CHAPTER

12

Closed Root Finding Methods

OUTLINE

We're Talking Root Down, I Put My Boot Down
And If You Want To Battle Me, You're Putting Loot Down

"Root Down" by the Beastie Boys

CHAPTER POINTS

- Finding the root of a nonlinear function requires an iterative method.
- The simplest and most robust method is bisection.
- Other methods can improve on bisection, but may be slower on some problems.

Up to this point we have only solved linear equations. In this chapter we solve nonlinear equations. Commonly, this process is called root finding because the problem is generally stated as determining a root of a function, i.e., find x so that

$$f(x) = 0.$$

The complication with nonlinear functions is that they may have many roots or not have any roots at all. For a single-variable linear function, $f(x) = a + bx$, we know that there is a single root, provided that $b \neq 0$.

A feature of nonlinear root finding is that it almost always requires an iterative method for finding the root: an initial guess is required and that guess is refined until the answer is *good enough*. In this chapter we will treat iterative methods for finding roots that start with an

DOI: 10.1016/B978-0-12-812253-2.00014-5

initial interval that we know the root is in, and then shrink that interval until we know the root is in a very small interval. These methods are called closed root finding methods because the root is known to be inside a closed interval.

12.1 BISECTION

The first closed method we use is the bisection algorithm. As with any closed method, to use it we need to first bracket the root. That is we need to find two points a and b where

$$f(a)f(b) < 0,$$

that is two points where $f(x)$ has a different sign. Therefore, we know by the intermediate value theorem that $f(x) = 0$ is in $[a, b]$, provided $f(x)$ is continuous in $[a, b]$.

Once we have two points a and b, we then pick the midpoint of the interval and call it c:

$$c = \frac{a+b}{2}.$$

Then we can determine which side of the interval root is on by comparing the sign of $f(c)$ to the sign of the function at the endpoints. Once we determine which side of the midpoint the root was on, we can change our interval to be that half. Algorithmically,

$$\text{if} \quad f(a)f(c) < 0 \qquad \text{then we set } b = c$$

because the root is between a and c. Otherwise,

$$\text{if} \quad f(b)f(c) < 0 \qquad \text{then we set } a = c$$

because the root is between c and b. Of course, if $f(a)f(c) = 0$, then c is the root, the probability of this happening is vanishingly small, however. If the function has multiple roots, it is possible that both $f(a)f(c) < 0$ and $f(b)f(c) < 0$. In this case we could take either half of the interval and still find a root. The result of this procedure is that the new interval is half the size of the previous interval: we can think of the result is that we have improved our estimate of the root by a factor of 2.

We can repeat this process of computing the sign of the function at the midpoint and shrinking the interval by a factor of 2 until the range is small enough that we can say we are done. Define the width of an interval after iteration n as $\Delta x_n = b - a$, with the initial interval width written as Δx_0. Using this definition and the fact that each iteration cuts the interval in half, we know that after n iterations the width of the interval is

$$\Delta x_n = 2^{-n}\Delta x_0.$$

If we want to know the root within a tolerance ϵ, then we can solve for n in the equation

$$\epsilon = 2^{-n}\Delta x_0,$$

which implies

$$n = \frac{\log(\Delta x_0/\epsilon)}{\log 2}.$$

BOX 12.1 NUMERICAL PRINCIPLE

To find the root of a nonlinear function almost always requires an iterative method. Bisection is the simplest method, but it is also very robust and almost always guaranteed to converge to the root.

Below is an implementation of the bisection algorithm in Python. As you can see, it is a simple algorithm.

```
In [1]: def bisection(f,a,b,epsilon=1.0e-6):
            """Find the root of the function f via bisection
            where the root lies within [a,b]
            Args:
                f: function to find root of
                a: left-side of interval
                b: right-side of interval
                epsilon: tolerance

            Returns:
                estimate of root
            """
            assert (b>a)
            fa = f(a)
            fb = f(b)
            assert (fa*fb < 0)
            delta = b - a
            print("We expect",
                  int(np.ceil(np.log(delta/epsilon)/np.log(2))),"iterations")
            iterations = 0
            while (delta > epsilon):
                c = (a+b)*0.5
                fc = f(c)
                if (fa*fc < 0):
                    b = c
                    fb = fc
                elif (fb*fc < 0):
                    a = c
                    fa = fc
                else:
                    return c
                delta = b-a
                iterations += 1
            print("It took",iterations,"iterations")
            return c #return midpoint of interval
```

Notice that we save the value of the function evaluations so that we only evaluate the function at a given point once. This will be important if it takes a long time to evaluate the function. For example, if the function evaluation involves the solution of a system of equations, as done in one of the exercises for this chapter, one does not want to be evaluating the function more times than necessary.

