CHAPTER

13

Open Root Finding Methods

OUTLINE

Two roads diverged in a yellow wood,
And sorry I could not travel both
And be one traveler, long I stood
And looked down one as far as I could
To where it bent in the undergrowth;

–"The Road Not Taken" by Robert Frost

CHAPTER POINTS

- Open root finding methods require an initial guess of the solution.
- These can converge faster than closed methods, but are not guaranteed to find a root.
- Newton's method uses the derivative of the function to generate a new estimate of the root.
- Newton's method can be generalized to multidimensional problems. Each iteration then requires the solution of a linear system of equations.

The previous chapter discussed closed root finding methods that bracket the root and then zoom in on the root by tightening the interval like a boa constrictor on a rat. Today we'll

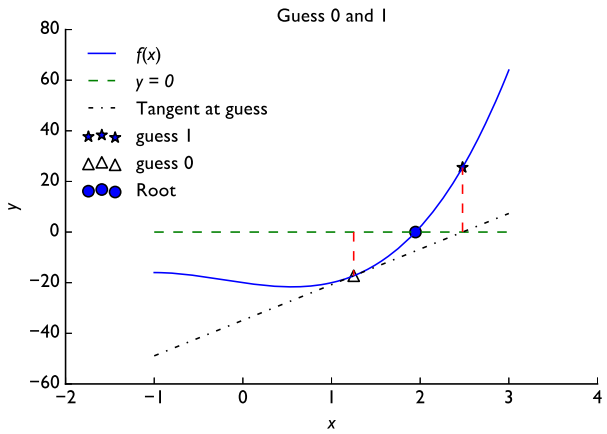
DOI: 10.1016/B978-0-12-812253-2.00015-7

discuss open methods that only need an initial guess, but not a range to begin. These are called open root finding methods.

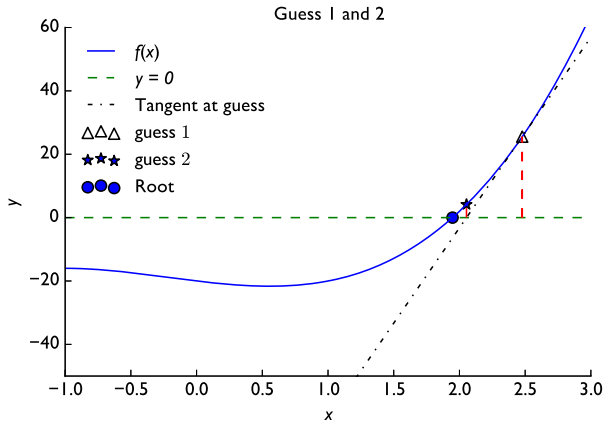
13.1 NEWTON'S METHOD

Newton's method is a rapidly convergent method that is a good choice provided that one has an estimate of the root. Newton's method is also known as the Newton–Raphson method because Isaac Newton is famous enough, and Raphson published the method before Newton did. However, the historical record indicates that Newton had used this method well before Raphson published it. Interestingly enough, Raphson also coined the term pantheism.

Newton's method is fairly robust. What the method does is compute the tangent at the guess x_i (via the derivative $f'(x_i)$), and uses where the tangent crosses zero to get the next guess. In the following figures, this procedure is illustrated. First, we show the procedure for going from the initial guess to the first calculated update:



Next, we compute the slope of the function at this point and find where it crosses the axis:



As you can see, we get closer to the root each iteration. This is due to the fact that over a short range, the tangent line is a reasonable approximation to the original function—just as the linear Taylor series can be a good approximation for small perturbations around the center of the expansion. Of course, using the slope to approximate the function behavior near a root is not always a good approximation, as we will see later.

The basic idea is to compute where the tangent line to the current root estimate crosses the x axis. We do this by equating the derivative of the function at the current estimate, x_i, to the slope of a line between the point $f(x_i)$ and the estimate of the function evaluated at an unknown point $f(x_{i+1})$:

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i}.$$

Then, we set $f(x_{i+1}) = 0$ to determine where the tangent crosses zero. This yields

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}.$$

Therefore, each iteration requires the evaluation of the slope at x_i and the value of the function at that same point. An implementation of Newton's method in Python is given below.

```
In [1]: def newton(f,fprime,x0,epsilon=1.0e-6, LOUD=False):
            """Find the root of the function f via Newton-Raphson method
            Args:
                f: function to find root of
                fprime: derivative of f
                x0: initial guess
                epsilon: tolerance

            Returns:
                estimate of root
            """
            x = x0
            if (LOUD):
                print("x0 =",x0)
            iterations = 0
            fx = f(x)
            while (np.fabs(fx) > epsilon):
                fprimex = fprime(x)
                if (LOUD):
                    print("x_",iterations+1,"=",x,"-",fx,
                          "/",fprimex,"=",x - fx/fprimex)
                x = x- fx/fprimex
                iterations += 1
                fx = f(x)
            print("It took",iterations,"iterations")
            return x #return estimate of root
```

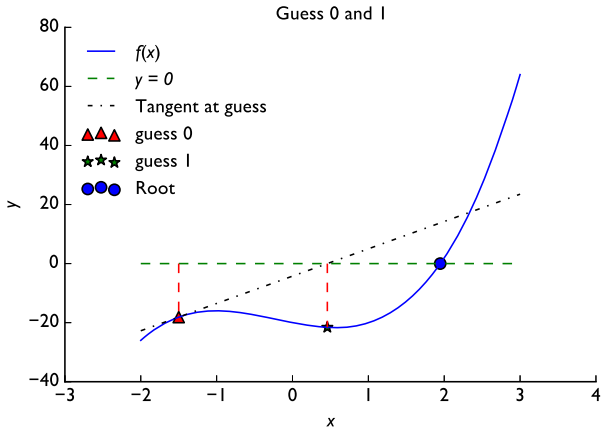
We will test the method on the cubic nonlinear function from the previous chapter. We will need to define a derivative function to use because Newton's method requires it. As a reference, bisection took 23 iterations and Ridder's method took 8. It is slightly difficult to

compare an open method to the closed methods because we cannot run the method under the same conditions and the open method only requires a single initial guess, not an interval that brackets the root. Furthermore, in Newton's method we have to evaluate the function and its derivative in each iteration. It is useful, however, to compare the overall convergence behavior between methods.

