$

Using the gradient notation, we can also write this as

$$J(\mathbf{x}) = \begin{bmatrix} \nabla f_1(\mathbf{x})^T \\ \vdots \\ \nabla f_n(\mathbf{x})^T \end{bmatrix} \tag{13}$$

1.4.3 Taylor Series and Taylor's Formula

For a univariate function f that is n times continuously differentiable, a *Taylor series* or *Taylor approximation* around x_0 is given by:

$$f(x) \approx f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2}f''(x_0)(x - x_0)^2 + \dots + \frac{1}{n!}f^{(n)}(x_0)(x - x_0)^n \tag{14}$$

Closely related to this is *Taylor's Theorem*: if f is, for example, twice continuously differentiable in an interval that contains x and x_0, then

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2}f''(c)(x - x_0)^2 \tag{15}$$

for some number c between x and x_0.

For a multivariate function $f : \mathbb{R}^n \to \mathbb{R}$, we can state Taylor's Theorem in the following way: if f is twice continuously differentiable and $p \in \mathbb{R}^n$, we have that

$$f(\mathbf{x}_0 + \mathbf{p}) = f(\mathbf{x}_0) + \nabla f(\mathbf{x}_0)^T\mathbf{p} + \frac{1}{2}\mathbf{p}^T\nabla^2 f(\mathbf{x}_0 + t\mathbf{p})\mathbf{p} \tag{16}$$

for some $t \in (0, 1)$.

1.5 Bisection

The simplest way to compute the root of a continuous univariate real-valued function is the *bisection method*. While simple, bisection captures two important features of most root-finding and optimization methods: it is a *local* method and it is based on an *iterative procedure*.

The key idea behind the bisection method is based on the *Intermediate Value Theorem*: if f is continuous and defined on the interval $[a, b]$, and if $f(a)$ and $f(b)$ are distinct values, then f must assume all values in between. Since we are interested in where f assumes the value 0, we need $f(a)$ and $f(b)$ to have *different signs*.

```
In [3]: ## cp. figure
        #
        #
        #
        #
        #
```

```
#
#
#
#
#
#
#
#
#
```

The bisection method implements the following "pseudo-code":

(i) Start with two distinct values a and b, $a < b$, such that $f(a)$ and $f(b)$ are defined and have different signs, i.e. $f(a) \cdot f(b) < 0$. Moreover, specify a "tolerance level" tol which should be a very small number, e.g. 1e-8.

(ii) Compute the midpoint between a and b, $x = \frac{a+b}{2}$.

(iii) If $f(x)$ has the same sign as $f(a)$, replace the left endpoint of the interval with x, i.e. $a = x$.

(iv) If $f(x)$ has the same sign as $f(b)$, replace the right endpoint of the interval with x, i.e. $b = x$.

(v) Check the *stopping rule*: if the absolute value of $f(x)$ is less than the tolerance level, $|f(x)| < tol$, stop and report the solution at x. If not, go back to (ii) and repeat.

Note the following:

- Bisection is an *iterative procedure*: at the beginning of each iteration step, the interval $[a, b]$ contains a root of f. The interval is then divided ("bisected") into two subintervals of equal length. One of the two subintervals must contain the root, and hence have endpoints of different signs. This subinterval is taken as the interval $[a, b]$ used for the next iteration. This process continues until the function value of the midpoint x of the current interval is sufficiently close to 0.
- Moreover, bisection is a *local* method: it will not give you all the roots of a function, but only one of the roots (in case there are multiple roots) between a and b. A corollary of this is that the outcome of bisection (and of local methods in general) is sensitive to the starting point chosen by the user, here the values for a and b.
- The bisection method is robust in the sense that it will find a root in a known number of iterations, assuming the initial choices for a and b lead to different signs for $f(a)$ and $f(b)$. The obvious downside of bisection is that it only works for univariate functions. Moreover, it is usually slower than the other methods discussed below.

In this week's problem set, you will be asked to code up the bisection method. Of course, most programming languages already have in-built implementations (e.g. in SciPy: **scipy.optimize.bisect**, as discussed below), so writing your own function may seem a bit redundant, but will help you to get used to the inner workings of many of the algorithms used in scientific computing.

1.6 Function Iteration

We have started to talk about iterative methods at the end of last lecture. To recap, the basic idea of iterative methods is to generate a sequence of approximations to the object of interest, e.g. the solution to linear or nonlinear system of equations, following an iteration rule:

$$x^{(k+1)} = g(x^{(k)}), \tag{17}$$

where k is an indicator counting the number of iterations. Hence, in words, the value for x in the $k+1$-iteration is obtained by applying function g on the value for x in the k-iteration. Ideally, these approximations become more and more precise with an increasing number of iterations. Recall that iterative methods, in contrast to direct methods, do not yield an exact solution.

When finding the root of a function f or solving for a system of nonlinear equations, the functional form of g is simply

$$g(x) = x - f(x). \tag{18}$$

This is intuitive: at the root \$ x = x^* \$, we have $f(x^*) = 0$ and hence $g(x^*) = x^*$. In other words, x^* is a *fixed point*.

