4 Roots of Equations

Find the solutions of $f(x) = 0$, where the function f is given

4.1 Introduction

A common problem encountered in engineering analysis is this: given a function $f(x)$, determine the values of x for which $f(x) = 0$. The solutions (values of x) are known as the *roots* of the equation $f(x) = 0$, or the *zeroes* of the function $f(x)$.

Before proceeding further, it might be helpful to review the concept of a *function*. The equation

$$y = f(x)$$

contains three elements: an input value x, an output value y, and the rule f for computing y. The function is said to be given if the rule f is specified. In numerical computing the rule is invariably a computer algorithm. It may be a function statement, such as

$$f(x) = \cosh(x)\cos(x) - 1$$

or a complex procedure containing hundreds or thousands of lines of code. As long as the algorithm produces an output y for each input x, it qualifies as a function.

The roots of equations may be real or complex. The complex roots are seldom computed, because they rarely have physical significance. An exception is the polynomial equation

$$a_0 + a_1 x + a_1 x^2 + \ldots + a_n x^n = 0$$

where the complex roots may be meaningful (as in the analysis of damped vibrations, for example). For the time being, we concentrate on finding the real roots of equations. Complex zeroes of polynomials are treated near the end of this chapter.

In general, an equation may have any number of (real) roots, or no roots at all. For example,

$$\sin x - x = 0$$

has a single root, namely, $x = 0$, whereas

$$\tan x - x = 0$$

has an infinite number of roots ($x = 0, \pm 4.493, \pm 7.725, \ldots$).

All methods of finding roots are iterative procedures that require a starting point, that is, an estimate of the root. This estimate can be crucial; a bad starting value may fail to converge, or it may converge to the "wrong" root (a root different from the one sought). There is no universal recipe for estimating the value of a root. If the equation is associated with a physical problem, then the context of the problem (physical insight) might suggest the approximate location of the root. Otherwise, a systematic numerical search for the roots can be carried out. One such search method is described in the next section. Plotting the function is another means of locating the roots, but it is a visual procedure that cannot be programmed.

It is highly advisable to go a step further and *bracket* the root (determine its lower and upper bounds) before passing the problem to a root-finding algorithm. Prior bracketing is, in fact, mandatory in the methods described in this chapter.

4.2 Incremental Search Method

The approximate locations of the roots are best determined by plotting the function. Often a very rough plot, based on a few points, is sufficient to give us reasonable starting values. Another useful tool for detecting and bracketing roots is the incremental search method. It can also be adapted for computing roots, but the effort would not be worthwhile, because other methods described in this chapter are more efficient for that.

The basic idea behind the incremental search method is simple: If $f(x_1)$ and $f(x_2)$ have opposite signs, then there is at least one root in the interval (x_1, x_2). If the interval is small enough, it is likely to contain a single root. Thus, the zeroes of $f(x)$ can be detected by evaluating the function at intervals Δx and looking for change in sign.

There are a couple of potential problems with the incremental search method:

- It is possible to miss two closely spaced roots if the search increment Δx is larger than the spacing of the roots.
- A double root (two roots that coincide) will not be detected.
- Certain singularities (poles) of $f(x)$ can be mistaken for roots. For example, $f(x) = \tan x$ changes sign at $x = \pm\frac{1}{2}n\pi$, $n = 1, 3, 5, \ldots$, as shown in Fig. 4.1. However, these locations are not true zeroes, since the function does not cross the x-axis.

■ `rootsearch`

This function searches for a zero of the user-supplied function *f(x)* in the interval `(a,b)` in increments of `dx`. It returns the bounds `(x1,x2)`of the root if the search

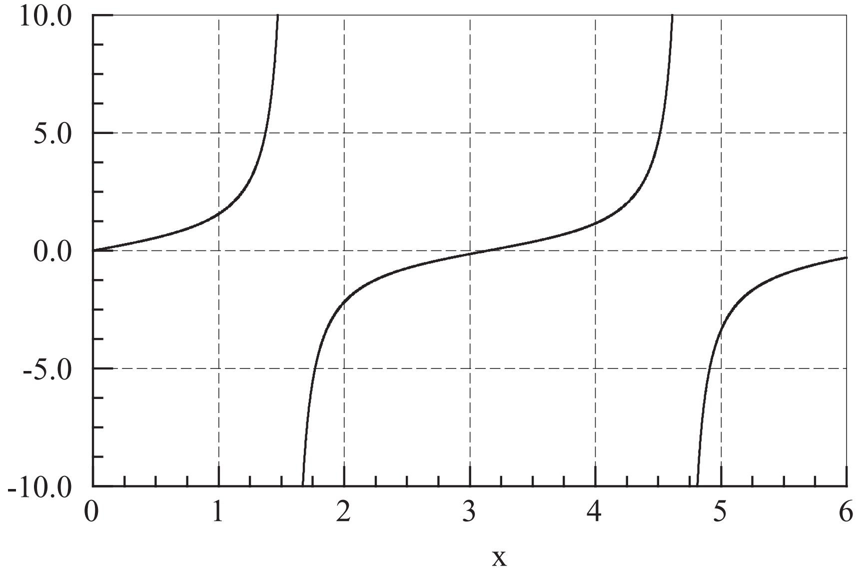


Figure 4.1. Plot of tan x.

was successful; `x1 = x2 = None` indicates that no roots were detected. After the first root (the root closest to `a`) has been detected, `rootsearch` can be called again with `a` replaced by `x2` in order to find the next root. This can be repeated as long as `rootsearch` detects a root.

```
## module rootsearch
''' x1,x2 = rootsearch(f,a,b,dx).
    Searches the interval (a,b) in increments dx for
    the bounds (x1,x2) of the smallest root of f(x).
    Returns x1 = x2 = None if no roots were detected.
'''
def rootsearch(f,a,b,dx):
    x1 = a; f1 = f(a)
    x2 = a + dx; f2 = f(x2)
    while f1*f2 > 0.0:
        if x1  >=  b: return None,None
        x1 = x2; f1 = f2
        x2 = x1 + dx; f2 = f(x2)
    else:
        return x1,x2
```

EXAMPLE 4.1

Use incremental search with $\Delta x = 0.2$ to bracket the smallest positive zero of $f(x) = x^3 - 10x^2 + 5$.

Solution We evaluate $f(x)$ at intervals $\Delta x = 0.2$, staring at $x = 0$, until the function changes its sign (the value of the function is of no interest to us, only its sign is relevant). This procedure yields the following results:

x	$f(x)$
0.0	5.000
0.2	4.608
0.4	3.464
0.6	1.616
0.8	−0.888

From the sign change of the function, we conclude that the smallest positive zero lies between $x = 0.6$ and $x = 0.8$.

4.3 Method of Bisection

After a root of $f(x) = 0$ has been bracketed in the interval (x_1, x_2), several methods can be used to close in on it. The method of bisection accomplishes this by successively halving the interval until it becomes sufficiently small. This technique is also known as the *interval halving method*. Bisection is not the fastest method available for computing roots, but it is the most reliable. Once a root has been bracketed, bisection will always close in on it.

The method of bisection uses the same principle as incremental search: If there is a root in the interval (x_1, x_2), then $f(x_1) \cdot f(x_2) < 0$. In order to halve the interval, we compute $f(x_3)$, where $x_3 = \frac{1}{2}(x_1 + x_2)$ is the midpoint of the interval. If $f(x_2) \cdot f(x_3) < 0$, then the root must be in (x_2, x_3), and we record this by replacing the original bound x_1 by x_3. Otherwise, the root lies in (x_1, x_3), in which case x_2 is replaced by x_3. In either case, the new interval (x_1, x_2) is half the size of the original interval. The bisection is repeated until the interval has been reduced to a small value ε, so that

$$|x_2 - x_1| \leq \varepsilon$$

It is easy to compute the number of bisections required to reach a prescribed ε. The original interval Δx is reduced to $\Delta x/2$ after one bisection, to $\Delta x/2^2$ after two bisections, and after n bisections it is $\Delta x/2^n$. Setting $\Delta x/2^n = \varepsilon$ and solving for n, we get

$$n = \frac{\ln(|\Delta x|/\varepsilon)}{\ln 2} \tag{4.1}$$

■ `bisection`

This function uses the method of bisection to compute the root of `f(x) = 0` that is known to lie in the interval `(x1,x2)`. The number of bisections `n` required to reduce

 https://doi.org/10.1017/CBO9780511812224.006

the interval to `tol` is computed from Eq. (4.1). By setting `switch = 1`, we force the routine to check whether the magnitude of `f(x)` decreases with each interval halving. If it does not, something may be wrong (probably the “root” is not a root at all, but a pole) and `root = None` is returned. Because this feature is not always desirable, the default value is `switch = 0`. The function `error.err`, which we use to terminate a program, is listed in Section 1.7.

```
## module bisection
''' root = bisection(f,x1,x2,switch=0,tol=1.0e-9).
    Finds a root of f(x) = 0 by bisection.
    The root must be bracketed in (x1,x2).
    Setting switch = 1 returns root = None if
    f(x) increases upon bisection.
'''
from math import log,ceil
import error

def bisection(f,x1,x2,switch=1,tol=1.0e-9):
    f1 = f(x1)
    if f1 == 0.0: return x1
    f2 = f(x2)
    if f2 == 0.0: return x2
    if f1*f2 > 0.0: error.err('Root is not bracketed')
    n = ceil(log(abs(x2 - x1)/tol)/log(2.0))
    for i in range(n):
        x3 = 0.5*(x1 + x2); f3 = f(x3)
        if (switch == 1) and (abs(f3) > abs(f1)) \
                         and (abs(f3) > abs(f2)):
            return None
        if f3 == 0.0: return x3
        if f2*f3 < 0.0: x1 = x3; f1 = f3
        else:           x2 = x3; f2 = f3
    return (x1 + x2)/2.0
```

EXAMPLE 4.2
Use bisection to find the root of $f(x) = x^3 - 10x^2 + 5 = 0$ that lies in the interval $(0.6, 0.8)$.

