5 Roots

In changing, it finds rest.

Heraclitus

5.1 Motivation

5.1.1 Examples from Physics

Nonlinear equations are omnipresent in physics. Here are a few examples:

1. **Van der Waals equation of state**
   As you may recall, the classical ideal gas "law" has the form:
   $$Pv = RT \tag{5.1}$$
   where P is the pressure, v is the molar volume, R is the gas constant, and T is the temperature. This ignores the finite size of the gas molecules, as well as the mutual attraction between molecules. Correcting for these features "by hand" (or via the formalism of statistical mechanics) leads to the so-called *van der Waals equation of state*:
   $$\left(P + \frac{a}{v^2}\right)(v - b) = RT \tag{5.2}$$
   where a and b are parameters that depend on the gas. Assume you know the pressure and the temperature and wish to evaluate the molar volume. As you can see by multiplying Eq. (5.2) with v^2, you get a cubic equation in v. Even though it's messy to do so, cubic equations can be solved analytically; however, van der Waals is not the final word on equations of state: you can keep adding more terms in an attempt to make your model more realistic (as is done in, e.g., the virial expansion), so the resulting equation becomes increasingly complicated.
2. **Quantum mechanics of a particle in a finite well**
   A standard part of any introduction to quantum mechanics is the study of a single particle in a one-dimensional finite square well potential. Assume that the potential is V_0 for $x < -a$ and $x > a$ and is 0 inside the well, i.e., for $-a < x < a$. Solving the one-dimensional time-independent Schrödinger equation leads to states of even and odd parity. The even solutions obey the transcendental equation:
   $$k \tan ka = \kappa \tag{5.3}$$

where k and κ are related to the mass m of the particle and to the energy of the state (and to V_0); k characterizes the wave function inside the well and κ that outside the well. These two quantities are related by:

$$k^2 + \kappa^2 = \frac{2mV_0}{\hbar^2} \tag{5.4}$$

In a first course on quantum mechanics you typically learn that these two equations must be solved for k and κ "graphically". But what does that really mean? And what would happen if the potential was slightly more complicated, leading to a more involved set of equations? It would be nice to have access to general principles and techniques.

3. **Gravitational force coming from two masses**

   The previous examples involved one or two polynomial or transcendental equations. We now turn to a case where several quantities are unknown and the equations we need to solve are nonlinear.

   Consider two masses, M_R and M_S, located at $\mathbf{R}$ and $\mathbf{S}$, respectively. Crucially, while the masses are known, the positions where they are located are unknown. That means that we need to determine the coordinates of $\mathbf{R}$ and $\mathbf{S}$, i.e., $2 \times 3 = 6$ numbers. Thankfully, we are free to make force measurements on a test mass m at positions of our choosing. First, we carry out a measurement of the force on m at the position $\mathbf{r}_0$ (which we pick):[1]

$$\mathbf{F}_0 = -GM_R m \frac{\mathbf{r}_0 - \mathbf{R}}{|\mathbf{r}_0 - \mathbf{R}|^3} - GM_S m \frac{\mathbf{r}_0 - \mathbf{S}}{|\mathbf{r}_0 - \mathbf{S}|^3} \tag{5.5}$$

   This is a vector relation so it corresponds to three scalar equations; however, we are dealing with six unknowns. To remedy the situation, we make another measurement of the force on m, this time at position $\mathbf{r}_1$:

$$\mathbf{F}_1 = -GM_R m \frac{\mathbf{r}_1 - \mathbf{R}}{|\mathbf{r}_1 - \mathbf{R}|^3} - GM_S m \frac{\mathbf{r}_1 - \mathbf{S}}{|\mathbf{r}_1 - \mathbf{S}|^3} \tag{5.6}$$

   The last two vector relations put together give us six scalar equations in six unknowns. Due to the nature of Newton's gravitational force, these are coupled nonlinear equations; this is not a problem you would enjoy approaching using paper and pencil.

5.1.2 The Problem(s) to Be Solved

At its most basic level, the problem we are faced with is as follows: you are given a function $f(x)$ and you need to find its *zeros*. In other words, you need to solve the equation $f(x) = 0$ (the solutions of this equation are known as *roots*).[2] Thus, our task is to solve:

$$f(x) = 0 \tag{5.7}$$

where $f(x)$ is generally nonlinear. Just like our main problems in the chapter on linear algebra, our equation this time is very easy to write down. We are faced with one equation

[1] Notice the formal similarity between this equation and Eq. (1.3).

[2] In practice, the words *root* and *zero* are often used interchangeably.

in one unknown: if the function $f(x)$ is linear, then the problem is trivial. If, however, it is nonlinear then arriving at a solution may be non-trivial. The $f(x)$ involved may be a polynomial, a special function, or a complicated programming routine (which makes each function evaluation quite costly). In what follows, we will denote our root by x^*.

To give an example: $f(x) = x^2 - 5 = 0$ is very easy to write down. Anyone can "solve" it by saying $x^2 = 5$, followed by $x = \pm\sqrt{5}$. This is where things are not so easy any more: what is the value of $\sqrt{5}$? Of course, you can use a calculator, or even `from math import sqrt` in Python. But how do *those* figure out which number gives 5 when squared? Roughly two thousand years ago, Hero of Alexandria came up with a specific prescription that addresses this question. In our case, we are not really interested in square roots, specifically, but in any nonlinear equation. We will encounter several general-use methods in section 5.2.

A complication arises when you are faced with a single polynomial and need to evaluate its zeros. In equation form:

$$c_0 + c_1 x + c_2 x^2 + c_3 x^3 + \cdots + c_{n-1} x^{n-1} = 0 \tag{5.8}$$

This is a single equation, but it has $n - 1$ roots. As we will discover in section 5.3, this problem presents its own set of challenges.

A more complicated version of the earlier problem (one nonlinear equation in one variable) is the case where we have n simultaneous nonlinear equations in n unknowns:

$$\begin{cases} f_0(x_0, x_1, \dots, x_{n-1}) & = 0 \\ f_1(x_0, x_1, \dots, x_{n-1}) & = 0 \\ & \vdots \\ f_{n-1}(x_0, x_1, \dots, x_{n-1}) & = 0 \end{cases} \tag{5.9}$$

which can be recast using our vector notation from chapter 4 as follows:

$$\mathbf{f}(\mathbf{x}) = \mathbf{0} \tag{5.10}$$

As we will discover in section 5.4, the crudest but most surefire method of solving a single nonlinear equation cannot help us when we are faced with several simultaneous equations.

The final problem we attack in this chapter is related yet distinct: instead of trying to find the zeros of a function, find the points where the function attains a minimum (or, as we will see, a maximum). Using the notation we introduced above, this problem would have the form:

$$\min \phi(\mathbf{x}) \tag{5.11}$$

where $\mathbf{x}$ bundles together the variables $x_0, x_1, \dots, x_{n-1}$ but ϕ produces scalar values.[3] This problem is tackled in section 5.5. As always, entire books have been written about each of these topics, so we will by necessity be selective in our exposition.

5.2 Nonlinear Equation in One Variable

As noted in the previous section, our problem is very easy to write down: $f(x) = 0$. In order to make things concrete, in what follows we will be trying to solve a specific equation using a variety of methods. That equation is:

$$e^{x-\sqrt{x}} - x = 0 \tag{5.12}$$

Obviously, in our case $f(x) = e^{x-\sqrt{x}} - x$. Our problem involves both an exponential and a square root, so it is clearly nonlinear.

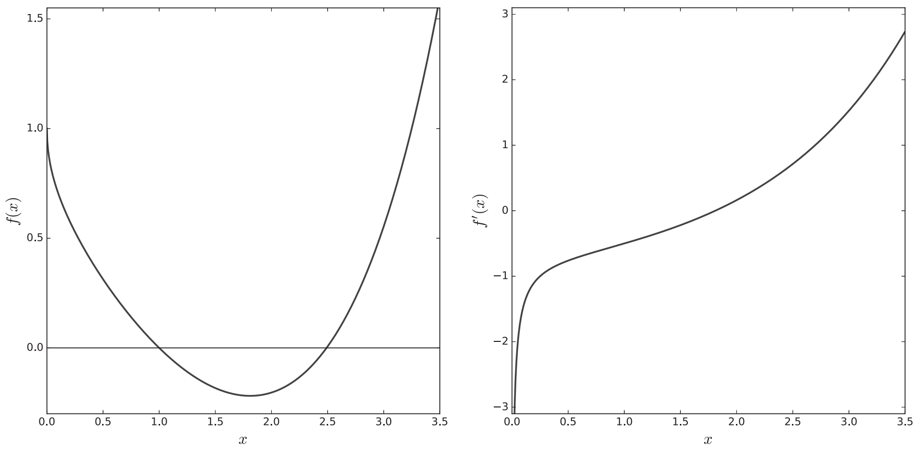
As a general lesson, before we start discussing detailed root-finding methods, we first try to get some intuition on the specific problem we are facing. Most simply, we can do that by plotting the function; the result is shown in the left panel of Fig. 5.1. Just by looking at this plot, we see that our function has two zeros, one near $x^* \approx 1$ and another one near $x^* \approx 2.5$. As a matter of fact, we can immediately see by substituting that the first root is exactly $x^* = 1$. In the same spirit of trying to understand as much as possible regarding our problem, we also went ahead and plotted the first derivative of our function in the right panel of Fig. 5.1. We find that the derivative varies from negative to positive values, with the change from one to the other happening between our two roots. Similarly, we notice that for x small and x large the derivative is large in magnitude, but it has smaller values for intermediate x's.

Now, turning to ways of solving our equation: we'll need a method that can find both roots, preferably in comparable amounts of time. (While we already analytically know one of the roots, we'll keep things general.) The most naive approach, having already plotted the function, is to try out different values (or perhaps even a very fine grid) near where we expect the roots to be. This is not a great idea, for several reasons: first, it is very wasteful, requiring a large number of function evaluations. What do you do if the first grid spacing you chose was not fine enough? You pick a smaller grid spacing and start evaluating the function again. Second, as noted in the previous section, in some cases even evaluating the function a single time may be very costly, so you might not even be able to plot the function in the first place. Finally, it is easy to see that this brute-force approach will immediately fail when you are faced with a multidimensional problem (since you'd need a multidimensional grid, with costly function evaluations at each point on the grid).

At a big-picture level, we can divide root-finding methods into two large classes:

- **Bracketing methods**: these are approaches which start with two values of x, let us call

[3] Obviously, if $n = 1$ you need to minimize a function of a single variable.



Nonlinear equation in one variable, for the case of $f(x) = e^{x-\sqrt{x}} - x$ Fig. 5.1

them x_0 and x_1, which bracket the root (i.e., we already know that $f(x_0)$ and $f(x_1)$ have opposite signs). Of the methods we discuss below, bisection and Ridders' are bracketing methods.

- **Non-bracketing methods**, also known as open-domain methods: these approaches may need one or two starting estimates of the root, but the root itself does not need to be bracketed in the first place. Of the methods introduced below, fixed-point iteration, Newton's, and the secant methods are all non-bracketing.

While the distinction between bracketing and non-bracketing methods is very important and should always be borne in mind when thinking about a given approach, we will *not* introduce these methods in that order. Instead, we will employ a pedagogical approach, starting with simple methods first (which end up being very slow) and increasing the sophistication level as we go. Let us note at the outset that we will provide Python implementations for three out of the five methods, inviting you to code up the rest in the problems at the end of the chapter. Similarly, while we provide a detailed study of convergence properties in the first few subsections, we don't do so for all five methods.

5.2.1 Conditioning

In section 4.2 we encountered the question of how $\mathbf{x}$, in the problem $\mathbf{Ax} = \mathbf{b}$, is impacted when we slightly perturb $\mathbf{A}$. This led us to introduce the condition number $\kappa(\mathbf{A})$ in Eq. (4.50), as a measure of how much (or how little) a perturbation in the coefficient matrix, $||\Delta\mathbf{A}||/||\mathbf{A}||$, is amplified when evaluating the effect on the solution vector, $||\Delta\mathbf{x}||/||\mathbf{x}||$.

We will now do something analogous, this time for the case of absolute changes, just like we did for the eigenvalue problem in Eq. (4.71). The problem we are now solving is the nonlinear equation $f(x) = 0$, so the perturbations in the system would be perturbations of the value of the function f. As it so happens, this is also closely related to what we studied in section 2.2.2 on error propagation for functions of one variable. To be specific, the exact root is here denoted by x^*, so our approximate value for it would be $\tilde{x}^*$. What we'll do is to Taylor expand $f(\tilde{x}^*)$ around x^*. This gives us:

$$\begin{aligned} f(\tilde{x}^*) - f(x^*) &= f(x^* + \Delta x^*) - f(x^*) = f(x^*) + f'(x^*)(\tilde{x}^* - x^*) + \cdots - f(x^*) \\ &\approx f'(x^*)(\tilde{x}^* - x^*) \end{aligned} \tag{5.13}$$

This is identical to the derivation we carried out in Eq. (2.36). Just like then, we made the assumption that $\tilde{x}^* - x^*$ is small, in order to drop higher-order terms.

We now remember that we are trying to consider the question of how well- or ill-conditioned a given nonlinear problem $f(x) = 0$ is; in other words, what we are after is a good estimate of x^*. To do that, we take our last result and solve it for $\tilde{x}^* - x^*$:

$$\tilde{x}^* - x^* \approx \frac{1}{f'(x^*)}\left[f(\tilde{x}^*) - f(x^*)\right] \tag{5.14}$$

allowing us to introduce a *condition number* for our problem:

$$\kappa_f = \frac{1}{f'(x^*)} \tag{5.15}$$

Clearly, the magnitude of κ_f tells us whether or not a small change in the value of the function near the root is amplified when evaluating the root itself. Qualitatively, we see that a function $f(x)$ which crosses the x axis "rapidly" leads to large $f'(x^*)$ and therefore a small condition number κ_f; this is a *well-conditioned* problem. On the other hand, a problem for which $f(x)$ is rather flat near the root, crossing the x axis "slowly", will correspond to a small $f'(x^*)$ and therefore a large condition number κ_f; this is an *ill-conditioned* problem. This should be conceptually easy to grasp: if a function is quite flat near the root, it will be difficult to tell the root apart from its neighbors. Just like in the earlier cases, we need some quasi-arbitrary criterion to help us distinguish between rapid and slow behavior.

5.2.2 Order of Convergence and Termination Criteria

Our goal is to find an x^* such that $f(x^*) = 0$. Except for very low-degree polynomials, this is a problem that does not have an analytical solution with a closed formula. In practice, we have to resort to *iterative methods*. As we saw in our discussion of the Jacobi iterative method for solving linear systems of equations in section 4.3.5, iterative methods start with a guess for the solution and then refine it until it stops changing; as a result, the number of iterations required is typically not known in advance.[4] Just like we did in that section (as well as in section 4.4 on eigenproblems), we will denote our initial root estimate by $x^{(0)}$: the superscript in parentheses tells us the iteration number (in this case we are starting, so it's the 0th iteration).

As noted, we won't actually be computing x^* itself. Instead, we will have a sequence of iterates, $x^{(0)}$, $x^{(1)}$, $x^{(2)}$, . . ., and $x^{(k)}$, which (we hope) will be approaching x^*. For now, let us assume that we are dealing with a convergent method and that $x^{(k)}$ will be close to the

[4] To make matters worse, in some cases an iterative method might not even converge.

root x^*; we'll discuss the meaning of "close" below. If there exist a constant $m \neq 0$ and a number p such that:

$$|x^{(k)} - x^*| \leq m|x^{(k-1)} - x^*|^p \tag{5.16}$$

for k sufficiently large, then m is called the *asymptotic error constant* and p is called the *order of convergence*. If $p = 1$ then we are dealing with a *linearly convergent* method, if $p = 2$ with a *quadratically convergent* method, and so on. For example, the fixed-point iteration method will turn out to be linearly convergent, whereas Newton's method will turn out to be quadratically convergent. It's worth noting that we can also find cases where p is not an integer: for example, the secant method turns out to have $p \approx 1.618$. We say that such methods are *superlinearly convergent*.[5]

Generally speaking, the order of convergence is a formal result associated with a given method, so it doesn't always help us when we are faced with an actual iterative procedure. As we found out in chapter 2 on numerics, sometimes general error bounds are too pessimistic; a general convergence trend doesn't tell you what you should expect for the specific $f(x)$ you are dealing with. To make things concrete, the order of convergence discussed in the previous paragraph relates, on the one hand, the distance between the current iterate and the true value of the root ($|x^{(k)} - x^*|$) with, on the other, the distance between the previous iterate and the true value of the root ($|x^{(k-1)} - x^*|$). In practice, we don't actually know the true value of the root (x^*): that is what we are trying to approximate. In other words, we wish to somehow quantify $|x^{(k)} - x^*|$ when all we have at our disposal are the iterates themselves, $x^{(0)}, x^{(1)}, x^{(2)}, \ldots, x^{(k)}$.

Thinking about how to proceed, one option might be to check the absolute value of the function and compare it to a small tolerance:

$$|f(x^{(k)})| \leq \epsilon \tag{5.17}$$

Unfortunately, this is an absolute criterion, which doesn't take into account the "typical" values of the function; it may be that the function is very steep near its root, in which case an ϵ like 10^{-8} would be too demanding (and therefore never satisfied). As another example, the function might have typical values of order 10^{15}: a test like our $\epsilon = 10^{-8}$ would simply be asking too much of our iterates. Conversely, the function might be very flat near its root (i.e., ill-conditioned) or involve tiny magnitudes, say 10^{-10}, even away from the root. You may be tempted, then, to somehow divide out a "typical" value of $f(x)$, in order to turn this criterion into a relative one. The problem is that it's not always clear what qualifies as typical.

In what follows, we will take another approach. Instead of checking to see how small the value of the function itself is, we will check the iterates themselves to see if they are converging to a given value. In other words, we will test the distance between the current

[5] Strictly speaking, superlinear order means that $|x^{(k)} - x^*| \leq d_k|x^{(k-1)} - x^*|$ and $d_k \to 0$ hold; for the case of quadratic convergence we can take $d_k = m|x^{(k-1)} - x^*| \to 0$, so we see that quadratic order is also superlinear.

iteration and the previous one, $|x^{(k)} - x^{(k-1)}|$.[6] We are already suspicious of using absolute criteria, so we come up with the following test:

$$\frac{|x^{(k)} - x^{(k-1)}|}{|x^{(k)}|} \leq \epsilon \tag{5.18}$$

Of course, if the numbers involved are tiny, even this criterion might lead us astray. We won't worry about this in our implementations below, but it may help you to know that one way of combining absolute and relative tolerance testing is as follows:

$$\frac{|x^{(k)} - x^{(k-1)}|}{1 + |x^{(k)}|} \leq \epsilon \tag{5.19}$$

If the iterates $x^{(k)}$ themselves are tiny and getting increasingly close to each other, here the numerator will be small but the denominator will be of order 1.

In closing, if you are feeling a bit uncomfortable about the transition from $|x^{(k)} - x^*|$ to (a possibly relativized) $|x^{(k)} - x^{(k-1)}|$ in our termination criterion, you should bring to mind the problem on the convergence properties of the Jacobi method in chapter 4. There, we directly proved a corresponding criterion for the case of that iterative method. If this is still not enough, you'll be pleased to find out that we will explicitly address the transition from one quantity to the other (for the case of fixed-point iteration) in one of the following subsections.

5.2.3 Fixed-Point Iteration

Algorithm

The first method we will introduce addresses a problem that is very similar, though not identical, to the $f(x) = 0$ problem that we started with. Specifically, this method solves the problem:

$$x = g(x) \tag{5.20}$$

As you may have already learned, a point x^* satisfying this relation is called a *fixed point* of g. For example, the function $g(x) = x^2 - 6$ has two fixed points, since $g(3) = 3$ and $g(-2) = -2$. Similarly, the function $g(x) = x + 2$ has no fixed points, since x is never equal to $x + 2$.

We could always take $g(x) = x$ and rewrite it as $f(x) = g(x) - x = 0$ so it appears that the problem we are now solving is equivalent to the one we started out with. (Not quite. Keep reading.) Thus, let us see how to solve $g(x) = x$ on the computer. Again, for concreteness, we study our problem from Eq. (5.12): $e^{x-\sqrt{x}} - x = 0$ can be trivially re-arranged to give $g(x) = e^{x-\sqrt{x}}$. We have plotted $y = x$ and $y = g(x)$ in the left panel of Fig. 5.2, where you can immediately see that this example leads to two fixed points, namely two points where

[6] As you may recall, in the case of linear algebra we were faced with an analogous but more complicated problem, since there the iterates themselves were vectors, so we had to somehow combine them together into a termination criterion; this led to Eq. (4.170), or the more robust Eq. (4.171).