The following piece of code implements function iteration. As a simple workhorse example, consider the function

$$f(x) = 4\ln(x) - 4, \tag{19}$$

which has a root at $x = e^1 = 2.718282$. For illustration, we print the current guess for $x^{(k)}$ for each iteration. As we can see, $x^{(k)}$ converges to x^* as the number of iterations increases.

```
In [4]: def fun(x):
            return 4*np.log(x) - 4

        def g(x):
            return x - fun(x)

In [5]: tol = 1e-8
        x = 4
        it = 0
        lst = []
        while abs((x - g(x))) > tol:
            it += 1
            x = g(x)
            lst.append(x)
            print(x)


        print("Number of iterations = {}".format(it) )
```

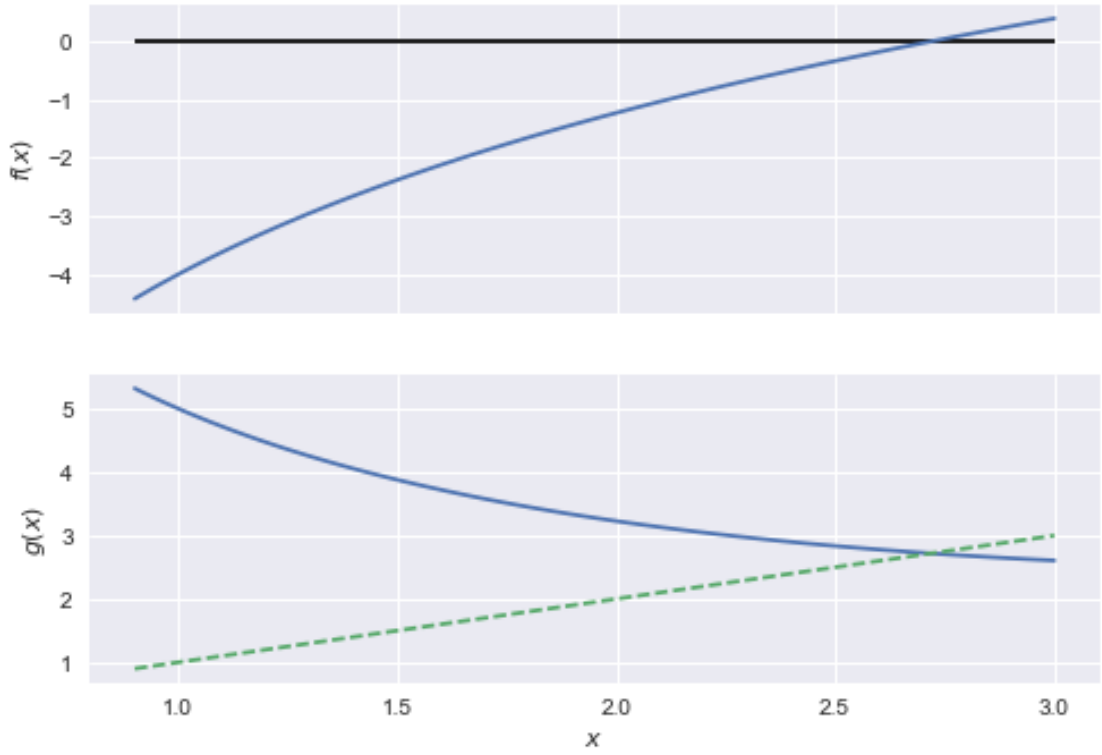
```
2.45482255552
2.86260463623
2.65567694682
2.74887857583
```

```
2.70410642422
2.72502036232
2.71511676033
2.7197769279
2.71757746733
2.71861408156
2.7181251951
2.71835569051
2.71824700267
2.71829824977
2.71827408559
2.71828547937
2.71828010699
2.71828264016
2.71828144573
2.71828200892
2.71828174337
2.71828186858
2.71828180954
2.71828183738
2.71828182425
Number of iterations = 25
```

We can Python's matplotlib package to illustrate how function iteration works.

```
In [6]: x = np.linspace(0.9, 3, 100)
        fig, ax = plt.subplots(2,1, sharex = True)
        # ax[0].ylabel($$f(x)$$)
        ax[0].plot(x, fun(x))
        ax[0].hlines(0, 0.9, 3)
        ax[1].plot(x, g(x))
        ax[1].plot(x, x, '--')
        ax[1].set_xlabel('$x$')
        ax[0].set_ylabel('$f(x)$')
        ax[1].set_ylabel('$g(x)$')

Out[6]: <matplotlib.text.Text at 0x3993db7470>
```



1.7 Newton's Method

Most algorithms used in practice to find the roots of a nonlinear system of equations are based on Newton's method. As function iteration, it is an iterative method. However, it uses additional information, namely about the derivative(s) of f.