Below we test the bisection function with a simple cubic, to find a single root:

```
In [2]: def nonlinear_function(x):
            #compute a nonlinear function for demonstration
            return 3*x**3 + 2*x**2 - 5*x-20
        root = bisection(nonlinear_function,1,2)
        print("The root estimate is",root,"\nf(",root,
              ") =",nonlinear_function(root))

We expect 20 iterations
It took 20 iterations
The root estimate is 1.9473047256469727
f( 1.9473047256469727 ) = -1.883242441280686e-05
```

Bisection is a useful algorithm because it is really easy to implement, and we know how long it should take to converge. Also, we know that it will converge. We can give it a really big range and as long as the root is in the range, it will find it. In this case we increase the initial interval from a width of 3 to 7. Bisection has no trouble finding the root.

```
In [3]: root = bisection(nonlinear_function,-5,2)
        print("The root estimate is",root,"\nf(",root,
             ") =",nonlinear_function(root))

We expect 23 iterations
It took 23 iterations
The root estimate is 1.9473060369491577
f( 1.9473060369491577 ) = 2.9577188165319512e-05
```

It did take more iterations, but with this larger interval, bisection still arrived at the answer.

12.1.1 Critical Radius of a Sphere

An example problem that we will use to compare root finding methods is that of finding the critical radius of a bare sphere reactor. From elementary nuclear reactor theory [7], one can show that for a critical, spherical reactor

$$\left(\frac{\pi}{R+2D}\right)^2 = \frac{\nu\Sigma_f - \Sigma_a}{D},$$

where D [cm] is the diffusion coefficient, ν is the number of neutrons born per fission, Σ_f [cm^{-1}] is the fission macroscopic cross-section, Σ_a [cm^{-1}] is the macroscopic cross-section

for absorption, and R [cm] is the radius of the reactor. Therefore, given D, $\nu\Sigma_f$, and Σ_a we can compute the critical radius. To use bisection we need to define the function we want to be zero:

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

We will solve the problem with $D = 9.21$ cm, $\nu\Sigma_f = 0.1570$ cm^{-1}, and $\Sigma_a = 0.1532$ cm^{-1}.

For the initial interval we will pick $a = 0$ and $b = 250$ cm, because $f(R)$ has different signs at these points. In the code below we define a simple function and pass it to our bisection algorithm. Notice that this function has all the arguments except for R have default arguments. This is because the bisection algorithm we defined earlier operates on a function of only a single variable.

The bisection algorithm finds the critical radius in 28 iterations.

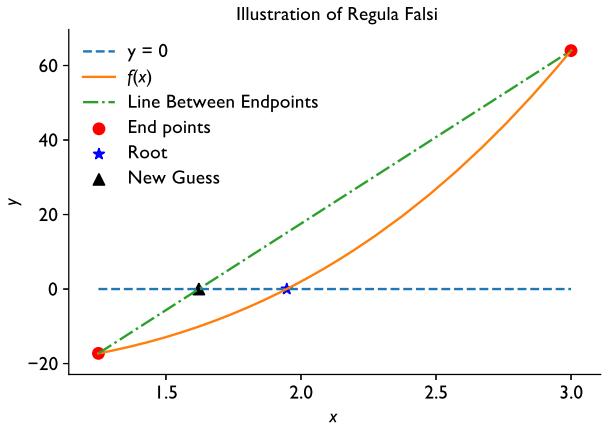
```
In [4]: #first define the function
        def Crit_Radius(R, D=9.21, nuSigf = 0.1570, Siga = 0.1532):
            return (np.pi/(R + 2*D))**2 - (nuSigf - Siga)/D
        a = 0
        b = 250
        Radius = bisection(Crit_Radius,a,b)
        print("The critical radius estimate is",Radius,
              "\nf(",Radius,") =",Crit_Radius(Radius))
```

```
We expect 28 iterations
It took 28 iterations
The critical radius estimate is 136.2435193732381
f( 136.2435193732381 ) = 2.175801910603986e-12
```

This will be the baseline for comparison as we examine new closed root finding methods.