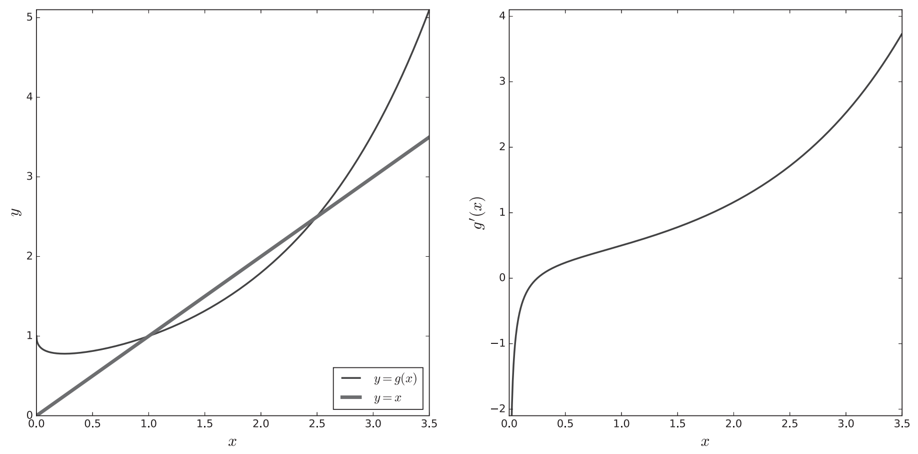


Fig. 5.2 Nonlinear equation of the fixed-point form, for the case of $g(x) = e^{x-\sqrt{x}}$

the curves meet. Unsurprisingly, these two fixed points are at $x^* \approx 1$ and $x^* \approx 2.5$, similarly to what we saw in our original problem.[7]

The simplest approach to solving Eq. (5.20) is basically to just iterate it, starting from an initial estimate $x^{(0)}$. Plug $x^{(0)}$ in to the right-hand side and from there get $x^{(1)}$:

$$x^{(1)} = g(x^{(0)}) \tag{5.21}$$

As you may have guessed, this process is repeated: $x^{(1)}$ is then plugged in to the right-hand side of Eq. (5.20), thereby producing a further improved solution $x^{(2)}$. For the general k-th iteration step, we now have our prescription for the *fixed-point iteration* method:

$$x^{(k)} = g(x^{(k-1)}), \quad k = 1, 2, \dots \tag{5.22}$$

If all is going well, the solution vector will asymptotically approach the exact solution. This process of guessing a given value, evaluating the next one, and so on, is illustrated in Fig. 5.3.[8]

Implementation

Observe that we have been making the (hyper-optimistic) assumption that everything will "go well". We will make this question concrete soon, when we discuss the convergence properties of the fixed-point iteration method. For now, we turn to a Python implementation

[7] It's not yet obvious why, but we have also plotted the first derivative of our $g(x)$ in the right panel of Fig. 5.2 – this is the same as what we saw in the right panel of Fig. 5.1, but with an offset.

[8] Incidentally, this is strongly reminiscent of what we did in the section on the Jacobi iterative method, when faced with Eq. (4.167). Of course, there we had to find many values all at once and the problem was linear.

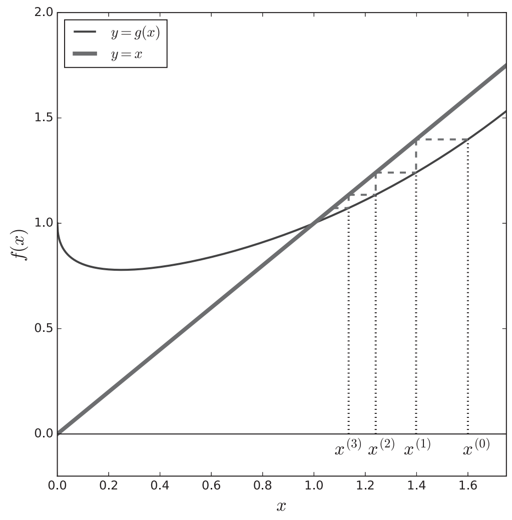


Fig. 5.3 Illustration of the fixed-point method for our example function

for the case of $g(x) = e^{x-\sqrt{x}}$, see Code 5.1. This trial-and-error approach will emphasize what "going well" and "going wrong" mean in practice.

This program is reasonably straightforward; in essence, it is a simpler version of the Jacobi code we saw in the chapter on linear algebra. After defining our $g(x)$ function, we create a function that implements the fixed-point iteration method. The latter function takes in as arguments: (a) $g(x)$, since we might wish to solve a different problem later on, (b) the initial estimate for the root $x^{(0)}$, (c) the maximum number of iterations we should keep going for (in order to make sure the process terminates even if we've been unsuccessful), and (d) the relative tolerance which will be used to determine whether we should stop iterating. Since we don't plan on changing the last two parameters often, they are given default values.

The body of the function contains a single loop which employs the `for-else` idiom that we also encountered in earlier chapters: as you may recall, the `else` is only executed if the main body of the `for` runs out of iterations without ever executing a `break` (which for us would mean that the error we calculated never became smaller than the pre-set error tolerance).

In addition to evaluating the new $x^{(k)}$ each time, we also keep track of the difference between it and the previous iterate, so that we can form the ratio $|x^{(k)} - x^{(k-1)}|/|x^{(k)}|$. We take the opportunity to print out our iteration counter, as well as the latest iterate and the (absolute) change in the value of the iterate. Obviously, you should comment out the call to `print()` inside `fixedpoint()` once you are ready to use the code for other purposes. We then say `xold = xnew` in preparation for the next iteration (should it occur).

In the main program, we simply pick a couple of values of $x^{(0)}$ and run our fixed-point iteration function using them. Wishing to reduce the number of required iterations, we pick our $x^{(0)}$ to be close to the actual root each time. Running this code we see that, for the first fixed point, we converge to the (analytically known) solution after 20 iterations. The fixedpoint.py Code 5.1

```
from math import exp, sqrt

def g(x):
     return exp(x - sqrt(x))

def fixedpoint(g,xold,kmax=200,tol=1.e-8):
     for k in range(1,kmax):
          xnew = g(xold)

          xdiff = xnew - xold
          print("{0:2d} {1:1.16f} {2:1.16f}".format(k,xnew,xdiff))

          if abs(xdiff/xnew) < tol:
               break

          xold = xnew
     else:
          xnew = None

     return xnew

if __name__ == '__main__':
     for xold in (0.99, 2.499):
          x = fixedpoint(g,xold)
          print(x)
```

absolute difference $|x^{(k)}-x^{(k-1)}|$ roughly gets halved at each iteration, so we eventually reach convergence. Obviously, the number of iterations required also depends on the starting value. For example, if we had picked $x^{(0)} = 1.2$ we would have needed 25 iterations.[9]

Turning to the second fixed point, things are more complicated. Even though we started very close to the fixed point (based on our knowledge of Fig. 5.2), we fail to converge. The absolute difference $|x^{(k)} - x^{(k-1)}|$ initially doubles at each iteration and then things get even worse, so we eventually crash. The `OverflowError` could have been avoided if we had introduced another test of the form $|x^{(k)}-x^{(k-1)}| \geq |x^{(k-1)}-x^{(k-2)}|$, intended to check whether or not we are headed in the right direction. In any case, at this point we don't really know what's different for the case of one fixed point vs the other. Thinking that maybe we picked the "wrong" starting estimate, we try again with $x^{(0)} = 2.49$; this time the run doesn't crash:

[9] Incidentally, if you're still uncomfortable with the for-else idiom, try giving `kmax` a value of, say, `10`.

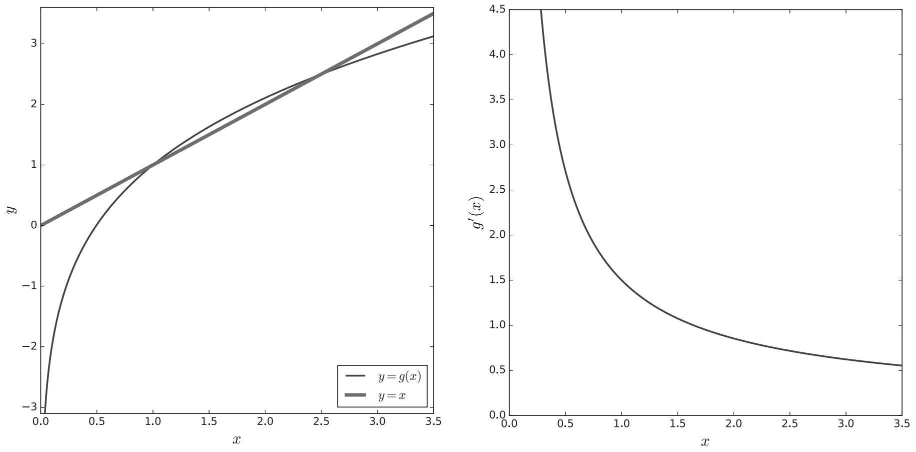


Fig. 5.4 Nonlinear equation of the fixed-point form, for the case of $g(x) = \ln x + \sqrt{x}$

the absolute difference $|x^{(k)} - x^{(k-1)}|$ first keeps increasing in magnitude, but at some point it starts decreasing, leading to eventual convergence. Unfortunately, though, this converges to the wrong fixed point! The method is giving us the first fixed point, which we had already found.

We'll let you try out several other initial estimates $x^{(0)}$ (whether by hand or systematically). That exercise won't add anything new: we always either crash or converge to the fixed point on the left. Since we're stuck, we decide to try out a different approach. Recall that the fixed-point iteration method solves the problem $x = g(x)$. Since the equation we wished to solve was $e^{x-\sqrt{x}} - x = 0$, we acted as if the two problems were equivalent and then proceeded. Here's the thing, though: our choice of $g(x)$ was *not unique*! Specifically, we could take the natural logarithm of both sides in $e^{x-\sqrt{x}} = x$ to get $x - \sqrt{x} = \ln x$, which then has the form $x = g(x)$, but this time for $g(x) = \ln x + \sqrt{x}$. We have plotted $y = x$ and this new $y = g(x)$ in the left panel of Fig. 5.4; as before, we have two fixed points, namely two points where the curves meet.[10] Unsurprisingly, these two fixed points are at $x^* \approx 1$ and $x^* \approx 2.5$, just like in the previous version of our problem. We have also plotted the first derivative of our new $g(x)$ in the right panel of Fig. 5.4; this is considerably different from both the original problem's derivative, Fig. 5.1, and our previous fixed-point derivative, Fig. 5.2.

If we code up our new function we find that our iterative procedure now crashes for the case of the *first* fixed point (using the same initial estimate as before)! Using $x^{(0)} = 2.49$, we now *do* converge to the fixed point on the right, even if we take a bit longer to get there. For the sake of completeness, note that $x^{(0)} = 2.499$ would have also worked, but would have required even more iterations to converge. Similarly, taking $x^{(0)} = 1.2$ also works, but it converges to the fixed point on the right (despite the starting value being so close to the fixed point on the left).

To summarize, for our first choice of $g(x)$ we either crash or converge to the fixed point

[10] Intriguingly, between the fixed points the two curves appear to be "closer" to each other than the corresponding two curves in Fig. 5.2 were.

on the left, whereas for our second choice of $g(x)$ we either crash or converge to the fixed point on the right. Needless to say, this does not feel like a robust solution to our initial problem, namely solving Eq. (5.12). It is now time to understand why this is taking place.

Convergence Properties

As before, we assume that our equation $x = g(x)$ has a fixed point x^* and we are using an interval in which this fixed point is unique. (That is, for our earlier examples, either near $x^* \approx 1$ or near $x^* \approx 2.5$.) In order to systematize the behavior exhibited by the fixed-point iteration method, we start with our defining relationship, Eq. (5.22):

$$x^{(k)} = g(x^{(k-1)}) \tag{5.23}$$

Since x^* is a fixed point, we know that:

$$x^* = g(x^*) \tag{5.24}$$

holds. Subtracting the last two equations gives us:

$$x^{(k)} - x^* = g(x^{(k-1)}) - g(x^*) \tag{5.25}$$

Let us now employ a Taylor expansion. This is something that we've already done repeatedly in chapter 3 on finite differences and that we'll keep on doing.[11] Let us write down the Taylor expansion of $g(x^*)$ around $x^{(k-1)}$:

$$g(x^*) = g(x^{(k-1)}) + (x^* - x^{(k-1)})g'(x^{(k-1)}) + \cdots \tag{5.26}$$

If we stop at first order, we can rewrite this expression as:

$$g(x^*) = g(x^{(k-1)}) + (x^* - x^{(k-1)})g'(\xi) \tag{5.27}$$

where ξ is a point between x^* and $x^{(k-1)}$.[12]

If we now plug $g(x^*)$ from Eq. (5.27) in to Eq. (5.25), we find:

$$x^{(k)} - x^* = g'(\xi)\left(x^{(k-1)} - x^*\right) \tag{5.28}$$

Note how on the right-hand side the order of x^* and $x^{(k-1)}$ has changed. If we now simply take the absolute value of both sides we find that our result is identical to that in Eq. (5.16) with $|g'(\xi)| \leq m$ and $p = 1$. We see that at each iteration the distance of our iterate from the true solution is multiplied by a factor of $|g'(\xi)|$; thus, if $|g'(\xi)| \leq m < 1$ we will be converging to the solution. More explicitly, if we start with a $x^{(0)}$ sufficiently close to x^*, we will have:

$$|x^{(k)} - x^*| \leq m|x^{(k-1)} - x^*| \leq m^2|x^{(k-2)} - x^*| \leq \ldots \leq m^k|x^{(0)} - x^*| \tag{5.29}$$

and since $m < 1$ we see that $x^{(k)}$ converges to x^* *linearly* (since $p = 1$). Note that each

[11] Most notably in chapter 7 on integration and in chapter 8 on differential equations.
[12] Taylor's theorem truncated to such a low order, as here, is known as the mean-value theorem.

application of the mean-value theorem needs its own ξ, but this doesn't matter as long as $|g'(\xi)| \leq m$ holds for each one of them. On the other hand, if $|g'(\xi)| > 1$ this process will diverge. Analogously, if $|g'(\xi)| < 1$ but close to 1, convergence will be quite slow. Finally, if $g'(x) = 0$ at or near the root, then the first-order term in the error vanishes; as a result, in this case the fixed-point iteration converges quadratically.[13]

Let us now apply these findings to our earlier examples. For the first $g(x)$ we have the derivative $g'(x)$ in the right panel of Fig. 5.2. We find that near the left fixed point $m < 1$ ($m = 0.5$ to be precise) and near the right fixed point $m > 1$. This is consistent with our findings using Python in the previous subsection: we cannot converge to the fixed point on the right but we do converge to the one on the left. Similarly, for the second $g(x)$ we have the derivative $g'(x)$ in the right panel of Fig. 5.4. We find that near the left fixed point $m > 1$ and near the right fixed point $m < 1$. Again, this is consistent with what we found using Python: we cannot converge to the fixed point on the left but we do converge to the one on the right. As it so happens, the latter case has $m \approx 0.72$, which explains why it took us a bit longer to converge in this case.

Before we conclude, let us return to the question of relating $|x^{(k)} - x^*|$ (which appears in the above derivation) to $|x^{(k)} - x^{(k-1)}|$ (which appears in our termination criterion). We can take Eq. (5.28) and add $g'(\xi)x^* - g'(\xi)x^{(k)}$ to both sides. After a trivial re-arrangement:

$$[1 - g'(\xi)]\left[x^{(k)} - x^*\right] = g'(\xi)\left[x^{(k-1)} - x^{(k)}\right] \tag{5.30}$$

If we now take the absolute value on both sides and assume $|g'(\xi)| \leq m < 1$, we find:

$$|x^{(k)} - x^*| \leq \frac{m}{1-m}|x^{(k)} - x^{(k-1)}| \tag{5.31}$$

Thus, as long as $m < 0.5$ (in which case $m/(1-m) < 1$) the "true convergence test" $|x^{(k)} - x^*|$ is even smaller than the quantity $|x^{(k)} - x^{(k-1)}|$ which we evaluate in our termination criterion. Conversely, if $m > 0.5$ it is possible that our termination criterion doesn't constrain true convergence very well; to take an extreme example, for $m = 0.99$ we have $m/(1-m) = 99$ so the two quantities may be two orders of magnitude apart.[14]

Finally, note that Eq. (5.31) does not include roundoff errors. The derivation above may be straightforwardly generalized to include those. The main feature is that the final computational error will only depend on the roundoff error made in the *final* iteration. This is good news, as it implies that iterative methods are *self-correcting*: minor issues in the first few iterations will not impact the final accuracy.

5.2.4 Bisection Method

Fixed-point iteration allowed us to study a root-finding method (including its convergence properties) in detail. Unfortunately, as we saw, the mapping from $f(x) = 0$ to $x = g(x)$ is not unique. Also, if $|g'(\xi)| > 1$ the fixed-point iteration method diverges. Thus, it would

[13] In a problem, you will see that we can recast Newton's method as a fixed-point iterative process.

[14] Even so, our result is merely an error *bound*, so the actual error can be much smaller.

be nice to see a different approach, one that always succeeds. This is precisely what the *bisection* method accomplishes;[15] it is a slow method, which doesn't really generalize to higher-dimensional problems, but it is safe and systematic.

Algorithm and Convergence

The bisection method assumes you have already bracketed the root; that means that you have found an x_0 and an x_1 for which $f(x_0)$ and $f(x_1)$ have opposite signs. We know from Bolzano's theorem that when a continuous function has values of opposite sign in a given interval, then the function has a root in that same interval (this is a corollary of the intermediate-value theorem). This works as follows:

- Take the interval (x_0, x_1) (for which we know $f(x_0)f(x_1) < 0$) and evaluate the midpoint:

$$x_2 = \frac{x_0 + x_1}{2} \tag{5.32}$$

- At this point, we will halve the original interval (x_0, x_1), thereby explaining why this approach is also known as the *internal halving* method. Thus, we are producing two subintervals: (x_0, x_2) and (x_2, x_1). The sign of $f(x_2)$ will determine in which of these two subintervals the root lies.
- If $f(x_0)f(x_2) < 0$, then the root is in (x_0, x_2). Thus, we could rename $x_1 \leftarrow x_2$ and repeat the entire process (which started with an x_0 and an x_1 that bracket the root).
- If, on the other hand, $f(x_0)f(x_2) > 0$, then the root is in (x_2, x_1). Thus, we could rename $x_0 \leftarrow x_2$ and repeat the entire process.
- There is also the extreme scenario where $f(x_0)f(x_2) = 0$ but, since we are dealing with floating-point numbers this is unlikely. Even so, if it does happen, it's merely informing you that you've already found the root.

Take a moment to realize that the notation we are using here is distinct from what we were doing above; x_0 and x_1 were never true "iterates", i.e., we never really thought x_1 was closer to the right answer than x_0. We could have just as easily called them a and b.[16] Actually, it may help you to think of the initial interval as (a, b); here a and b never change (since they simply correspond to the original bracket) whereas x_0 and x_1 are the numbers that keep getting updated (and simply start out with the values $x_0 = a$ and $x_1 = b$).

The best estimate at a given iteration is x_2, given by Eq. (5.32). Thus, using our earlier notation $x^{(0)}$ is the first x_2, $x^{(1)}$ is the next x_2 (after we've halved the interval and found the next midpoint), $x^{(2)}$ is the next x_2 (once we've halved the interval again), and so on. In general, at the k-th iteration $x^{(k)}$ will be the latest midpoint. This is illustrated in Fig. 5.5: for this case, the initial midpoint is $x^{(0)}$, the midpoint between that and b is $x^{(1)}$, and so on.

[15] Well, almost. Keep reading.
[16] However, this notation would get really confusing below, since Ridders' method needs a, b, c, and d.

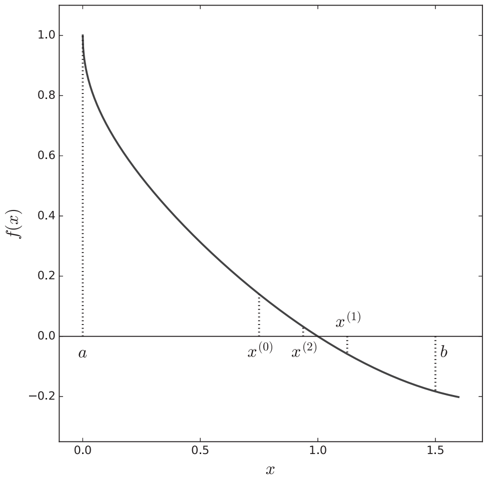


Fig. 5.5 Illustration of the bisection method for our example function

Since we are always maintaining the bracketing of the root and the interval in which the root is bracketed always gets halved, we find that:

$$|x^{(k)} - x^*| \leq \frac{b-a}{2^{k+1}} \tag{5.33}$$

Remember that the original interval had length $b - a$. Instead of taking Eq. (5.33) for granted, let's quickly derive it. For $k = 0$ we have evaluated our first midpoint, but have not actually halved the interval (i.e., we haven't carried out a selection of one of the two subintervals). Since the first midpoint is at $(b + a)/2$, the largest possible error we've made in our determination of the root is either:

$$\frac{a+b}{2} - a = \frac{b-a}{2} \tag{5.34}$$

or:

$$b - \frac{a+b}{2} = \frac{b-a}{2} \tag{5.35}$$

which amounts to the same answer. Now, for $k = 1$ we have already picked one of the subintervals (say, the one on the left) and evaluated the midpoint of *that*: this is $[a+(a+b)/2]/2$. The largest possible error in *that* determination is the distance between the latest midpoint and either of the latest endpoints; this comes out to be $(b - a)/4$. The general case follows the same pattern. In the end, Eq. (5.33) tells us that as k becomes sufficiently large we converge to x^*. Since there's always a factor of $1/2$ relating $|x^{(k)} - x^*|$ with $|x^{(k-1)} - x^*|$, we see from our general result on convergence, Eq. (5.16), that here we have $m = 1/2$ and $p = 1$, i.e., *linear* convergence.

You should keep in mind that many textbooks use the interval itself in their termination criterion. In contradistinction to this, we are here employing at each iteration only a given number, $x^{(k)}$, as an iterate (which happens to be the latest incarnation of the midpoint, x_2). This allows us to employ the same termination criterion as always, Eq. (5.18):

$$\frac{|x^{(k)} - x^{(k-1)}|}{|x^{(k)}|} \leq \epsilon \tag{5.36}$$

with the same disclaimer holding for the case where the magnitudes involved are tiny.

Implementation

Given what we've seen so far, it's reasonably straightforward to implement the bisection method in Python; the result is Code 5.2. Note that we had to first define $f(x)$, since the fixed-point iteration code employed $g(x)$, instead. Our `bisection()` method employs the interface you'd expect, first taking in $f(x)$. Then, it accepts the points determining the starting interval, x_0 and x_1.[17] The `kmax` and `tol` parameters are identical to what we had before.