Univariate Function Let us start with the case of a univariate function f. Recall that we want to find a root x^* of the function f, i.e. where $f(x^*) = 0$. Start with an initial guess for x^*, denoted by x_0. We can approximate f with a first-order Taylor approximation around x_0:

$$f(x) \approx f(x_0) + (x - x_0)f'(x_0) \tag{20}$$

Setting $f(x) = 0$ - our target value - and solving this expression for x gives us the "best guess" for x^* given the initial guess and the properties of the function (i.e. its value and derivative) at x_0:

$$x_1 \approx x_0 - \frac{f(x_0)}{f'(x_0)} \tag{21}$$

Iterating on this step, we can again generate a sequence $x^{(1)}, x^{(2)}, ..., x^{(n)}$; hence the iteration rule is given by:

$$x^{(k+1)} = x^{(k)} - \frac{f(x^{(k)})}{f'(x^{(k)})} \tag{22}$$

In other words, the functional form of g is now

$$g(x) = x - \frac{f(x)}{f'(x)}. \tag{23}$$

The key idea of Newton's method is *successive linerarization*: a nonlinear problem is replaced with a sequence of linear problems whose solutions converge to the solution of a nonlinear problem. This is illustrated by the following figure.

```
In [7]: ## cp. figure in class
        #
        #
        #
        #
        #
        #
        #
        #
        #
        #
        #
        #
        #
        #
        #
        #
        #
        #
        #
```

Comparing to simple function iteration above, we have one additional term, the derivative of f at $x^{(k)}$. Hence, we use more information on the properties of the function than above. More precisely, we put a weight on the distance between the old guess $x^{(k)}$ and the new guess $x^{(k+1)}$. With function iteration, the distance was given by $f(x^{(k)})$, while in Newton's method, it is $f(x^{(k)})/f'(x^{(k)})$. It is intuitive why this is an improvement: - if the absolute value of $f'(x^{(k)})$ is small, this means the function is relatively flat at $x^{(k)}$; in this case, it is likely that the current guess $x^{(k)}$ is still far from the root, and hence the jump to the next guess should be large - if the absolute value of $f'(x^{(k)})$ is large, the function is relatively steep, making it more likely that we are close to the root; hence the jump to the next guess should be small

Newton's method implements the following pseudo-code:

(i) Specify tolerance levels $tol1$ and $tol2$ and choose a starting guess $x^{(0)}$.

(ii) Compute the next iterate as

$$x^{(k+1)} = x^{(k)} - \frac{f(x^{(k)})}{f'(x^{(k)})} \tag{24}$$

(iii) Check the stopping rule: if $|x^{(k+1)} - x^{(k)}| < tol1$, stop. If not, go back to (ii).

(iv) If $|f(x^{(k+1)})| < tol2$, report $x^{(k+1)}$ as the solution. Otherwise, report failure.

The following code implements Newton's method. As before, the current guess for x is printed in every iteration. Unsurprisingly given the intuition above, Newton's method needs considerably fewer iterations than simple function iteration.

```
In [8]: def fd(x):
            return 4/x

        def g_newton(fun, fun_d, x):
            """
            Implements the iteration rule for Newton's method.
            """
            f, fd = fun(x), fun_d(x)
            return x - f * fd**(-1)

        def my_newton(fun, fun_d, x, tol1 = 1e-8, tol2 = 1e-8):
            """
            Implements Newton's method.
            """
            eps = 1
            it = 0

            while eps > tol1:
                it += 1
                x_new = g_newton(fun, fun_d, x)
                eps = abs(x - x_new)
                x = x_new
                print(x_new)

            print("Number of iterations = {}".format(it) )

            if abs(fun(x)) < tol2:
                return x
            else:
                print("No solution found!")
```

```
In [9]: x_root = my_newton(fun, fd, 1)
```

```
2.0
2.61370563888
2.71624392636
2.71828106436
2.71828182846
2.71828182846
Number of iterations = 6
```

The stopping criteria should not be set too loosely. For some functions, a higher $tol1$ does not impact the solution from Newton's method:

```
In [10]: x_root = my_newton(fun, fd, 1, tol1 = 1e-4)
```

```
2.0
2.61370563888
2.71624392636
2.71828106436
2.71828182846
Number of iterations = 5
```

For other functions, however, setting $tol1$ too high can result in finding a "root" quite far away from zero. This is true for functions that are quite flat around its root. For these functions, we also see very slow convergence. Note that in the example below, $f(x) = x^6$, a small $tol2$ does not compensate for a rather high $tol1$.

```
In [11]: def fun2(x):
             return x**6
         def fd2(x):
             return 6 * x**(5)

         x_root = my_newton(fun2, fd2, 1, tol1 = 1e-4, tol2 = 1e-10)
         print(fun2(x_root))
```

```
0.8333333333333334
0.6944444444444444
0.5787037037037037
0.48225308641975306
0.4018775720164609
0.3348979766803841
0.2790816472336535
0.2325680393613779
0.19380669946781492
0.16150558288984576
0.13458798574153813
0.11215665478461512
0.09346387898717926
0.07788656582264938
0.06490547151887449
0.05408789293239541
0.04507324411032951
0.03756103675860792
0.031300863965506596
0.02608405330458883
0.021736711087157357
0.018113925905964463
0.015094938254970386
```

```
0.01257911521247532
0.010482596010396101
0.008735496675330084
0.00727958056277507
0.006066317135645892
0.00505526427970491
0.004212720233087425
0.0035106001942395207
0.002925500161866267
0.0024379168015552224
0.002031597334629352
0.0016929977788577933
0.0014108314823814943
0.0011756929019845785
0.0009797440849871487
0.0008164534041559572
0.0006803778367966309
0.0005669815306638591
0.0004724846088654926
Number of iterations = 42
1.1125665436700811e-20
```