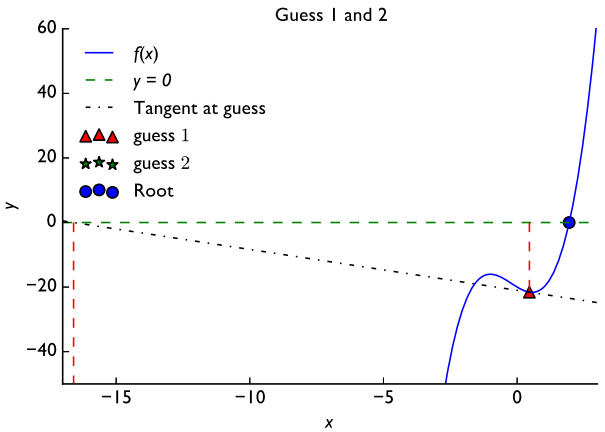
```
In [2]: def Dnonlinear_function(x):
            #compute a nonlinear function for demonstration
            return 9*x**2 + 4*x - 5
        root = newton(nonlinear_function,Dnonlinear_function,-1.5,LOUD=True)
        print("The root estimate is",root,"\nf(",root,") =",
              nonlinear_function(root))


x0 = -1.5
x_ 1 = -1.5 - -18.125 / 9.25 = 0.45945945945945943
x_ 2 = 0.45945945945945943 - -21.584111503760884 / -1.2622352081811545
= -16.64045295295295
x_ 3 = -16.64045295295295 - -13206.446010659996 / 2420.5802585031524
= -11.184552053047796
x_ 4 = -11.184552053047796 - -3911.2567600472344 / 1076.1096334338297
= -7.54992539557002
x_ 5 = -7.54992539557002 - -1159.3159776918205 / 477.81265972577813
= -5.123627234367753
x_ 6 = -5.123627234367753 - -345.3883141179978 / 210.7694953933235
= -3.484925611661592
x_ 7 = -3.484925611661592 - -105.25615528465704 / 90.36265622268793
= -2.320106429882939
x_ 8 = -2.320106429882939 - -35.100340028952424 / 34.165618894325654
= -1.2927478991471761
x_ 9 = -1.2927478991471761 - -16.675175982276617 / 4.869782580156233
= 2.131465750600949
x_ 10 = 2.131465750600949 - 7.479645608829035 / 44.41417921626759
= 1.963059044886359
x_ 11 = 1.963059044886359 - 0.5864442011938849 / 37.53464350293673
= 1.9474349667957278
x_ 12 = 1.9474349667957278 - 0.004789634744607696 / 36.92226641627101
= 1.947305244673835
It took 12 iterations
The root estimate is 1.947305244673835
f( 1.947305244673835 ) = 3.2858902798693634e-07
```

In this example we had a bad initial guess so the method went the wrong way at first, but it eventually honed in on the solution. This highlights a feature of open methods: the root estimate can get worse, and even diverge. This is in comparison with closed methods where the root is confined to an interval. On the other hand, open methods only require an initial guess instead of knowledge of an interval where the root lies. In the first iteration, the method did move closer to the root, but the new estimate was close to a local minimum:



The slope at this point then moves the method long distance in the wrong direction:



This estimate is much worse than that after the first iteration. Nevertheless, after this point the method, does move in the correct direction.

The other test we performed on the closed root finding methods was the critical sphere problem:

$$f(R) = \left(\frac{\pi}{R+2D}\right)^2 - \frac{\nu\Sigma_\mathrm{f} - \Sigma_\mathrm{a}}{D}.$$

For Newton's method, we need to compute the derivative:

$$f'(R) = \frac{-2\pi^2}{(R+2D)^3}.$$

Recall that regula falsi (false position) had a very hard time with this problem because the function is flat for a large range of the radius and sharply changing past a transition point. Newton's method does not suffer from these problems.

First, we define the function and a function to compute its derivative:

```
In [3]: def Crit_Radius(R, D=9.21, nuSigf = 0.1570, Siga = 0.1532):
            return (np.pi/(R + 2*D))**2 - (nuSigf - Siga)/D
        def DCrit_Radius(R, D=9.21, nuSigf = 0.1570, Siga = 0.1532):
            return (-2.0*np.pi**2/(R + 2*D)**3)
```

For this problem bisection took 28 iterations, false position took 260, and Ridder's method took 3 iterations. Here, we will run Newton's method with an initial guess of 120 (about the midpoint of the initial range used in the previous chapter).

```
In [4]: Radius = newton(Crit_Radius,DCrit_Radius,120,LOUD=True)
        print("The critical radius estimate is",Radius,
         "\nf(",Radius,") =",Crit_Radius(Radius))

x0 = 120
x_ 1 = 120 - 0.00010251745512686794 / -7.442746142981495e-06
= 133.77414373101277
x_ 2 = 133.77414373101277 - 1.3497467451998235e-05 / -5.599328100744185e-06
= 136.1846949987987
It took 2 iterations
The critical radius estimate is 136.1846949987987
f( 136.1846949987987 ) = 3.140322315689872e-07
```

This is even faster than Ridder's method, though the comparison is not exactly fair. We did, however, need to know the derivative of the function and evaluate this at each iteration.