Solution The best way to implement the method is to use the following table. Note that the interval to be bisected is determined by the sign of $f(x)$, not its magnitude.

x	$f(x)$	Interval
0.6	1.616	–
0.8	−0.888	(0.6, 0.8)
$(0.6 + 0.8)/2 = 0.7$	0.443	(0.7, 0.8)
$(0.8 + 0.7)/2 = 0.75$	−0.203	(0.7, 0.75)
$(0.7 + 0.75)/2 = 0.725$	0.125	(0.725, 0.75)
$(0.75 + 0.725)/2 = 0.7375$	−0.038	(0.725, 0.7375)
$(0.725 + 0.7375)/2 = 0.73125$	0.044	(0.7375, 0.73125)
$(0.7375 + 0.73125)/2 = 0.73438$	0.003	(0.7375, 0.73438)
$(0.7375 + 0.73438)/2 = 0.73594$	−0.017	(0.73438, 0.73594)
$(0.73438 + 0.73594)/2 = 0.73516$	−0.007	(0.73438, 0.73516)
$(0.73438 + 0.73516)/2 = 0.73477$	−0.002	(0.73438, 0.73477)
$(0.73438 + 0.73477)/2 = 0.73458$	0.000	–

The final result $x = 0.7346$ is correct within four decimal places.

EXAMPLE 4.3

Find *all* the zeroes of $f(x) = x - \tan x$ in the interval (0, 20) by the method of bisection. Utilize the functions `rootsearch` and `bisection`.

Solution Note that $\tan x$ is singular and changes sign at $x = \pi/2, 3\pi/2, \ldots$. To prevent `bisection` from mistaking these point for roots, we set `switch` = `1`. The closeness of roots to the singularities is another potential problem that can be alleviated by using small Δx in rootsearch. Choosing $\Delta x = 0.01$, we arrive at the following program:

```
#!/usr/bin/python
## example4_3
from math import tan
from rootsearch import *
from bisection import *

def f(x): return x - tan(x)

a,b,dx = (0.0, 20.0, 0.01)
print "The roots are:"
while 1:
    x1,x2 = rootsearch(f,a,b,dx)
    if x1 != None:
        a = x2
        root = bisection(f,x1,x2,1)
        if root != None: print root
    else:
        print "\nDone"
        break
raw_input("Press return to exit")
```

The output from the program is:

```
The roots are:
0.0
4.4934094581
7.72525183707
10.9041216597
14.0661939129
17.2207552722
Done
```

4.4 Methods Based on Linear Interpolation

Secant and False Position Methods

The secant and the false position methods are closely related. Both methods require two starting estimates of the root, say, x_1 and x_2. The function $f(x)$ is assumed to be approximately linear near the root, so that the improved value x_3 of the root can be estimated by linear interpolation between x_1 and x_2.
Referring to Fig. 4.2, we obtain from similar triangles (shaded in the figure)

$$\frac{f_2}{x_3 - x_2} = \frac{f_1 - f_2}{x_2 - x_1}$$

where we used the notation $f_i = f(x_i)$. This yields for the improved estimate of the root

$$x_3 = x_2 - f_2 \frac{x_2 - x_1}{f_2 - f_1} \tag{4.2}$$

The false position method (also known as *regula falsi*) requires x_1 and x_2 to bracket the root. After the improved root is computed from Eq. (4.2), either x_1 or x_2 is replaced by x_3. If f_3 has the same sign as f_1, we let $x_1 \leftarrow x_3$; otherwise, we choose $x_2 \leftarrow x_3$. In this manner, the root is always bracketed in (x_1, x_2). The procedure is then repeated until convergence is obtained.

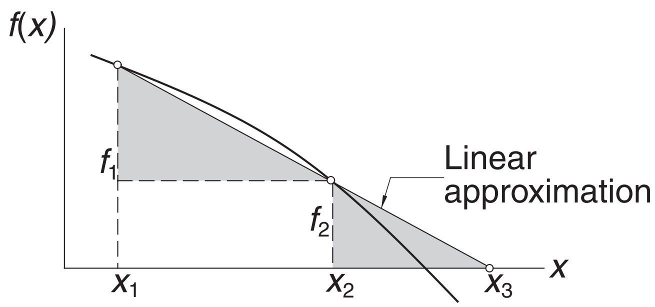


Figure 4.2. Linear interpolation.

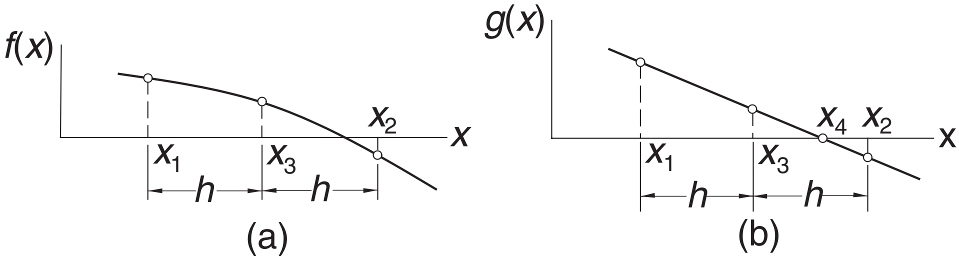


Figure 4.3. Mapping used in Ridder's method.

The secant method differs from the false-position method in two details: (1) It does not require prior bracketing of the root; and (2) the oldest prior estimate of the root is discarded, that is, after x_3 is computed, we let $x_1 \leftarrow x_2$, $x_2 \leftarrow x_3$.

The convergence of the secant method can be shown to be superlinear, the error behaving as $E_{k+1} = cE_k^{1.618...}$ (the exponent 1.618... is the "golden ratio"). The precise order of convergence for the false position method is impossible to calculate. Generally, it is somewhat better than linear, but not by much. However, because the false position method always brackets the root, it is more reliable. We will not delve further into these methods, because both of them are inferior to Ridder's method as far as the order of convergence is concerned.

Ridder's Method

Ridder's method is a clever modification of the false position method. Assuming that the root is bracketed in (x_1, x_2), we first compute $f_3 = f(x_3)$, where x_3 is the midpoint of the bracket, as indicated in Fig. 4.3(a). Next, we the introduce the function

$$g(x) = f(x)e^{(x-x_1)Q} \tag{a}$$

where the constant Q is determined by requiring the points (x_1, g_1), (x_2, g_2), and (x_3, g_3) to lie on a straight line, as shown in Fig. 4.3(b). As before, the notation we use is $g_i = g(x_i)$. The improved value of the root is then obtained by linear interpolation of $g(x)$ rather than $f(x)$.

Let us now look at the details. From Eq. (a) we obtain

$$g_1 = f_1 \qquad g_2 = f_2 e^{2hQ} \qquad g_3 = f_3 e^{hQ} \tag{b}$$

where $h = (x_2 - x_1)/2$. The requirement that the three points in Fig. 4.3b lie on a straight line is $g_3 = (g_1 + g_2)/2$, or

$$f_3 e^{hQ} = \frac{1}{2}(f_1 + f_2 e^{2hQ})$$

which is a quadratic equation in e^{hQ}. The solution is

$$e^{hQ} = \frac{f_3 \pm \sqrt{f_3^2 - f_1 f_2}}{f_2} \tag{c}$$

Linear interpolation based on points (x_1, g_1) and (x_3, g_3) now yields for the improved root

$$x_4 = x_3 - g_3 \frac{x_3 - x_1}{g_3 - g_1} = x_3 - f_3 e^{hQ} \frac{x_3 - x_1}{f_3 e^{hQ} - f_1}$$

where in the last step we utilized Eqs. (b). As the final step, we substitute for e^{hQ} from Eq. (c) and obtain after some algebra

$$x_4 = x_3 \pm (x_3 - x_1) \frac{f_3}{\sqrt{f_3^2 - f_1 f_2}} \tag{4.3}$$

It can be shown that the correct result is obtained by choosing the plus sign if $f_1 - f_2 > 0$, and the minus sign if $f_1 - f_2 < 0$. After the computation of x_4, new brackets are determined for the root and Eq. (4.3) is applied again. The procedure is repeated until the difference between two successive values of x_4 becomes negligible.

Ridder's iterative formula in Eq. (4.3) has a very useful property: if x_1 and x_2 straddle the root, then x_4 is always within the interval (x_1, x_2). In other words, once the root is bracketed, it stays bracketed, making the method very reliable. The downside is that each iteration requires two function evaluations. There are competitive methods that get by with only one function evaluation per iteration (e.g., Brent's method), but they are more complex, with elaborate book-keeping.

Ridder's method can be shown to converge quadratically, making it faster than either the secant or the false position method. It is the method to use if the derivative of $f(x)$ is impossible or difficult to compute.

■ ridder

The following is the source code for Ridder's method:

```
## module ridder
''' root = ridder(f,a,b,tol=1.0e-9).
    Finds a root of f(x) = 0 with Ridder's method.
    The root must be bracketed in (a,b).
'''
import error
from math import sqrt

def ridder(f,a,b,tol=1.0e-9):
    fa = f(a)
    if fa == 0.0: return a
    fb = f(b)
    if fb == 0.0: return b
    if fa*fb > 0.0: error.err('Root is not bracketed')
    for i in range(30):
      # Compute the improved root x from Ridder's formula
        c = 0.5*(a + b); fc = f(c)
        s = sqrt(fc**2 - fa*fb)
```

```
            if s == 0.0: return None
            dx = (c - a)*fc/s
            if (fa - fb) < 0.0: dx = -dx
            x = c + dx; fx = f(x)
          # Test for convergence
            if i > 0:
                if abs(x - xOld) < tol*max(abs(x),1.0): return x
            xOld = x
          # Re-bracket the root as tightly as possible
            if fc*fx > 0.0:
                if fa*fx < 0.0: b = x; fb = fx
                else:           a = x; fa = fx
            else:
                a = c; b = x; fa = fc; fb = fx
        return None
        print 'Too many iterations'
```

EXAMPLE 4.4

Determine the root of $f(x) = x^3 - 10x^2 + 5 = 0$ that lies in $(0.6, 0.8)$ with Ridder's method.