The core of our new function implements Eq. (5.32) and the bullet points that follow. That implies an evaluation of a midpoint, then a halving of the interval, followed by a comparison between old and new midpoints. Note that we are trying not to be wasteful in our function evaluations, calling `f()` only once per iteration; to do that, we introduce the new variables `f0` and `f2` which get updated appropriately. We *are* being wasteful in evaluating `x2` at the start of the loop, since that will always be the same as the midpoint we had computed the last time through the loop (of course, this is a simple operation involving floats and therefore less costly than a function evaluation). When printing out values (which you can later comment out, as usual) we decided this time to also print out the (absolute) value of the function itself; even though our disclaimers above about how this could lead one astray are still valid, it's nice to have extra information in our output. Observe that the termination criterion is identical to what we had before; as per Eq. (5.18), we evaluate $|x^{(k)} - x^{(k-1)}|/|x^{(k)}|$, only this time the $x^{(k)}$ are the midpoints.

In the main program, we run our function twice, once for each root. The bracketing in this case is chosen to be quite wide: we simply look at the left panel of Fig. 5.1 and try to be conservative. Running this code leads us to finding both roots; already, things are much better than with the fixed-point iteration method. In each case we need more than a couple dozen iterations (so our approach to the final answer was somewhat slow), but success was guaranteed. Looking at the ouput in a bit more detail, we recall that the first column contains the iteration counter, the second column has the latest midpoint, the third column is the (absolute) difference in the last two midpoints, and the final column contains the absolute value of the function itself. Notice that the difference in the third columns gets halved (exactly) from one iteration to the next. As far as the function value itself goes, we generally tend to reduce its magnitude when iterating, though for the case of the root on the right the value of the function first grows a little.

[17] The fixed-point iteration method was different, taking in only a single starting value, $x^{(0)}$.

Code 5.2 **bisection.py**

```
from math import exp, sqrt

def f(x):
     return exp(x - sqrt(x)) - x

def bisection(f,x0,x1,kmax=200,tol=1.e-8):
     f0 = f(x0)
     for k in range(1,kmax):
          x2 = (x0+x1)/2
          f2 = f(x2)

          if f0*f2 < 0:
               x1 = x2
          else:
               x0, f0 = x2, f2

          x2new = (x0+x1)/2
          xdiff = abs(x2new-x2)
          rowf = "{0:2d} {1:1.16f} {2:1.16f} {3:1.16f}"
          print(rowf.format(k,x2new,xdiff,abs(f(x2new))))

          if abs(xdiff/x2new) < tol:
               break
     else:
          x2new = None

     return x2new

if __name__ == '__main__':
     root = bisection(f,0.,1.5)
     print(root); print("")
     root = bisection(f,1.5,3.)
     print(root)
```

5.2.5 Newton's Method

We now turn to *Newton's method* (sometimes also called the *Newton–Raphson method*): this is the simplest fast method used for root-finding. It also happens to generalize to largerdimensional problems in a reasonably straightforward manner. At a big-picture level, Newton's method requires more input than the approaches we saw earlier: in addition to being able to evaluate the function $f(x)$, one must also be able to evaluate its first derivative $f'(x)$. This is obviously trivial for our example above, where $f(x)$ is analytically known, but may not be so easy to access for the case where $f(x)$ is an externally provided (costly) routine. Furthermore, to give the conclusion ahead of time: there are many situations in which Newton's method can get in trouble, so it always pays to think about your specific problem instead of blindly trusting a canned routine. Even so, if you already have a reasonable estimate of where the root may lie, Newton's method is usually a fast and reliable solution.

Algorithm and Interpretation

We will assume that $f(x)$ has continuous first and second derivatives. Also, take $x^{(k-1)}$ to be the last iterate we've produced (or just an initial guess). Similarly to what we did in Eq. (5.26) above, we will now write down a Taylor expansion of $f(x)$ around $x^{(k-1)}$; this time we go up to one order higher:

$$f(x) = f(x^{(k-1)}) + \left(x - x^{(k-1)}\right) f'(x^{(k-1)}) + \frac{1}{2}\left(x - x^{(k-1)}\right)^2 f''(\xi) \tag{5.37}$$

where ξ is a point between x and $x^{(k-1)}$. If we now take $x = x^*$ then we have $f(x^*) = 0$. If we further assume that $f(x)$ is linear (in which case $f''(\xi) = 0$), we get:

$$0 = f(x^{(k-1)}) + \left(x^* - x^{(k-1)}\right) f'(x^{(k-1)}) \tag{5.38}$$

which can be re-arranged to give:

$$x^* = x^{(k-1)} - \frac{f(x^{(k-1)})}{f'(x^{(k-1)})} \tag{5.39}$$

In words: for a linear function, an initial guess can be combined with the values of the function and the first derivative (at that initial guess) to locate the root.

This motivates Newton's method: if $f(x)$ is nonlinear, we still use the same formula, Eq. (5.39), this time in order to evaluate not the root but our next iterate (which, we hope, brings us closer to the root):

$$x^{(k)} = x^{(k-1)} - \frac{f(x^{(k-1)})}{f'(x^{(k-1)})}, \quad k = 1, 2, \dots \tag{5.40}$$

As we did in the previous paragraph, we are here neglecting the second derivative term in the Taylor expansion. However if we are, indeed, converging, then $(x^{(k)} - x^{(k-1)})^2$ in Eq. (5.37) will actually be smaller than $x^{(k)} - x^{(k-1)}$, so all will be well.

Another way to view the prescription in Eq. (5.40) is *geometrically*: we approximate $f(x)$ with its tangent at the point $(x^{(k-1)}, f(x^{(k-1)}))$; then, the next iterate, $x^{(k)}$, is the point where that tangent intercepts the x axis. If this is not clear to you, look at Eq. (5.38) again. Figure 5.6 shows this step being applied repeatedly for our example function from Eq. (5.12). We make an initial guess, $x^{(0)}$, and then evaluate the tangent at the point $(x^{(0)}, f(x^{(0)}))$. We

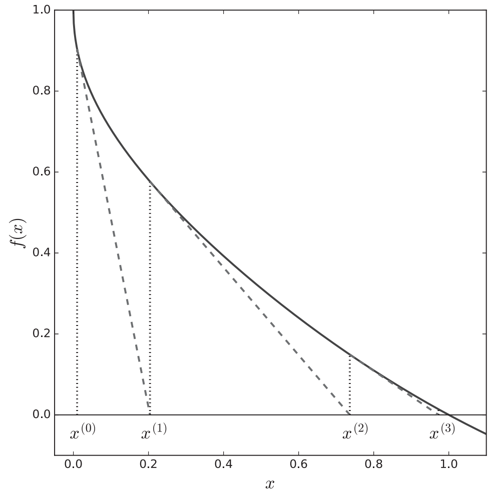


Fig. 5.6 Illustration of Newton's method for our example function

call $x^{(1)}$ the point where that tangent intercepts the x axis and repeat. For our example, this process brings us very close to the root in just a few steps.

Convergence Properties

We now turn to the convergence properties of Newton's method. To orient the reader, what we'll try to do is to relate $x^{(k)} - x^*$ to $x^{(k-1)} - x^*$, as per Eq. (5.16). We will employ our earlier Taylor expansion, Eq. (5.37), and take $x = x^*$, but this time without assuming that the second derivative vanishes. Furthermore, we assume that we are dealing with a simple root x^*, for which we therefore have $f'(x^*) \neq 0$; that means we can also assume $f'(x) \neq 0$ in the vicinity of the root. We have:

$$0 = f(x^{(k-1)}) + \left(x^* - x^{(k-1)}\right) f'(x^{(k-1)}) + \frac{1}{2}\left(x^* - x^{(k-1)}\right)^2 f''(\xi) \tag{5.41}$$

where the left-hand side is the result of $f(x^*) = 0$.

Dividing by $f'(x^{(k-1)})$ and re-arranging, we find:

$$-\frac{f(x^{(k-1)})}{f'(x^{(k-1)})} - x^* + x^{(k-1)} = \frac{\left(x^* - x^{(k-1)}\right)^2 f''(\xi)}{2f'(x^{(k-1)})} \tag{5.42}$$

The first and third terms on the left-hand side can be combined together to give $x^{(k)}$, as per Newton's prescription in Eq. (5.40). This leads to:

$$x^{(k)} - x^* = \left[\frac{f''(\xi)}{2f'(x^{(k-1)})}\right]\left(x^{(k-1)} - x^*\right)^2 \tag{5.43}$$

This is our desired result, relating $x^{(k)} - x^*$ to $x^{(k-1)} - x^*$. Taking the absolute value of both

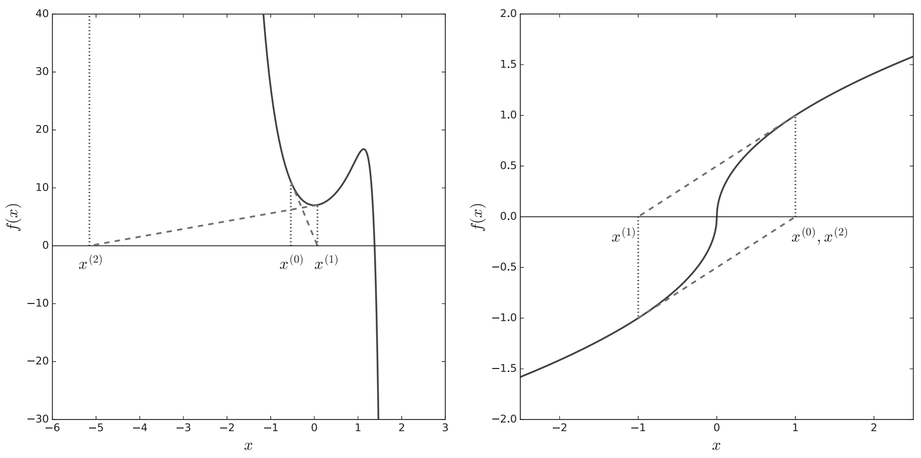


Fig. 5.7 Cases where Newton's method struggles

sides, our result is identical to that in Eq. (5.16), under the assumption that:

$$\frac{f''(\xi)}{2f'(x^{(k-1)})} \leq m \tag{5.44}$$

Since the right-hand side of Eq. (5.43) contains a square, we find that, using the notation of Eq. (5.16), $p = 2$, so if $m < 1$ holds then *Newton's method is quadratically convergent*, sufficiently close to the root.[18] This explains why our iterates approached the root so rapidly in Fig. 5.6, even though we intentionally picked a poor starting point.

Multiple Roots and Other Issues

Of course, as already hinted at above, there are situations where Newton's method can misbehave. As it so happens, our starting point in Fig. 5.6 was *near* 0, but not actually 0. For our example function $f(x) = e^{x-\sqrt{x}} - x$, the first derivative has a $\sqrt{x}$ in the denominator, so picking $x^{(0)} = 0$ would have gotten us in trouble. This can easily be avoided by picking another starting point. There are other problems that relate not to our initial guess, but to the behavior of $f(x)$ itself.

An example is given in the left panel of Fig. 5.7. Our initial guess $x^{(0)}$ is perfectly normal and does not suffer from discontinuities, or other problems. However, by taking the tangent and finding the intercept with the x axis, our $x^{(1)}$ happens to be near a local extremum (minimum in this case); since $f'(x^{(k-1)})$ appears in the denominator in Newton's prescription in Eq. (5.40), a small derivative leads to a large step, considerably away from the root. Note that, for this misbehavior to arise, our previous iterate doesn't even need to be "at" the extremum, only in its neighborhood. It's worth observing that, for this case, the root $x^* \approx 1.4$ actually *is* found by Newton's method after a few dozen iterations: after slowly creeping back toward the minimum, one iterate ends up overshooting to the right this time (past the root), after which point the solution is slowly but surely reached.

[18] Again, a problem recasts Newton's method as a special case of fixed-point iteration to find the same result.

Things are even worse in the case of the right panel of Fig. 5.7, where our procedure enters an infinite cycle. Here we start with a positive $x^{(0)}$, then the tangent brings us to a negative iterate (which happens to be the exact opposite of where we started, $x^{(1)} = -x^{(0)}$), and then taking the tangent and finding the intercept brings us back to where we started, $x^{(2)} = x^{(0)}$. After that, the entire cycle keeps repeating, without ever coming any closer to the root which we can see is at $x^* = 0$. Of course, this example is somewhat artificial; in a real-world application, the function would likely not be perfectly symmetric.

Another problem arises when we are faced with a root that is *not* simple. As you may recall, in our study of the convergence properties for Newton's method above, we assumed that $f'(x^*) \neq 0$. This is not the case when you are faced with a multiple root.[19] This is a problem not only for the convergence study but also for the prescription itself: Eq. (5.40) contains the derivative in the denominator, so $f'(x^*) = 0$ can obviously lead to trouble.

As an example, in Fig. 5.8 we are plotting the function:

$$f(x) = x^4 - 9x^3 + 25x^2 - 24x + 4 \tag{5.45}$$

which has one root at $x^* \approx 0.21$, another one at $x^* \approx 4.79$, as well as a *double root* at $x^* = 2$. This is easier to see if we rewrite our function as:

$$f(x) = (x-2)^2 \, (x^2 - 5x + 1) \tag{5.46}$$

As it turns out, Newton's method can find the double root, but it takes a bit longer than you'd expect to do so. Intriguingly, this is an example where the bisection method (and *all* bracketing methods) cannot help us: the function does not change sign at the root $x^* = 2$ (i.e., the root is not bracketed). As you will find out when you solve the corresponding problem, Newton's method for this case converges linearly! If you would like to have it converge quadratically (as is usually the case), you need to implement Newton's prescription in a slightly modified form, namely:

$$x^{(k)} = x^{(k-1)} - 2\frac{f(x^{(k-1)})}{f'(x^{(k-1)})}, \quad k = 1, 2, \dots \tag{5.47}$$

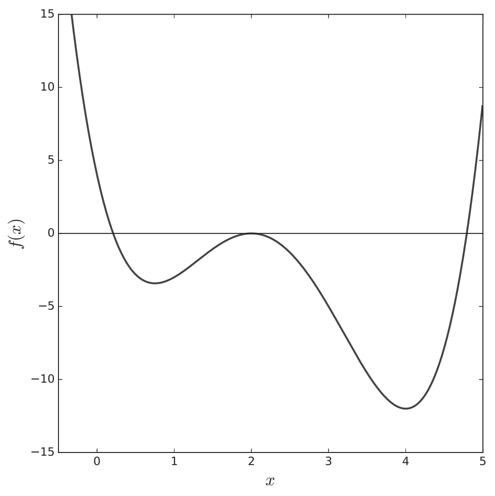
as opposed to Eq. (5.40). The aforementioned problem discusses the source of this mysterious 2 prefactor.

Unfortunately, in practice we don't know ahead of time that we are dealing with a double root (or a triple root, etc.), so we cannot pick such a prefactor that guarantees quadratic convergence. As you will discover in another problem, a useful trick (which applies regardless of your knowledge of the multiplicity of the root) is to apply Newton's prescription, but this time not on the function $f(x)$ but on the function:

$$w(x) = \frac{f(x)}{f'(x)} \tag{5.48}$$

As you will show in that problem, $w(x)$ has a simple root at x^*, regardless of the multiplicity of the root of $f(x)$ at x^*. Of course, there's no free lunch, and you end up introducing cancellations (which may cause headaches) when evaluating the derivative $w'(x)$, which you'll need to do in order to apply Eq. (5.40) to $w(x)$.

[19] Applying our definition in Eq. (5.15), we see that for multiple roots, where $f'(x^*) = 0$, the root-finding problem is always ill-conditioned. This is challenging for all approaches, not only for Newton's method.



A case where the equation has a double root

Fig. 5.8

Note, finally, that there are several other modifications to the plain Newton algorithm that one could carry out (e.g., what is known as *backtracking*, where a "bad" step is discarded). Some of these improvements are explored in the problems at the end of the chapter. Many such tricks happen to be of wider applicability, meaning that we can employ them even when we're not using Newton's method.

5.2.6 Secant Method

Given the simplicity of Newton's method, as well as the fact that it generalizes to other problems, it would be natural to provide an implementation at this point. However, since (modified) versions of Newton's method are implemented below, we don't give one now for the basic version of the algorithm. Instead, we proceed to discuss other approaches, starting from the *secant method*, which is quite similar to Newton's method. As you may recall, the biggest new requirement that the latter introduced was that you needed to evaluate not only the function $f(x)$, but also the derivative $f'(x)$. In some cases that is too costly, or even impossible. You might be tempted to approximate the derivative using the finite-difference techniques we introduced in chapter 3. However, those would necessitate (at least) two function evaluations per step. The method we are about to introduce is better than that, so will be our go-to solution for the cases where we wish to avoid derivatives.

Algorithm and Interpretation

As already hinted at, the secant method replaces the evaluation of the derivative, needed for Newton's method, by a single function evaluation. Although a naive finite-difference calculation would be bad (how do you pick h?), the secant method is inspired by such an

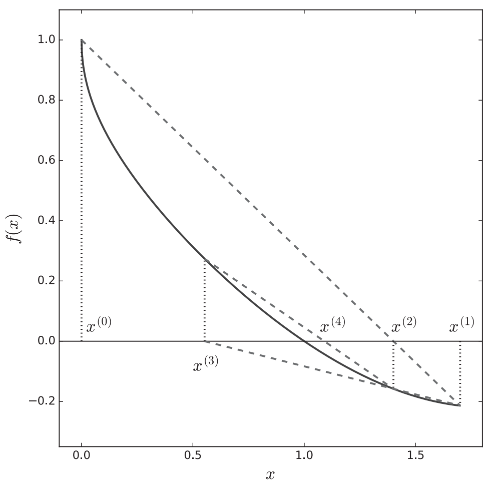


Fig. 5.9 Illustration of the secant method for our example function

approach. Specifically, this new method starts from Newton's Eq. (5.40):

$$x^{(k)} = x^{(k-1)} - \frac{f(x^{(k-1)})}{f'(x^{(k-1)})}, \quad k = 1, 2, \ldots \tag{5.49}$$

and approximates the derivative $f'(x^{(k-1)})$ using the latest two points and function values:

$$f'(x^{(k-1)}) \approx \frac{f(x^{(k-1)}) - f(x^{(k-2)})}{x^{(k-1)} - x^{(k-2)}} \tag{5.50}$$

which looks like a finite difference, where the spacing h is picked not by being on a grid or by hand, but by whatever values our latest iterates have; this implies that the spacing changes at every iteration.

Combining the last two equations, our prescription for the secant method is:

$$x^{(k)} = x^{(k-1)} - f(x^{(k-1)})\frac{x^{(k-1)} - x^{(k-2)}}{f(x^{(k-1)}) - f(x^{(k-2)})}, \quad k = 2, 3, \ldots \tag{5.51}$$

where we start the iteration at $k = 2$, since we need *two* initial guesses as starting points. Once we get going, though, we only need to evaluate the function *once* per iteration. Note how this prescription doesn't need the derivative $f'(x)$ anywhere.

You may be wondering where the name "secant" comes from. This word refers to a straight line that cuts a curve in two or more points.[20] You can understand why the method is so named if you think about the geometric interpretation of our prescription in Eq. (5.51). From elementary geometry, you may recall that the line that goes through the two points (x_0, y_0) and (x_1, y_1) is:

$$y - y_0 = \frac{y_1 - y_0}{x_1 - x_0}(x - x_0) \tag{5.52}$$

[20] Secant comes from the Latin *secare*, "to cut".

If you now compute the x axis intercept of this straight line, you will find that it is precisely of the same form as in Eq. (5.51). Thus, in the secant method, at a given step, we are producing a straight line[21] as the secant through the two points $(x^{(k-1)}, f(x^{(k-1)}))$ and $(x^{(k-2)}, f(x^{(k-2)}))$. The intercept of that straight line with the x axis gives us the next iterate, and so on.

This is nicely illustrated in Fig. 5.9 for our usual example function. Even though we (purposely) begin with two initial points that are far away from the root, we see that this process of constructing a line through the points, finding the x axis intercept, and then using the latest two points again, quickly approaches the root. Crucially, we do *not* require that our two initial guesses bracket the root. In this example, $x^{(0)}$ and $x^{(1)}$ do happen to bracket the root, but then $x^{(1)}$ and $x^{(2)}$ don't; this is irrelevant to us, since we're simply drawing a straight line that goes through two points.

From a purely numerical perspective, we observe that the secant method requires the evaluation of the ratio $[x^{(k-1)} - x^{(k-2)}]/[f(x^{(k-1)}) - f(x^{(k-2)})]$. As you may have suspected, typically both numerator and denominator will be small, so this ratio will be evaluated with poor relative accuracy. Fortunately, it's the coefficient in front, $f(x^{(k-1)})$, that matters more. In the problem set, you will show that the order of convergence of the secant method is the golden ratio, $p \approx 1.618$. This is superlinear convergence: not as fast as Newton's method, but the fact that we don't need to evaluate the derivative may make this trade-off worthwhile.

Implementation

Code 5.3 is a Python program that implements our prescription from Eq. (5.51). Observe that the parameter list is identical to that of our `bisection()` function from a previous section.[22] The rest of the function is pretty straightforward: we keep forming the ratio of the difference between abscissas and function values, use it to produce a new iterate, and evaluate the difference between our best current estimate and the previous one. This part of the code happens to be easier to read (and write) than `bisection()` was, since there we needed notation for the old vs the new midpoint. As usual, we print out a table of values for informative purposes and then check to see if we should break out of the loop. It's worth observing that we are trying to avoid being wasteful when computing `ratio` and `x2`: similarly to what we did in `bisection()`, we introduce intermediate variables `f0` and `f1` to ensure that there is only one function evaluation per iteration.

The main program uses initial guesses for each of the roots. For the first root, we pick $x^{(0)}$

[21] This time not as the tangent at the point $(x^{(k-1)}, f(x^{(k-1)}))$, as we did in Newton's method.

[22] Of course, there the `x0` and `x1` were required to bracket the root, whereas here they don't have to. While we're on the subject, note that in our earlier program `x0`, `x1`, and `x2` stood for x_0, x_1, and x_2, whereas now `x0`, `x1`, and `x2` stand for $x^{(k-2)}$, $x^{(k-1)}$, and $x^{(k)}$.