BOX 13.1 NUMERICAL PRINCIPLE

Newton's method converges rapidly and has a degree of robustness. It also only requires an initial guess, not an interval bounding the root. You do need to know the derivative of the target function to use Newton's method without modification.

13.2 INEXACT NEWTON

What if we do not know the derivative of the function? There may be cases where the derivative of the function is unknown or not easily calculable. In these cases we can use a method that is known as inexact Newton. This method estimates the derivative using a finite difference:

$$f'(x_i) \approx \frac{f(x_i + \delta) - f(x_i)}{\delta},$$

which will converge to the derivative as $\delta \to 0$. Indeed, the limit of this approximation as $\delta \to 0$ is the definition of a derivative. To implement this we need to only make a small change to the code to estimate the derivative instead of calling a derivative function. The downside

is that we need an extra function evaluation to estimate the derivative. This extra cost is partially offset by eliminating the need to evaluate a function that gives the derivative. In the code below, we modify the Newton's method implementation above to perform inexact Newton iterations.

```
In [5]: def inexact_newton(f,x0,delta = 1.0e-7, epsilon=1.0e-6, LOUD=False):
            """Find the root of the function f via Newton-Raphson method
            Args:
                f: function to find root of
                x0: initial guess
                delta: finite difference parameter
                epsilon: tolerance

            Returns:
                estimate of root
            """
            x = x0
            if (LOUD):
                print("x0 =",x0)
            iterations = 0
            fx = f(x)
            while (np.fabs(fx) > epsilon):
                fxdelta = f(x+delta)
                slope = (fxdelta - fx)/delta
                if (LOUD):
                    print("x_",iterations+1,"=",x,"-",fx,"/",slope,"=",
                          x - fx/slope)
                x = x - fx/slope
                fx = f(x)
                iterations += 1
            print("It took",iterations,"iterations")
            return x #return estimate of root
```

To compute the derivative we need to define the value of δ. In general, a reasonable value for this parameter is 10^{-7}, though it should be adjusted if the value of x in the function evaluation is very large or small.

On the critical radius problem, inexact Newton performs the same as the original Newton method:

```
In [5]: Radius = inexact_newton(Crit_Radius,120,LOUD=True)
        print("The critical radius estimate is",Radius,
              "\nf(",Radius,") =",Crit_Radius(Radius))

x0 = 120
x_ 1 = 120 - 0.00010251745512686794 / -7.442742169100347e-06
= 133.77415108540026
x_ 2 = 133.77415108540026 - 1.3497426272372716e-05 / -5.599325256927523e-06
= 136.18469622307978
It took 2 iterations
The critical radius estimate is 136.18469622307978
f( 136.18469622307978 ) = 3.1402569209483836e-07
```

Notice that we get the same answer and it took the same number of iterations.

BOX 13.2 NUMERICAL PRINCIPLE

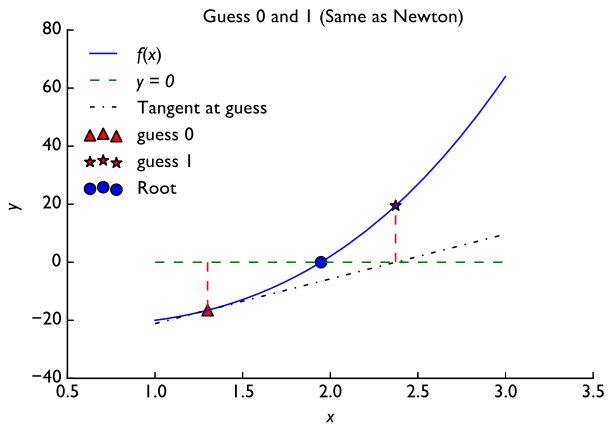
Inexact Newton gives a way around the necessity of knowing the function's derivative. The cost is an extra function evaluation at each iteration.

13.3 SECANT METHOD

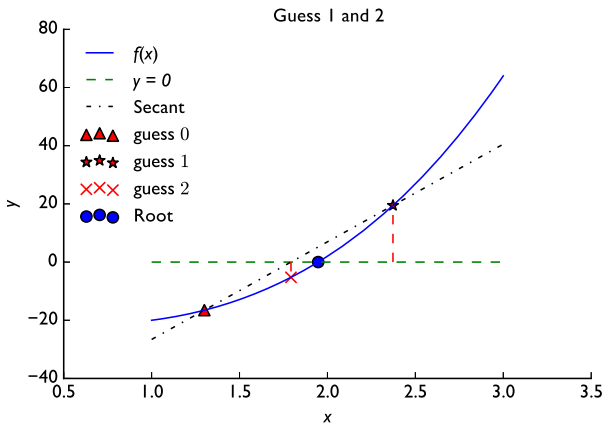
The secant method is a variation on the theme of Newton's method. It takes its name from the fact that it constructs a straight line that intersects the curve at two points: such a line is called a secant. In this case we use the previous two guesses to construct the slope:

$$f'(x_i) \approx \frac{f(x_i) - f(x_{i-1})}{x_i - x_{i-1}}.$$

The benefit of this is that it does not require an additional function evaluation, nor do we have to evaluate a derivative function. This will be a big savings if it takes a long time to do a function evaluation. One issue is that we need two points to get started. Therefore, we can use inexact Newton for the first step and then use secant from then on. In a graphical demonstration, we first take a step of inexact Newton:



Then we draw the estimate of the derivative using x_0 and x_1, and find where this crosses the x axis:



Notice that this estimate of the root behaves similarly to that from Newton's method, with only one function evaluation per iteration. A Python implementation of the secant method is given next:

```
In [6]:  def secant(f,x0,delta = 1.0e-7, epsilon=1.0e-6, LOUD=False):
             """Find the root of the function f via Newton-Raphson method
             Args:
                 f: function to find root of
                 x0: initial guess
                 delta: finite difference parameter
                 epsilon: tolerance

             Returns:
                 estimate of root
             """
             x = x0
             if (LOUD):
                 print("x0 =",x0)
             #first time use inexact Newton
             x_old = x
             fold = f(x_old)
             fx = fold
             slope = (f(x_old+delta) - fold)/delta
             x = x - fold/slope
             if (LOUD):
                 print("Inexact Newton\nx_",1,"=",x,"-",fx,"/",slope,"=",
                       x - fx/slope,"\nStarting Secant")
             fx = f(x)
             iterations = 1
             while (np.fabs(fx) > epsilon):
                 slope = (fx - fold)/(x - x_old)
                 fold = fx
                 x_old = x
                 if (LOUD):
                     print("x_",iterations+1,"=",x,"-",fx,
                           "/",slope,"=",x - fx/slope)
                 x = x - fx/slope
```

```
            fx = f(x)
            iterations += 1
        print("It took",iterations,"iterations")
        return x #return estimate of root
```

Running the secant method on the critical radius problem, we observe that it takes one more iteration the Newton's method:

```
In [7]:  Radius = secant(Crit_Radius,120,LOUD=True)
         print("The critical radius estimate is",Radius,"\nf(",Radius,
         ") =",Crit_Radius(Radius))


x0 = 120
Inexact Newton
x_ 1 = 133.77415108540026 - 0.00010251745512686794 / -7.442742169100347e-06
= 147.54830217080053
Starting Secant
x_ 2 = 133.77415108540026 - 1.3497426272372716e-05 / -6.462832322846442e-06
= 135.86262028870274
x_ 3 = 135.86262028870274 - 2.0397790242225296e-06 / -5.486146135184707e-06
= 136.23442573718336
It took 3 iterations
The critical radius estimate is 136.23442573718336
f( 136.23442573718336 ) = 4.8524539480966286e-08
```

Despite the fact that it took one more iteration, secant only required four function evaluations to compute the root: one per iteration, plus one extra for the slope estimation in the first step.

BOX 13.3 NUMERICAL PRINCIPLE

The secant method is a middle ground between inexact Newton and Newton's method. It uses existing function evaluations to approximate the derivative of the function. It does converge slightly slower than Newton's method, however.

13.4 SLOW CONVERGENCE

Newton's method, including its inexact variant, and the secant method can converge slowly in the presence of the following:

1. Multiple roots or closely spaced roots,
2. Complex roots,
3. Bad initial guess.

BOX 13.4 NUMERICAL PRINCIPLE

General nonlinear functions can behave in many different ways, and some of these behaviors make root-finding difficult. In practical applications, finding a good guess at the root is often necessary to find a root efficiently.

The function

$$f(x) = x^7$$

has multiple roots at 0: it converges slowly with Newton's method.

```
In [8]:  mult_root = lambda x: 1.0*x**7
         Dmult_root = lambda x: 7.0*x**6
         root = newton(mult_root,Dmult_root,1.0,LOUD=True)
         print("The root estimate is",root,"\nf(",root,") =",mult_root(root))

x0 = 1.0
x_ 1 = 1.0 - 1.0 / 7.0 = 0.8571428571428572
x_ 2 = 0.8571428571428572 - 0.33991667708911394 / 2.7759861962277634
= 0.7346938775510204
x_ 3 = 0.7346938775510204 - 0.11554334736330486 / 1.1008713373781547
= 0.6297376093294461
x_ 4 = 0.6297376093294461 - 0.03927511069548781 / 0.43657194805493627
= 0.5397750937109538
...
x_ 12 = 0.18347855622969242 - 6.9999864354836515e-06 / 0.00026706066395597614
= 0.15726733391116493
x_ 13 = 0.15726733391116493 - 2.3794121288184727e-06 / 0.00010590810238531585
= 0.13480057192385567
It took 13 iterations
The root estimate is 0.13480057192385567
f( 0.13480057192385567 ) = 8.088018642535101e-07
```

If we decrease the number of roots to 1,

$$f(x) = \sin(x),$$

we see that a similar problem converges faster.

```
In [9]:  mult_root = lambda x: np.sin(x)
         Dmult_root = lambda x: np.cos(x)
         root = newton(mult_root,Dmult_root,1.0,LOUD=True)
         print("The root estimate is",root,"\nf(",root,") =",mult_root(root))

x0 = 1.0
x_ 1 = 1.0 - 0.841470984808 / 0.540302305868
= -0.557407724655
x_ 2 = -0.557407724655 - -0.52898809709 / 0.848629243626
= 0.0659364519248
x_ 3 = 0.0659364519248 - 0.0658886845842 / 0.997826979613
```

```
= -9.57219193251e-05
x_ 4 = -9.57219193251e-05 - -9.57219191789e-05 / 0.999999995419
= 2.92356620141e-13
It took 4 iterations
The root estimate is 2.92356620141e-13
f( 2.92356620141e-13 ) = 2.92356620141e-13
```