Solution The starting points are

$$x_1 = 0.6 \qquad f_1 = 0.6^3 - 10(0.6)^2 + 5 = 1.6160$$
$$x_2 = 0.8 \qquad f_2 = 0.8^3 - 10(0.8)^2 + 5 = -0.8880$$

First iteration

Bisection yields the point

$$x_3 = 0.7 \qquad f_3 = 0.7^3 - 10(0.7)^2 + 5 = 0.4430$$

The improved estimate of the root can now be computed with Ridder's formula:

$$s = \sqrt{f_3^2 - f_1 f_2} = \sqrt{0.4330^2 - 1.6160(-0.8880)} = 1.2738$$

$$x_4 = x_3 \pm (x_3 - x_1)\frac{f_3}{s}$$

Because $f_1 > f_2$, we must use the plus sign. Therefore,

$$x_4 = 0.7 + (0.7 - 0.6)\frac{0.4430}{1.2738} = 0.7348$$

$$f_4 = 0.7348^3 - 10(0.7348)^2 + 5 = -0.0026$$

As the root clearly lies in the interval (x_3, x_4), we let

$$x_1 \leftarrow x_3 = 0.7 \qquad f_1 \leftarrow f_3 = 0.4430$$

$$x_2 \leftarrow x_4 = 0.7348 \qquad f_2 \leftarrow f_4 = -0.0026$$

which are the starting points for the next iteration.

Second iteration

$$x_3 = 0.5(x_1 + x_2) = 0.5(0.7 + 0.7348) = 0.7174$$

$$f_3 = 0.7174^3 - 10(0.7174)^2 + 5 = 0.2226$$

$$s = \sqrt{f_3^2 - f_1 f_2} = \sqrt{0.2226^2 - 0.4430(-0.0026)} = 0.2252$$

$$x_4 = x_3 \pm (x_3 - x_1)\frac{f_3}{s}$$

Because $f_1 > f_2$, we again use the plus sign, so that

$$x_4 = 0.7174 + (0.7174 - 0.7)\frac{0.2226}{0.2252} = 0.7346$$

$$f_4 = 0.7346^3 - 10(0.7346)^2 + 5 = 0.0000$$

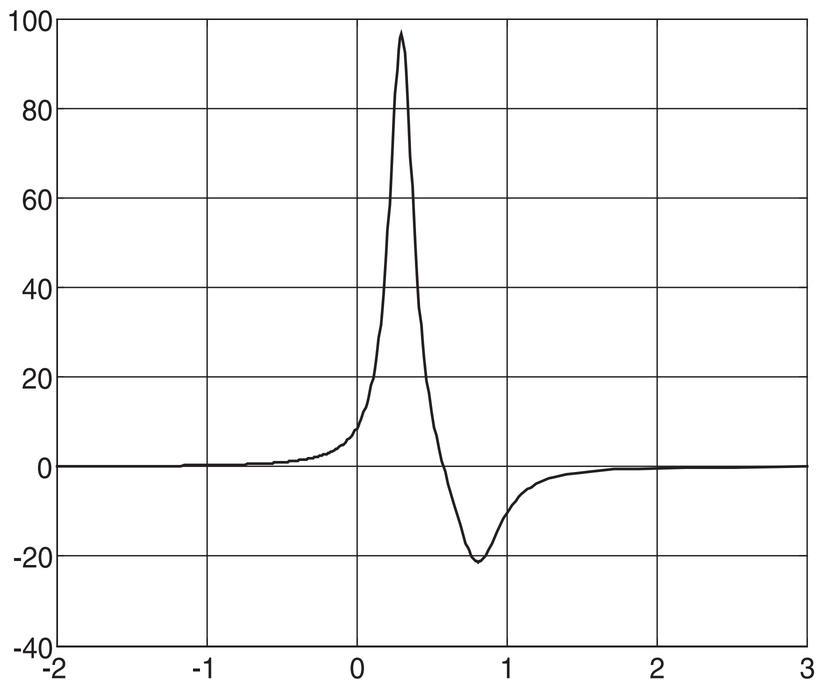
Thus the root is $x = 0.7346$, accurate to at least four decimal places.

EXAMPLE 4.5

Compute the zero of the function

$$f(x) = \frac{1}{(x-0.3)^2 + 0.01} - \frac{1}{(x-0.8)^2 + 0.04}$$

Solution We obtain the approximate location of the root by plotting the function.



It is evident that the root of $f(x) = 0$ lies between $x = 0.5$ and 0.7. We can extract this root with the following program:

```
#!/usr/bin/python
## example4_5
from math import cos
from ridder import *

def f(x):
    a = (x - 0.3)**2 + 0.01
    b = (x - 0.8)**2 + 0.04
    return 1.0/a - 1.0/b

print "root =",ridder(f,0.5,0.7)
raw_input("Press return to exit")
```

The result is

```
root = 0.58
```

4.5 Newton–Raphson Method

The Newton–Raphson algorithm is the best-known method of finding roots for a good reason: it is simple and fast. The only drawback of the method is that it uses the derivative $f'(x)$of the function as well as the function $f(x)$ itself. Therefore, the Newton–Raphson method is usable only in problems where $f'(x)$ can be readily computed.

The Newton–Raphson formula can be derived from the Taylor series expansion of $f(x)$ about x:

$$f(x_{i+1}) = f(x_i) + f'(x_i)(x_{i+1} - x_i) + O(x_{i+1} - x_i)^2 \tag{a}$$

where $\mathcal{O}(z)$ is to be read as "of the order of z" – see Appendix A1. If x_{i+1} is a root of $f(x) = 0$, Eq. (a) becomes

$$0 = f(x_i) + f'(x_i)\,(x_{i+1} - x_i) + O(x_{i+1} - x_i)^2 \tag{b}$$

Assuming that x_i is close to x_{i+1}, we can drop the last term in Eq. (b) and solve for x_{i+1}. The result is the Newton–Raphson formula

$$x_{i+1} = x_i - \frac{f(x)}{f'(x)} \tag{4.3}$$

Letting x denote the true value of the root, the error in x_i is $E_i = x - x_i$. It can be shown that if x_{i+1} is computed from Eq. (4.3), the corresponding error is

$$E_{i+1} = -\frac{f''(x)}{2f'(x)}\,E_i^2$$

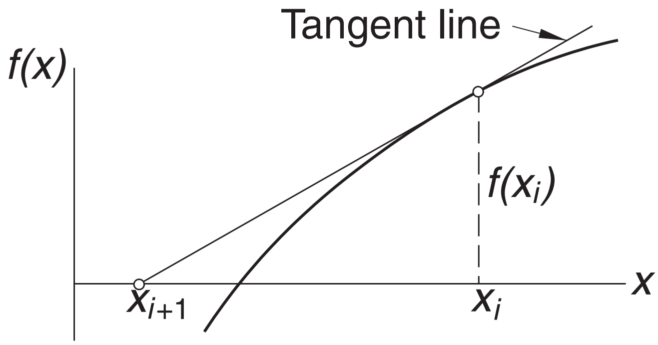


Figure 4.4. Graphical interpretation of Newon–Raphson formula.

indicating that the Newton–Raphson method converges *quadratically* (the error is the square of the error in the previous step). As a consequence, the number of significant figures is roughly doubled in every iteration, provided that x_i is close to the root.

A graphical depiction of the Newton–Raphson formula is shown in Fig. 4.4. The formula approximates $f(x)$ by the straight line that is tangent to the curve at x_i. Thus, x_{i+1} is at the intersection of the x-axis and the tangent line.

The algorithm for the Newton–Raphson method is simple: it repeatedly applies Eq. (4.3), starting with an initial value x_0, until the convergence criterion

$$|x_{i+1} - x_i| < \varepsilon$$

is reached, ε being the error tolerance. Only the latest value of x has to be stored. Here is the algorithm:

1. Let x be a guess for the root of $f(x) = 0$.
2. Compute $\Delta x = -f(x)/f'(x)$.
3. Let $x \leftarrow x + \Delta x$ and repeat steps 2–3 until $|\Delta x| < \varepsilon$.

Although the Newton–Raphson method converges fast near the root, its global convergence characteristics are poor. The reason is that the tangent line is not always an acceptable approximation of the function, as illustrated in the two examples in Fig. 4.5. But the method can be made nearly fail-safe by combining it with bisection.

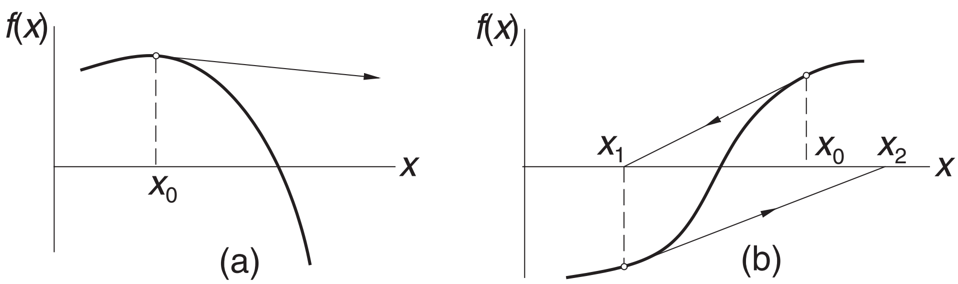


Figure 4.5. Examples where the Newton–Raphson method diverges.

■ newtonRaphson

The following *safe version* of the Newton–Raphson method assumes that the root to be computed is initially bracketed in (a,b). The midpoint of the bracket is used as the initial guess of the root. The brackets are updated after each iteration. If a Newton–Raphson iteration does not stay within the brackets, it is disregarded and replaced with bisection. Because newtonRaphson uses the function f(x) as well as its derivative, function routines for both (denoted by f and df in the listing) must be provided by the user.

```
## module newtonRaphson
''' root = newtonRaphson(f,df,a,b,tol=1.0e-9).
    Finds a root of f(x) = 0 by combining the Newton--Raphson
    method with bisection. The root must be bracketed in (a,b).
    Calls user-supplied functions f(x) and its derivative df(x).
'''
def newtonRaphson(f,df,a,b,tol=1.0e-9):
    import error
    fa = f(a)
    if fa == 0.0: return a
    fb = f(b)
    if fb == 0.0: return b
    if fa*fb > 0.0: error.err('Root is not bracketed')
    x = 0.5*(a + b)
    for i in range(30):
        fx = f(x)
        if abs(fx) < tol: return x
      # Tighten the brackets on the root
        if fa*fx < 0.0:
            b = x
        else:
            a = x
      # Try a Newton-Raphson step
        dfx = df(x)
      # If division by zero, push x out of bounds
        try: dx = -fx/dfx
        except ZeroDivisionError: dx = b - a
        x = x + dx
      # If the result is outside the brackets, use bisection
        if (b - x)*(x - a) < 0.0:
            dx = 0.5*(b-a)
            x = a + dx
      # Check for convergence
        if abs(dx) < tol*max(abs(b),1.0): return x
    print 'Too many iterations in Newton-Raphson'
```

EXAMPLE 4.6
A root of $f(x) = x^3 - 10x^2 + 5 = 0$ lies close to $x = 7$. Compute this root with the Newton–Raphson method.