Code 5.3 secant.py

```
from bisection import f

def secant(f,x0,x1,kmax=200,tol=1.e-8):
    f0 = f(x0)
    for k in range(1,kmax):
        f1 = f(x1)
        ratio = (x1 - x0)/(f1 - f0)
        x2 = x1 - f1*ratio

        xdiff = abs(x2-x1)
        x0, x1 = x1, x2
        f0 = f1

        rowf = "{0:2d} {1:1.16f} {2:1.16f} {3:1.16f}"
        print(rowf.format(k,x2,xdiff,abs(f(x2))))

        if abs(xdiff/x2) < tol:
            break
    else:
        x2 = None

    return x2

if __name__ == '__main__':
    root = secant(f,0.,1.7)
    print(root); print("")
    root = secant(f,2.,2.1)
    print(root)
```

and $x^{(1)}$ to match what we saw in Fig. 5.9. For the second root, we pick two values, close to where we expect the root to lie, as is commonly done in practice. Running this code, we see that not too many iterations are required to reach the desired tolerance target. This is much faster than the fixed-point iteration; even more importantly, the secant method allows us to evaluate both roots, without having to carry out any analytical manipulations first. Of course, there was some guesswork involved when picking the two pairs of initial guesses. The secant method also took considerably fewer iterations than the bisection method and barely more than Newton's method. All in all, with basically no bookkeeping and no derivative-evaluations, it has allowed us to finds both roots fast.

5.2.7 Ridders' Method

We now turn to *Ridders'* method: this approach is "only" 40 years old, making it one of the most modern topics in this volume.[23] We back up for a moment to realize that the last two methods we've encountered (Newton's, secant) produced the next iterate by finding the x axis intercept of a straight line; they only differ in how that straight line is produced. Ridders' method follows the same general idea, of finding the x axis intercept of a straight line, but chooses this line not by the value of the function $f(x)$ or its derivative $f'(x)$, but through a clever trick. While our derivation below will be somewhat long, the prescription we arrive at is quite simple, requiring very little bookkeeping.

Ridders' is a bracketing method, so it assumes $f(x_0)f(x_1) < 0$. The main idea is to multiply $f(x)$ with the unique exponential function which turns it into a straight line:

$$R(x) = f(x)\, e^{Qx} \tag{5.53}$$

That is, regardless of the shape of $f(x)$, the new $R(x)$ will be linear, $R(x) = c_0 + c_1 x$. Since we have three undetermined parameters (c_0, c_1, and Q), we will use three points to pin them down. We take these to be our initial bracket's endpoints, x_0 and x_1, as well as their midpoint, $x_2 = (x_0 + x_1)/2$. A moment's thought will convince you that, since $R(x)$ is a straight line, we will have:

$$R_2 = \frac{R_0 + R_1}{2} \tag{5.54}$$

where we are using the notation $R_i \equiv R(x_i)$; we will soon also employ the corresponding $f_i \equiv f(x_i)$. Let us further define:

$$d = x_2 - x_0 = x_1 - x_2 \tag{5.55}$$

Using Eq. (5.53) and Eq. (5.54), we find:

$$f_0\, e^{Qx_0} + f_1\, e^{Qx_1} - 2f_2\, e^{Qx_2} = 0 \tag{5.56}$$

Factoring out e^{Qx_0} leads to:

$$f_1\, e^{2Qd} - 2f_2\, e^{Qd} + f_0 = 0 \tag{5.57}$$

which you can see is a quadratic equation in e^{Qd}. Solving it leads to:

$$e^{Qd} = \frac{f_2 - \text{sign}(f_0)\sqrt{f_2^2 - f_0 f_1}}{f_1} \tag{5.58}$$

Since we are starting with the assumption that $f_0 f_1 < 0$, we see that $f_2^2 - f_0 f_1 > 0$, as it should be. For the same reason, the square root of $f_2^2 - f_0 f_1$ is larger in magnitude than f_2.

[23] And, in all likelihood, in your undergraduate education.

The $-\mathrm{sign}(f_0)$ is deduced from the fact that $e^{Qd} > 0$. To see this, we distinguish between two possible scenarios, (a) $f_0 < 0$ and $f_1 > 0$, (b) $f_0 > 0$ and $f_1 < 0$. Taking the first scenario, we see that the denominator in Eq. (5.58) is positive, so the numerator should also be positive. For this scenario, there are two possibilities regarding the sign of f_2. If $f_2 > 0$, then the square root term has to come in with a plus sign in order to get a positive numerator. Similarly, if $f_2 < 0$ then the first term in the numerator is negative so, again, the square root term (which is larger in magnitude) has to come in with a plus sign to ensure $e^{Qd} > 0$. An analogous line of reasoning leads to a minus sign in front of the square root for the second scenario, $f_0 > 0$ and $f_1 < 0$. Overall, we need $-\mathrm{sign}(f_0)$, as in Eq. (5.58).

Looking at Eq. (5.58), we see that we have determined Q in terms of the known values d, f_0, f_1, and f_2. That means that we have fully determined the full $R(x)$, as per our starting Eq. (5.53). Given that $R(x)$ is a straight line, we can play the same game as in the secant method, namely we can find the x axis intercept. To be clear, that means that we will produce a new point x_3 as the x axis intercept of the line going through (x_1, R_1) and (x_2, R_2), which of course also goes through (x_0, R_0); we apply Eq. (5.51):

$$x_3 = x_2 - R_2\frac{x_2 - x_1}{R_2 - R_1} = x_2 - \frac{d}{R_1/R_2 - 1} = x_2 + \mathrm{sign}(f_0)\frac{f_2 d}{\sqrt{f_2^2 - f_0 f_1}} \tag{5.59}$$

In the second equality we brought the R_2 to the denominator and identified d in the numerator (along with changing the sign of both numerator and denominator). In the third equality we employed the fact that:

$$\frac{R_1}{R_2} = \frac{f_1\, e^{Qx_1}}{f_2\, e^{Qx_2}} = \frac{f_1}{f_2}\, e^{Qd} \tag{5.60}$$

together with our result in Eq. (5.58) for e^{Qd}. To reiterate, we found the x axis intercept of $R(x)$; we then re-expressed everything in terms of the function $f(x)$, which was how our initial problem was formulated.

Our last result in Eq. (5.59) is something you could immediately implement programmatically. It only requires $f(x)$ evaluations to produce a root estimate, x_3. We'll follow Ridders' original work in making an extra step, in order to remove the factor $\mathrm{sign}(f_0)$.[24] Divide both the numerator and denominator with f_0 to find:

$$x_3 = x_2 + (x_1 - x_2)\frac{f_2/f_0}{\sqrt{(f_2/f_0)^2 - f_1/f_0}} \tag{5.61}$$

where we also took the oportunity to re-express d as per Eq. (5.55).[25] We now notice a nice property of Ridders' prescription: since $f_0 f_1 < 0$, we see that the denominator is larger in

[24] In the literature after Ridders' paper, that factor is given as $\mathrm{sign}(f_0 - f_1)$. However, since we already know that $f_0 f_1 < 0$, it's simpler to use $\mathrm{sign}(f_0)$.

[25] This form of the equation for x_3 might make you a bit uncomfortable from a numerical perspective: f_0, f_1, and f_2 are expected to become very small in magnitude as we approach the root, so perhaps we shouldn't be dividing these small numbers with each other and then subtracting the ratios.

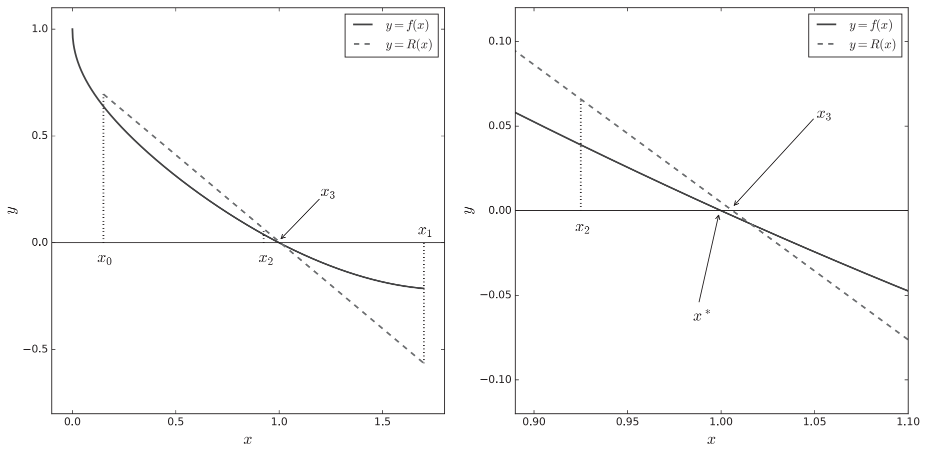


Illustration of Ridders' method for our example function

Fig. 5.10

magnitude than the numerator. That shows that x_3 will certainly stay within the original bracket between x_0 and x_1; thus, Ridders' method is guaranteed to converge.

So far, we've discussed only the procedure that allows us to produce our first root estimate, x_3, i.e., we haven't seen how to iterate further. Even so, it may be instructive at this point to see Ridders' method in action; in the left panel of Fig. 5.10 we show our example function, as usual. We have picked our initial bracket x_0 and x_1 in such a way as to highlight that the function $f(x)$ and the straight line $R(x)$ are totally different entities. As a reminder, in Ridders' method we start from x_0 and x_1, we evaluate the midpoint x_2, and then use the function values f_0, f_1, and f_2 as per Eq. (5.58) and Eq. (5.53) to produce the line $R(x)$. Our root estimate x_3 is the point where $R(x)$ crosses the x axis. As you can see in Fig. 5.10, for this problem just one iteration of this algorithm was enough to get very close to the true root x^*. In order to clarify that the x axis intercepts of $f(x)$ and of this first $R(x)$ are not necessarily the same, we provide a zoomed-in version of the plot in the right panel of Fig. 5.10.

After we've evaluated our first estimate for the root, in the way just discussed, Ridders' method goes on to form a new bracket, made up of x_3 and one of the existing x_i values. Specifically, we check if $f_3 f_i < 0$ for $i = 0, 1, 2$ to ensure that the root is still bracketed between x_3 and a specific x_i. We then rename the two endpoints appropriately, i.e., to x_0 and x_1, and repeat the entire process.

Overall, Ridders' method is a reliable bracketing method. As we've seen in our example case, it converges more rapidly than the bisection method, typically quadratically. Of course, this is slightly misleading, since each iteration of Ridders' method requires two function evaluations, one for the midpoint and one for an endpoint (which was an x axis intercept in the previous iteration). Even so, this is still superlinear and therefore faster than the bisection method.

5.2.8 Summary of One-Dimensional Methods

We've already encountered five different methods that solve nonlinear equations. Here's a quick summary of their main features:

- *Fixed-point method*: a non-bracketing method; simple; doesn't always converge.
- *Bisection method*: a bracketing method; simple; always converges; linear convergence.
- *Newton's method*: a non-bracketing method; requires the function derivative; usually quadratic convergence; sometimes slow or worse.[26]
- *Secant method*: a non-bracketing method; similar to Newton's method but doesn't require the function derivative; usually superlinear convergence; sometimes slow or worse.
- *Ridders' method*: a bracketing method; superlinear convergence; rarely slow.

There are even more speedy and reliable methods out there. Most notably, *Brent's method* is a bracketing approach which is as reliable as the bisection method, with a speed similar to that of the Newton or secant methods (for non-pathological functions). However, the bookkeeping it requires is more complicated, so in the spirit of the rest of this volume we omit it from our presentation. Note that for all bracketing methods our earlier disclaimer applies: if you have a root that is not bracketed, like in Fig. 5.8, you will not be able to use such an approach.

Combining all of the above pros and cons, in what follows we will generally reach for Newton's method (especially when generalizing to multiple dimensions below) or the secant method (when we don't have any information on the derivative, as in chapter 8). When you need the speed of these methods and the reliability of bracketing methods, you can employ a "hybrid" approach, as touched upon in the problem set.

5.3 Zeros of Polynomials

As noted at the start of this chapter, a special case of solving a nonlinear equation in one variable has to do with polynomial equations, of the form:

$$c_0 + c_1 x + c_2 x^2 + c_3 x^3 + \cdots + c_{n-1} x^{n-1} = 0 \tag{5.62}$$

The reason this gets its own section is as follows: we know from the fundamental theorem of algebra that a polynomial of degree $n-1$ has $n-1$ roots, so this problem is already different from what we were dealing with above, where only one or two roots needed to be evaluated.

[26] Leading some people to say that Newton's method should be avoided; this is too strong a statement.

5.3.1 Challenges

Let us start out by discussing the special challenges that emerge in polynomial root-finding. We've already encountered one of these in a previous section; as you may recall, for *multiple roots* we have $f'(x^*) = 0$ so, using our definition from Eq. (5.15), we saw that the root-finding problem in this case is always ill-conditioned. We already learned that bracketing methods may be inapplicable in such scenarios, whereas methods (like Newton's) which are usually lightning fast, slow down.

Based on the intuition you developed when studying error analysis in linear algebra, in section 4.2, you probably understand what being ill-conditioned means: if you change the coefficients of your problem slightly, the solution is impacted by a lot. Let's look at a specific example, motivated by Eq. (5.46):

$$x^2 - 4x + 4 = 0 \tag{5.63}$$

This has a double root at $x^* = 2$, as you can easily see by gathering the terms together into $(x-2)^2 = 0$. A small perturbation of this problem has the form:

$$x^2 - 4x + 3.99 = 0 \tag{5.64}$$

Even though we've only perturbed one of the coefficients by 0.25%, the equation now has two *distinct* roots, at $x^* = 2.1$ and $x^* = 1.9$. Each of these has changed its value by 5% in comparison to the unperturbed problem. This effect is considerably larger than the change in the coefficient. Things can get even worse, though. Consider the perturbed equation:

$$x^2 - 4x + 4.01 = 0 \tag{5.65}$$

This one, too, has a single coefficient perturbed by 0.25%. Evaluating the roots in this case, however, leads to $x^* = 2 + 0.1i$ and $x^* = 2 - 0.1i$, where i is the *imaginary unit*! A small change in one coefficient of the polynomial changed the real double root into a complex conjugate pair, with a sizable imaginary part (i.e., there are no longer real roots).

Even so, we don't want you to get the impression that polynomials are ill-conditioned only for the case of multiple roots. To see that, we consider the infamous "Wilkinson polynomial":

$$\begin{aligned} W(x) &= \prod_{k=1}^{20}(x-k) = (x-1)(x-2)\dots(x-20) \\ &= x^{20} - 210x^{19} + 20,615x^{18} + \cdots + 2\,432\,902\,008\,176\,640\,000 \end{aligned} \tag{5.66}$$

Clearly, this polynomial has the 20 roots $x^* = 1, 2, \dots, 20$. The roots certainly appear to be well-spaced, i.e., nowhere near being multiple. One would naively expect that a tiny change in the coefficients would not have a large effect on the locations of the roots. Wilkinson examined the effect of changing the coefficient of x^{19} from `-210` to:

```
-210 - 2**(-23) = -210.0000001192093
```

which is a single-coefficient modification of relative magnitude $\approx 6 \times 10^{-8}\%$. The result

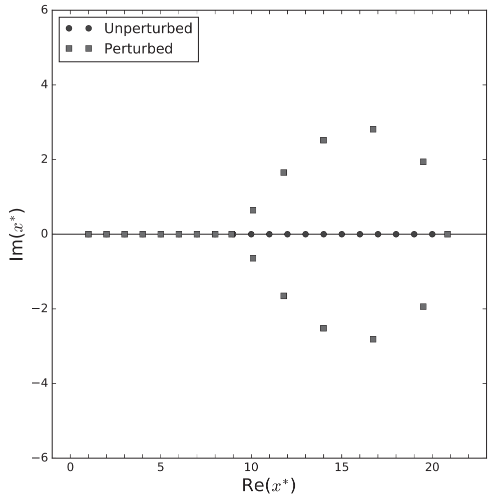


Fig. 5.11 Roots of the Wilkinson polynomial on the complex plane

of this tiny perturbation ends up being dramatic, as shown in Fig. 5.11. Half the roots have now become complex, with large imaginary parts; the largest effect is on the roots that used to be $x^* = 15$ and 16. It's important to emphasize that these dramatic changes are not errors in our root-finding: we are calculating the true roots of the perturbed polynomial, these just happen to be very different from the true roots of the original polynomial. Whatever root-finding algorithm you employ, you should make sure the errors it introduces are small enough not to impact the interpretation of your actual polynomial equation.

You may recall that in chapter 2 we studied the question of how to *evaluate* a polynomial when you have access to the coefficients c_i, using what is known as *Horner's rule*. In this section, we've been examining the problem of solving for the polynomial's zeros, when you have access to the coefficients c_i. Our conclusion has been that the problem of finding the roots starting from the coefficients can be very ill-conditioned, so this approach should be avoided if possible. While this is an extreme example, the general lesson is that you should be very careful in the earlier mathematical steps which lead to a polynomial problem. Given such idiosyncratic behavior, you will not be surprised to hear that there exist specialized algorithms, designed specifically for finding zeros of polynomials. Some of these techniques suppress an already-found root, as touched upon in the problem set. In what follows, we begin with a more pedestrian approach.

5.3.2 One Root at a Time: Newton's Method

The easiest way to solve for a polynomial's zeros is to treat the problem as if it was any other nonlinear equation and employ one of the five methods we introduced in the previous section, or other ones like them. Specifically, you may recall that, if you have access to the function's derivative and you start close enough to the root, Newton's method is a very fast way of finding an isolated root. For a high-degree polynomial this will entail finding many roots one after the other. Thus, it becomes crucial that you have a good first estimate of where the roots lie. Otherwise, you will be forced to guess, for example picking a grid of starting values; many of these will likely end up finding the same roots over and over, thereby wasting computational resources. Even worse, in order to find *all* the zeros of the polynomial, you may be forced to repeat the entire exercise with a finer grid.

We will address a specific problem, that of finding the roots of *Legendre polynomials*. As you may recall, we first encountered these in the context of the multipole expansion (chapter 2) and then saw them again in our discussion of Gram–Schmidt orthogonalization (chapter 4). We will re-encounter them when we introduce the theory of Gaussian quadrature (chapter 7); as it so happens, at that point we will need their roots, so here we are providing both scaffolding and a pedagogical example.[27]

Fortunately, Legendre polynomials have been extensively studied and tabulated, so we can use several of their properties. Take $P_n(x)$ to be the n-th degree Legendre polynomial[28] and denote its i-th root by x_i^*; as usual, we use 0-indexing, meaning that the n roots are labelled using $i = 0, 1, \ldots, n-1$. If we convert Eq. (22.16.6) from Ref. [1] to use our counting scheme, we learn that the true roots x_i^* obey the following inequality:

$$\cos\left(\frac{2i+1}{2n+1}\pi\right) \leq x_i^* \leq \cos\left(\frac{2i+2}{2n+1}\pi\right) \tag{5.67}$$

This allows us to start Newton's algorithm with the following initial guess:

$$x_i^{(0)} = \cos\left(\frac{4i+3}{4n+2}\pi\right) \tag{5.68}$$

where we are using a subscript to denote the cardinal number of the root[29] and a superscript in parentheses to denote the starting value in Newton's method. Note that we multiplied both numerator and denominator with 2 in order to avoid having a $3/2$ in the numerator.

We have now accomplished what we set out to do, namely to find a reasonably good starting estimate of the n roots of our polynomial. The other requirement that Newton's method has is access to the function derivative. Fortunately for us, we have already produced, in `legendre.py`, a function that returns both the Legendre polynomial and its first derivative at a given x. Intriguingly, we accomplished this without ever producing the coefficients of the Legendre polynomials, so we are not too worried about questions of conditioning. Thus, we are now ready to implement in Python a root-finder for Legendre polynomials; the result is Code 5.4. We start out by importing our earlier `legendre()` and then define two new functions; let's discuss each of these in turn.

The function `legnewton()` is designed specifically to find a single root of a Legendre polynomial. In other words, it looks quite a bit like what a Newton's method function would have looked like, had we implemented it in an earlier section; however, it does *not*

[27] Note also how we are following the separation of concerns principle: Legendre-polynomial evaluation is different from Legendre-polynomial root-finding, and both of these are different from Gaussian quadrature, so there should be (at least) three Python functions involved here.

[28] Don't get confused: $P_n(x)$ involves x^n, just like our $(n-1)$-th degree polynomial in Eq. (5.62) involves x^{n-1}.

[29] Here x_i is referring to distinct roots, *not* to bracketing endpoints, like x_0 and x_1 above.

Code 5.4 legroots.py

```python
from legendre import legendre
import numpy as np

def legnewton(n,xold,kmax=200,tol=1.e-8):
    for k in range(1,kmax):
        val, dval = legendre(n,xold)
        xnew = xold - val/dval

        xdiff = xnew - xold
        if abs(xdiff/xnew) < tol:
            break

        xold = xnew
    else:
        xnew = None
    return xnew

def legroots(n):
    roots = np.zeros(n)
    npos = n//2
    for i in range(npos):
        xold = np.cos(np.pi*(4*i+3)/(4*n+2))
        root = legnewton(n,xold)
        roots[i] = -root
        roots[-1-i] = root
    return roots

if __name__ == '__main__':
    roots = legroots(9); print(roots)
```

take in as parameters a general function `f` and its derivative `fprime`. Instead, it is designed to work only with `legendre()` which, in its turn, had been designed to return the values of a Legendre polynomial and its first derivative as a tuple.[30] The rest of the function, with its termination criterion and so on, looks very similar to what we saw in, say, `secant()`.