For the case of complex roots we will consider a function that has complex roots near the actual root. One such function is

$$f(x) = x(x-1)(x-3) + 3.$$

The derivative of this function is

$$f'(x) = 3x^3 - 8x + 3.$$

The root is at $x = -0.546818$.

```
In [10]: x = np.linspace(-1,4,200)
         comp_root = lambda x: x*(x-1)*(x-3) + 3
         d_comp_root = lambda x: 3*x**2 - 8*x + 3
         root = newton(comp_root,d_comp_root,2.0,LOUD=True)
         print("The root estimate is",root,"\nf(",root,") =",mult_root(root))
```

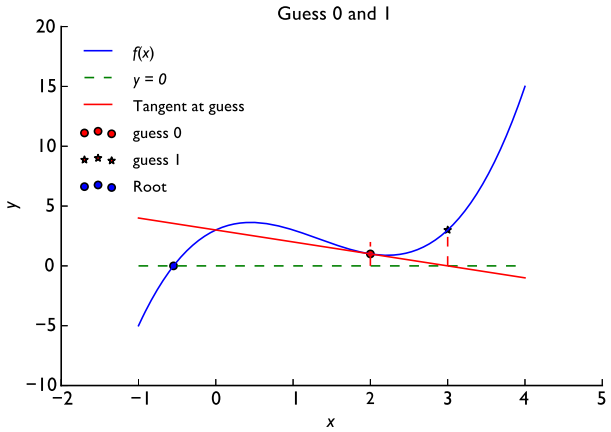
```
x0 = 2.0
x_ 1 = 2.0 - 1.0 / -1.0 = 3.0
x_ 2 = 3.0 - 3.0 / 6.0 = 2.5
x_ 3 = 2.5 - 1.125 / 1.75 = 1.8571428571428572
x_ 4 = 1.8571428571428572 - 1.1807580174927113 / -1.5102040816326543
= 2.6389961389961383
...

x_ 42 = -0.6654802789331873 - -1.062614102742372 / 9.652434236412477
= -0.5553925977621718
x_ 43 = -0.5553925977621718 - -0.07133846535161004 / 8.368523595044415
= -0.5468679799438203
x_ 44 = -0.5468679799438203 - -0.0004111366150030271 / 8.272137602014066
= -0.5468182785685793
It took 44 iterations
The root estimate is -0.5468182785685793
f( -0.5468182785685793 ) = -0.519972092294
```

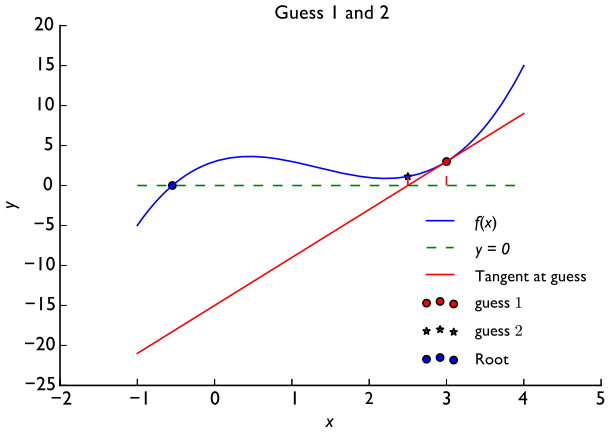
This converged slowly because the complex roots at $x = 2.2734 \pm 0.5638i$ make the slope of the function change so that tangents do not necessarily point to a true root.

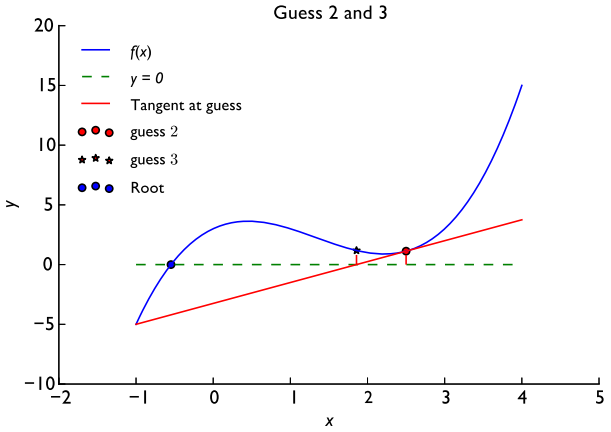
We can see this graphically by looking at each iteration.

The first iteration moves in the wrong direction:

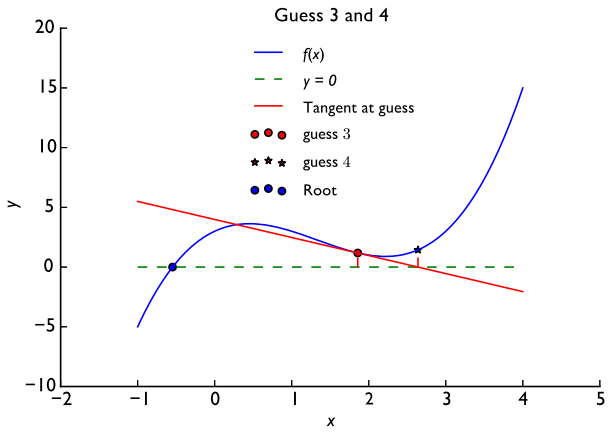


Iterations two and three move in the correct direction:





Step four then moves in wrong direction: it has been two steps forward, one step back:



The presence of the complex root causes the solution to oscillate around the local minimum of the function. Eventually, the method will converge on the root, but it takes many iterations to do so. The upside, however, is that it does eventually converge.