Solution The derivative of the function is $f'(x) = 3x^2 - 20x$, so that the Newton–Raphson formula in Eq. (4.3) is

$$x \leftarrow x - \frac{f(x)}{f'(x)} = x - \frac{x^3 - 10x^2 + 5}{3x^2 - 20x} = \frac{2x^3 - 10x^2 - 5}{x\,(3x - 20)}$$

It takes only two iterations to reach five-decimal-place accuracy:

$$x \leftarrow \frac{2(0.7)^3 - 10(0.7)^2 - 5}{0.7\,[3(0.7) - 20]} = 0.735\,36$$

$$x \leftarrow \frac{2(0.735\,36)^3 - 10(0.735\,36)^2 - 5}{0.735\,36\,[3(0.735\,36) - 20]} = 0.734\,60$$

EXAMPLE 4.6
Use the Newton–Raphson method to obtain successive approximations of $\sqrt{2}$ as the ratio of two integers.

Solution The problem is equivalent to finding the root of $f(x) = x^2 - 2 = 0$. Here the Newton–Raphson formula is

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

Starting with $x = 1$, successive iterations yield

$$x \leftarrow \frac{(1)^2 + 2}{2(1)} = \frac{3}{2}$$

$$x \leftarrow \frac{(3/2)^2 + 2}{2(3/2)} = \frac{17}{12}$$

$$x \leftarrow \frac{(17/12)^2 + 2}{2(17/12)} = \frac{577}{408}$$

$$\vdots$$

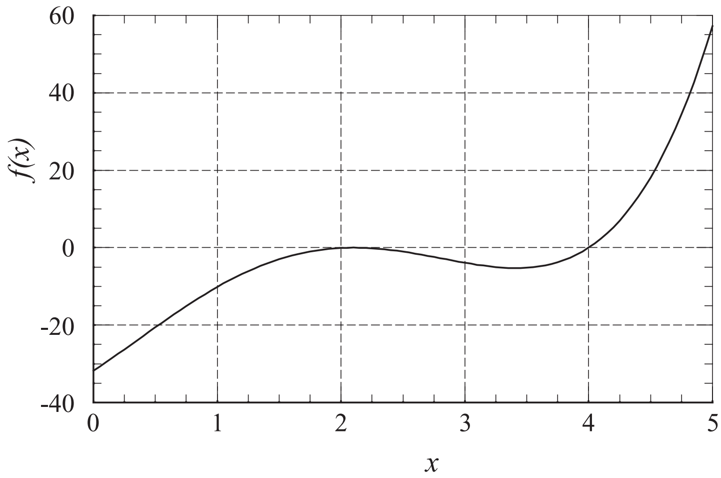
Note that $x = 577/408 = 1.1414216$ is already very close to $\sqrt{2} = 1.1414214$.

The results are dependent on the starting value of x. For example, $x = 2$ would produce a different sequence of ratios.

EXAMPLE 4.7
Find the smallest positive zero of

$$f(x) = x^4 - 6.4x^3 + 6.45x^2 + 20.538x - 31.752$$

Solution Inspecting the plot of the function, we suspect that the smallest positive zero is a double root at about $x = 2$. Bisection and Ridder's method would not work here, because they depend on the function changing its sign at the root. The same



argument applies to the function `newtonRaphson`. But there is no reason why the unrefined version of the Newton–Raphson method should not succeed. We used the following program, which prints the number of iterations in addition to the root:

```
#!/usr/bin/python
## example4_7

def f(x): return x**4 - 6.4*x**3 + 6.45*x**2 + 20.538*x - 31.752
def df(x): return 4.0*x**3 - 19.2*x**2 + 12.9*x + 20.538

def newtonRaphson(x,tol=1.0e-9):
    for i in range(30):
        dx = -f(x)/df(x)
        x = x + dx
        if abs(dx) < tol: return x,i
    print 'Too many iterations\n'

root,numIter = newtonRaphson(2.0)
print 'Root =',root
print 'Number of iterations =',numIter
raw_input(''Press return to exit'')
```

The output is

```
Root = 2.09999998403
Number of iterations = 23
```

The true value of the root is $x = 2.1$. It can be shown that near a multiple root, the convergence of the Newton–Raphson method is linear rather than quadratic, which explains the large number of iterations. Convergence to a multiple root can be speeded up by replacing the Newton–Raphson formula in Eq. (4.3) with

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

where m is the multiplicity of the root ($m = 2$ in this problem). After making the change in the above program, we obtained the result in only five iterations.

4.6 Systems of Equations

Introduction

Up to this point, we have confined our attention to solving the single equation $f(x) = 0$. Let us now consider the n-dimensional version of the same problem, namely

$$\mathbf{f}(\mathbf{x}) = \mathbf{0}$$

or, using scalar notation,

$$\begin{aligned} f_1(x_1, x_2, \ldots, x_n) &= 0 \\ f_2(x_1, x_2, \ldots, x_n) &= 0 \\ &\vdots \\ f_n(x_1, x_2, \ldots, x_n) &= 0 \end{aligned} \tag{4.4}$$

The solution of n simultaneous, nonlinear equations is a much more formidable task than finding the root of a single equation. The trouble is the lack of a reliable method for bracketing the solution vector $\mathbf{x}$. Therefore, we cannot always provide the solution algorithm with a good starting value of $\mathbf{x}$, unless such a value is suggested by the physics of the problem.

The simplest, and the most effective means of computing $\mathbf{x}$ is the Newton–Raphson method. It works well with simultaneous equations, provided that it is supplied with a good starting point. There are other methods that have better global convergence characteristics, but all of them are variants of the Newton–Raphson method.

Newton–Raphson Method

In order to derive the Newton–Raphson method for a system of equations, we start with the Taylor series expansion of $f_i(\mathbf{x})$ about the point $\mathbf{x}$:

$$f_i(\mathbf{x} + \mathbf{\Delta x}) = f_i(\mathbf{x}) + \sum_{j=1}^{n} \frac{\partial f_i}{\partial x_j} \Delta x_j + O(\Delta x^2) \tag{4.5a}$$

Dropping terms of order Δx^2, we can write Eq. (4.5a) as

$$\mathbf{f}(\mathbf{x} + \mathbf{\Delta x}) = \mathbf{f}(\mathbf{x}) + \mathbf{J}(\mathbf{x})\,\mathbf{\Delta x} \tag{4.5b}$$

where $\mathbf{J}(\mathbf{x})$ is the *Jacobian matrix* (of size $n \times n$) made up of the partial derivatives

$$J_{ij} = \frac{\partial f_i}{\partial x_j} \tag{4.6}$$

Note that Eq. (4.5b) is a linear approximation (vector $\mathbf{\Delta x}$ being the variable) of the vector-valued function $\mathbf{f}$ in the vicinity of point $\mathbf{x}$.

Let us now assume that $\mathbf{x}$ is the current approximation of the solution of $\mathbf{f}(\mathbf{x}) = \mathbf{0}$, and let $\mathbf{x} + \mathbf{\Delta x}$ be the improved solution. To find the correction $\mathbf{\Delta x}$, we set $\mathbf{f}(\mathbf{x} + \mathbf{\Delta x}) = \mathbf{0}$ in Eq. (4.5b). The result is a set of linear equations for $\mathbf{\Delta x}$:

$$\mathbf{J}(\mathbf{x})\,\mathbf{\Delta x} = -\mathbf{f}(\mathbf{x}) \tag{4.7}$$

The following steps constitute the Newton–Raphson method for simultaneous, nonlinear equations:

1. Estimate the solution vector $\mathbf{x}$.
2. Evaluate $\mathbf{f}(\mathbf{x})$.
3. Compute the Jacobian matrix $\mathbf{J}(\mathbf{x})$ from Eq. (4.6).
4. Set up the simultaneous equations in Eq. (4.7) and solve for $\mathbf{\Delta x}$.
5. Let $\mathbf{x} \leftarrow \mathbf{x} + \mathbf{\Delta x}$ and repeat steps 2–5.

The foregoing process is continued until $|\mathbf{\Delta x}| < \varepsilon$, where ε is the error tolerance. As in the one-dimensional case, the success of the Newton–Raphson procedure depends entirely on the initial estimate of $\mathbf{x}$. If a good starting point is used, convergence to the solution is very rapid. Otherwise, the results are unpredictable.

Because analytical derivation of each $\partial f_i/\partial x_j$ can be difficult or impractical, it is preferable to let the computer calculate the partial derivatives from the finite difference approximation

$$\frac{\partial f_i}{\partial x_j} \approx \frac{f_i(\mathbf{x} + \mathbf{e}_j h) - f_i(\mathbf{x})}{h} \tag{4.8}$$

where h is a small increment of applied to x_j and $\mathbf{e}_j$ represents a unit vector in the direction of x_j. This formula can be obtained from Eq. (4.5a) after dropping the terms of order Δx^2 and setting $\mathbf{\Delta x} = \mathbf{e}_j h$. We get away with the approximation in Eq. (4.8) because the Newton–Raphson method is rather insensitive to errors in $\mathbf{J}(\mathbf{x})$. By using this approximation, we also avoid the tedium of typing the expressions for $\partial f_i/\partial x_j$ into the computer code.

■ `newtonRaphson2`

This function is an implementation of the Newton–Raphson method. The nested function `jacobian` computes the Jacobian matrix from the finite difference approximation in Eq. (4.8). The simultaneous equations in Eq. (4.7) are solved by Gauss elimination with row pivoting using the function `gaussPivot` listed in Section 2.5. The function subroutine `f` that returns the array $\mathbf{f}(\mathbf{x})$ must be supplied by the user.

```
## module newtonRaphson2
''' soln = newtonRaphson2(f,x,tol=1.0e-9).
    Solves the simultaneous equations f(x) = 0 by
    the Newton-Raphson method using {x} as the initial
    guess. Note that {f} and {x} are vectors.
'''
from numpy import zeros,dot
from gaussPivot import *
from math import sqrt

def newtonRaphson2(f,x,tol=1.0e-9):

    def jacobian(f,x):
        h = 1.0e-4
        n = len(x)
        jac = zeros((n,n))
        f0 = f(x)
        for i in range(n):
            temp = x[i]
            x[i] = temp + h
            f1 = f(x)
            x[i] = temp
            jac[:,i] = (f1 - f0)/h
        return jac,f0

    for i in range(30):
        jac,f0 = jacobian(f,x)
        if sqrt(dot(f0,f0)/len(x)) < tol: return x
        dx = gaussPivot(jac,-f0)
        x = x + dx
        if sqrt(dot(dx,dx)) < tol*max(max(abs(x)),1.0): return x
    print 'Too many iterations'
```

Note that the Jacobian matrix $\mathbf{J}(\mathbf{x})$ is recomputed in each iterative loop. Because each calculation of $\mathbf{J}(\mathbf{x})$ involves $n+1$ evaluations of $\mathbf{f}(\mathbf{x})$ (n is the number of equations), the expense of computation can be high depending on n and the complexity of $\mathbf{f}(\mathbf{x})$. It is often possible to save computer time by neglecting the changes in the Jacobian matrix between iterations, thus computing $\mathbf{J}(\mathbf{x})$ only once. This will work provided that the initial $\mathbf{x}$ is sufficiently close to the solution.