The next function, `legroots()`, takes care of some rather messy bookkeeping. In essence, all it does is to repeatedly call `legnewton()` with different initial guesses corresponding

[30] Note that if you had tried to use a pre-existing general `newton()` function, you would have had to go through syntactic contortions in order to make the two interfaces match.

to each root, as per Eq. (5.68). The messiness appears because we have decided to halve our workload: instead of calculating n roots for the n-th degree polynomial, we notice that these are symmetrically distributed around 0. Specifically, if n is even, then we can simply evaluate only $n/2$ roots (say, the positive ones) and we can rest assured that the other $n/2$ can be trivially arrived at. If n is odd, then 0 is also a root, so we need only evaluate $(n - 1)/2$ positive roots. The code does just that, by starting out with an array full of zeros, `roots`, which all get overwritten in the case of n-even but provide us with the needed 0 for the case of n-odd. Notice that we employed Python's integer division to find the number of positive roots, `n//2`, which works correctly for both n-even and n-odd. We start `roots` out with the negative roots and then introduce the corresponding positive roots counting from the end: as you may recall, `roots[-1-i]` is the idiomatic way of accessing the last element, the second-to-last element, and so on.

Unsurprisingly, `numpy` has a function that carries out the same task as our `legroots()`; you may wish to use `numpy.polynomial.legendre.leggauss()` to compare with our results; this actually returns two arrays, but we are interested only in the first one. In the main program we try out the specific case of `n = 9`, in order to test what we said earlier about the case of n-odd. The output of running this program is:

```
[-0.96816024 -0.83603111 -0.61337143 -0.32425342 0. 0.32425342
  0.61337143  0.83603111  0.96816024]
```

As expected, the roots here are symmetrically distributed around 0. You will see in the problem set two other ways of arriving at the same roots.

5.3.3 All the Roots at Once: Eigenvalue Approach

The approach we covered in the previous section, of evaluating Legendre polynomial roots one at a time, seems perfect: we didn't have to go through the intermediate step of computing the polynomial coefficients, so we didn't get in trouble with the conditioning of the root-finding problem. However, there was one big assumption involved, namely that we had good initial guesses for each root available, see Eq. (5.68). But what if we didn't? More specifically, what if you are dealing with some new polynomials which have not been studied before? In that case, our earlier approach would be inapplicable. We now turn to a different method, which uses matrix techniques to evaluate all of a polynomial's roots at once. We state at the outset that this approach requires knowledge of the coefficients of the polynomial, so our earlier disclaimers regarding the conditioning of root-finding in that case also apply here. Another disclaimer: this technique is more costly than polynomial-specific approaches like Laguerre's method (which we won't discuss).

Qualitatively, the idea involved here is as follows: we saw in chapter 4 that we shouldn't evaluate a matrix's eigenvalues by solving the characteristic equation; you now know that this was due to the conditioning issues around polynomial root-finding. However, there's nothing forbidding you from taking the reverse route: take a polynomial equation, map it onto an eigenvalue problem and then use your already-developed robust eigensolvers.

Motivated by Eq. (5.62), we define the following polynomial:

$$p(x) = c_0 + c_1x + c_2x^2 + c_3x^3 + \cdots + c_{n-1}x^{n-1} + x^n \tag{5.69}$$

Note that $p(x)$ also includes an x^n term, with a coefficient of 1. This is known as a *monic polynomial*; you can think of it as a monomial plus a polynomial of one degree lower. Even though it may seem somewhat arbitrary, we now define the following *companion matrix* which has dimensions $n \times n$:[31]

$$\mathbf{C} = \begin{pmatrix} 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ 0 & 0 & 0 & \ddots & \vdots \\ \vdots & \vdots & \ddots & 0 & 1 \\ -c_0 & -c_1 & \dots & -c_{n-2} & -c_{n-1} \end{pmatrix} \tag{5.70}$$

We picked this matrix because it has a very special *characteristic polynomial*. As you may recall from Eq. (4.13), the characteristic polynomial for our companion matrix $\mathbf{C}$ will be formally arrived at using a determinant, i.e., $\det(\mathbf{C} - \lambda\mathcal{I})$. Now here is the remarkable property of our companion matrix: if you evaluate this determinant, it will turn out to be (as a function of λ) equal to our starting polynomial (up to an overall sign), i.e., $p(\lambda)$! Therefore, if you are interested in the solutions of the equation $p(\lambda) = 0$, which is our starting problem in Eq. (5.69), you can simply calculate the eigenvalues using a method of your choosing. We spent quite a bit of time in chapter 4 developing robust eigenvalue-solvers, so you could use one of them; note that you have hereby mapped one nonlinear equation with n roots to an $n \times n$ eigenvalue problem. Intriguingly, this provides you with *all* the eigenvalues at once, in contradistinction to what we saw in the previous subsection, where you were producing them one at a time.

In the previous paragraph we merely stated this remarkable property of companion matrices. Actually there were two interrelated properties: (a) the characteristic polynomial of $\mathbf{C}$ is (up to a sign) equal to $p(\lambda)$, and therefore (b) the roots of $p(\lambda)$ are the eigenvalues of $\mathbf{C}$. Here, we are really interested only in the latter property; let's prove it. Assume x^* is a root of our polynomial $p(x)$, i.e., $p(x^*) = 0$ holds. It turns out that the eigenvector of $\mathbf{C}$ is simply a tower of monomials in x^*, specifically $(1 \; x^* \; (x^*)^2 \; \dots \; (x^*)^{n-2} \; (x^*)^{n-1})^T$; observe that this is an $n \times 1$ column vector. Let's explicitly see that this is true, by acting with our

[31] Amusingly enough, mathematicians have gone on to also define a "comrade matrix" and a "colleague matrix".

companion matrix on this vector:

$$\mathbf{C}\begin{pmatrix} 1 \\ x^* \\ \vdots \\ (x^*)^{n-2} \\ (x^*)^{n-1} \end{pmatrix} = \begin{pmatrix} x^* \\ (x^*)^2 \\ \vdots \\ (x^*)^{n-1} \\ -c_0 - c_1 x^* - c_2(x^*)^2 - \ldots - c_{n-1}(x^*)^{n-1} \end{pmatrix} = \begin{pmatrix} x^* \\ (x^*)^2 \\ \vdots \\ (x^*)^{n-1} \\ (x^*)^n \end{pmatrix} = x^* \begin{pmatrix} 1 \\ x^* \\ \vdots \\ (x^*)^{n-2} \\ (x^*)^{n-1} \end{pmatrix} \tag{5.71}$$

In the first equality we simply carried out the matrix multiplication. In the second equality we used the fact that x^* is a root, so $p(x^*) = 0$, which can be re-expressed in the form $(x^*)^n = -c_0 - c_1 x^* - c_2(x^*)^2 - \cdots - c_{n-1}(x^*)^{n-1}$. In the third equality we pulled out a factor of x^*. Thus, we have managed to prove not only that the column vector we said was an eigenvector is, indeed, an eigenvecor, but also that x^* is the corresponding eigenvalue![32]

To summarize, we can evaluate the roots of our polynomial $p(x)$ from Eq. (5.69) by computing the eigenvalues of the companion matrix from Eq. (5.70). Make sure you remember that Eq. (5.69) is a *monic* polynomial. If in your problem the coefficient of x^n is different from 1, simply divide all the coefficients with it; you're interested in the roots, and this operation doesn't impact them. From there onwards, everything we've said in this subsection applies. To be concrete, for a monic polynomial of degree 4, you will have a 4×4 companion matrix and a 4×1 eigenvector. One of the problems asks you to implement this approach in Python, using the eigenvalue solvers we developed in the previous chapter, `qrmet.py` or `eig.py`.

5.4 Systems of Nonlinear Equations

We now turn to a more complicated problem, that of n simultaneous nonlinear equations in n unknowns. Employing the notation of Eq. (5.10), this can be expressed simply as $\mathbf{f}(\mathbf{x}) = \mathbf{0}$. In the earlier section on the one-variable problem, the approach that was the easiest to reason about and was also guaranteed to converge was the bisection method. Unfortunately, this doesn't trivially generalize to the many-variable problem: you would need to think about all the possible submanifolds and evaluate the function (which, remember, may be a costly operation) a very large number of times to check if it changed sign. On the other hand, the fixed-point iteration method generalizes to the n-variable problem straightforwardly (as you will discover in a problem) but, as before, is not guaranteed to converge. What we would like is a fast method that can naturally handle an n-dimensional space. It turns out that Newton's method fits the bill, so in the present section we will discuss mainly variations of this general approach. It should come as no surprise that, in that process, we will use matrix-related functionality from chapter 4.

Writing down $\mathbf{f}(\mathbf{x}) = \mathbf{0}$ may appear misleadingly simple, so let's make things concrete by looking at a two-dimensional problem:

[32] Note that we didn't have to evaluate a determinant anywhere.

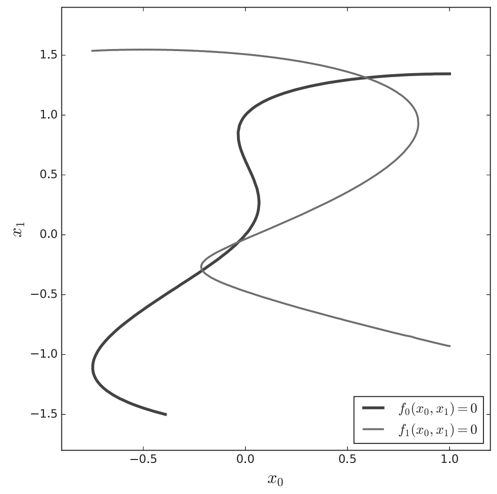


Fig. 5.12 Example of two coupled nonlinear equations

$$\begin{cases} f_0(x_0, x_1) = x_0^2 - 2x_0 + x_1^4 - 2x_1^2 + x_1 = 0 \\ f_1(x_0, x_1) = x_0^2 + x_0 + 2x_1^3 - 2x_1^2 - 1.5x_1 - 0.05 = 0 \end{cases} \tag{5.72}$$

We have two unknowns, x_0 and x_1, and two equations; the latter are given in the form $f_i = 0$, where each of the f_i's is a function of the two variables x_0 and x_1. Note that here we are truly faced with two variables, x_0 and x_1, in contradistinction to the previous section, where x_0, x_1, and so on were used to denote distinct roots of a single polynomial.[33] You should convince yourself that you cannot fully solve this pair of coupled nonlinear equations by hand; in this simple case, it's possible to solve, say, for x_0 in terms of x_1, and then get a single nonlinear equation, but the question arises how you would be able to do the same for the case of a 10-dimensional system. To get some insight into the problem, Fig. 5.12 shows the curves described by our two equations. Note that even the simple task of producing this plot is not totally trivial. We find three intersection points between the two curves.[34]

5.4.1 Newton's Method

The derivation of Newton's multidimensional method will basically be a straightforward generalization of what we saw in section 5.2.5, with a Taylor expansion at its core. We assume that $\mathbf{f}$ has bounded first and second derivatives; the actual solution of our problem is $\mathbf{x}^*$ and we will be trying to approximate it using iterates, which this time are themselves vectors, $\mathbf{x}^{(k)}$.[35] In order to make the transition to the general problem as simple as possible, let's start from a Taylor expansion of a single function component f_i around our latest

[33] Also in contradistinction to the bracketing endpoints, x_0 and x_1, of earlier sections.
[34] Actually, there is also a fourth one, if you look farther to the right.
[35] Just like in our disussion of the Jacobi iterative method for solving linear systems of equations in section 4.3.5.

iterate, $\mathbf{x}^{(k-1)}$:

$$f_i(\mathbf{x}) = f_i(\mathbf{x}^{(k-1)}) + \left(\nabla f_i(\mathbf{x}^{(k-1)})\right)^T \left(\mathbf{x} - \mathbf{x}^{(k-1)}\right) + O\left(\|\mathbf{x} - \mathbf{x}^{(k-1)}\|^2\right) \tag{5.73}$$

where, as usual, $i = 0, 1, \dots, n-1$. We can rewrite the second term on the right-hand side in terms of vector components:

$$\left(\nabla f_i(\mathbf{x}^{(k-1)})\right)^T \left(\mathbf{x} - \mathbf{x}^{(k-1)}\right) = \sum_{j=0}^{n-1} \left.\frac{\partial f_i}{\partial x_j}\right|_{x_j^{(k-1)}} \left(x_j - x_j^{(k-1)}\right) \tag{5.74}$$

With a view to collecting the n function components together, we now introduce the *Jacobian matrix*:

$$\mathbf{J}(\mathbf{x}) = \left\{\frac{\partial f_i}{\partial x_j}\right\} = \begin{pmatrix} \frac{\partial f_0}{\partial x_0} & \frac{\partial f_0}{\partial x_1} & \dots & \frac{\partial f_0}{\partial x_{n-1}} \\ \frac{\partial f_1}{\partial x_0} & \frac{\partial f_1}{\partial x_1} & \dots & \frac{\partial f_1}{\partial x_{n-1}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_{n-1}}{\partial x_0} & \frac{\partial f_{n-1}}{\partial x_1} & \dots & \frac{\partial f_{n-1}}{\partial x_{n-1}} \end{pmatrix} \tag{5.75}$$

You may sometimes see this denoted by $\mathbf{J}_f(\mathbf{x})$, in order to keep track of which function it's referring to. Using this matrix, we can now rewrite Eq. (5.73) to group together all n function components:

$$\mathbf{f}(\mathbf{x}) = \mathbf{f}(\mathbf{x}^{(k-1)}) + \mathbf{J}(\mathbf{x}^{(k-1)})\left(\mathbf{x} - \mathbf{x}^{(k-1)}\right) + O\left(\|\mathbf{x} - \mathbf{x}^{(k-1)}\|^2\right) \tag{5.76}$$

Keep in mind that this is nothing more than a generalization of the Taylor expansion in Eq. (5.37). In the spirit of that derivation, we now drop the second-order term and assume that we have found the solution, $\mathbf{f}(\mathbf{x}^*) = \mathbf{0}$:

$$\mathbf{0} = \mathbf{f}(\mathbf{x}^{(k-1)}) + \mathbf{J}(\mathbf{x}^{(k-1)})\left(\mathbf{x}^* - \mathbf{x}^{(k-1)}\right) \tag{5.77}$$

In practice, one iteration will not be enough to find the solution so, instead, we use our latest formula to introduce the *prescription* of Newton's method for the next iterate, $\mathbf{x}^{(k)}$:

$$\mathbf{J}(\mathbf{x}^{(k-1)})\left(\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\right) = -\mathbf{f}(\mathbf{x}^{(k-1)}) \tag{5.78}$$

Since all quantities at the location of our previous iterate, $\mathbf{x}^{(k-1)}$, are known, this equation has the form of $\mathbf{Ax} = \mathbf{b}$, i.e., it is a *linear* system of n equations in n unknowns. Assuming $\mathbf{J}(\mathbf{x}^{(k-1)})$ is non-singular, we can solve this system and then we will be able to find all the $x_j^{(k)}$. This process is repeated, until we satisfy a termination criterion, which could be taken to be that in Eq. (4.170):

$$\sum_{j=0}^{n-1} \left|\frac{x_j^{(k)} - x_j^{(k-1)}}{x_j^{(k)}}\right| \le \epsilon \tag{5.79}$$

or something fancier.

In principle, you could further manipulate Eq. (5.78) so as to write it in the form:

$$\mathbf{x}^{(k)} = \mathbf{x}^{(k-1)} - \left(\mathbf{J}(\mathbf{x}^{(k-1)})\right)^{-1} \mathbf{f}(\mathbf{x}^{(k-1)}) \tag{5.80}$$

which looks very similar to Eq. (5.40) for the one-variable Newton's method. In practice, as you know from chapter 4, explicitly evaluating the inverse of a matrix is usually a bad idea. Instead, you simply solve the linear system Eq. (5.78), say by Gaussian elimination or the Jacobi iterative method (if the Jacobian matrix is sparse). A straightforward generalization of the convergence study in section 5.2.5 will convince you that, even in many dimensions, Newton's method is *quadratically convergent*, assuming your initial guess (vector) $\mathbf{x}^{(0)}$ is close enough to the true solution $\mathbf{x}^*$.

5.4.2 Discretized Newton Method

Note that, in order to solve the linear system in Eq. (5.78), you need to have access to the Jacobian matrix, $\mathbf{J}(\mathbf{x})$. This is completely analogous to the one-variable version of Newton's method, where you needed to be able to evaluate $f'(x)$. For our two-dimensional problem in Eq. (5.72) this is very easy to do: you simply take the derivatives analytically. In general, however, this may not be possible, e.g., if your function components f_i are external complicated subroutines.

Algorithm

You may recall that in the case of the one-variable Newton's method we had advised against using a finite-difference approach to evaluate the first derivative $f'(x)$. This was both because we didn't know which spacing h to pick, and also because it implied two function evaluations at each iteration. In the present, multidimensional, case we are willing to deal with these problems, given the paucity of alternatives.

This gives rise to the *discretized Newton method*; this is simply a result of using the forward difference to approximate the derivatives that make up the Jacobian matrix:

$$\frac{\partial f_i}{\partial x_j} \approx \frac{f_i(\mathbf{x} + \mathbf{e}_j h) - f_i(\mathbf{x})}{h} = \frac{f_i(x_0, x_1, \dots, x_j + h, \dots, x_{n-1}) - f_i(x_0, x_1, \dots, x_j, \dots, x_{n-1})}{h} \tag{5.81}$$

As shown explicitly in the second equality, $\mathbf{e}_j$ is the j-th column of the identity matrix, as we had already seen back in Eq. (4.148). The algorithm of the discretized Newton method then consists of employing Eq. (5.78), where the Jacobian is approximated as per Eq. (5.81).

Since each of i and j takes up n possible values, you can see that in order to evaluate all the $f_i(\mathbf{x} + \mathbf{e}_j h)$ we need n^2 function-component evaluations. To that you must add another n function-component evaluations that lead to $f_i(\mathbf{x})$; you actually need those for the right-hand side of Eq. (5.78), so you compute them before you start forming the forward differences of Eq. (5.81). In addition to this cost, each iteration of the discretized Newton method requires the linear system of Eq. (5.78) to be solved, for which as you may recall the operation count is $\sim 2n^3/3$ for Gaussian elimination.[36]

[36] Which is the method you'd likely employ for a dense matrix.

Implementation

At this point, it should be relatively straightforward to implement the discretized Newton method in Python; the result is Code 5.5. We start out by importing the Gaussian-elimination-with-pivoting function from the last chapter: since we have no guarantees about the structure of our matrices, we try to be as general as possible. We then introduce three new functions. Taking these in turn: first, we create a function that evaluates the n function components f_i given a position argument $\mathbf{x}$. We do this for our two-variable problem in Eq. (5.72), but the rest of the code is set up to be completely general. In other words, the dimensionality of the problem is located *only* in one part of the code, i.e., in the function `fs()`. Since the rest of the code will be manipulating matrices, we make sure to produce a one-dimensional `numpy` array to hold the values of $\mathbf{f}(\mathbf{x})$. Note that in `fs()` we start by unpacking the input argument `xs` into two local variables; this is not necessary, since we could have simply used `xs[0]` and `xs[1]` below. However, all those square brackets are error prone so we avoid them.

Our next function, `jacobian()`, is designed to build up the Jacobian matrix `Jf` via the forward difference, as per Eq. (5.81). Note that we are being careful not to waste function evaluations: `fs0` is calculated once, outside the loop. After this, we are reasonably idiomatic, producing one column of the Jacobian matrix, Eq. (5.75), at a time. To get all the columns, we are iterating through the j index in $f_i(\mathbf{x} + \mathbf{e}_j h)$, while using `numpy` functionality to step through the i's, without the need for a second explicit index. It's worth pausing to compare this to the way we evaluated the second derivative all the way back in section 3.5. At the time, we didn't have access to `numpy` arrays, so we would shift a given position by h, in order to avoid rounding error creep. Here, we have the luxury of using a new vector for each shifted component separately.

The function `multi_newton()` looks a lot like our earlier `legnewton()` but, of course, it applies to any multidimensional function. As a result, it uses `numpy` arrays throughout. After evaluating the Jacobian matrix at the latest iterate as per Eq. (5.81), we solve the linear system Eq. (5.78) and thereby produce a new iterate. As usual, we test against the termination criterion to see if we should keep going. The print-out statement could have included the value(s) of `fs(xnews)`, to highlight that each function component f_i gets closer to zero as the algorithm progresses. Before exiting the loop, we prepare for the next iteration by renaming our latest iterate; we could have also accomplished this by saying `xolds = xnews`, but it's good to get into the habit of using `numpy.copy()`.

The main part of the program simply makes up an initial guess and runs our multidimensional Newton function. We then print out the final solution vector, as well as the values of the function components f_i at the solution vector. Running this code, we succeed, after a small number of iterations, in finding a solution which leads to tiny function components. You should try different initial guesses to produce the other roots.

5.4.3 Broyden's Method

As it now stands, the discretized Newton method we just discussed works quite well for problems that are not too large; we will make use of it in the projects at the end of this

Code 5.5 multi_newton.py

```
from gauelim_pivot import gauelim_pivot
from jacobi import termcrit
import numpy as np

def fs(xs):
    x0, x1 = xs
    f0 = x0**2 - 2*x0 + x1**4 - 2*x1**2 + x1
    f1 = x0**2 + x0 + 2*x1**3 - 2*x1**2 - 1.5*x1 - 0.05
    return np.array([f0,f1])

def jacobian(fs,xs,h=1.e-4):
    n = xs.size
    iden = np.identity(n)
    Jf = np.zeros((n,n))
    fs0 = fs(xs)
    for j in range(n):
        fs1 = fs(xs+iden[:,j]*h)
        Jf[:,j] = (fs1 - fs0)/h
    return Jf, fs0

def multi_newton(fs,jacobian,xolds,kmax=200,tol=1.e-8):
    for k in range(1,kmax):
        Jf, fs_xolds = jacobian(fs, xolds)
        xnews = xolds + gauelim_pivot(Jf, -fs_xolds)

        err = termcrit(xolds, xnews)
        print(k, xnews, err)
        if err < tol:
            break

        xolds = np.copy(xnews)
    else:
        xnews = None
    return xnews

if __name__ == '__main__':
    xolds = np.array([1.,1.])
    xnews = multi_newton(fs, jacobian, xolds)
    print(xnews); print(fs(xnews))
```

and following chapters. However, let's back up and see which parts of the algorithm are wasteful. As you may recall, this method entailed two large costs: first, it involved $n^2 + n$ function-component evaluations, n^2 for the Jacobian matrix as per Eq. (5.81) and n for the right-hand side of the Newton update, as per Eq. (5.78). Second, the Newton update itself involved solving a linear system of equations, which for Gaussian elimination costs $\sim 2n^3/3$ operations.