Multivariate Case The logic from the univariate case translates to a vector-valued function $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$. Recall that its Jacobian is defined as:

$$J(\mathbf{x}) = \begin{bmatrix} \partial f_1/\partial x_1 & ... & \partial f_1/\partial x_n \\ \vdots & \ddots & \vdots \\ \partial f_n/\partial x_1 & ... & \partial f_n/\partial x_n \end{bmatrix} \tag{25}$$

Start with a first-order Taylor approximation around $\mathbf{x}_0$:

$$0 = \mathbf{f}(\mathbf{x}) \approx \mathbf{f}(\mathbf{x}_0) + J(\mathbf{x})(\mathbf{x} - \mathbf{x}_0) \tag{26}$$

As a side note, to see that a Taylor approximation works also in the case of vector-valued functions, note that we can apply it element-wise, i.e.

$$f_j(\mathbf{x}) \approx f_j(\mathbf{x}_0) + f_j(\mathbf{x})^T(\mathbf{x} - \mathbf{x}_0) \tag{27}$$

and recall from above that

$$J(\mathbf{x}) = \begin{bmatrix} \nabla f_1(\mathbf{x})^T \\ \vdots \\ \nabla f_n(\mathbf{x})^T \end{bmatrix} \tag{28}$$

Hence,

$$\mathbf{x} \approx \mathbf{x}_0 - J^{-1}(\mathbf{x}_0) f(\mathbf{x}_0) \tag{29}$$

The key idea is to use this relation iteratively, i.e. generate a sequence $\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, ..., \mathbf{x}^{(m)}$ where

$$\mathbf{x}_{k+1} \approx \mathbf{x}_k - J^{-1}(\mathbf{x}_k) f(\mathbf{x}_k) \tag{30}$$

As an example, consider the function

$$f(\mathbf{x}) = \begin{bmatrix} x_2^2 - 1 \\ \sin x_1 - x_2 \end{bmatrix} \tag{31}$$

To apply Newton's method, start by coding up the function and its Jacobian:

```
In [12]: def foc(x):
             """
             Implements a system of equation in two unknowns, here f(x) = [x2**2 -
             """
             return np.array( (x[1]**2 - 1 , np.sin(x[0]) - x[1] ) )


         def foc_J(x):
             """
             Implements the Jacobian system of equation in two unknowns above
             """
             f_00 = 0
             f_01 = 2 * x[1]
             f_10 = np.cos(x[0])
             f_11 = -1

             return np.array([[f_00, f_01], [f_10, f_11]])
```

```
In [13]: def my_newton_mult(fun, fun_d, x,  tol = 1e-8):
             """
             Implements Newton's method for a vector-valued function
             """
             eps = 1
             it = 0
             while eps > tol:
                 it += 1
                 f, J = fun(x), fun_d(x)
                 x_new = x - np.linalg.inv(J) @ f
                 eps = np.linalg.norm(x - x_new)
                 x = x_new

             print("Number of iterations = {}".format(it) )

             return x

         x_init = [1.5,0.9]
         x = my_newton_mult(foc, foc_J, x_init)
         print(x)
```

```
Number of iterations = 24
[ 1.57079633  1.        ]
```

```
In [14]: x_init = [3,-0.8]
         x = my_newton_mult(foc, foc_J, x_init)
         print(x)

Number of iterations = 27
[ 4.71238897 -1.        ]
```

As bisection, Newton's method is a local method, i.e. does not necessarily find the global optimum. In the case of multiple solutions, like in the example above, it converges to one solution - which one depends on the starting point.

Moreover, it is not guaranteed that the iterates in Newton's method converge, in particular for "erratic" functions with high derivatives that change sign frequently. In such cases, Newton's method converges only for initial starting values that are sufficiently close to a root $\mathbf{x}^*$ (always assuming that the Jacobian J is invertible and well-conditioned at $\mathbf{x}^*$).