12.2 FALSE POSITION (REGULA FALSI)

Another closed root finding method is the false position method (also called regula falsi, if one prefers Latin names). This method draws a line between the two endpoints of the interval $[a, b]$ and uses the position where that line intersects the x axis as the value of c (the guess for the new endpoint). Below is a graphical example:



The false position methods works in almost the exact same way as bisection except that c is not directly in the middle of the interval, rather it is where the interpolating line intersects the x axis. The reason this may be a good idea, is that if the function is linear (or near linear) in the interval, this value of c will be the root.

To derive false position, we first define the slope between the two endpoints:

$$m \equiv \frac{f(b) - f(a)}{b - a}.$$

For a line the slope is the same everywhere so we know that

$$\frac{f(b) - f(a)}{b - a} = \frac{f(c) - f(a)}{c - a}.$$

We also want $f(c)$ to be zero, so this simplifies to

$$\frac{f(b) - f(a)}{b - a} = \frac{-f(a)}{c - a}.$$

Solving for c gives

$$c = a - \frac{f(a)}{m}.$$

We have to use a different convergence criterion than we did for bisection as the interval size is not the best measure in false position because after each iteration our guess is c. In this case we will use $|f(c)| < \epsilon$ for our convergence criteria.

A Python implementation of this algorithm is below.

```
In [5]:  def false_position(f,a,b,epsilon=1.0e-6):
             """Find the root of the function f via false position
             where the root lies within [a,b]
             Args:
                 f: function to find root of
                 a: left-side of interval
```

```
        b: right-side of interval
        epsilon: tolerance

    Returns:
        estimate of root
    """
    assert (b>a)
    fa = f(a)
    fb = f(b)
    assert (fa*fb< 0)
    delta = b - a
    iterations = 0
    residual = 1.0
    while (np.fabs(residual) > epsilon):
        m = (fb-fa)/(b-a)
        c = a - fa/m
        fc = f(c)
        if (fa*fc < 0):
            b = c
            fb = fc
        elif (fb*fc < 0):
            a = c
            fa = fc
        else:
            print("It took",iterations,"iterations")
            return c
        residual = fc
        iterations += 1
    print("It took",iterations,"iterations")
    return c #return c
```

We will apply false position to the cubic function we defined above:

```
In [6]:  root = false_position(nonlinear_function,1,2)
         print("The root estimate is",root,"\nf(",root,
               ") =",nonlinear_function(root))

It took 5 iterations
The root estimate is 1.9473052141477751
f( 1.9473052141477751 ) = -7.983476244532994e-07
```

This is faster than bisection on this short interval because bisection took 20 iterations. This factor of 4 improvement is impressive. If we give it a bigger interval, it is still faster than bisection by a factor of 3.

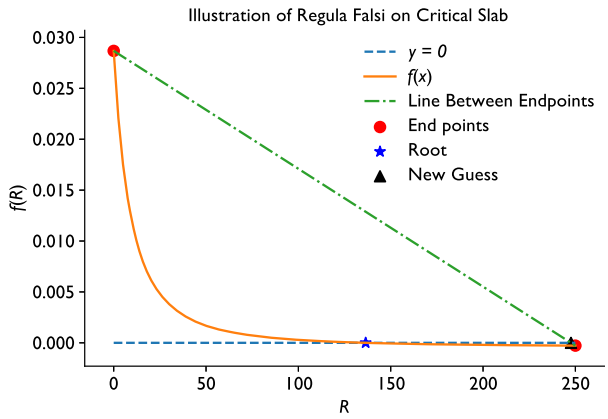
```
In [7]:  root = false_position(nonlinear_function,-5,2)
         print("The root estimate is",root,"\nf(",root,") =",
               nonlinear_function(root))

It took 9 iterations
The root estimate is 1.947305257431454
f( 1.947305257431454 ) = 7.995646100766862e-07
```

Finally, we apply it on the critical slab case (bisection took 28 iterations for this problem and initial interval).