A reasonable assumption to make at this point is that, of these two costs, the function (component) evaluations are the most time-consuming. You can imagine problems where n is, say, 10, so solving a 10×10 linear system is pretty straightforward, but each function-component evaluation could be a lengthy separate calculation involving many other math-emetical operations; then, the n^2 evaluations required for the discretized Newton method will dominate the total runtime. To address this issue, we are inspired by the secant method for the one-variable case: what we did there was to use the function values at the two latest iterates in order to approximate the value of the derivative, see Eq. (5.50). The idea was that, since we're going to be evaluating the function value at each iterate, we might as well hijack it to also provide information on the derivative. As the iterates get closer to each other, this leads to a reasonably good approximation of the first derivative.

With the secant method as our motivation, we then come up with the idea of using the last two guess vectors, $\mathbf{x}^{(k)}$ and $\mathbf{x}^{(k-1)}$, and the corresponding function values, in order to approximate the Jacobian matrix. In equation form, this is:

$$\mathbf{J}(\mathbf{x}^{(k)})\left(\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\right) \approx \mathbf{f}(\mathbf{x}^{(k)}) - \mathbf{f}(\mathbf{x}^{(k-1)}) \tag{5.82}$$

This *secant equation* has to be obeyed but, unfortunately, it cannot be solved on its own, since it is *underdetermined*. To see this, think of our problem as $\mathbf{A}\mathbf{x} = \mathbf{b}$, where we know $\mathbf{x}$ and $\mathbf{b}$ but not $\mathbf{A}$. In order to make progress, we need to impose some further condition.

This problem was tackled by Broyden, who had the idea of not evaluating $\mathbf{J}(\mathbf{x}^{(k)})$ from scratch at each iteration. Instead, he decided to take a previous estimate of the Jacobian, $\mathbf{J}(\mathbf{x}^{(k-1)})$, and update it. The notation can get messy, so let's define some auxiliary variables:

$$\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)} \equiv \mathbf{q}^{(k)}, \qquad \mathbf{f}(\mathbf{x}^{(k)}) - \mathbf{f}(\mathbf{x}^{(k-1)}) \equiv \mathbf{y}^{(k)} \tag{5.83}$$

Our secant equation then becomes simply:

$$\mathbf{J}(\mathbf{x}^{(k)})\mathbf{q}^{(k)} \approx \mathbf{y}^{(k)} \tag{5.84}$$

Broyden chose to impose the further requirement that:

$$\mathbf{J}(\mathbf{x}^{(k)})\mathbf{p} = \mathbf{J}(\mathbf{x}^{(k-1)})\mathbf{p} \tag{5.85}$$

for any vector $\mathbf{p}$ that is orthogonal to $\mathbf{q}^{(k)}$, i.e., for which $\left(\mathbf{q}^{(k)}\right)^T \mathbf{p} = 0$ holds: if you change $\mathbf{x}$ in a direction perpendicular to $\mathbf{q}^{(k)}$ then you don't learn anything about the rate of change of $\mathbf{f}$. The last two equations are enough to uniquely determine the update from $\mathbf{J}(\mathbf{x}^{(k-1)})$ to $\mathbf{J}(\mathbf{x}^{(k)})$. However, instead of constructing this update explicitly, we will follow an easier route: we'll state Broyden's prescription, and then see that it satisfies both the required conditions, Eq. (5.84) and Eq. (5.85). Broyden's update is:

$$\mathbf{J}(\mathbf{x}^{(k)}) = \mathbf{J}(\mathbf{x}^{(k-1)}) + \frac{\mathbf{y}^{(k)} - \mathbf{J}(\mathbf{x}^{(k-1)})\mathbf{q}^{(k)}}{(\mathbf{q}^{(k)})^T \mathbf{q}^{(k)}} \left(\mathbf{q}^{(k)}\right)^T \tag{5.86}$$

(We state at the outset that we can get in trouble if $\mathbf{q}^{(k)} = \mathbf{x}^{(k)} - \mathbf{x}^{(k-1)} = 0$.) Let's check if our two requirements are satisfied.

First, we take Eq. (5.86) and multiply with $\mathbf{q}^{(k)}$ on the right. We have:

$$\begin{aligned}\mathbf{J}(\mathbf{x}^{(k)})\mathbf{q}^{(k)} &= \mathbf{J}(\mathbf{x}^{(k-1)})\mathbf{q}^{(k)} + \frac{\mathbf{y}^{(k)} - \mathbf{J}(\mathbf{x}^{(k-1)})\mathbf{q}^{(k)}}{(\mathbf{q}^{(k)})^T \mathbf{q}^{(k)}} \left(\mathbf{q}^{(k)}\right)^T \mathbf{q}^{(k)} \\ &= \mathbf{J}(\mathbf{x}^{(k-1)})\mathbf{q}^{(k)} + \mathbf{y}^{(k)} - \mathbf{J}(\mathbf{x}^{(k-1)})\mathbf{q}^{(k)} = \mathbf{y}^{(k)}\end{aligned} \tag{5.87}$$

In the second equality we cancelled the denominator and in the third equality we cancelled the other two terms. We find that our first requirement, Eq. (5.84), is satisfied.

Next, we take Eq. (5.86) again, this time multiplying with $\mathbf{p}$ on the right:

$$\mathbf{J}(\mathbf{x}^{(k)})\mathbf{p} = \mathbf{J}(\mathbf{x}^{(k-1)})\mathbf{p} + \frac{\mathbf{y}^{(k)} - \mathbf{J}(\mathbf{x}^{(k-1)})\mathbf{q}^{(k)}}{(\mathbf{q}^{(k)})^T \mathbf{q}^{(k)}} \left(\mathbf{q}^{(k)}\right)^T \mathbf{p} \tag{5.88}$$

Since we know that $\left(\mathbf{q}^{(k)}\right)^T \mathbf{p} = 0$ holds, we see that the second term on the right-hand side vanishes, so we are left with Eq. (5.85), as desired.

Thus, Broyden's update in Eq. (5.86) satisfies our two requirements in Eq. (5.84) and Eq. (5.85). In other words, we have been able to produce the next Jacobian matrix using only the previous Jacobian matrix and the last two iterates (and corresponding function values). Let us summarize the entire prescription using our original notation, which employs fewer extra variables:

$$\begin{aligned}\mathbf{J}(\mathbf{x}^{(k-1)})\left(\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\right) &= -\mathbf{f}(\mathbf{x}^{(k-1)}) \\ \mathbf{J}(\mathbf{x}^{(k)}) &= \mathbf{J}(\mathbf{x}^{(k-1)}) + \frac{\mathbf{f}(\mathbf{x}^{(k)})\left(\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\right)^T}{\|\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\|_E^2}\end{aligned} \tag{5.89}$$

We start with a known $\mathbf{J}(\mathbf{x}^{(0)})$; this could be produced using a forward-difference scheme (just once at the start), or could even be taken to be the identity matrix. From then and onwards, we simply apply Newton's step from Eq. (5.78), in the first line, and then use Broyden's formula, Eq. (5.86), in the second line to update the Jacobian matrix; we also took the opportunity to use the first line to cancel some terms in the numerator and to identify the Euclidean norm in the denominator. Remember, even though Broyden's update was designed to respect the secant equation, Eq. (5.82), this equation does not appear in the prescription itself. Observe that the first line requires n function-component evaluations for $\mathbf{f}(\mathbf{x}^{(k-1)})$ and the next line another n function-component evaluations, for $\mathbf{f}(\mathbf{x}^{(k)})$, but the latter will be re-used in the guise of $\mathbf{f}(\mathbf{x}^{(k-1)})$ the next time through the loop. Note, finally, that there is no derivative in sight. Keep in mind that we won't exhibit quadratic convergence (like Newton's method for analytical derivatives does) but superlinear convergence, similarly to what we saw in the one-variable case.

At the end of the day, our prescription above still requires us to solve a linear system of equations in the first line. Broyden actually did not carry out our cancellation in the numerator; instead, he went on to make the prescription even more efficient, but the main concept should be clear from our discussion. The crucial point is that we are avoiding the n^2 function-component evaluations; this approach is often much faster than the discretized Newton method we covered in the previous subsection. A problem asks you to implement this version of Broyden's method in Python; you should keep in mind that the numerator in the second line of Eq. (5.89) involves the product of a column vector and a row vector, so it is similar to the outer product in producing an $n \times n$ matrix.[37]

5.5 Minimization

We turn to the last problem of this chapter: instead of finding function zeros, we are now going to be locating function minima. Note that we will be studying the problem of *unconstrained minimization*, meaning that we will not be imposing any further constraints on our variables; we are simply looking for the variable values that minimize a scalar function.

5.5.1 One-Dimensional Minimization

For simplicity, let's start from the case of a function of a single variable, $\phi(x)$.[38] As you may recall from elementary calculus, a *stationary point* (which, for a differentiable function, is also known as a *critical point*) is a point at which the derivative vanishes, namely:

$$\phi'(x^*) = 0 \tag{5.90}$$

where we are now using x^* to denote the stationary point. If $\phi''(x^*) > 0$ we are dealing with a *local minimum*, whereas if $\phi''(x^*) < 0$ a *local maximum*. Minima and maxima together are known as *extrema*.

A simple example is our function $\phi(x) = e^{x-\sqrt{x}} - x$ from Fig. 5.1; in an earlier section we saw that it has two zeros, at ≈ 1 and at ≈ 2.5, but we are now interested in the minimum, which is located at $x^* \approx 1.8$. It's easy to see that $\phi''(x^*) > 0$ (simply by looking) so we have a (single) minimum. From a practical perspective, to compute the location of the minimum we can use one of our five one-variable root-finding algorithms from section 5.2, this time applied not to $\phi(x)$ but to $\phi'(x)$. By simply applying root-finding algorithms to the $\phi'(x)$ function, you will find a stationary point, but you won't know if it is a minimum,

[37] So you should *not* use `numpy`'s `@` here, since it would lead to a scalar; use `numpy.outer()` instead.
[38] You will soon see why we are switching our notation from $f(x)$ to $\phi(x)$.

maximum, or something else (see below). As usual, more specialized (and "safe") methods exist that locate only function minima, but here we are trying to give you only a flavor for the subject. A problem asks you to apply the secant method to this problem.

As a second example, we look at Fig. 5.8, which was plotting the function:

$$\phi(x) = x^4 - 9x^3 + 25x^2 - 24x + 4 \tag{5.91}$$

As you may recall, that function had one root at ≈ 0.21, another one at ≈ 4.79, as well as a double root at 2. Of course, we are now not trying to evaluate roots but minima or maxima. Just by looking at the plot we can see that we have a local minimum at $x^* \approx 0.7$, a local maximum at $x^* = 2$ (which coincides with the location of the double root), as well as another minimum at $x^* \approx 4$. The second derivative of our function at each of these extrema, $\phi''(x^*)$, is positive, negative, and positive, respectively. This is a nice opportunity to point out that the minimum at $x^* \approx 4$ happens to be the *global minimum*, i.e., the point where the function reaches its smallest value not only locally but in general. It should be easy to see that our iterative root-finding algorithms, which start with an initial guess $x^{(0)}$ and proceed from there cannot guarantee that a global minimum (or, for other functions, a global maximum) will be reached. This applies even if you have somehow taken into account the value of the second derivative, i.e., even if you can guarantee that you are dealing with a local minimum, you cannot guarantee that you have found the global minimum. You can think of placing a marble (point) on this curve: it will roll to the minimum point that is accessible to it, but may not roll to the absolute minimum. We won't worry too much about this, but there are techniques that can move you out of a local minimum, in search of nearby (deeper) local minima.

At this point we realize that, when classifying critical points above, we forgot about one possibility, namely that $\phi''(x^*) = 0$. In this case, one must study the behavior of higher-order derivatives (this is called the *extremum test*):[39] the point may turn out to be a minimum, a maximum, or what is known as a *saddle point*.[40] In order to elucidate that last concept, let us look at a third example, namely the function:

$$\phi(x) = 4x^5 - 15x^4 + 14x^3 + 4 \tag{5.92}$$

In Fig. 5.13 we are plotting the function $\phi(x)$ itself in the left panel, the first derivative $\phi'(x)$ in the middle panel, and the second derivative $\phi''(x)$ in the right panel. We have a local minimum at $x^* \approx 1.89$, a local maximum at $x^* \approx 1.11$, and something new at $x^* = 0$: the function "flattens" but then keeps going in the same direction. Looking at the first derivative in the middle panel, we see that all three points are critical points (i.e., have vanishing first derivative). Then, turning to the second derivative in the right panel, examining these three points starting from the rightmost, $\phi''(x^*)$ is positive, negative, and zero, respectively. You can see from the way $\phi''(x)$ crosses the x axis at $x = 0$ that we will find $\phi'''(0) \neq 0$, which is what leads to a saddle point.

To summarize, locating critical points using a one-dimensional root-finding algorithm is reasonably straightforward. After this, one has to be a little careful in distinguishing

[39] This is actually more general, also covering the case where the first several derivatives are zero.

[40] There are situations where you can have an *inflection point* that is not a critical point (so it doesn't get called a saddle point). For example: $\phi(x) = \sin x$ leads to $\phi''(0) = 0$, but for $\phi'(0) \neq 0$.

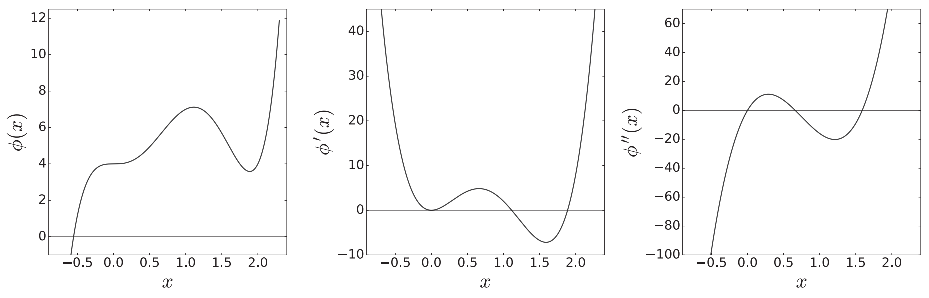


Fig. 5.13 Example illustrating the concept of a saddle point

between (local and global) minima, maxima, and saddle points; this is easier to do when function-derivative information is available.

5.5.2 Multidimensional Minimization

The problem of multidimensional minimization is, in general, much harder to solve; as the dimensionality grows one cannot even visualize what's going on very effectively. We start with some mathematical aspects of the problem, then turn to a two-dimensional example, and after that discuss specific minimization methods.

General Features

Consider a scalar function of many variables, i.e., $\phi(\mathbf{x})$, where $\mathbf{x}$ bundles together the variables $x_0, x_1, \ldots, x_{n-1}$ but ϕ produces scalar values. This is somewhat analogous to the single function components f_i that we encountered in section 5.4. As we did in Eq. (5.73), we will now employ a multidimensional Taylor expansion, this time going to one order higher. Also, in order to keep things general, we will not expand around our latest iterate, $\mathbf{x}^{(k-1)}$, since we are not introducing a specific method right now; we are simply trying to explore features of the problem of minimizing $\phi(\mathbf{x})$.[41]

We assume $\phi(\mathbf{x})$ has bounded first, second, and third derivatives. Then, employing notation inspired by Eq. (5.83), we Taylor expand in each of the components of the vector $\mathbf{x}$; these can be bundled together using vector notation, see Eq. (C.5):

$$\phi(\mathbf{x}+\mathbf{q}) = \phi(\mathbf{x}) + (\nabla\phi(\mathbf{x}))^T\,\mathbf{q} + \frac{1}{2}\mathbf{q}^T\mathbf{H}(\mathbf{x})\mathbf{q} + O\left(\|\mathbf{q}\|^3\right) \tag{5.93}$$

Here the first-order term involves $\nabla\phi(\mathbf{x})$, the *gradient* vector of ϕ at $\mathbf{x}$. This is:

$$\nabla\phi(\mathbf{x}) = \left(\frac{\partial\phi}{\partial x_0}\;\frac{\partial\phi}{\partial x_1}\;\cdots\;\frac{\partial\phi}{\partial x_{n-1}}\right)^T \tag{5.94}$$

Similarly to Eq. (5.74), we can express the first-order term in our expansion in terms of the

[41] Whatever we discover on the question of minimizing $\phi(\mathbf{x})$ will also apply to the problem of maximizing $-\phi(\mathbf{x})$.

vector components:

$$(\nabla\phi(\mathbf{x}))^T\,\mathbf{q} = \sum_{j=0}^{n-1} \frac{\partial\phi}{\partial x_j} q_j \tag{5.95}$$

We will come back to this point below but, for now, note that $\nabla\phi(\mathbf{x})$ is the direction of *steepest ascent*. To see this, observe that, for small $\mathbf{q}$, the term linear in $\mathbf{q}$ is the dominant contribution, since $||\mathbf{q}||^2 \ll ||\mathbf{q}||$. From elementary vector calculus we know that the dot product $(\nabla\phi(\mathbf{x}))^T\,\mathbf{q}$ will be maximized when $\mathbf{q}$ points in the direction of $\nabla\phi(\mathbf{x})$.

Assuming $\mathbf{x}^*$ is a local minimum of ϕ and ignoring higher-order terms in Eq. (5.93):

$$\phi(\mathbf{x}^* + \mathbf{q}) \approx \phi(\mathbf{x}^*) + (\nabla\phi(\mathbf{x}^*))^T\,\mathbf{q} \tag{5.96}$$

Reversing our argument from the previous paragraph, the first-order term will lead to the largest possible decrease when $\mathbf{q}$ points in the direction of $-\nabla\phi(\mathbf{x}^*)$. If $\nabla\phi(\mathbf{x}^*) \neq \mathbf{0}$, we will have found a $\phi(\mathbf{x}^* + \mathbf{q})$ that is smaller than $\phi(\mathbf{x}^*)$; by definition, this is impossible, since we said that $\mathbf{x}^*$ is a local minimum of ϕ. This leads us to the following multidimensional generalization of our criterion for being a critical point in Eq. (5.90):

$$\nabla\phi(\mathbf{x}^*) = \mathbf{0} \tag{5.97}$$

where both the left-hand side and the right-hand side are vectors.

Having established that the gradient vector vanishes at a critical point, we now turn to the second-order term in Eq. (5.93), which involves the *Hessian matrix*, $\mathbf{H}(\mathbf{x})$. To see what this is, we expand the quadratic form as follows:

$$\frac{1}{2}\mathbf{q}^T\mathbf{H}(\mathbf{x})\mathbf{q} = \frac{1}{2}\sum_{i,j=0}^{n-1} \frac{\partial\phi}{\partial x_i \partial x_j} q_i q_j \tag{5.98}$$

In matrix form:

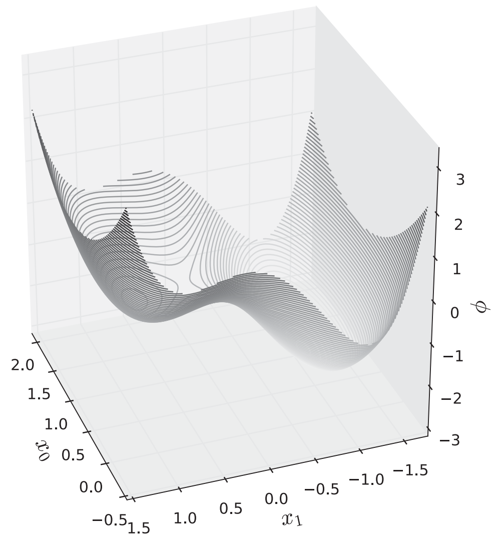
$$\mathbf{H}(\mathbf{x}) = \left\{\frac{\partial\phi}{\partial x_i \partial x_j}\right\} = \begin{pmatrix} \frac{\partial^2\phi}{\partial x_0^2} & \frac{\partial^2\phi}{\partial x_0 \partial x_1} & \cdots & \frac{\partial^2\phi}{\partial x_0 x_{n-1}} \\ \frac{\partial^2\phi}{\partial x_1 \partial x_0} & \frac{\partial^2\phi}{\partial x_1^2} & \cdots & \frac{\partial^2\phi}{\partial x_1 \partial x_{n-1}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2\phi}{\partial x_{n-1} \partial x_0} & \frac{\partial^2\phi}{\partial x_{n-1} \partial x_1} & \cdots & \frac{\partial^2\phi}{\partial x_{n-1}^2} \end{pmatrix} \tag{5.99}$$

Since for us the second partial derivatives will always be continuous, it is easy to see that our Hessian matrix will be *symmetric*. Let us now apply Eq. (5.93), for the case of $\mathbf{x}^*$, a local minimum of ϕ. We have:

$$\phi(\mathbf{x}^* + \mathbf{q}) = \phi(\mathbf{x}^*) + \frac{1}{2}\mathbf{q}^T\mathbf{H}(\mathbf{x}^*)\mathbf{q} + O\left(||\mathbf{q}||^3\right) \tag{5.100}$$

where we have not included the first-order term, since the gradient vector vanishes, as per Eq. (5.97). If we now further assume that $\mathbf{H}(\mathbf{x}^*)$ is *positive definite*,[42] then we can see that, indeed, $\phi(\mathbf{x}^* + \mathbf{q}) > \phi(\mathbf{x}^*)$, as it should, since $\mathbf{x}^*$ is a minimum.

[42] Recall from section 4.3.4 that a matrix $\mathbf{A}$ is positive definite if $\mathbf{x}^T\mathbf{A}\mathbf{x} > 0$ for all $\mathbf{x} \neq \mathbf{0}$.