We can modify Newton's algorithm as outlined above to increase the likelihood of convergence, by incorporating *backstepping*. Intuitively, if the sequence of iterates $\mathbf{x}^{(k)}$ "goes in the wrong direction" - that is, if the distance to the root gets larger instead of smaller as k increases - we decrease the size of the "step" to the next iterate.

```
In [15]: ## cp. classroom notes
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
```

If J is ill-conditioned at $\mathbf{x}^*$, it can lead to inaccurately computed updates $\mathbf{x}^{(k)}$, which may prevent Newton's method from converging. As outlined in the last lecture, ill-conditioning can stem from units of measurement that vary vastly in their order of magnitude. Rescaling variables so that their values have comparable orders of magnitudes may be a remedy here (and is a good idea in general).

1.8 Numerical Differentiation

Before moving on, it is useful to look at *numerical differentiation*: instead of working with precise derivatives of a function, we can use numerical approximations for these derivatives. This is very useful in particular when the function is complicated, and hence its precise derivatives are hard to obtain.

Numerical derivatives are based on *finite differences*. In the one-dimensional case, we have

$$\frac{\partial f}{\partial x} \approx \frac{f(x+\epsilon) - f(x)}{\epsilon} \tag{32}$$

where ϵ is small. In other words, the first derivative at x is approximated by the response to a small perturbation of x. Note that the approximation above is called the *forward-difference* or *one-sided-difference*. Similarly, the *central-difference* is given by:

$$\frac{\partial f}{\partial x} \approx \frac{f(x+\epsilon) - f(x-\epsilon)}{2\epsilon} \tag{33}$$

Both expression are intuitive to derive graphically.

```
In [16]: ## cp. figure in class
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
```

Formally, finite differencing is based on Taylor's Theorem, as shown below for a multivariate function $f : \mathbb{R}^n \to \mathbb{R}$. In this case, we can write the forward-difference formula as:

$$\frac{\partial f}{\partial x_i} \approx \frac{f(x + \epsilon e_i) - f(x)}{\epsilon}, \tag{34}$$

where e_i is the corresponding canonical vector.

```
In [17]: ## cp. classroom notes: finite difference and Taylor's Theorem
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
```

What value should be chosen for ϵ? A good rule of thumb is the square root of machine epsilon:

$$\epsilon = \sqrt{\epsilon_{DP}} \approx 10^{-8} \tag{35}$$

For a vector-valued function, a similar derivation gives:

$$J(\mathbf{x})\mathbf{p} \approx \frac{\mathbf{f}(\mathbf{x}_0 + \epsilon\mathbf{p}) - \mathbf{f}(\mathbf{x}_0)}{\epsilon} \tag{36}$$

Note that this expression does not give us an approximation for the full Jacobian, but rather for the product $J(\mathbf{x})\mathbf{p}$. It requires two function evaluations; it can be shown that a complete approximation of the Jacobian would require $n+1$ function evaluations for a function of dimension n.

In Python Numerical differentation is helpful when using Newton's method, as outlined above. One common problem with Newton's method are programming errors when coding the Jacobian, in particular for complicated functions. An easy check is to compare the analytic Jacobian with its finite-difference counterpart. Python has a package **statsmodels** that includes a routine for numerical differentation of a vector-valued function, as illustrated by the following example:

```
In [18]: foc_J(x)

Out[18]: array([[  0.00000000e+00,  -2.00000000e+00],
                [ -1.08160913e-08,  -1.00000000e+00]])

In [19]: import statsmodels.api as sm

         sm.tools.numdiff.approx_fprime(x, foc)

Out[19]: array([[  0.00000000e+00,  -1.99999999e+00],
                [  2.37159347e-08,  -1.00000000e+00]])
```

As a side note, the Scipy package (discussed in more detail below) has a function of the same name, **scipy.optimize.approx_fprime**. In contrast to the **statsmodels** version, this works only for univariate and multivariate scalar functions; in the multivariate case, this computes the gradient.

```
In [20]: ## precise derivative of the example function below
         fd(3.5)

Out[20]: 1.1428571428571428

In [21]: ## numerical derivative
         scipy.optimize.approx_fprime([3.5], fun, [1e-8])

Out[21]: array([ 1.1428571])
```

1.9 Quasi-Newton Methods

There is an obvious cost of using Newton's method as outlined above: we need to provide the analytical derivative of a univariate scalar function or the Jacobian of a vector-valued function, respectively. While this may be not a big deal for simple functions as in the examples above, for more complicated problems, this step may involve a large cost in terms of time for computing the derivatives and for coding them up. Moreover, as mentioned above, coding up complicated derivatives increases the risk of programming errors.

Hence, in practice we often rely on "derivative-free" or *Quasi-Newton* methods. In a nutshell, their basic idea is the same as in Newton method's - successive linearization - but instead of using the precise derivatives of a function, we approximate them numerically. A drawback is that we have to provide an initial guess for the function's derivative or Jacobian.