EXAMPLE 4.8

Determine the points of intersection between the circle $x^2 + y^2 = 3$ and the hyperbola $xy = 1$.

Solution The equations to be solved are

$$f_1(x, y) = x^2 + y^2 - 3 = 0 \tag{a}$$

$$f_2(x, y) = xy - 1 = 0 \tag{b}$$

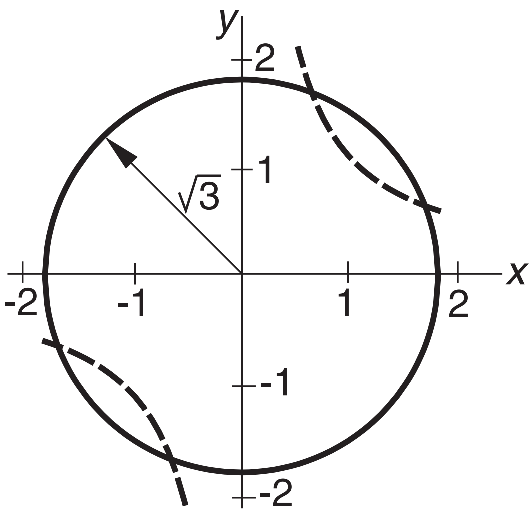
The Jacobian matrix is

$$\mathbf{J}(x, y) = \begin{bmatrix} \partial f_1/\partial x & \partial f_1/\partial y \\ \partial f_2/\partial x & \partial f_2/\partial y \end{bmatrix} = \begin{bmatrix} 2x & 2y \\ y & x \end{bmatrix}$$

Thus, the linear equations $\mathbf{J}(\mathbf{x})\,\mathbf{\Delta x} = -\mathbf{f}(\mathbf{x})$ associated with the Newton–Raphson method are

$$\begin{bmatrix} 2x & 2y \\ y & x \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} -x^2 - y^2 + 3 \\ -xy + 1 \end{bmatrix} \tag{c}$$

By plotting the circle and the hyperbola, we see that there are four points of intersection. It is sufficient, however, to find only one of these points, as the others can be deduced from symmetry. From the plot, we also get rough estimate of the coordinates of an intersection point: $x = 0.5,\ y = 1.5$, which we use as the starting values.



The computations then proceed as follows.

First iteration

Substituting $x = 0.5$, $y = 1.5$ in Eq. (c), we get

$$\begin{bmatrix} 1.0 & 3.0 \\ 1.5 & 0.5 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} 0.50 \\ 0.25 \end{bmatrix}$$

the solution of which is $\Delta x = \Delta y = 0.125$. Therefore, the improved coordinates of the intersection point are

$$x = 0.5 + 0.125 = 0.625 \qquad y = 1.5 + 0.125 = 1.625$$

Second iteration
Repeating the procedure using the latest values of x and y, we obtain

$$\begin{bmatrix} 1.250 & 3.250 \\ 1.625 & 0.625 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} -0.031250 \\ -0.015625 \end{bmatrix}$$

which yields $\Delta x = \Delta y = -0.00694$. Thus,

$$x = 0.625 - 0.006\,94 = 0.618\,06 \qquad y = 1.625 - 0.006\,94 = 1.618\,06$$

Third iteration
Substitution of the latest x and y into Eq. (c) yields

$$\begin{bmatrix} 1.236\,12 & 3.23612 \\ 1.618\,06 & 0.61806 \end{bmatrix} \begin{bmatrix} \Delta x \\ \Delta y \end{bmatrix} = \begin{bmatrix} -0.000\,116 \\ -0.000\,058 \end{bmatrix}$$

The solution is $\Delta x = \Delta y = -0.00003$, so that

$$x = 0.618\,06 - 0.000\,03 = 0.618\,03$$

$$y = 1.618\,06 - 0.000\,03 = 1.618\,03$$

Subsequent iterations would not change the results within five significant figures. Therefore, the coordinates of the four intersection points are

$$\pm(0.618\,03,\ 1.618\,03) \text{ and } \pm\ (1.618\,03,\ 0.618\,03)$$

Alternate solution
If there are only a few equations, it may be possible to eliminate all but one of the unknowns. Then we would be left with a single equation which can be solved by the methods described in Sections 4.2–4.5. In this problem, we obtain from Eq. (b)

$$y = \frac{1}{x}$$

which, upon substitution into Eq. (a), yields $x^2 + 1/x^2 - 3 = 0$, or

$$x^4 - 3x^2 + 1 = 0$$

The solutions of this biquadratic equation; $x = \pm 0.618\,03$ and $\pm 1.618\,03$, agree with the results obtained by the Newton–Raphson method.

EXAMPLE 4.9
Find a solution of

$$\sin x + y^2 + \ln z - 7 = 0$$

$$3x + 2^y - z^3 + 1 = 0$$

$$x + y + z - 5 = 0$$

using `newtonRaphson2`. Start with the point $(1, 1, 1)$.

Solution Letting $x_1 = x$, $x_2 = y$ and $x_3 = z$, we obtain the following program:

```
#!/usr/bin/python
## example4_9
from numpy import zeros,array
from math import sin,log
from newtonRaphson2 import *

def f(x):
    f = zeros(len(x))
    f[0] = sin(x[0]) + x[1]**2 + log(x[2]) - 7.0
    f[1] = 3.0*x[0] + 2.0**x[1] - x[2]**3 + 1.0
    f[2] = x[0] + x[1] + x[2] - 5.0
    return f

x = array([1.0, 1.0, 1.0])
print newtonRaphson2(f,x)
raw_input ("\nPress return to exit")
```

The output from this program is

```
[ 0.59905376  2.3959314   2.00501484]
```

PROBLEM SET 4.1

1. Use the Newton–Raphson method and a four-function calculator ($+ - \times \div$ operations only) to compute $\sqrt[3]{75}$ with four-significant-figure accuracy.
2. Find the smallest positive (real) root of $x^3 - 3.23x^2 - 5.54x + 9.84 = 0$ by the method of bisection.
3. The smallest positive, nonzero root of $\cosh x \cos x - 1 = 0$ lies in the interval $(4, 5)$. Compute this root by Ridder's method.
4. Solve Problem 3 by the Newton–Raphson method.
5. A root of the equation $\tan x - \tanh x = 0$ lies in $(7.0, 7.4)$. Find this root with three-decimal-place accuracy by the method of bisection.
6. Determine the two roots of $\sin x + 3\cos x - 2 = 0$ that lie in the interval $(-2, 2)$. Use the Newton–Raphson method.
7. Solve Prob. 6 using the secant formula, Eq. (4.2).
8. Draw a plot of $f(x) = \cosh x \cos x - 1$ in the range $0 \le x \le 10$. (a) Verify from the plot that the smallest positive, nonzero root of $f(x) = 0$ lies in the interval $(4, 5)$. (b) Show graphically that the Newton–Raphson formula would not converge to this root if it is started with $x = 4$.
9. The equation $x^3 - 1.2x^2 - 8.19x + 13.23 = 0$ has a double root close to $x = 2$. Determine this root with the Newton–Raphson method within four decimal places.
10. ■ Write a program that computes all the roots of $f(x) = 0$ in a given interval with Ridder's method. Utilize the functions `rootsearch` and `ridder`. You may use

the program in Example 4.3 as a model. Test the program by finding the roots of $x \sin x + 3\cos x - x = 0$ in $(-6, 6)$.

11. ■ Repeat Prob. 10 with the Newton–Raphson method.
12. ■ Determine all real roots of $x^4 + 0.9x^3 - 2.3x^2 + 3.6x - 25.2 = 0$.
13. ■ Compute all positive real roots of $x^4 + 2x^3 - 7x^2 + 3 = 0$.
14. ■ Find all positive, nonzero roots of $\sin x - 0.1x = 0$.
15. ■ The natural frequencies of a uniform cantilever beam are related to the roots β_i of the frequency equation $f(\beta) = \cosh\beta\cos\beta + 1 = 0$,
where

$$\beta_i^4 = (2\pi f_i)^2 \frac{mL^3}{EI}$$

f_i = ith natural frequency (cps)

m = mass of the beam

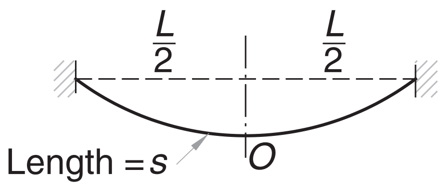
L = length of the beam

E = modulus of elasticity

I = moment of inertia of the cross section

Determine the lowest two frequencies of a steel beam 0.9 m long, with a rectangular cross section 25 mm wide and 2.5 mm high. The mass density of steel is 7850 kg/m^3 and $E = 200$ GPa.

16. ■



A steel cable of length s is suspended as shown in the figure. The maximum tensile stress in the cable, which occurs at the supports, is

$$\sigma_{\max} = \sigma_0 \cosh\beta$$

where

$$\beta = \frac{\gamma L}{2\sigma_0}$$

σ_0 = tensile stress in the cable at O

γ = weight of the cable per unit volume

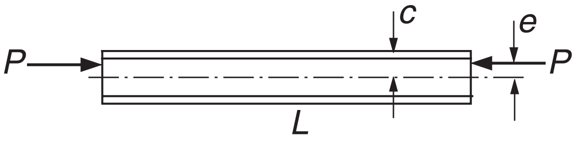
L = horizontal span of the cable

The length to span ratio of the cable is related to β by

$$\frac{s}{L} = \frac{1}{\beta}\sinh\beta$$

Find $\sigma_{\max}$ if $\gamma = 77 \times 10^3$ N/m^3 (steel), $L = 1000$ m, and $s = 1100$ m.