13.5 NEWTON'S METHOD FOR SYSTEMS OF EQUATIONS

Finding the root of a single function is interesting, but in general a simple problem. It is very common that a problem you encounter will require the root of a multidimensional function. This is the situation we analyze now.

Say that we have a function of n variables $\mathbf{x} = (x_1, \ldots x_n)$:

$$\mathbf{F}(\mathbf{x}) = \begin{pmatrix} f_1(x_1, \ldots, x_n) \\ \vdots \\ f_n(x_1, \ldots, x_n) \end{pmatrix},$$

the root is $\mathbf{F}(\mathbf{x}) = \mathbf{0}$. For this scenario we no longer have a tangent line at point $\mathbf{x}$, rather we have a Jacobian matrix that contains the derivative of each component of $\mathbf{F}$ with respect to each component of $\mathbf{x}$:

$$\mathbf{J}(\mathbf{x}) = \begin{pmatrix} \frac{\partial f_1}{\partial x_1}(x_1, \ldots, x_n) & \ldots & \frac{\partial f_1}{\partial x_n}(x_1, \ldots, x_n) \\ \vdots & & \vdots \\ \frac{\partial f_n}{\partial x_1}(x_1, \ldots, x_n) & \ldots & \frac{\partial f_n}{\partial x_n}(x_1, \ldots, x_n) \end{pmatrix}.$$

We reformulate Newton's method for a single equation

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)},$$

as

$$f'(x_i)(x_{i+1} - x_i) = -f(x_i).$$

The multidimensional analog of this equation is

$$\mathbf{J}(\mathbf{x}_i)(\mathbf{x}_{i+1} - \mathbf{x}_i) = -\mathbf{F}(\mathbf{x}_i).$$

Note this is a linear system of equations to solve to get a vector of changes for each x: $\boldsymbol{\delta} = \mathbf{x}_{i+1} - \mathbf{x}_i$. Therefore, in each step we solve the system

$$\mathbf{J}(\mathbf{x}_i)\boldsymbol{\delta} = -\mathbf{F}(\mathbf{x}_i),$$

and set

$$\mathbf{x}_{i+1} = \mathbf{x}_i + \boldsymbol{\delta}.$$

In multidimensional rootfinding we can observe the importance of having a small number of iterations: we need to solve a linear system of equations at each iteration. If this system is large, the time to find the root could be prohibitively long.

BOX 13.5 NUMERICAL PRINCIPLE

Finding the root of a multi-dimensional function is often computationally expensive. There is the additionally complication of determining the Jacobian matrix. The efficient methods for finding roots of functions in many dimensions are beyond the scope of this work. Suffice it to say that these methods are based on approximating the Jacobian of the system with finite differences and employing efficient linear solvers.

13.5.1 Rectangular Parallelepiped (Shoebox) Reactor Example

As an example of a multidimensional root finding problem, we will consider the problem of designing a parallelepiped reactor. For this type of reactor the critical equation given by one-group diffusion theory, i.e., when geometric and materials buckling are equal, is

$$\left(\frac{\pi}{a+2D}\right)^2 + \left(\frac{\pi}{b+2D}\right)^2 + \left(\frac{\pi}{c+2D}\right)^2 = \frac{\nu\Sigma_\mathrm{f} - \Sigma_\mathrm{a}}{D},$$

where a, b, and c are length of the sides. We will solve this with $D = 9.21$ cm, $\nu\Sigma_\mathrm{f} = 0.1570\ \mathrm{cm}^{-1}$, and $\Sigma_\mathrm{a} = 0.1532\ \mathrm{cm}^{-1}$.

To make this a system, we stipulate that the surface area should be $1.2 \times 10^6\ \mathrm{cm}^2$, and that $a = b$. This makes

$$\mathbf{F}(\mathbf{x}) = \begin{pmatrix} \left(\frac{\pi}{a+2D}\right)^2 + \left(\frac{\pi}{b+2D}\right)^2 + \left(\frac{\pi}{c+2D}\right)^2 - \frac{\nu\Sigma_\mathrm{f}-\Sigma_\mathrm{a}}{D} \\ 2(ab+bc+ac) - 1.2\times 10^6 \\ a - b \end{pmatrix},$$

with $\mathbf{x} = (a, b, c)$.

The Jacobian is

$$\mathbf{J}(\mathbf{x}) = \begin{pmatrix} \frac{2\pi}{(a+18.42)^2} & \frac{2\pi}{(b+18.42)^2} & \frac{2\pi}{(c+18.42)^2} \\ 2(b+c) & 2(a+c) & 2(a+b) \\ 1 & -1 & 0 \end{pmatrix}.$$

Start with an initial guess of $a = b = 7000$ and $c = 100$ all in cm:

$$\mathbf{F}(\mathbf{x}_0) = \begin{pmatrix} 0.000292 \\ 9.96 \times 10^7 \\ 0 \end{pmatrix}.$$

Solving the system

$$\mathbf{J}(\mathbf{x}_0)\boldsymbol{\delta}_1 = -\mathbf{F}(\mathbf{x}_0),$$

via Gauss elimination gives

$$\boldsymbol{\delta}_1 = \begin{pmatrix} -6357.33535866 \\ -6357.33535866 \\ 45.4741297 \end{pmatrix},$$

which gives

$$\mathbf{x}_1 = \begin{pmatrix} 642.66464134 \\ 642.66464134 \\ 145.4741297 \end{pmatrix}.$$