Univariate Functions: Secant Method In the one-dimensional case, an obvious idea would be to obtain the derivative in Newton's method by numerical differentation:

$$f'(x^{(k)}) \approx \frac{f(x^{(k)} + \epsilon) - f(x^{(k)})}{\epsilon} \tag{37}$$

While this would work, in practice we use the *secant method*, where the finite-difference approximation of $f'(x)$ is constructed from the function values at the current and the previous iterate:

$$f'(x^{(k)}) \approx \frac{f(x^{(k)}) - f(x^{(k-1)})}{x^{(k)} - x^{(k-1)}} \equiv (f')^{(k)} \tag{38}$$

The main advantage is that this reduces the number of function evaluations: since $f(x^{(k-1)})$ was evaluated anyway when updating the guess for x, it is easy to store and use for the update of $f'(x)$. In the first expression, we would have to *additionally* evaluate $f(x^{(k)} + \epsilon)$ in every iteration. Note that the computational cost may not be large for simple function, but can be substantial for more complicated functions.

```
In [22]: ## cp. classroom notes
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
```

Note that the secant method generates not only a sequence $x^{(k)}$, but also a sequence $(f')^{(k)}$ of approxiations of the derivative. In general, while $x^{(k)}$ converge to the root of the the function, $(f')^{(k)}$ does *not* converge to the derivative of f at the root.

In this week's problem set, you are asked to to implement this function numerically. Intuitively, the secant method requires more iterations and hence more function evaluations than Newton's method due to the approximation of the derivative. For more complex functions, however, this doesn't necessarily translate into a longer running time, as the secant method does not require to evaluate the function's derivative.

Multivariate Functions: Broyden's Method The cost of computing exact analytical derivatives increases quadratically in the number of dimensions. Hence, in practice, we usually rely on the multidimensional equivalent to the secant method, *Broyden's Method*.

Recall from above that we can approximate the Jacobian of a vector-valued function with

$$J(\mathbf{x})\mathbf{p} \approx \frac{\mathbf{f}(\mathbf{x}_0 + \epsilon\mathbf{p}) - \mathbf{f}(\mathbf{x}_0)}{\epsilon} \tag{39}$$

Analogous to the one-dimensional case, we will work with an approximation A where $\epsilon = 1$. Setting $\mathbf{x}_1 = \mathbf{x}_0 + \mathbf{p}$, define A such that

$$A\mathbf{p} = \mathbf{f}(\mathbf{x}_1) - \mathbf{f}(\mathbf{x}_0). \tag{40}$$

In other words, a numerical approximation A of the Jacobian should satisfy the *secant condition*.

```
In [23]: ## cp. classroom notes
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
         #
```

1.10 Convergence

All of the methods above were iterative, i.e. generating sequences $x^{(k)}$ (hopefully) converging to the root of the function. Importanly, a sequence of iterates $x^{(k)}$ converges to x^* at a rate of order p if there is a constant C such that

$$||x^{(k+1)} - x^*|| \leq C||x^{(k)} - x^*||^p \tag{41}$$

for sufficiently large k. We can determine C and p for the different methods above:

- Bisection converges with $C = 0.5$ and $p = 1$, and hence at a *linear rate*
- Function iteration converges also at a linear rate, with C equal to $f'(x^*)$
- The secant/Broyden's method converges at a *superlinear rate*, with $1 < p \approx 1.62 < 2$
- Newton's method converges at a *quadratic rate* of $p = 2$

Compare the example in M&F, section 3.6.

1.11 Summary: What method to pick?

- Newton's method has the fastest rate of convergence, but can require a lot of "developmental" effort; it should be used for nonlinear equations of small dimensions, when the derivatives are not too hard to find and to code and when a equation is to be solved many times
- The secant/Broyden's method has a smaller rate of convergence (but still more than linear) and requires less programming time; it should be used when the derivatives are expensive to compute and code, and the equation is not solved too often
- In the univariate case, the bisection method is the most robust; for highly irregular functions, it can be used in combination with Newton's method, to find a close initial guess

---

1.12 The Scipy Package

Modern programming languages have built-in implementations of the algorithms outlined above. In Python, we mainly rely on the **Scipy** package, which comes with the Anaconda distribution. For rootfinding and numerical optimization (in the next lecture), we use Scipy's subpackage **scipy.optimize**, which we first need to import:

```
In [24]: import scipy.optimize
```

1.12.1 One dimension

For the univariate case, consider again the function

$$f(x) = 4\ln(x) - 4 \tag{42}$$

We define the function and use the **bisect()** function, an implementation of the bisection method outlined above:

```
In [25]: def fun(x):
             return 4*np.log(x) - 4

         print(scipy.optimize.bisect(fun,1,4))
```

```
2.718281828459567
```

bisect(fun,a,b) takes three arguments: the function **fun** (which can be built-in or user-written), and an upper and lower initial guess for the root. In other words, you tell the algorithm to look for a root in the interval $[a, b]$. The important thing to remember here is that $f(a)$ and $f(b)$ must have different signs - if they do not, you will get an error message (in this case, change a or b and try again).