17. ■



The aluminum W 310 × 202 (wide flange) column is subjected to an eccentric axial load P as shown. The maximum compressive stress in the column is given by the *secant formula*:

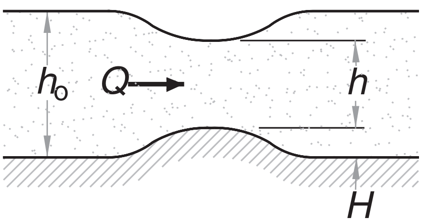
$$\sigma_{\max} = \bar{\sigma}\left[1 + \frac{ec}{r^2}\sec\left(\frac{L}{2r}\sqrt{\frac{\bar{\sigma}}{E}}\right)\right]$$

where

$$\bar{\sigma} = P/A = \text{average stress}$$
$$A = 25,\ 800\ \text{mm}^2 = \text{cross-sectional area of the column}$$
$$e = 85\ \text{mm} = \text{eccentricity of the load}$$
$$c = 170\ \text{mm} = \text{half-depth of the column}$$
$$r = 142\ \text{mm} = \text{radius of gyration of the cross section}$$
$$L = 7100\ \text{mm} = \text{length of the column}$$
$$E = 71 \times 10^9\ \text{Pa} = \text{modulus of elasticity}$$

Determine the maximum load P that the column can carry if the maximum stress is not to exceed 120×10^6 Pa.

18. ■



Bernoulli's equation for fluid flow in an open channel with a small bump is

$$\frac{Q^2}{2gb^2h_0^2} + h_0 = \frac{Q^2}{2gb^2h^2} + h + H$$

where

$$Q = 1.2\ \mathrm{m}^3/\mathrm{s} = \text{volume rate of flow}$$

$$g = 9.81\ \mathrm{m/s}^2 = \text{gravitational acceleration}$$

$$b = 1.8\ \mathrm{m} = \text{width of channel}$$

$$h_0 = 0.6\ \mathrm{m} = \text{upstream water level}$$

$$H = 0.075\ \mathrm{m} = \text{height of bump}$$

$$h = \text{ water level above the bump}$$

Determine h.

19. ■ The speed v of a Saturn V rocket in vertical flight near the surface of the earth can be approximated by

$$v = u \ln \frac{M_0}{M_0 - \dot{m}t} - gt$$

where

$$u = 2510\ \mathrm{m/s} = \text{velocity of exhaust relative to the rocket}$$

$$M_0 = 2.8 \times 10^6\ \mathrm{kg} = \text{mass of rocket at liftoff}$$

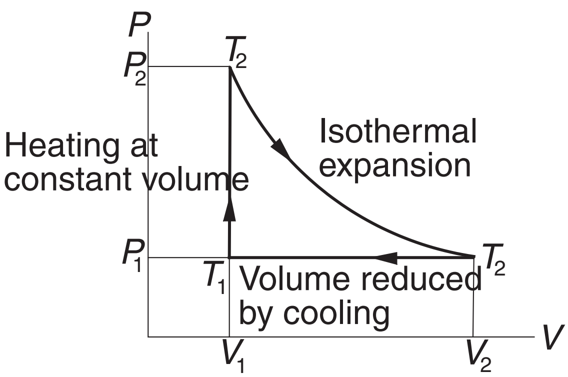
$$\dot{m} = 13.3 \times 10^3\ \mathrm{kg/s} = \text{rate of fuel consumption}$$

$$g = 9.81\ \mathrm{m/s}^2 = \text{gravitational acceleration}$$

$$t = \text{time measured from liftoff}$$

Determine the time when the rocket reaches the speed of sound (335 m/s).

20. ■



The figure shows the thermodynamic cycle of an engine. The efficiency of this engine for monatomic gas is

$$\eta = \frac{\ln(T_2/T_1) - (1 - T_1/T_2)}{\ln(T_2/T_1) + (1 - T_1/T_2)/(\gamma - 1)}$$

where T is the absolute temperature and $\gamma = 5/3$. Find T_2/T_1 that results in 30% efficiency ($\eta = 0.3$).

21. ■ The Gibbs free energy of 1 mole of hydrogen at temperature T is

$$G = -RT \ln\left[(T/T_0)^{5/2}\right] \text{ J}$$

where $R = 8.314\,41$ J/K is the gas constant and $T_0 = 4.444\,18$ K. Determine the temperature at which $G = -10^5$ J.

22. ■ The chemical equilibrium equation in the production of methanol from CO and H_2 is[1]

$$\frac{\xi(3-2\xi)^2}{(1-\xi)^3} = 249.2$$

where ξ is the *equilibrium extent of the reaction*. Determine ξ.

23. ■ Determine the coordinates of the two points where the circles $(x-2)^2 + y^2 = 4$ and $x^2 + (y-3)^2 = 4$ intersect. Start by estimating the locations of the points from a sketch of the circles, and then use the Newton–Raphson method to compute the coordinates.

24. ■ The equations

$$\sin x + 3\cos x - 2 = 0$$
$$\cos x - \sin y + 0.2 = 0$$

have a solution in the vicinity of the point $(1, 1)$. Use the Newton–Raphson method to refine the solution.

25. ■ Use any method to find *all* real solutions of the simultaneous equations

$$\tan x - y = 1$$
$$\cos x - 3\sin y = 0$$

in the region $0 \le x \le 1.5$.

26. ■ The equation of a circle is

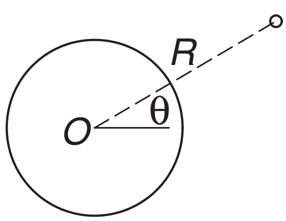
$$(x-a)^2 + (y-b)^2 = R^2$$

where R is the radius and (a, b) are the coordinates of the center. If the coordinates of three points on the circle are

x (in.)	8.21	0.34	5.96
y (in.)	0.00	6.62	−1.12

determine R, a, and b.

27. ■



[1] From R. A. Alberty, *Physical Chemistry*, 7th ed. (Wiley, 1987).

The trajectory of a satellite orbiting the earth is

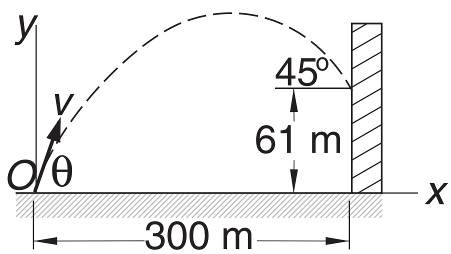
$$R = \frac{C}{1 + e\sin(\theta + \alpha)}$$

where (R, θ) are the polar coordinates of the satellite, and C, e, and α are constants (e is known as the eccentricity of the orbit). If the satellite was observed at the three positions

θ	-30°	0°	30°
R (km)	6870	6728	6615

determine the smallest R of the trajectory and the corresponding value of θ.

28. ■

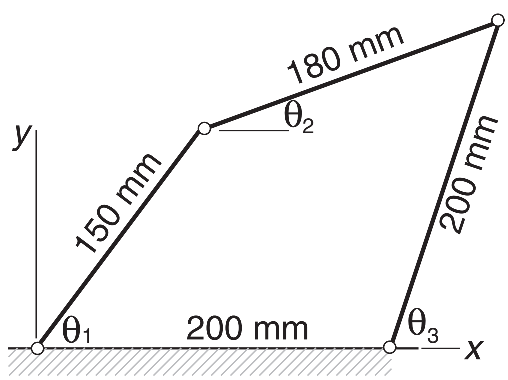


A projectile is launched at O with the velocity v at the angle θ to the horizontal. The parametric equation of the trajectory is

$$x = (v\cos\theta)t$$
$$y = -\frac{1}{2}gt^2 + (v\sin\theta)t$$

where t is the time measured from instant of launch, and $g = 9.81$ m/s^2 represents the gravitational acceleration. If the projectile is to hit the target at the 45° angle shown in the figure, determine v, θ, and the time of flight.

29. ■



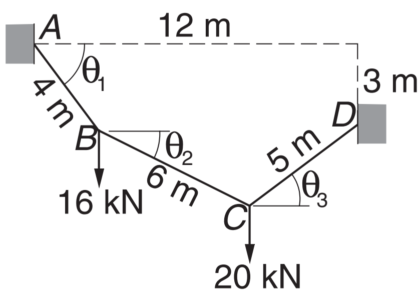
The three angles shown in the figure of the four-bar linkage are related by

$$150\cos\theta_1 + 180\cos\theta_2 - 200\cos\theta_3 = 200$$

$$150\sin\theta_1 + 180\sin\theta_2 - 200\sin\theta_3 = 0$$

Determine θ_1 and θ_2 when $\theta_3 = 75^\circ$. Note that there are two solutions.

30. ■



The 15-m cable is suspended from A and D and carries concentrated loads at B and C. The vertical equilibrium equations of joints B and C are

$$T(-\tan\theta_2 + \tan\theta_1) = 16$$

$$T(\tan\theta_3 + \tan\theta_2) = 20$$

where T is the horizontal component of the cable force (it is the same in all segments of the cable). In addition, there are two geometric constraints imposed by the positions of the supports:

$$-4\sin\theta_1 - 6\sin\theta_2 + 5\sin\theta_2 = -3$$

$$4\cos\theta_1 + 6\cos\theta_2 + 5\cos\theta_3 = 12$$

Determine the angles θ_1, θ_2, and θ_3.

*4.7 Zeroes of Polynomials

Introduction

A polynomial of degree n has the form

$$P_n(x) = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n \tag{4.9}$$

where the coefficients a_i may be real or complex. We concentrate on polynomials with real coefficients, but the algorithms presented in this section also work with complex coefficients.

The polynomial equation $P_n(x) = 0$ has exactly n roots, which may be real or complex. If the coefficients are real, the complex roots always occur in conjugate pairs $(x_r + ix_i,\ x_r - ix_i)$, where x_r and x_i are the real and imaginary parts, respectively.

For real coefficients, the number of real roots can be estimated from the *rule of Descartes*:

- The number of positive, real roots equals the number of sign changes in the expression for $P_n(x)$, or less by an even number.
- The number of negative, real roots is equal to the number of sign changes in $P_n(-x)$, or less by an even number.