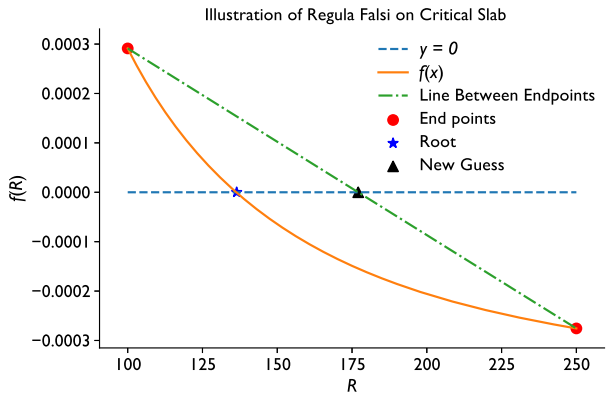
```
In [8]:  a = 0
         b = 250
         Radius = false_position(Crit_Radius,a,b)
         print("The critical radius estimate is",Radius,"\nf(",Radius,
               ") =",Crit_Radius(Radius))
```

```
It took 260 iterations
The critical radius estimate is 136.42803805393132
f( 136.42803805393132 ) = -9.827177585040653e-07
```

What happened? False position was much faster on the cubic function, but is now 9 times slower than bisection. It is important to note that it still converged (that's good), but it converged slowly (that's bad). To see why, we can look at the first iteration by plotting the function and the false position update:



Notice how the function is very steep near 0 and much less steep to the right of the root. This makes the value of c very close to b. Since the right endpoint moves so slowly, it takes a long time to converge. If we gave it a better initial range, it should work better. With the minimum value of the initial interval shifted to $R = 100$ we get:



```
In [9]:  a = 100
         b = 250
         Radius = false_position(Crit_Radius,a,b)
         print("The critical radius estimate is",Radius,"\nf(",Radius,
               ") =",Crit_Radius(Radius))


It took 6 iterations
The critical radius estimate is 136.42658989359794
f( 136.42658989359794 ) = -9.750187400692396e-07
```

This demonstrates that when we tried to get better than bisection we sometimes are slower. Bisection is the tank of nonlinear solvers: it is slow, but it will get there. Regula falsi can be faster than the tank, but it can also be slowed down in the mud. There is a lot of mud in the realm of solving nonlinear equations.

BOX 12.2 NUMERICAL PRINCIPLE

Faster methods are not always better in terms of robustness. False position can be much faster than bisection, but it also can be much slower than bisection in certain cases.

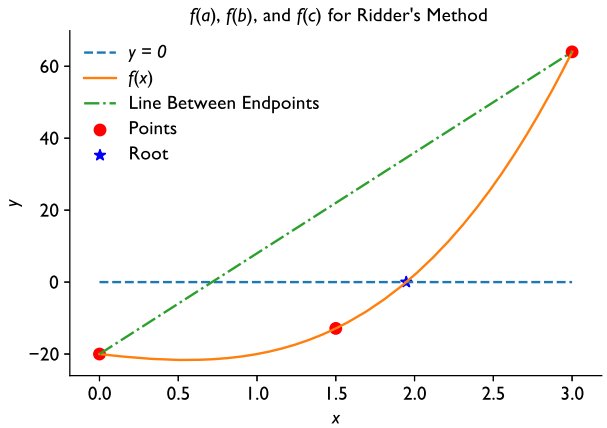
12.3 RIDDER'S METHOD

An obvious way to quantify the speed of a rootfinding method is the number of iterations. This, however, is not a perfect comparison between methods because the number of function evaluations per iteration may differ between methods. For both bisection and false position, the number of additional function evaluations needed is one, at $x = c$. This will not always be the case. The next method requires an extra function evaluation in each iteration.

BOX 12.3 NUMERICAL PRINCIPLE

An appropriate measure of a root finding methods speed is the number of function evaluations. This is the case because the function in many applications may be difficult to compute and might require the solution of a linear system of equations.

In Ridder's method we interpolate between a and b in a way that is different than linear interpolation. In particular, given points a and b which bracket the root, we compute the values $f(a)$, $f(b)$, $f(c)$ where c is the midpoint of a and b.