Rather than continuing by hand, we will write a Python function to solve the problem. We will define a function inside of our Newton function to compute the finite difference Jacobian on the fly.

```
In [11]: #first import our Gauss Elim function
         from GaussElim import *
         def newton_system(f,x0,delta = 1.0e-7, epsilon=1.0e-6, LOUD=False):
             """Find the root of the function f via inexact Newton-Raphson method
             Args:
                 f: function to find root of
                 x0: initial guess
                 delta: finite difference parameter
                 epsilon: tolerance

             Returns:
                 estimate of root
             """
             def Jacobian(f,x,delta = 1.0e-7):
                 N = x0.size
                 J = np.zeros((N,N))
                 idelta = 1.0/delta
                 x_perturbed = x.copy() #copy x to add delta
                 fx = f(x) #only need to evaluate this once
                 for i in range(N):
                     x_perturbed[i] += delta
```

```
                col = (f(x_perturbed) - fx) * idelta
                x_perturbed[i] = x[i]
                J[:,i] = col
            return J

        x = x0
        if (LOUD):
            print("x0 =",x0)
        iterations = 0
        fx = f(x)
        while (np.linalg.norm(fx) > epsilon):
            J = Jacobian(f,x,delta)

            RHS = -fx;
            delta_x = GaussElimPivotSolve(J,RHS)
            x = x + delta_x
            fx = f(x)
            if (LOUD):
                print("Iteration",iterations+1,": x =",x," norm(f(x)) =",
                      np.linalg.norm(fx))
            iterations += 1
        print("It took",iterations,"iterations")
        return x #return estimate of root
```

To use this function we have to define the function we want to minimize:

```
In [12]: def Reactor(x, D=9.21, nuSigf = 0.1570, Siga = 0.1532):
             """This function is defined in the equation above
             """
             answer = np.zeros((3))
             answer[0] = (np.pi/(x[0] + 2*D))**2 +
                         (np.pi/(x[1] + 2*D))**2 +
                         (np.pi/(x[2] + 2*D))**2 - (nuSigf - Siga)/D
             answer[1] = 2*(x[0]*x[1] + x[1]*x[2] + x[0]*x[2])-1.2e6
             answer[2] = x[0] - x[1]
             return answer
```

We can now set up our initial guess and then solve:

```
In [13]: x0 = np.array([7000.0,7000.0,100.0])
         x = newton_system(Reactor,x0,LOUD=True, epsilon=1.0e-8, delta = 1.0e-10)
         #check
         print("The surface area is",2.0*(x[0]*x[1] + x[1]*x[2] + x[0]*x[2]))
         D=9.21; nuSigf = 0.1570; Siga = 0.1532;
         print("The geometric buckling is",(np.pi/(x[0] + 2*D))**2 +
               (np.pi/(x[1] + 2*D))**2 + (np.pi/(x[2] + 2*D))**2)
         print("The materials buckling is",(nuSigf - Siga)/D)

x0 = [ 7000.  7000.   100.]
Iteration 1 : x = [ 3457.80758198  3457.80758198   124.53337225]
norm(f(x)) = 24435316.3032
...
Iteration 9 : x = [ 642.66464134  642.66464134  145.4741297 ]
norm(f(x)) = 1.08420217249e-19
```

```
It took 9 iterations
The surface area is 1200000.0
The geometric buckling is 0.000412595005429
The materials buckling is 0.0004125950054288814
```

The solution is that the a and b are about 642.66 cm and the height of the reactor is 145.47 cm. There are multiple solutions to the problem. We can change the initial condition so that the method finds a reactor that is taller than this one by guessing a thin and tall parallelepiped.

```
In [14]: x0 = np.array([100.0,100.0,10000.0])
         x = newton_system(Reactor,x0,LOUD=True, epsilon=1.0e-8, delta = 1.0e-10)
         #check
         print("The surface area is",2.0*(x[0]*x[1] + x[1]*x[2] + x[0]*x[2]))
         D=9.21; nuSigf = 0.1570; Siga = 0.1532;
         print("The geometric buckling is",(np.pi/(x[0] + 2*D))**2 +
                (np.pi/(x[1] + 2*D))**2 + (np.pi/(x[2] + 2*D))**2)
         print("The materials buckling is",(nuSigf - Siga)/D)

x0 = [   100.    100.  10000.]
Iteration 1 : x = [  141.87143819   141.87143819 -1265.89519199]
norm(f(x)) = 1878122.47601
...
Iteration 8 : x = [  201.6439505    201.6439505   1386.94891624]
norm(f(x)) = 1.86264514923e-09
It took 8 iterations
The surface area is 1200000.0
The geometric buckling is 0.000412595005429
The materials buckling is 0.0004125950054288814
```