In the example above, solving for the root using Python is not really necessary. The real advantage of numerical root finding is in situations where finding a solution to $f(x) = 0$ analytically is not feasible. Consider, for example,

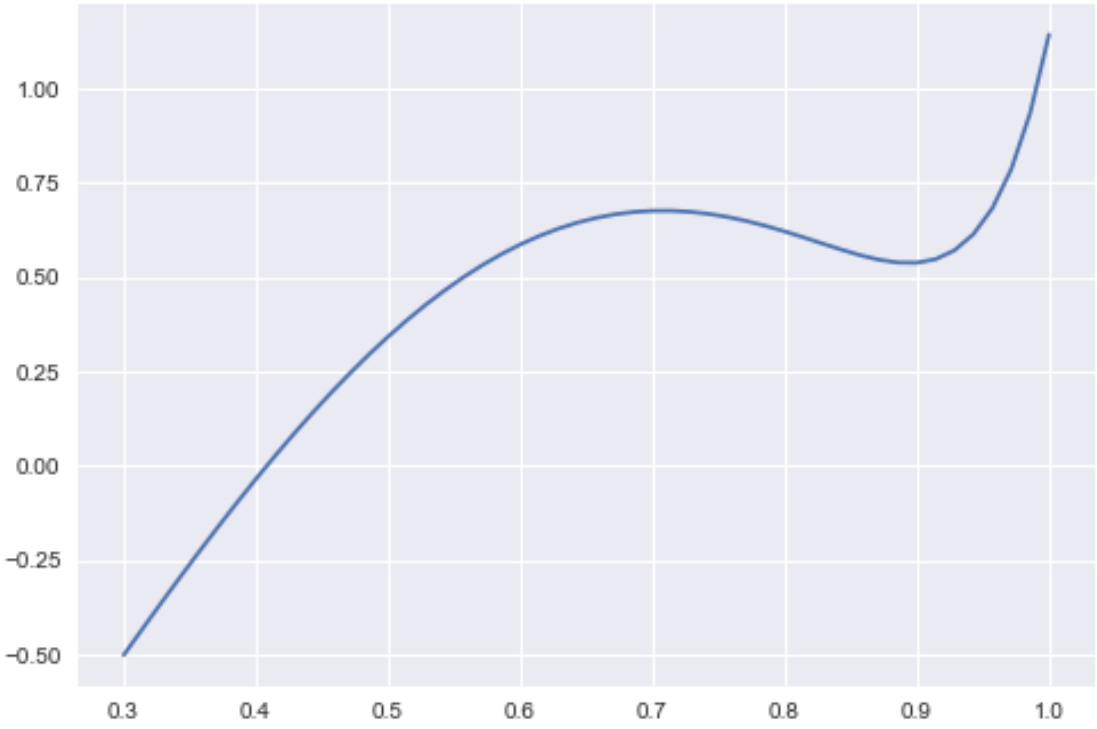
$$f(x) = \sin(4(x - 1/4)) + x + x^{20} - 1 \tag{43}$$

We can Matplotlib to plot the function:

```
In [26]: def fun(x):
             return np.sin(4 * (x - 0.25)) + x + x**20 - 1

         fig, ax = plt.subplots()
         x = np.linspace(0.3, 1, 50)
         ax.plot(x, fun(x))

Out[26]: [<matplotlib.lines.Line2D at 0x39956612b0>]
```



Finding a root via the bisection method is straightforward:

```
In [27]: print(scipy.optimize.bisect(fun,0,2))
```

```
0.4082935042806639
```

The Scipy function **newton(fun, x, fprime = None)** implements both Newton's method (if the derivate of the function is given as **fprime**) and the secant method for univariate rootfinding:

```
In [28]: def fun_d(x):
             return np.cos(4 * (x - 0.25)) * 4 + 1 + 20 * x**19

         scipy.optimize.newton(fun,0.6, fun_d)

Out[28]: 0.408293504279367

In [29]: scipy.optimize.newton(fun,0.6)

Out[29]: 0.40829350427936667
```

We can use Jupyter's **%timeit** magic to compare the running time of each of the three methods:

```
In [30]: %timeit -n1 scipy.optimize.bisect(fun,0,2)

1 loop, best of 3: 308 µs per loop


In [31]: %timeit -n1 scipy.optimize.newton(fun,0.6)

1 loop, best of 3: 97.1 µs per loop


In [32]: %timeit -n1 scipy.optimize.newton(fun,0.6, fun_d)

1 loop, best of 3: 111 µs per loop
```

For comparison, our implementation of Newton's method above takes about the same running time:

```
In [33]: def my_newton(fun, fun_d, x, tol1 = 1e-8, tol2 = 1e-8):

             eps = 1
             it = 0
             maxit = 100

             while eps > tol1 and it < maxit:
                 it += 1
                 x_new = g_newton(fun, fun_d, x)
                 eps = abs(x - x_new)
                 x = x_new

             if abs(fun(x)) < tol2:
```

```
            return x
        else:
            print("No solution found!")


my_newton(fun, fun_d, 0.99)
```

```
Out[33]: 0.40829350427936706
```

```
In [34]: %timeit -n1 my_newton(fun, fun_d, 0.6)
```

```
1 loop, best of 3: 120 µs per loop
```

1.12.2 Multiple Dimensions: NGM Revisited

As an example for a multidimensional system of nonlinear equation, let's go back to our NGM model. Recall that a steady state is given by (k_s, h_s) such that

$$\begin{bmatrix} S_1 \\ S_2 \end{bmatrix} = \begin{bmatrix} \beta\left[f_k(k_s, h_s) + 1 - \delta\right] - 1 \\ \left[f(k_s, h_s) - \delta k\right]^{-\nu} f_h(k_s, h_s) - B h_s^{\eta} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{44}$$

To solve this system numerically, we first need to assign values to the model parameters.