Then we use these function values to estimate the function:

$$g(x) = f(x)e^{(x-a)Q}.$$

The function $g(x)$ will only be zero when $f(x)$ is zero. Furthermore, it touches the original function at a, and not at b. To determine the value of Q in $g(x)$ we require that $g(a)$, $g(b)$, and $g(c)$ all lie on a line (or they are collinear). To do this we first look at each of these values:

$$g(a) = f(a), \qquad g(b) = f(b)e^{2hQ}, \qquad g(c) = f(c)e^{hQ},$$

where $h = c - a$. If these three points are to lie on a straight line then,

$$g(c) = \frac{g(a) + g(b)}{2},$$

or

$$f(c)e^{hQ} = \frac{f(a) + f(b)e^{2hQ}}{2}.$$

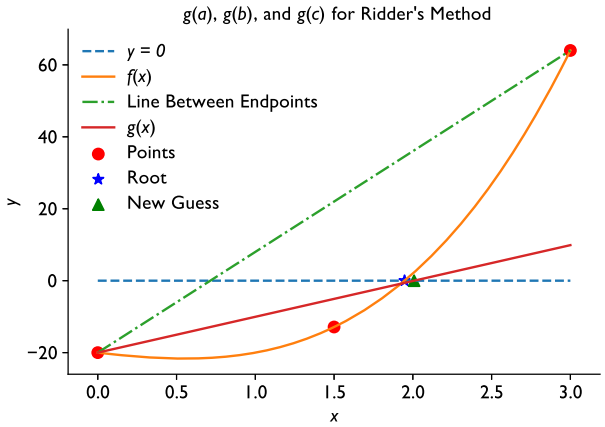
Solving this equation for e^{hQ} we get

$$e^{hQ} = \frac{f(c) \pm \sqrt{f(c)^2 - f(a)f(b)}}{f(b)}.$$

The root we choose is the "+" root if $f(a) - f(b) > 0$ and the "−" root otherwise. Once we have our $g(x)$ function, we then linearly interpolate from $(a, g(a))$ and $(c, g(c))$ to find where $g(d) = 0$ and use this as the next guess at the root:

$$d = c \pm \frac{(c-a)f(c)}{\sqrt{f(c)^2 - f(a)f(b)}}.$$

The following figures shows this procedure graphically.



As you can see the new guess is really close. This appears to be much better than false position. This is one of the benefits of performing interpolation on the exponentially varying function $g(x)$ rather than using the original function $f(x)$.

The general algorithm for Ridder's method is below.

```
In [10]: def ridder(f,a,b,epsilon=1.0e-6):
             """Find the root of the function f via Ridder's Method
             where the root lies within [a,b]
             Args:
                 f: function to find root of
                 a: left-side of interval
                 b: right-side of interval
                 epsilon: tolerance

             Returns:
                 estimate of root
             """
             assert (b>a)
             fa = f(a)
             fb = f(b)
             assert (fa*fb < 0)
             delta = b - a
             iterations = 0
             residual = 1.0
             while (np.fabs(residual) > epsilon):
                 c = 0.5*(b+a)
                 d = 0.0
                 fc = f(c)
                 if (fa - fb > 0):
                     d = c + (c-a)*fc/np.sqrt(fc**2-fa*fb)
                 else:
                     d = c - (c-a)*fc/np.sqrt(fc**2-fa*fb)
                 fd = f(d)
                 #now see which part of interval root is in
                 if (fa*fd < 0):
                     b = d
```

```
            fb = fd
        elif (fb*fd < 0):
            a = d
            fa = fd
        residual = fd
        iterations += 1
    print("It took",iterations,"iterations")
    return d #return c
```

On the example using a cubic function from before Ridder performs the best of the methods we have seen (bisection took 23 iterations and false position took 9):

```
In [20]: root = ridder(nonlinear_function,-5,2)
         print("The root estimate is",root,"\nf(",root,
               ") =",nonlinear_function(root))

It took 7 iterations
The root estimate is 1.94730524023
f( 1.94730524023 ) = 1.64623202181e-07
```

Now we try our criticality problem (the one that false position required 260 iterations and bisection took 28). This is where Ridder's method shines: it can handle the large change in our function's behavior.

```
In [21]: a = 0
         b = 250
         Radius = ridder(Crit_Radius,a,b)
         print("The critical radius estimate is",Radius,"\nf(",Radius,
               ") =",Crit_Radius(Radius))

It took 3 iterations
The critical radius estimate is 136.242306777
f( 136.242306777 ) = 6.47192578831e-09
```

That was much faster than either of the previous methods.