As an example, consider $P_3(x) = x^3 - 2x^2 - 8x + 27$. Because the sign changes twice, $P_3(x) = 0$ has either two or zero positive real roots. On the other hand, $P_3(-x) = -x^3 - 2x^2 + 8x + 27$ contains a single sign change; hence $P_3(x)$ possesses one negative real zero.

The real zeroes of polynomials with real coefficients can always be computed by one of the methods already described. But if complex roots are to be computed, it is best to use a method that specializes in polynomials. Here we present a method due to Laguerre, which is reliable and simple to implement. Before proceeding to Laguerre's method, we must first develop two numerical tools that are needed in any method capable of determining the zeroes of a polynomial. The first of these is an efficient algorithm for evaluating a polynomial and its derivatives. The second algorithm we need is for the *deflation* of a polynomial, that is, for dividing the $P_n(x)$ by $x - r$, where r is a root of $P_n(x) = 0$.

Evaluation Polynomials

It is tempting to evaluate the polynomial in Eq. (4.9) from left to right by the following algorithm (we assume that the coefficients are stored in the array **a**):

```
p = 0.0
for i in range(n+1):
    p = p + a[i]*x**i
```

Because x^k is evaluated as $x \times x \times \cdots \times x$ ($k - 1$ multiplications), we deduce that the number of multiplications in this algorithm is

$$1 + 2 + 3 + \cdots + n - 1 = \frac{1}{2}n(n-1)$$

If n is large, the number of multiplications can be reduced considerably if we evaluate the polynomial from right to left. For an example, take

$$P_4(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + a_4x^4$$

After rewriting the polynomial as

$$P_4(x) = a_0 + x\{a_1 + x[a_2 + x(a_3 + xa_4)]\}$$

the preferred computational sequence becomes obvious:

$$P_0(x) = a_4$$
$$P_1(x) = a_3 + xP_0(x)$$
$$P_2(x) = a_2 + xP_1(x)$$
$$P_3(x) = a_1 + xP_2(x)$$
$$P_4(x) = a_0 + xP_3(x)$$

For a polynomial of degree n, the procedure can be summarized as

$$P_0(x) = a_n$$
$$P_i(x) = a_{n-i} + xP_{i-1}(x), \quad i = 1, 2, \ldots, n \tag{4.10}$$

leading to the algorithm

```
p = a[n]
for i in range(1,n+1):
    p = a[n-i] + p*x
```

The last algorithm involves only n multiplications, making it more efficient for $n > 3$. But computational economy is not the prime reason why this algorithm should be used. Because the result of each multiplication is rounded off, the procedure with the least number of multiplications invariably accumulates the smallest roundoff error.

Some root-finding algorithms, including Laguerre's method, also require evaluation of the first and second derivatives of $P_n(x)$. From Eq. (4.10) we obtain by differentiation

$$P_0'(x) = 0 \quad P_i'(x) = P_{i-1}(x) + xP_{i-1}'(x), \quad i = 1, 2, \ldots, n \tag{4.11a}$$
$$P_0''(x) = 0 \quad P_i''(x) = 2P_{i-1}'(x) + xP_{i-1}''(x), \quad i = 1, 2, \ldots, n \tag{4.11b}$$

■ evalPoly

Here is the function that evaluates a polynomial and its derivatives:

```
## module evalPoly
''' p,dp,ddp = evalPoly(a,x).
    Evaluates the polynomial
    p = a[0] + a[1]*x + a[2]*xˆ2 +...+ a[n]*xˆn
    with its derivatives dp = p' and ddp = p''
    at x.
'''
def evalPoly(a,x):
    n = len(a) - 1
    p = a[n]
```

```
    dp = 0.0 + 0.0j
    ddp = 0.0 + 0.0j
    for i in range(1,n+1):
        ddp = ddp*x + 2.0*dp
        dp = dp*x + p
        p = p*x + a[n-i]
    return p,dp,ddp
```

Deflation of Polynomials

After a root r of $P_n(x) = 0$ has been computed, it is desirable to factor the polynomial as follows:

$$P_n(x) = (x - r)\, P_{n-1}(x) \tag{4.12}$$

This procedure, known as deflation or *synthetic division*, involves nothing more than computing the coefficients of $P_{n-1}(x)$. Because the remaining zeroes of $P_n(x)$ are also the zeroes of $P_{n-1}(x)$, the root-finding procedure can now be applied to $P_{n-1}(x)$ rather than $P_n(x)$. Deflation thus makes it progressively easier to find successive roots, because the degree of the polynomial is reduced every time a root is found. Moreover, by eliminating the roots that have already been found, the chances of computing the same root more than once are eliminated.

If we let

$$P_{n-1}(x) = b_0 + b_1 x + b_2 x^2 + \cdots + b_{n-1} x^{n-1}$$

then Eq. (4.12) becomes

$$\begin{aligned} & a_0 + a_1 x + a_2 x^2 + \cdots + a_{n-1} x^{n-1} + a_n x^n \\ &= (x - r)(b_0 + b_1 x + b_2 x^2 + \cdots + b_{n-1} x^{n-1}) \end{aligned}$$

Equating the coefficients of like powers of x, we obtain

$$b_{n-1} = a_n \qquad b_{n-2} = a_{n-1} + r b_{n-1} \qquad \cdots \qquad b_0 = a_1 + r b_1 \tag{4.13}$$

which leads to *Horner's deflation algorithm:*

```
b[n-1] = a[n]
for i in range(n-2,-1,-1):
    b[i] = a[i+1] + r*b[i+1]
```

Laguerre's Method

Laquerre's formulas are not easily derived for a general polynomial $P_n(x)$. However, the derivation is greatly simplified if we consider the special case where the polynomial has a zero at $x = r$ and $(n - 1)$ zeroes at $x = q$. Hence, the polynomial can be written as

$$P_n(x) = (x-r)(x-q)^{n-1} \tag{a}$$

Our problem is now this: given the polynomial in Eq. (a) in the form

$$P_n(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$$

determine r (note that q is also unknown). It turns out that the result, which is exact for the special case considered here, works well as an iterative formula with any polynomial.

Differentiating Eq. (a) with respect to x, we get

$$\begin{aligned} P_n'(x) &= (x-q)^{n-1} + (n-1)(x-r)(x-q)^{n-2} \\ &= P_n(x)\left(\frac{1}{x-r} + \frac{n-1}{x-q}\right) \end{aligned}$$

Thus,

$$\frac{P_n'(x)}{P_n(x)} = \frac{1}{x-r} + \frac{n-1}{x-q} \tag{b}$$

which upon differentiation yields

$$\frac{P_n''(x)}{P_n(x)} - \left[\frac{P_n'(x)}{P_n(x)}\right]^2 = -\frac{1}{(x-r)^2} - \frac{n-1}{(x-q)^2} \tag{c}$$

It is convenient to introduce the notation

$$G(x) = \frac{P_n'(x)}{P_n(x)} \qquad H(x) = G^2(x) - \frac{P_n''(x)}{P_n(x)} \tag{4.14}$$

so that Eqs. (b) and (c) become

$$G(x) = \frac{1}{x-r} + \frac{n-1}{x-q} \tag{4.15a}$$

$$H(x) = \frac{1}{(x-r)^2} + \frac{n-1}{(x-q)^2} \tag{4.15b}$$

If we solve Eq. (4.15a) for $x-q$ and substitute the result into Eq. (4.15b), we obtain a quadratic equation for $x-r$. The solution of this equation is the *Laguerre's formula*

$$x - r = \frac{n}{G(x) \pm \sqrt{(n-1)\left[nH(x) - G^2(x)\right]}} \tag{4.16}$$

The procedure for finding a zero of a general polynomial by Laguerre's formula is:

1. Let x be a guess for the root of $P_n(x) = 0$ (any value will do).
2. Evaluate $P_n(x)$, $P_n'(x)$, and $P_n''(x)$ using the procedure outlined in Eqs. (4.11).
3. Compute $G(x)$ and $H(x)$ from Eqs. (4.14).
4. Determine the improved root r from Eq. (4.16) choosing the sign that results in the *larger magnitude of the denominator* (this can be shown to improve convergence).
5. Let $x \leftarrow r$ and repeat steps 2–5 until $|P_n(x)| < \varepsilon$ or $|x - r| < \varepsilon$, where ε is the error tolerance.

One nice property of Laguerre's method is that it converges to a root, with very few exceptions, from any starting value of x.

■ polyRoots

The function polyRoots in this module computes all the roots of $P_n(x) = 0$, where the polynomial $P_n(x)$ defined by its coefficient array $\mathbf{a} = [a_0, a_1, \ldots, a_n]$. After the first root is computed by the nested function laguerre, the polynomial is deflated using deflPoly and the next zero computed by applying laguerre to the deflated polynomial. This process is repeated until all n roots have been found. If a computed root has a very small imaginary part, it is more than likely that it represents roundoff error. Therefore, polyRoots replaces a tiny imaginary part by zero.

```
from evalPoly import *
from numpy import zeros,complex
from cmath import sqrt
from random import random

def polyRoots(a,tol=1.0e-12):

    def laguerre(a,tol):
        x = random()   # Starting value (random number)
        n = len(a) - 1
        for i in range(30):
            p,dp,ddp = evalPoly(a,x)
            if abs(p) < tol: return x
            g = dp/p
            h = g*g - ddp/p
            f = sqrt((n - 1)*(n*h - g*g))
            if abs(g + f) > abs(g - f): dx = n/(g + f)
            else: dx = n/(g - f)
            x = x - dx
            if abs(dx) < tol: return x
        print 'Too many iterations'

    def deflPoly(a,root):  # Deflates a polynomial
        n = len(a)-1
        b = [(0.0 + 0.0j)]*n
        b[n-1] = a[n]
        for i in range(n-2,-1,-1):
            b[i] = a[i+1] + root*b[i+1]
        return b

    n = len(a) - 1
    roots = zeros((n),dtype=complex)
```

```
    for i in range(n):
        x = laguerre(a,tol)
        if abs(x.imag) < tol: x = x.real
        roots[i] = x
        a = deflPoly(a,x)
    return roots
    raw_input("\nPress return to exit")
```

Because the roots are computed with finite accuracy, each deflation introduces small errors in the coefficients of the deflated polynomial. The accumulated roundoff error increases with the degree of the polynomial and can become severe if the polynomial is ill conditioned (small changes in the coefficients produce large changes in the roots). Hence, the results should be viewed with caution when dealing with polynomials of high degree.