Example of a scalar function of two variables

Fig. 5.14

To summarize: (a) a necessary condition for $\mathbf{x}^*$ being a local minimum is that it be a critical point, i.e., that its gradient vector vanish, and (b) a sufficient condition for the critical point $\mathbf{x}^*$ being a local minimum is that its Hessian matrix be positive definite.

A Two-Dimensional Example

As in the case of solving nonlinear coupled equations, it's easier to build your intuition by studying a two-variable problem. This means we will be dealing with a (single) scalar function ϕ that takes in two variables, x_0 and x_1, and produces a single number, the function value. As above, our goal is to minimize this function, i.e., to find the values of x_0 and x_1 that produce the smallest possible $\phi(x_0, x_1)$. We decide to look at the f_0 from our earlier example, Eq. (5.72), this time being considered not as a function component but as a single function; also, as noted, we are no longer looking at the zeros of this function, but its entire behavior, with a view to minimizing it. It is:

$$\phi(x_0, x_1) = x_0^2 - 2x_0 + x_1^4 - 2x_1^2 + x_1 \tag{5.101}$$

To help you get more comfortable with higher dimensions, Fig. 5.14 shows what this function looks like. This is attempting both to visualize the third dimension and to draw equipotential curves (also known as contour lines). For example, if you take $\phi = 0$ then you will reproduce our earlier curve from Fig. 5.12.

We find that we are dealing with two local minima. The one on the "right" leads to smaller/more negative function values, so it appears to be the global minimum. As a reminder of some of our earlier points: if you place a marble somewhere near these two wells, it will roll down to one of the minima; which of the two minima you end up in depends on where you start. Note, finally, that our discussion above on saddle points in one dimension needs to be generalized: you may have a saddle point when you are at a maximum along, say, the x_0 direction and at a minimum along the x_1 direction. Incidentally, this also explains why the word "saddle" is used in this context.

5.5.3 Gradient Descent

We now turn to a simple and intuitively clear approach to multidimensional minimization. This method, known as *gradient descent*, does not exhibit great convergence properties and can get in trouble for non-differentiable functions. Even so, it is a pedagogical, straightforward approach.

Algorithm and Interpretation

Recall from our discussion of general features that $\nabla\phi(\mathbf{x})$ is the direction of steepest ascent. This leads to the conclusion that $-\nabla\phi(\mathbf{x})$ is the direction of *steepest descent*: as in our discussion around Eq. (5.96), we know that choosing $\mathbf{q}$ to point along the negative gradient guarantees that the function value decrease will be the fastest. The method we are about to introduce, which employs $-\nabla\phi(\mathbf{x})$, is known as *gradient descent*.[43] Qualitatively, this approach makes use of *local* information: if you're exploring a mountainous region (with your eyes closed), you can take a small step downhill *at that point*; this doesn't mean that you're always actually moving in the direction that will most quickly bring you to a (possibly distant) local minimum, simply that you are moving in a downward direction.

Implicit in our discussion above is the fact that the steps we make will be *small*: while $-\nabla\phi(\mathbf{x})$ helps you pick the direction, it doesn't tell you how far in that direction you should go, i.e., how large a $||\mathbf{q}||$ you should employ. The simplest possible choice, analogously to our closed-eyes example, is to make small fixed steps, quantified by a parameter γ. Using notation similar to that in Eq. (5.80), this leads to the following prescription:

$$\mathbf{x}^{(k)} = \mathbf{x}^{(k-1)} - \gamma\nabla\phi(\mathbf{x}^{(k-1)}) \tag{5.102}$$

Note that the right-hand side involves an evaluation of the gradient, not of the function itself. At each step, this method picks the direction that is perpendicular to the contour line. Note also that there is no (costly) Jacobian computation being carried out here; similarly, there is no matrix inversion or linear system solution: all that's being done is that the gradient is computed and a (small) step is taken along that direction. Of course, the question arises what "small" means, i.e., of how to pick γ. In what follows, we will take an empirical, i.e., trial-and-error, approach. In one of the problems, you will explore a more systematic solution, where this magnitude parameter is allowed to change at each iteration, i.e., you

[43] Since it employs the direction of steepest descent, this method is also known by that name, though it should not be confused with the "method of steepest descent", which arises in the study of contour integration.

will be dealing with $\gamma^{(k)}$ and will employ an extra criterion to help you pick this parameter at each step.

In practice, we may not know the gradient analytically. In complete analogy to Eq. (5.81), we typically approximate it using a forward-difference scheme. In equation form, this means that Eq. (5.94) becomes:

$$\nabla\phi(\mathbf{x}) = \begin{pmatrix} [\phi(\mathbf{x} + \mathbf{e}_0 h) - \phi(\mathbf{x})]/h \\ [\phi(\mathbf{x} + \mathbf{e}_1 h) - \phi(\mathbf{x})]/h \\ \dots \\ [\phi(\mathbf{x} + \mathbf{e}_{n-1} h) - \phi(\mathbf{x})]/h \end{pmatrix} \tag{5.103}$$

for a given spacing h. This involves "only" $n+1$ function evaluations, so it is not too costly (it doesn't involve any really demanding linear-algebra steps).

Implementation

Our prescription in Eq. (5.102) is quite simple, so you will not be surprised to hear that it is straightforward to implement. Code 5.6 is along the lines you'd expect. We first define our scalar function $\phi(\mathbf{x})$; again, notice that this is the only part of the code where the problem is shown to be two-dimensional, implementing Eq. (5.101). In other words, changing `phi()` would be all you would need to do to solve a 10-dimensional problem.

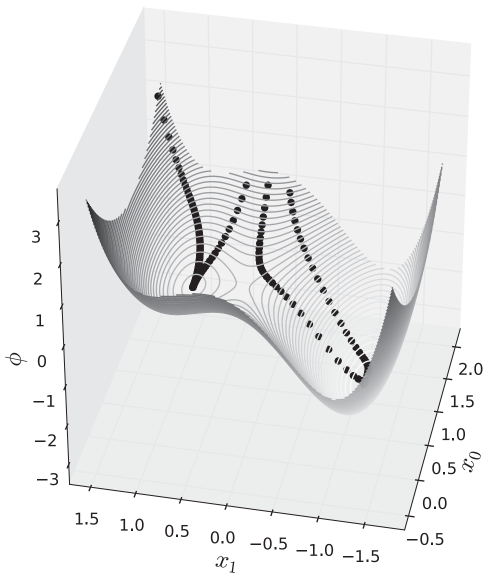
We then introduce another function, `gradient()`, which computes the gradient using a forward-difference approximation, as per Eq. (5.103). It's important to keep in mind that ϕ is a scalar, i.e., `phi()` returns a single number; as a result, $\nabla\phi$ is a column vector, so `gradient()` returns a `numpy` array. We could have employed here an explicit loop, as you are asked to do in one of the problems. Let's look at `Xph` a bit more closely: in essence, what we are doing is producing a matrix made up of n copies of the position $\mathbf{x}$ (or $\mathbf{x}^{(k-1)}$) and then adding in h to each of the position-components separately. The transposition is taking place because we intend to call `phi()` and wish to have each shifted position vector, $\mathbf{x}+\mathbf{e}_j h$, in its own *column*; then, each of `x0` and `x1` will end up being a `numpy` array instead of an individual number; this allows us to carry out all the needed calculations at once.

The function `descent()` takes in the ϕ to be minimized, as well as another function parameter, which will allow you to call a different gradient-computing prescription in the future, if you so desire. We then pass a γ with a reasonably small default value. Note that this is something you will need to tune by hand for each problem and possibly also for each initial-guess vector. The function body itself is quite similar to our earlier iterative codes. As a matter of fact, it's much less costly, since the main updating of Eq. (5.102) is so straightforward. At each iteration we are printing out the value of our latest iterate, $\mathbf{x}^{(k)}$, a measure of the change in the iterates, as well as the function value at our latest iterate, $\phi(\mathbf{x}^{(k)})$. Once again, note that we are *not* looking for function zeros, but for local minima. You can compare the final function value you find using different initial guesses, to see if the minimum you arrived at is lower than what you produced in previous runs.

Code 5.6 descent.py

```
from jacobi import termcrit
import numpy as np

def phi(xs):
     x0, x1 = xs
     return x0**2 - 2*x0 + x1**4 - 2*x1**2 + x1

def gradient(phi,xs,h=1.e-6):
     n = xs.size
     phi0 = phi(xs)
     Xph = (xs*np.ones((n,n))).T + np.identity(n)*h
     grad = (phi(Xph) - phi0)/h
     return grad

def descent(phi,gradient,xolds,gamma=0.15,kmax=200,tol=1.e-8):
     for k in range(1,kmax):
          xnews = xolds - gamma*gradient(phi,xolds)

          err = termcrit(xolds,xnews)
          print(k, xnews, err, phi(xnews))
          if err < tol:
               break

          xolds = np.copy(xnews)
     else:
          xnews = None
     return xnews

if __name__ == '__main__':
     xolds = np.array([2.,0.25])
     xnews = descent(phi, gradient, xolds)
     print(xnews)
```

The main part of the program simply makes an initial guess and runs our gradient descent function. We then print out the final solution vector. Running this code, we succeed in finding the global minimum, but we need considerably more than a handful of iterations to get there. Of course, the number of iterations needed depends on the initial-guess vector



Example of gradient descent applied to a scalar function of two variables

Fig. 5.15

and on the value of γ. In order to help you understand how the gradient-descent method reaches the minimum, in Fig. 5.15 we are visualizing its progress.[44] Shown are four distinct “trajectories”, i.e., collections of iterates for four distinct initial guesses. As you can see from these trajectories, each time the step taken is perpendicular to the contour line. Sometimes that means you will go to the minimum pretty straightforwardly, whereas other times you will bend away from your earlier trajectory, always depending on what the contour line looks like locally. Notice also that you may end up in the global minimum or not, depending on where you started.

5.5.4 Newton’s Method

Gradient descent is nice and simple but, as we will see in the problem set, either your γ is very small and you waste iterations or you are carrying out a line-search at each step to determine the optimal $\gamma^{(k)}$, which is starting to get costly.

A distinct approach goes as follows: instead of using only the value of the gradient at a given point, perhaps we should be building in more information. As you may recall, that is roughly what we were doing in an earlier section when employing the multidimensional Newton’s method. Of course, there we were in the business of (multiple-equation) root-finding, whereas now we are trying to find (single) function minima. However, it turns out that these two problems can be mapped onto each other. Specifically, Eq. (5.97) told us that the gradient vanishes at a critical point. Combine that with the fact that the gradient of a scalar function is a column vector, and you can recast that equation as a set of n coupled

[44] We have changed the viewing angle, as `matplotlib` conveniently allows, but it’s the same problem.

nonlinear equations:

$$\mathbf{f}(\mathbf{x}) = \nabla\phi(\mathbf{x}) = \mathbf{0} \tag{5.104}$$

In other words, finding a critical point is simply a special case of solving a nonlinear system of equations. Thus, we can use approaches such as Newton's or Broyden's methods that we developed above. Importantly, by comparing the Jacobian matrix of Eq. (5.75), made up of first derivatives of function components, with the Hessian matrix of Eq. (5.99), made up of second derivatives of the scalar function, we realize that:

$$\mathbf{H}(\mathbf{x}) = \mathbf{J}_{\nabla\phi}(\mathbf{x}) \tag{5.105}$$

due to the fact that the Hessian is symmetric. Note that we have now seen the notation where the Jacobian gets a subscript pay off. In words, the Jacobian matrix of the gradient is the Hessian matrix of our scalar function. This enables us to apply Newton's method to the problem of scalar-function minimization and thereby produce the next iterate, $\mathbf{x}^{(k)}$. Thus, Eq. (5.78) now becomes:

$$\mathbf{J}_{\nabla\phi}(\mathbf{x}^{(k-1)})\left(\mathbf{x}^{(k)} - \mathbf{x}^{(k-1)}\right) = -\nabla\phi(\mathbf{x}^{(k-1)}) \tag{5.106}$$

Observe that this equation contains the gradient at the previous iterate, just like gradient descent in Eq. (5.102), but now also builds in further information, in the form of the Hessian/Jacobian on the left-hand side. Of course, this is accomplished at the cost of having to solve a linear system of equations in order to produce the next iterate, $\mathbf{x}^{(k)}$.

In a problem, you are asked to apply this method to our example scalar function of Eq. (5.101). The easiest way to do this is to analytically produce and code up the gradient. At that point, all the infrastructure we built in `multi_newton.py` above applies directly to our problem. Alternatively, you could envision evaluating both the gradient and the Hessian using finite-difference approximations. The qualitative point to realize is that we will have to carry out a larger number of function evaluations at each iteration, in addition to also having to solve a linear system each step of the way. On the other hand, Newton's method converges quadratically if you are close enough to the root/minimum, so the higher cost at each iteration is usually compensated by the smaller number of required iterations.

You might protest at this point: Newton's method as just described finds critical points, but how do you know that you reached a minimum? As it turns out, we (implicitly) addressed this question in our earlier discussion in section 5.5.2: once you've reached a critical point, you can test your Hessian matrix to see if it is *positive definite*;[45] as you may recall, since the Hessian is symmetric, this simply means that if you find all the eigenvalues of the Hessian to be *positive*, then you'll know you've found a minimum. Similarly, if they're all negative you are at a maximum. In the problem set you will also explore the case of a multidimensional saddle point. Of course, there's another question that we haven't addressed: can you guarantee that you have found the global minimum? Generally, the answer is no. Many refinements of Newton's method exist that attempt to make it more

[45] Note, however, that the Hessian might *not* be positive definite away from the minimum.

safe, more efficient, as well as more globally oriented, but what we've covered should be enough for now.

Before closing this section, we realize that our linear-algebra machinery from chapter 4 has turned out to be pretty useful when dealing with the zeros of polynomials, with systems of nonlinear equations, and with multidimensional minimization. These are quite different applications, some of which used Gaussian elimination and others an eigenvalue solver. What unites them is the power of linear algebra, even when you're solving problems that themselves are nonlinear.

5.6 Project: Extremizing the Action in Classical Mechanics

As in previous chapters, we now turn to a physics application of the infrastructure we've developed in earlier sections. We will be studying a problem from classical mechanics, more specifically the dynamics of a single particle. Crucially, we will *not* be solving differential equations; that's the topic of chapter 8. Here, after some background material on Lagrangian mechanics, we will see how multidimensional minimization techniques, like those we just covered, can help you find the trajectory of a particle. This is a nice concretization of topics that students encounter in introductory courses; it's one thing to *say* that we minimize the action and quite another to see it happen.

5.6.1 Defining and Extremizing the Action

Working in a Cartesian system, let us study a single particle in one dimension. We can denote the particle's location by $x(t)$, where we're explicitly showing that the position is a function of time. For a more complicated problem, we would introduce generalized coordinates, but what we have here is enough for our purposes.

The kinetic energy of the particle will be a function of only $\dot{x}(t)$, i.e., of the time derivative of the position: $K = K(\dot{x}(t))$. Specifically, since we are dealing with a single particle, we know that:

$$K = \frac{1}{2}m\dot{x}^2 \tag{5.107}$$

where m is the mass of the particle. Similarly, in the absence of time-dependent external fields, the potential energy is a function of only $x(t)$: $V = V(x(t))$. The difference of these two quantities is defined as the *Lagrangian*:

$$L(x(t), \dot{x}(t)) \equiv K(\dot{x}(t)) - V(x(t)) \tag{5.108}$$

where, for our case, there is no explicit dependence of the Lagrangian on time.

We are interested in studying the particle from time $t = 0$ to time $t = T$. Then, one can define the *action functional* as the integral of the Lagrangian over time:

$$S[x(t)] \equiv \int_0^T dt\, L(x(t), \dot{x}(t)) = \int_0^T dt \left(\frac{1}{2}m\dot{x}^2 - V(x)\right) \tag{5.109}$$

where we applied Eq. (5.108) to our case. Notice that we called the action a *functional* and used square brackets on the left-hand side. Roughly speaking, a *functional* is a function of a function. A reminder: an ordinary function ϕ takes us from one number t to another number $\phi(t)$. A functional is an entity that takes in an entire function and gives back a number. In other words, a functional is a mapping from a space of functions into the real (or complex) numbers.

In case you haven't encountered functionals before, let us start with a simple case: a functional F of $\phi(t)$ (where t is a regular variable – this is a one-dimensional problem): $F[\phi(t)]$. Being a functional, F depends simultaneously on the values of ϕ at all points t but *it does not depend on t itself*: we provide it with the entire function and it provides us with one number as a result. A trivial example: $F[\phi] = \int_0^1 dt\phi(t)$ gives us one number for $\phi(t) = t$ and a different number for $\phi(t) = t^2$. In both cases, however, the answer is an ordinary number that does not depend on t.

Applied to our mechanics problem, we see that the action S depends on the position of the particle $x(t)$ at *all* times from 0 to T, but not on t directly, since t has been "integrated out". For a given trajectory $x(t)$ from $t = 0$ to $t = T$, the action produces a single number, S.[46] The question then arises: which trajectory $x(t)$ from $t = 0$ to time $t = T$ does the particle actually "choose"? The answer comes from *Hamilton's principle*: of all possible paths, the path that is actually followed is that which minimizes the action. As it so happens, the action only needs to be *stationary*, i.e., we are extremizing and not necessarily minimizing, but we will usually be dealing with a minimum. This extremization is taking place with the endpoints kept fixed, i.e., $x(0)$ and $x(T)$ are not free to vary.

5.6.2 Discretizing the Action

At this point our exposition will diverge from that of a standard mechanics textbook. This means that we will not proceed to write a possible trajectory as the physical trajectory plus a small perturbation. Instead, we will take the action from Eq. (5.109) and *discretize* it. This simply means that we will assume the positions $x(t)$ from $t = 0$ to $t = T$ can be accessed only at a discrete set of n points; applying our discussion of points on a grid from section 3.3.6, we have:

$$t_k = k\Delta t = k\frac{T}{n-1} \tag{5.110}$$

[46] As an aside, observe that, since the Lagrangian has units of energy, the action has units of energy times time. Intriguingly, these are also the units of Planck's constant, $\hbar$, but here we're studying classical mechanics.

where, as usual, $k = 0, 1, \dots, n-1$.[47] We will further employ the notation $x_k \equiv x(t_k)$ to denote the (possible) position of the particle at each of our time-grid points.

We promised to discretize the action from Eq. (5.109). This involves two separate steps. First, we have to discretize the integral: we haven't seen how to carry out numerical integration yet (the subject of chapter 7). Even so, you are already familiar with the simplest-possible way of carrying out an integral, namely the *rectangle rule*: assume that the area under L from t_k to t_{k+1} can be approximated by the area of a rectangle, with width Δt. Equivalently, the rectangle rule approximates L as a constant from t_k to t_{k+1}, namely a straight (horizontal) line. Then, the areas of all the individual rectangles have to be added up. Second, we have to discretize the time derivative $\dot{x}$. This is a process that you are even more familiar with, having studied chapter 3. The simplest thing to do is to employ a forward-difference scheme.[48] Putting these two discretization steps together allows us to introduce the finite-n analogue of Eq. (5.109):

$$S_n \equiv \sum_{k=0}^{n-2} \Delta t \left[\frac{1}{2} m \left(\frac{x_{k+1} - x_k}{\Delta t} \right)^2 - V(x_k) \right] \tag{5.111}$$

where the sum goes up to $n-2$: the last rectangle gets determined by the value of the function on the left. Since the continuum version of Hamilton's principle involved determining $x(t)$ with $x(0)$ and $x(T)$ kept fixed, we will now carry out the corresponding extremization of S_n, with x_0 and x_{n-1} kept fixed. This means that we are dealing with n points in total, but only $n-2$ variable points. To keep thing simple, we define $n_{var} = n - 2$. This means that our discrete action, S_n, is a function of these n_{var} variables:

$$S_n = S_n(x_1, x_2, \dots, x_{n_{var}}) \tag{5.112}$$

There you have it: our problem has now become that of finding a minimum of S_n in this n_{var}-dimensional space. But this is precisely the problem we tackled in section 5.5.2, when we were faced with a scalar function of many variables, $\phi(\mathbf{x})$. Crucially, Eq. (5.112) shows us that S_n is a *function* of a discrete number of variables, x_1, x_2, and so on; there is no integral, no derivative, and no functional left anymore.

5.6.3 Newton's Method for the Discrete Action

We now decide to employ Newton's method from section 5.5.4 for this multidimensional minimization problem. While we haven't yet reached the stage of implementing things in Python, let us try to make sure we won't get confused by the notation. We are dealing with n positions x_k; of these, x_0 and x_{n-1} are kept fixed, so the only "true" variables are $x_1, x_2, \dots, x_{n_{var}}$. These n_{var} variables will be determined by our Newton's method; the function we will employ to do that (`multi_newton()`) will expect our variables to employ the

[47] Note that this k is a dummy summation variable over our discretization index, so it has nothing to do with the iteration counter, e.g., $\mathbf{x}^{(k)}$, we encountered earlier.
[48] A problem invites you to do better.

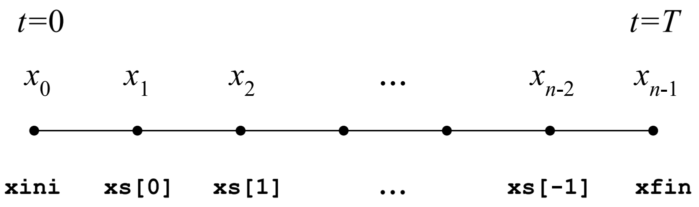


Fig. 5.16 Discretized coordinates, also showing the corresponding Python variables

standard Python 0-indexing. Thus, we will employ distinct Python names for the first and last point: x_0 and x_{n-1} will be `xini` and `xfin`, respectively. The n_{var} actual variables, which are to be determined, will be stored in our usual `numpy` array by the name of `xs`, which will be indexed from `0` up to `nvar-1`. We attempt to summarize these facts in Fig. 5.16: notice how `xs[0]` corresponds to x_1, the first true variable; similarly, `xs[-1]` is another name for `xs[nvar-1]` and corresponds to x_{n-2}, the last true variable.