The total cost of Ridder's method per iteration is one function evaluation more per iteration than false position or bisection: we need to evaluate the function at the guess from the previous iteration, $f(d)$, and at the midpoint of the new interval. Therefore, to be more efficient than false position or bisection, the number of iterations needs to be 2 times fewer than for either of those methods. In the example above Ridder's method met this hurdle, except in the case where Ridder had 7 iterations and false position had 9 (to have the same number of function evaluations Ridder's method needed 4.5 iterations). This slight miss is mitigated by the fact that Ridder's method seems much more robust than false position.

CODA

We have reviewed root finding methods that work by bracketing the root and then zooming in on the root either simply, as in bisection, or through interpolation, as in false position

or Ridder's method. One observation we can make at this point is that the methods all converged to the root, even though they could be slow.

In the next chapter we will cover open root finding methods that require only a single initial guess and do not bracket a root. These methods can converge quickly, though they generally require information about the slope of the function.

FURTHER READING

There are additional closed root finding methods that we have not discussed. A popular one is Brent's method the details of this method can be found in Atkinson's text [18], among others.

PROBLEMS

Short Exercises

12.1. Find a root of $\cos x$ using the three methods discussed in this section and an initial interval of $[0, 10]$. Compare the solutions and the number of iterations to find the root for each method.

12.2. 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 the three methods specified above.

Programming Projects

1. 2-D Heat Equation Optimization

Previously, in Algorithm 9.5 we gave code to solve the heat equation:

$$-k\nabla^2 T = q, \qquad \text{for } x \in [0, L_x] \quad y \in [0, L_y].$$

With the boundary condition

$$T(x, y) = 0 \qquad \text{for } x, y \text{ on the boundary.}$$

Your have been tasked to determine what value of k will make the maximum temperature equal to 3 when $L_x = L_y = 1$ and the source, q, is given by

$$q = \begin{cases} 1 & 0.25 \le x \le 0.75 \qquad 0.25 \le y \le 0.75 \\ 0 & \text{otherwise} \end{cases}.$$

Your particular tasks are as follows:

- Define a function called `max_temperature` that finds the maximum value of $T(x)$ in the domain. This function will take as it's only argument k. Inside the function solve the heat equation with $\Delta x = \Delta y = 0.025$. The function `np.max` will be helpful here.
- Find the value of k for which the max temperature equals 3 using bisection, false position, and Ridder's method. Use an initial interval of $k \in [0.001, 0.01]$. Remember that the root-finding methods find when a function is equal to 0. You will have to define a function that is equal to 0 when the maximum temperature is equal to 3. How many iterations does each method take?
- The Python package, `time`, has a function `time.clock()` that returns the system time. Using this function time how long it takes for each method to find the value of k that makes the maximum temperature equal to 3. Which method is the fastest?

This problem will demonstrate why it is important to be parsimonious with the number of function evaluations.

2. Peak Xenon Time

In Programming Project 1 from Chapter 7 you determined the equilibrium concentrations of ^{135}Xe, ^{135}I and ^{135}Te. After shutdown of the reactor, the ^{135}Xe concentration, $X(t)$ can be written as

$$X(t) = X_0 e^{-\lambda_X t} + \frac{\lambda_I}{\lambda_I - \lambda_X} I_0 \left(e^{-\lambda_X t} - e^{-\lambda_I t} \right),$$

where t is the time since shutdown, X_0 and I_0 are the equilibrium concentrations of ^{135}Xe and ^{135}I during reactor operation, and λ_X and λ_I are the decay constants for ^{135}Xe and ^{135}I.

Using a root finding method of your choice, compute how long after shutdown the maximum concentration of ^{135}Xe is reached, as well as the value of the concentration at those times for power densities of 5, 50, and 100 W/cm^3.