The errors caused by deflation can be reduced by recomputing each root using the original, undeflated polynomial. The roots obtained previously in conjunction with deflation are employed as the starting values.

EXAMPLE 4.10

A zero of the polynomial $P_4(x) = 3x^4 - 10x^3 - 48x^2 - 2x + 12$ is $x = 6$. Deflate the polynomial with Horner's algorithm, that is, find $P_3(x)$ so that $(x - 6)\,P_3(x) = P_4(x)$.

Solution With $r = 6$ and $n = 4$, Eqs. (4.13) become

$$b_3 = a_4 = 3$$
$$b_2 = a_3 + 6b_3 = -10 + 6(3) = 8$$
$$b_1 = a_2 + 6b_2 = -48 + 6(8) = 0$$
$$b_0 = a_1 + 6b_1 = -2 + 6(0) = -2$$

Therefore,

$$P_3(x) = 3x^3 + 8x^2 - 2$$

EXAMPLE 4.11

A root of the equation $P_3(x) = x^3 - 4.0x^2 - 4.48x + 26.1$ is approximately $x = 3 - i$. Find a more accurate value of this root by one application of Laguerre's iterative formula.

Solution Use the given estimate of the root as the starting value. Thus,

$$x = 3 - i \qquad x^2 = 8 - 6i \qquad x^3 = 18 - 26i$$

Substituting these values in $P_3(x)$ and its derivatives, we get

$$\begin{aligned}
P_3(x) &= x^3 - 4.0x^2 - 4.48x + 26.1 \\
&= (18 - 26i) - 4.0(8 - 6i) - 4.48(3 - i) + 26.1 = -1.34 + 2.48i \\
P_3'(x) &= 3.0x^2 - 8.0x - 4.48 \\
&= 3.0(8 - 6i) - 8.0(3 - i) - 4.48 = -4.48 - 10.0i \\
P_3''(x) &= 6.0x - 8.0 = 6.0(3 - i) - 8.0 = 10.0 - 6.0i
\end{aligned}$$

Equations (4.14) then yield

$$\begin{aligned}
G(x) &= \frac{P_3'(x)}{P_3(x)} = \frac{-4.48 - 10.0i}{-1.34 + 2.48i} = -2.36557 + 3.08462i \\
H(x) &= G^2(x) - \frac{P_3''(x)}{P_3(x)} = (-2.36557 + 3.08462i)^2 - \frac{10.0 - 6.0i}{-1.34 + 2.48i} \\
&= 0.35995 - 12.48452i
\end{aligned}$$

The term under the square root sign of the denominator in Eq. (4.16) becomes

$$\begin{aligned}
F(x) &= \sqrt{(n-1)\left[n\,H(x) - G^2(x)\right]} \\
&= \sqrt{2\left[3(0.35995 - 12.48452i) - (-2.36557 + 3.08462i)^2\right]} \\
&= \sqrt{5.67822 - 45.71946i} = 5.08670 - 4.49402i
\end{aligned}$$

Now we must find which sign in Eq. (4.16) produces the larger magnitude of the denominator:

$$\begin{aligned}
|G(x) + F(x)| &= |(-2.36557 + 3.08462i) + (5.08670 - 4.49402i)| \\
&= |2.72113 - 1.40940i| = 3.06448 \\
|G(x) - F(x)| &= |(-2.36557 + 3.08462i) - (5.08670 - 4.49402i)| \\
&= |-7.45227 + 7.57864i| = 10.62884
\end{aligned}$$

Using the minus sign, Eq. (4.16) yields the following improved approximation for the root:

$$\begin{aligned}
r &= x - \frac{n}{G(x) - F(x)} = (3 - i) - \frac{3}{-7.45227 + 7.57864i} \\
&= 3.19790 - 0.79875i
\end{aligned}$$

Thanks to the good starting value, this approximation is already quite close to the exact value $r = 3.20 - 0.80i$.

EXAMPLE 4.12
Use `polyRoots` to compute *all* the roots of $x^4 - 5x^3 - 9x^2 + 155x - 250 = 0$.

Solution The commands

```
>>> from polyRoots import *
>>> print polyRoots([-250.0,155.0,-9.0,-5.0,1.0])
```

resulted in the output

```
[ 2.+0.j  4.-3.j  4.+3.j -5.+0.j]
```

PROBLEM SET 4.2

Problems 1–5 A zero $x = r$ of $P_n(x)$ is given. Verify that r is indeed a zero, and then deflate the polynomial, that is, find $P_{n-1}(x)$ so that $P_n(x) = (x - r)P_{n-1}(x)$.

1. $P_3(x) = 3x^3 + 7x^2 - 36x + 20$, $r = -5$.
2. $P_4(x) = x^4 - 3x^2 + 3x - 1$, $r = 1$.
3. $P_5(x) = x^5 - 30x^4 + 361x^3 - 2178x^2 + 6588x - 7992$, $r = 6$.
4. $P_4(x) = x^4 - 5x^3 - 2x^2 - 20x - 24$, $r = 2i$.
5. $P_3(x) = 3x^3 - 19x^2 + 45x - 13$, $r = 3 - 2i$.

Problems 6–9 A zero $x = r$ of $P_n(x)$ is given. Determine all the other zeroes of $P_n(x)$ by using a calculator. You should need no tools other than deflation and the quadratic formula.

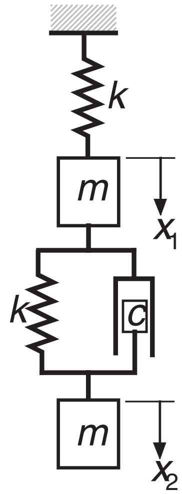
6. $P_3(x) = x^3 + 1.8x^2 - 9.01x - 13.398$,, $r = -3.3$.
7. $P_3(x) = x^3 - 6.64x^2 + 16.84x - 8.32$, $r = 0.64$.
8. $P_3(x) = 2x^3 - 13x^2 + 32x - 13$, $r = 3 - 2i$.
9. $P_4(x) = x^4 - 3x^2 + 10x^2 - 6x - 20$, $r = 1 + 3i$.

Problems 10–15 Find all the zeroes of the given $P_n(x)$.

10. ■ $P_4(x) = x^4 + 2.1x^3 - 2.52x^2 + 2.1x - 3.52$.
11. ■ $P_5(x) = x^5 - 156x^4 - 5x^3 + 780x^2 + 4x - 624$.
12. ■ $P_6(x) = x^6 + 4x^5 - 8x^4 - 34x^3 + 57x^2 + 130x - 150$.
13. ■ $P_7(x) = 8x^7 + 28x^6 + 34x^5 - 13x^4 - 124x^3 + 19x^2 + 220x - 100$.
14. ■ $P_8(x) = x^8 - 7x^7 + 7x^6 + 25x^5 + 24x^4 - 98x^3 - 472x^2 + 440x + 800$.
15. ■ $P_4(x) = x^4 + (5 + i)x^3 - (8 - 5i)x^2 + (30 - 14i)x - 84$.
16. ■

    The two blocks of mass m each are connected by springs and a dashpot. The stiffness of each spring is k, and c is the coefficient of damping of the dashpot. When the system is displaced and released, the displacement of each block during the ensuing motion has the form

$$x_k(t) = A_k e^{\omega_r t}\cos(\omega_i t + \phi_k),\ k = 1, 2$$

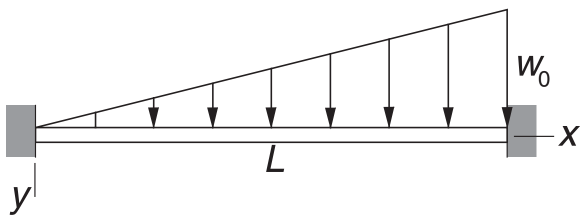


where A_k and ϕ_k are constants, and $\omega = \omega_r \pm i\omega_i$ are the roots of

$$\omega^4 + 2\frac{c}{m}\omega^3 + 3\frac{k}{m}\omega^2 + \frac{c}{m}\frac{k}{m}\omega + \left(\frac{k}{m}\right)^2 = 0$$

Determine the two possible combinations of ω_r and ω_i if $c/m = 12\ \mathrm{s}^{-1}$ and $k/m = 1500\ \mathrm{s}^{-2}$.

17.



The lateral deflection of the beam shown is

$$y = \frac{w_0}{120EI}(x^5 - 3L^2x^3 + 2L^3x^2)$$

where ω_0 is the maximum load intensity and EI represents the bending rigidity. Determine the value of x/L where the maximum displacement occurs.

Other Methods

The most prominent root-finding algorithm omitted from this chapter is *Brent's method*, which combines bisection and quadratic interpolation. It is potentially more efficient than Ridder's method, requiring only one function evaluation per iteration (as compared to two evaluations in Ridder's method), but this advantage is somewhat negated by elaborate bookkeeping.

There are many methods for finding zeroes of polynomials. Of these, the *Jenkins-Traub algorithm*[2] deserves special mention because of its robustness and widespread use in packaged software.

The zeroes of a polynomial can also be obtained by calculating the eigenvalues of the $n \times n$ "companion matrix"

$$\mathbf{A} = \begin{bmatrix} -a_{n-1}/a_n & -a_2/a_n & \cdots & -a_1/a_n & -a_0/a_n \\ 1 & 0 & \cdots & 0 & 0 \\ 0 & 0 & & 0 & 0 \\ \vdots & \vdots & & \vdots & \vdots \\ 0 & 0 & \cdots & 1 & 0 \end{bmatrix}$$

where a_i are the coefficients of the polynomial. The characteristic equation (see Section 9.1) of this matrix is

$$x^n + \frac{a_{n-1}}{a_n} x^{n-1} + \frac{a_{n-2}}{a_n} x^{n-2} + \cdots + \frac{a_1}{a_n} x + \frac{a_0}{a_n} = 0$$

which is equivalent to $P_n(x) = 0$. Thus the eigenvalues of $\mathbf{A}$ are the zeroes of $P_n(x)$. The eigenvalue method is robust, but considerably slower than Laguerre's method. But it is worthy of consideration if a good program for eigenvalue problems is available.

[2] M. Jenkins and J. Traub, *SIAM Journal on Numerical Analysis*, Vol. 7 (1970), p. 545.