Parameters

```
In [35]: ## utility
         beta = 0.8       # discount factor
         nu = 2        # risk-aversion coefficient for consumption
         eta = 1          # elasticity parameter for labor supply
         eps = 1e-6       # lower bound of consumption and labor supply
         ## production
         alpha = 0.25
         delta = 0
         ## derived
         # A = (1 - beta * (1 - delta))/(alpha*beta) # normalization parameter for
         # B = (1 - alpha) * A * (A - delta)**nu      # parameter for utility funct
         B = 0.8
         A = 1.1
```

Functions Next, it will be useful to define some auxiliary functions that implement the Cobb-Douglas production function, as well as its first and second derivatives.

```
In [36]: def cobb_douglas(x, alpha, A):
             """
             Evaluates the Cobb-Douglas function with coefficient alpha and shift p
             """
             return A * x[0]**alpha * x[1]**(1 - alpha)
```

```
        def cd_diff(x, alpha, A):
            """
            Evaluates the first derivatives (returned as a tuple) of the Cobb-Doug
            """
            return (A * alpha * cobb_douglas(x, alpha, A) / x[0],
                    A * (1 - alpha) * cobb_douglas(x, alpha, A) / x[1])

        def cd_diff2(x, alpha, A):
            """
            Evaluates the second derivative (returned as a tuple, with the cross d
            """
            return (A * alpha * (alpha - 1) * cobb_douglas(x, alpha, A) / x[0]**2,
                    A * (1 - alpha) * (-alpha) * cobb_douglas(x, alpha, A) / x[1]*
                    A * alpha * (1 - alpha) * cobb_douglas(x, alpha, A) / (x[0] *
```

Finally, we can code up the system of nonlinear equations S as a Numpy array.

```
In [37]: def steady(x):
             """
             Returns the vector-valued function consisting of the steady-state cond
             """
             y = np.zeros(2)
             mp = cd_diff(x, alpha, A)

             y[0] = beta * (mp[0] + 1 - delta) - 1
             y[1] = (cobb_douglas(x, alpha, A) - delta * x[0])**(-nu) * mp[1] - B *

             return y
```

For Newton's method, we also need to provide the Jacobian, i.e.

$$J(k,h) = \begin{bmatrix} \partial S_1/\partial k & \partial S_1/\partial h \\ \partial S_2/\partial k & \partial S_2/\partial h \end{bmatrix} \tag{45}$$

```
In [38]: def steady_jac(x):
             """
             Returns the Jacobian of the vector-valued function consisting of the s
             """
             J = np.zeros((2,2))
             mp = cd_diff(x, alpha, A)
             mp2 = cd_diff2(x, alpha, A)

             Q = cobb_douglas(x, alpha, A) - delta * x[0]

             J[0,0] = beta * mp2[0]
             J[0,1] = beta * mp2[2]
             J[1,1] = -nu * Q**(-nu-1) * mp[1]**2 + Q**(-nu) * mp2[1] - B * eta * x
             J[1,0] = -nu * Q**(-nu-1) * mp[1] * (mp[0] - delta) + Q**(-nu) * mp2[2

             return J
```

Solve for the steady state Start by using our implementation of Newton's method written above. We also need to provide an initial guess x_0:

```
In [39]: x0 = np.array([0.5, 0.5])
         my_newton_mult(steady, steady_jac, x0)

Number of iterations = 10


Out[39]: array([ 1.23548993,  0.95820563])
```

Next, we use Scipy's **optimize.broyden1** function, an implementation of Broyden's method outlined above. As it is derivative-free, we do not have to provide the Jacobian:

```
In [40]: res = scipy.optimize.broyden1(steady, x0, f_tol = 1e-8)
         print(res)

[ 1.23548994  0.95820563]
```

Note that alternatively, you can also call Scipy's **optimize.root** function, which is essentially a "wrapper" around different algorithms for solving nonlinear systems of equations, not only Broyden's method. I usually use the **root** function since it provides a more informative output, in particular on function values and number of iterations.

```
In [41]: res = scipy.optimize.root(steady, x0,  tol = 1e-8, method = "broyden1")
         print(res)

     fun: array([  7.35234096e-12,  -1.00914832e-11])
 message: 'A solution was found at the specified tolerance.'
     nit: 20
  status: 1
 success: True
       x: array([ 1.23548993,  0.95820563])
```

As expected, Broyden's method takes more iterations to solve the system than Newton's method.