This reactor is almost 14 meters tall and has a geometric cross-section that is about 2 by 2 meters. The fact that there are multiple solutions to the problem is one of the features of nonlinear root finding to be aware of. One of these solutions may be better for a particular application, and picking a different initial guess will influence which root the method converges to. Furthermore, in the presence of multiple roots, an initial guess in the middle of the two roots may slow the convergence:

```
In [81]: x0 = np.array([421.0,421.0,750.0])
         x = newton_system(Reactor,x0,LOUD=True, epsilon=1.0e-8, delta = 1.0e-10)
         #check
         print("The surface area is",2.0*(x[0]*x[1] + x[1]*x[2] + x[0]*x[2]))
         D=9.21; nuSigf = 0.1570; Siga = 0.1532;
         print("The geometric buckling is",(np.pi/(x[0] + 2*D))**2 +
               (np.pi/(x[1] + 2*D))**2 + (np.pi/(x[2] + 2*D))**2)
         print("The materials buckling is",(nuSigf - Siga)/D)

x0 = [ 421.  421.  750.]
Iteration 1 : x = [ -403.19253367  -403.19253367  2792.77379015]
norm(f(x)) = 5378973.72321
...
Iteration 16 : x = [  201.6439505    201.6439505   1386.94891624]
norm(f(x)) = 4.65661287308e-10
```

```
It took 16 iterations
The surface area is 1200000.0
The geometric buckling is 0.000412595005429
The materials buckling is 0.0004125950054288814
```

This guess basically doubled the number of iterations.

CODA

In this, and the previous lecture, we have discussed a range of rootfinding methods. Open methods are good when you only have a single initial guess. These methods usually require either a knowledge of the functions's derivative or require an approximation to the derivative. Closed methods are more robust because the root estimate will always be in the initial bounds. This robustness comes at the cost of determining bounds for the root.

In practice, the appropriate method will be problem dependent. With the root finding tools we have covered, you have a rich toolset to find roots for just about any problem you will encounter.

PROBLEMS

Short Exercises

13.1. Apply Newton's method to the function $f(x) = (1-x)^2 + 100(1-x^2)^2 = 0$, using the initial guess of 2.5.

13.2. You are given a radioactive sample with an initial specific activity of 10^4 Bq/kg, and you are told the half-life is 19 days. Compute the time it will take to get the specific activity of Brazil nuts (444 Bq/kg) using Newton's method, inexact Newton, and secant.

Programming Projects

1. Roots of Bessel Function

Consider the Bessel function of the first kind defined by

$$J_\alpha(x) = \sum_{m=0}^{\infty} \frac{(-1)^m}{m!\,\Gamma(m+\alpha+1)} \left(\frac{x}{2}\right)^{2m+\alpha}.$$

13.1. Write a Python code that prompts the user asks if they want to use bisection or Newton's method. Then the user enters an initial guess or initial range depending on the method selected.

13.2. Using the input find a root to J_0. Each iteration print out to the user the value of $J_0(x)$ for the current guess and the change in the guess.

13.3. For testing, there is a root at $x \approx 2.4048$. Also, `math.gamma(x)`, will give you $\Gamma(x)$.

2. *Nonlinear Heat Conduction*

We will consider heat conduction in a cylindrical nuclear fuel pellet. In order to simplify the model, we will

- suppose a steady state operation,
- neglect the axial heat conduction,
- suppose that the heat is uniformly generated radially,
- neglect the presence of the clad.

Under these assumption, the heat conduction equation is:

$$\frac{1}{r}\frac{\partial}{\partial r}\left(rk(T)\frac{\partial T}{\partial r}\right) = -q''' \quad \text{for } 0 \le r \le R, \tag{13.1}$$

$$T(R) = T_R,$$

$$\left.\frac{\partial T}{\partial r}\right|_{r=0} = 0,$$

where $T = T(r)$ is the temperature inside the pellet, $k = k(T)$ is the temperature dependent conductivity, in W/(m·C), and q''' is the heat source (W/m^3). The temperature T_R is the temperature at the surface of the pellet. Solving for the temperature distribution within the pellet can be transformed into the following statement:

$$\int_{T_R}^{T(r)} k(T)\,dT = q'''\frac{R^2 - r^2}{4}. \tag{13.2}$$

Suppose you are given a formula for $k(T)$. You can then compute the conductivity integral (i.e., the antiderivative)

$$I(u) = \int k(u)\,du + C.$$

Finally, the problem boils down to solving the following nonlinear equation of one variable:

$$I(T(r)) = q'''\frac{R^2 - r^2}{4} + I(T_R).$$

If you solve the above equation at various radii r for $T(r)$, you will then get the temperature profile (i.e., the temperature at these different positions).

The data below (see Table 13.1) provides you with the conductivity formula $k(T)$ (which is easy to integrate), the pellet radius R, and the boundary condition T_R. The rest of the data will be useful to determine the average power density q''' (power per unit volume) for the entire core. The core is a typical Westinghouse PWR reactor, containing a given number of fuel assemblies. Each fuel assembly is loaded with a given number of fuel rods. Be careful with your unit conversions.

TABLE 13.1 Problem Definition

Conductivity, W/(m·C)	$k = 1.05 + 2150/(T + 200)$
Total power generated in the core	4200 MWth
Number of fuel assemblies (FA)	205
Number of fuel pins per FA	264
Core height	14 ft.
Pellet radius R	0.41 cm
Temperature at R	400°C

Your assignment

- Derive Eq. (13.2) using the heat conduction equation and it's boundary conditions.
- Write a clean and clear Python code to solve the above problem using the following methods: (1) bisection and (2) Newton's.
  - Use 11 grids points for the temperature profile (i.e., $r_i = \frac{i-1}{10} R$ for $i = 1 \ldots 11$).
  - Compare graphically your results with the case where the conductivity is assumed to be the following constant:

$$k = k(500°\text{C}) = 4.12\ \text{W}/(\text{m} \cdot \text{C}).$$

Note that when the conductivity is constant, you have an analytical solution. Provide the analytical solution $T^A(r)$.