As mentioned, we plan to employ Newton's method for minimization. We recall from Eq. (5.106) that this will require the gradient vector $\nabla\phi(\mathbf{x})$ as well as the Hessian, i.e., the Jacobian of the gradient, $\mathbf{J}_{\nabla\phi}(\mathbf{x})$; of course, here we are faced with S_n instead of ϕ. Significantly, our approach in earlier sections was to evaluate any needed derivatives using a finite-difference scheme. In the present case, however, we have already employed a forward difference scheme once, in order to get to our discrete action, Eq. (5.111). More importantly, Eq. (5.111) contains the sum of a large number of contributions, but any derivative with respect to a given position will not involve so many contributions; in other words, if we try to evaluate the first and second derivatives of S_n numerically, we will be asking for trouble. Instead, we decide to take any required derivatives analytically; we will then code up our results in the following subsection.

To produce the components of the gradient vector, we need to take the derivative of S_n with respect to x_i, for $i = 1, 2, \ldots, n_{var}$. As repeatedly mentioned, x_0 and x_{n-1} are kept fixed so we will not be taking any derivatives with respect to them. Let's look at a given component:

$$\begin{aligned}\frac{\partial S_n}{\partial x_i} &= \sum_{k=0}^{n-2} \Delta t \left[\frac{m}{(\Delta t)^2}(x_{k+1} - x_k)(\delta_{i,k+1} - \delta_{i,k}) - \frac{\partial V(x_k)}{\partial x_i}\delta_{i,k}\right] \\ &= \frac{m}{\Delta t}(2x_i - x_{i-1} - x_{i+1}) - \Delta t\,\frac{\partial V(x_i)}{\partial x_i}\end{aligned} \tag{5.113}$$

In the first line, we saw the $1/2$ get cancelled by taking the derivative of $(x_{k+1} - x_k)^2$; observe that only specific values of k contribute to this sum. This fact is taken into account in the second line, where the Kronecker deltas are used to eliminate most of the terms in the sum, leaving us with a simple expression.[49] Note that, even though we're only taking the derivative with respect to x_i for $i = 1, 2, \ldots, n_{var}$, the result in Eq. (5.113) also involves x_0 and $x_{n_{var}+1} = x_{n-1}$, due to the presence of the x_{i-1} and x_{i+1} terms, respectively.

Similarly, we can find the Hessian by taking the derivative of the gradient vector com-

[49] In the study of differential equations (the subject of chapter 8) our result is known as the *Verlet algorithm*.

ponents with respect to x_j, where $j = 1, 2, \ldots, n_{var}$. We have:

$$\frac{\partial S_n}{\partial x_j \partial x_i} = \frac{m}{\Delta t}(2\delta_{j,i} - \delta_{j,i-1} - \delta_{j,i+1}) - \Delta t \frac{\partial^2 V(x_i)}{\partial x_i^2}\delta_{j,i} \tag{5.114}$$

We see that the Hessian matrix will only have elements on the main diagonal and on the two diagonals next two it; in other words, it will be a tridiagonal matrix. Of course, this only holds for our specific prescription of how to discretize the action.

Armed with the gradient in Eq. (5.113) and the Hessian in Eq. (5.114), we are now ready to employ Newton's multidimensional minimization method from Eq. (5.106). Note that we haven't specified anything about $V(x_i)$ so far; this is a feature, not a bug. It means that our formalism of discretizing and then extremizing the action will apply to any physical problem where the potential energy at x_i depends only on x_i,[50] regardless of the specific form V takes.

5.6.4 Implementation

Before we implement our approach in Python, we have to make things concrete. Let's pick a specific form for the potential energy, that of the *quartic oscillator*:

$$V = \frac{1}{4}x^4 \tag{5.115}$$

where we didn't include an extra coefficient in front, for the sake of simplicity. Of course, the classical quartic oscillator is a problem that's been studied extensively; we are merely using it as a case where: (a) you won't be able to think of the analytical solution off the top off your head, and (b) the dependence on x is nonlinear, so we'll get to test the robustness of our minimization methodology for a challenging case. You can generalize our approach to more complicated scenarios later. Given the forms of Eq. (5.113) and Eq. (5.114), we realize that we are really only interested in the first and second derivative of the potential energy, namely:

$$F(x) = -\frac{\partial V}{\partial x} = -x^3, \qquad F'(x) = -\frac{\partial^2 V}{\partial x^2} = -3x^2 \tag{5.116}$$

where we also took the opportunity to define the force (and introduce its first derivative).

Code 5.7 starts by importing our earlier function `multi_newton()`. We then define a function to hold several parameters; we do this here in order to help our future selves, who may be interested in trying out different endpoint values, total times, masses,[51] or numbers of variables, n_{var}. This way of localizing all the parameters in one function and then having other parts of the code call it to get the parameter values is somewhat crude,[52] but still better than using global variables (i.e., Python variables defined outside a function and never explicitly communicated to it). In our example we employ $n_{var} = 99$; this was chosen such that $n = 101$, which in its turn, see Eq. (5.110), leads to $\Delta t = 0.01$ in the

[50] Of course, even this requirement can be loosened, by appropriately generalizing the above derivation.
[51] For simplicity, we take $m = 1$.
[52] This is a case where object orientation would really help.

Code 5.7 **action.py**

```
from multi_newton import multi_newton
import numpy as np

def params():
     nvar = 99; m = 1.
     xini, xfin = 2., 0.
     tt = 1.; dt = tt/(nvar+1)
     return nvar, m, xini, xfin, dt

def fod(der,x):
     return -x**3 if der==0 else -3*x**2

def actfs(xs):
     nvar, m, xini, xfin, dt = params()
     arr = np.zeros(nvar)
     arr[0] = (m/dt)*(2*xs[0]-xini-xs[1]) + dt*fod(0,xs[0])
     arr[1:-1] = (m/dt)*(2*xs[1:-1] - xs[:-2] - xs[2:])
     arr[1:-1] += dt*fod(0,xs[1:-1])
     arr[-1] = (m/dt)*(2*xs[-1]-xs[-2]-xfin) + dt*fod(0,xs[-1])
     return arr

def actjac(actfs,xs):
     nvar, m, xini, xfin, dt = params()
     Jf = np.zeros((nvar,nvar))
     np.fill_diagonal(Jf, 2*m/dt + fod(1,xs)*dt)
     np.fill_diagonal(Jf[1:,:], -m/dt)
     np.fill_diagonal(Jf[:,1:], -m/dt)
     actfs_xs = actfs(xs)
     return Jf, actfs_xs

if __name__ == '__main__':
     nvar, m, xini, xfin, dt = params()
     xolds = np.array([2-0.02*i for i in range(1,nvar+1)])
     xnews = multi_newton(actfs, actjac, xolds); print(xnews)
```

appropriate units. Importantly, this means that we have to solve a minimization problem in ~100 variables, which is a larger-scale problem than any we solved in earlier sections.

We then define a simple one-liner function to evaluate the force-or-derivative. This is the only other place in our program where specifics about the physics problem are encoded. A point that will come in handy below: the parameter `x` of our function `fod()` can handle either single numbers or entire `numpy` arrays.

The bulk of our code is to be found in the following two functions. We recall from Code 5.5 that `multi_newton()` has a very specific interface: as its first argument it expects a function that returns a 1d `numpy` array and as its second argument a function that returns a 2d `numpy` array and a 1d array. These functions implement the gradient vector of Eq. (5.113) and the Jacobian/Hessian of Eq. (5.114). Crucially, both these functions do not employ *any* explicit loops or indices (like `i`, `j` and so on). In most of our codes we have prioritized legibility over efficiency. Even here, despite the fact that our slicing is more efficient than the indexed alternative, our guiding principle was actually clarity: the grid in Fig. 5.16 can be confusing, so we'd rather avoid having to deal with indices that go up to `nvar-1` or perhaps `nvar-2` (or even worse: `n-3` or `n-4`) when we step through our arrays. As it so happens, a problem in chapter 1 asked you to rewrite these two functions using explicit indices.

Since `actfs()` is following our prescription of fixed endpoints, x_0 and x_{n-1}, the lines setting up `arr[0]` and `arr[-1]` need to be different from what we do for the "middle" points.[53] Similarly, `actjac()` needs to treat the main and other diagonals differently. We accomplish the latter task by slicing the arrays we pass as the first argument to `numpy.fill_diagonal()`. Note how for the main diagonal we pass the entire `xs` to `fod()`.

The main part of the program is incredibly simple. It sets up an initial guess vector, $\mathbf{x}^{(0)}$, designed to start near fixed endpoint x_0 and end near our other fixed endpoint x_{n-1}. It then calls our earlier `multi_newton()` function, passing in our tailored gradient and Hessian functions. Importantly, by using `params()` we have ensured that `actfs()` and `actjac()` have the same interface as our earlier `fs()` and `jacobian()` functions, so everything works together smoothly. It may be surprising to see that we've been able to solve for the dynamical trajectory of a particle by minimizing in $\sim$100 variables, using a code that takes up less than a page. Of course, this is slightly misleading, since we are also calling `multi_newton()`, which in its turn calls `gauelim_pivot()` and `termcrit()`. The former then calls `backsub()`. Thus, our latest program fits on one page, but only because we can so smoothly make use of several functions we developed earlier.

Instead of showing you the 99 numbers that get printed on the screen when you run this code, we have decided to plot them in Fig. 5.17. Our `xolds` is labelled with "Initial guess #1". The converged solution of `xnews` is labelled with "Minimized". You may be wondering how we know that our converged solution corresponds to a minimum; if so, compute the eigenvalues of the Hessian corresponding to `xnews`. Wanting to make sure that our result was not a fluke that was directly tied to a "magic" choice of our initial guess vector, we try out another `xolds`, labelled with "Initial guess #2": this leads to the exact same minimum as before, thereby increasing our confidence in our result.

It's worth pausing for a second to marvel at our accomplishment. In this project, we went beyond the formal statement that the action "could" be minimized or "should" be

[53] This is analogous to our discussion of the central difference in section 3.3.2.

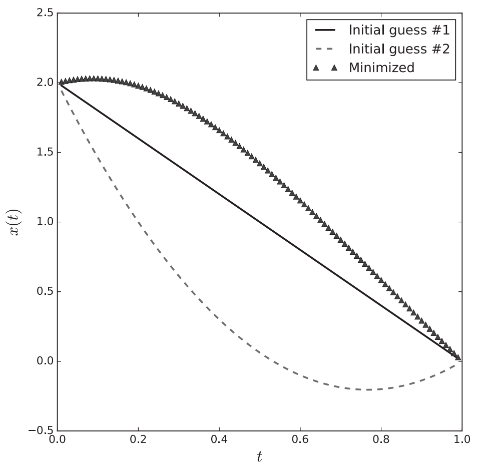


Fig. 5.17 Finding the trajectory of a particle via minimization of the action

minimized and actually minimized the action. We did this by using a (somewhat crude) discretization scheme and stepping through a many-variable minimization process which, incidentally, only required roughly half a dozen iterations to converge. You may become more convinced of the importance of this fact after you attempt the problem that asks you to carry out a minimization for the (easier) case of the *harmonic* oscillator using the gradient descent method. You will discover that multidimensional minimization is a fickle procedure, with few guarantees. A combination of analytical work and earlier coding has allowed us to find a practical avenue toward making the action stationary and thereby determining the physical trajectory of the particle.

Now that we've marvelled at how great we did, let's tone things down: had we actually followed the standard textbook-route, we would have used Hamilton's principle to find the Euler–Lagrange equation(s), thereby getting an equation of motion, which would be a differential equation. For our case, this would have been simply $m\ddot{x} = -x^3$. This differential equation, in turn, can straightforwardly be solved using its own discretization scheme, as you will soon learn (in chapter 8). In contradistinction to this, in our present approach we had to repeatedly employ 99×99 matrices, even though the problem is reasonably simple. When all is said and done, it's nice to see that you can calculate dynamical trajectories without having to solve a differential equation.

5.7 Problems

1. Come up with an example where the relative termination criterion of Eq. (5.18) fails. Then, replace it with Eq. (5.19) and re-run your root-finding program.
2. This problem studies fixed-point iteration for the case of $g(x) = cx(1 - x)$. Examine:

(a) $c = 0.9$: play around with the number of iterations, tolerance, termination criterion, and initial guess to see where all your solutions tend to go to.
(b) $c = 1.5$: see how long it takes you to find a fixed point.
(c) $c = 2.8$: does this take longer than the $c = 1.5$ case?
(d) $c = 3.2$: start with $x^{(0)} = 0.2$ and observe that a new pattern emerges.
(e) $c = 3.5$: start with $x^{(0)} = 0.2$ and see another pattern emerge.
(f) $c = 3.6$: start with $x^{(0)} = 0.2$ and attempt to discern any pattern.

You may need to plot the $x^{(k)}$ as a function of k in order to (attempt to) discern the patterns. For future reference, this problem is known as a "logistic map".

3. Use bisection for $f(x) = 1/(x-3)$ in the interval $(0, 5)$. Is your answer correct?
4. Code up Newton's method for our example function $f(x) = e^{x-\sqrt{x}} - x$ and reproduce Fig. 5.6. You should evaluate $f'(x)$ analytically.
5. Analytically use Newton's method to write $\sqrt{2}$ as a ratio of two *integers*. Provide a few (increasingly better) approximations.
6. Code up a *safe* version of Newton's method, by combining it with the bisection method. Specifically, start with a bracketing interval and carry out a Newton step: if the iterate "wants" to leave the interval, carry out a bisection step, instead. Repeat.
7. We will now recast Newton's method as a special case of the fixed-point iteration method. Specifically, if your problem is $f(x) = 0$, then one way of transforming it into the fixed-point form $x = g(x)$ is to take:

$$g(x) = x - \frac{f(x)}{f'(x)} \tag{5.117}$$

Assume x^* is a simple root (i.e., $f(x^*) = 0$ and $f'(x^*) \neq 0$). You should analytically evaluate $g'(x^*)$. You should then combine your result with our analysis of the convergence properties of the fixed-point iteration method to find the order of convergence for the method of Eq. (5.117).

8. Having recast Newton's method as a version of fixed-point iteration in the previous problem, we will now build on that result. Assume x^* is a root of multiplicity m. This means we can write:

$$f(x) = (x - x^*)^m q(x) \tag{5.118}$$

where $q(x^*) \neq 0$. Now:

(a) Analytically derive a formula for the $g(x)$ of Eq. (5.117) where you've employed Eq. (5.118).
(b) Use that result to show that $g'(x^*) = (m-1)/m$. From there, since $m = 2$ or higher, draw the conclusion that Newton's method converges linearly for multiple roots.
(c) It should now be straightforward to see that if you had used:

$$g(x) = x - m\frac{f(x)}{f'(x)} \tag{5.119}$$

instead of Eq. (5.117), then you would have found $g'(x^*) = 0$, in which case Newton's method would converge quadratically (again).

9. In practice, we don't know the multiplicity of a root ahead of time, so the utility of Eq. (5.119) is limited. In the first part of the previous problem, you incidentally also derived a formula for $w(x) = f(x)/f'(x)$ for the case of Eq. (5.118). Similarly, in the second part of that problem you found a formula for $w'(x)$, which leads to $w'(x^*) = 1/m$. But this means that $w'(x^*) \neq 0$, while $w(x^*) = 0$; in other words, $w(x)$ has a simple root at x^* regardless of what the multiplicity of $f(x)$ at x^* is.

   We can take advantage of this fact: whenever you think you may have a multiple root, you could apply Newton's method not to $f(x)$, but to $w(x)$. Code this approach up in Python for the case of:

$$f(x) = x^5 - 3x^4 + x^3 + 5x^2 - 6x + 2 \tag{5.120}$$

   Compare your runs to how long it takes the (unmodified) Newton's method to converge. While you're at it, also compare to the result of using Eq. (5.119), where you'll need to guess the value of m.

10. When using Newton's method, it can sometimes be frustrating to start from different initial guesses but end up producing the same root over and over. A trick that can be used to suppress an already-found root is to apply Newton's method not to $f(x)$ but to $u(x) = f(x)/(x-a)$, where a is the root you've already found and are no longer looking for. Implement this strategy for the case of our example function $f(x) = e^{x-\sqrt{x}} - x$, where you are suppressing the root $a = 1$. This means that you should always be converging to the other root, regardless of your initial guess.[54]

11. We now guide you toward deriving the secant method's order of convergence. First introduce notation similar to that in Eq. (2.5), i.e., $\Delta_k = x^{(k)} - x^*$. Now, Taylor expand $f(x^{(k)})$, $f(x^{(k-1)})$, and $f(x^{(k)})$ around x^*; note that this is different from what we did in Eq. (5.37) and elsewhere, where we Taylor expanded around $x^{(k-1)}$. Plug these three relations into Eq. (5.51) to show that Δ_k is equal to the product of Δ_{k-1} and Δ_{k-2} (times another factor). Motivated by Eq. (5.16), take $\Delta_k = m\Delta_{k-1}^p$ and $\Delta_{k-1} = m\Delta_{k-2}^p$, and combine these with your earlier result. This should lead to a quadratic equation in p, which is solved by $p = (1 + \sqrt{5})/2 \approx 1.618$.

12. Implement Ridders's method from Eq. (5.59) for our example function $f(x) = e^{x-\sqrt{x}} - x$.

13. There exists a method known as *regula falsi* or *false position* which is very similar to the secant method. The difference is that *regula falsi* is a bracketing method, so it starts with x_0 and x_1 such that $f(x_0)f(x_1) < 0$. Like the secant method, it then employs the line that goes through the two points (x_0, y_0) and (x_1, y_1), as per Eq. (5.52), and then finds the x axis intercept. The false-position method then ensures that it continues to bracket the root. Implement this approach for our example function $f(x) = e^{x-\sqrt{x}} - x$, starting with the interval $(0, 1.7)$. Do you see the method converging on both sides or not? Do you understand what that implies for the case where one of your initial points is "bad", i.e., very far from the root?

[54] Root suppression feels similar to, yet is distinct from, the concept of *deflation* for polynomials, which amounts to using *synthetic division* and then solving a smaller-sized problem.

14. Another approach, known as *Steffensen's method*, iterates according to:

$$x^{(k)} = x^{(k-1)} - \frac{f(x^{(k-1)})}{g(x^{(k-1)})} \tag{5.121}$$

where:

$$g(x^{(k-1)}) = \frac{f\left(x^{(k-1)} + f(x^{(k-1)})\right) - f(x^{(k-1)})}{f(x^{(k-1)})} \tag{5.122}$$

This achieves quadratic convergence, at the cost of requiring two function evaluations per iteration. Implement Steffensen's method in Python for our example function $f(x) = e^{x-\sqrt{x}} - x$ and compare the required number of iterations to those of other methods.

15. Explore the roots of:

$$f(x) = -x^5 + 4x^4 - 4x^3 + x^2e^x - 2x^2 - 4xe^x + 8x + 4e^x - 8 \tag{5.123}$$

using a method of your choice.

16. Take the cubic equation:

$$x^3 - 21x^2 + 120x - 100 = 0 \tag{5.124}$$

and find its roots. Now perturb the coefficient of x^3 so that it becomes 0.99 (first) or 1.01 (next) and discuss how the roots are impacted.

17. In the Project of chapter 3 we computed the values of Hermite polynomials, see Code 3.3, i.e., `psis.py`. You should now find the roots of Hermite polynomials, by writing a code similar to `legroots.py`. Digging up Eq. (6.31.19) in Ref. [90] and converting to our notation gives us:

$$\frac{\pi(4i+3-(-1)^n)}{4\sqrt{2n+1}} \le x_i^{(0)} \le \frac{4i+6-(-1)^n}{\sqrt{2n+1}} \tag{5.125}$$

for the positive Hermite-polynomial zeros. Figure out how to use this to produce a function that gives correct roots for n up to, say, 20. Compare the zeros you find with the output of the (alternatively produced) `numpy.polynomial.hermite.hermgauss()`.

18. Code up our companion matrix from Eq. (5.70) for a general polynomial. Then, use it to find all the roots of our polynomial from Eq. (5.120).

19. In `legroots.py` we computed the roots of a Legendre polynomial using Newton's method and calling `legendre.py` for the function values (and derivatives). We now see that it is possible to directly employ the recurrence relation from section 2.5.2 to evaluate the roots of a Legendre polynomial, by casting the problem into matrix form. This generalizes to other sets of polynomials, even if you don't have good initial guesses for the roots. It is part of the *Golub–Welsch algorithm*.

    (a) Take Eq. (2.86) and write it in the form:

    $$\frac{j+1}{2j+1}P_{j+1}(x) + \frac{j}{2j+1}P_{j-1}(x) = xP_j(x) \tag{5.126}$$

    If x^* is one of the zeros of $P_n(x)$, i.e., $P_n(x^*) = 0$, we can apply the above relation at $x = x^*$ for $j = 0, 1, \dots, n-1$; the result takes the form, $\mathbf{J}\,\mathbf{p}(x^*) = x^*\mathbf{p}(x^*)$. Verify that $\mathbf{J}$, known as a *Jacobi matrix* (not to be confused with a *Jacobian matrix*), is tridiagonal. The matrix elements of the **J** you found can be immediately determined from the coefficients in the recurrence relation, Eq. (5.126).

(b) Observe that $\mathbf{J}\,\mathbf{p}(x^*) = x^*\mathbf{p}(x^*)$ is an eigenvalue problem for the zeros of $P_n(x)$. Using Python, compute the eigenvalues of **J** and verify that they coincide with the output of `legroots.py`.

20. We now generalize the fixed-point iteration method of Eq. (5.22) to the problem of n equations in n unknowns. In equation form, this is simply $\mathbf{x}^{(k)} = \mathbf{g}(\mathbf{x}^{(k-1)})$ where, crucially, all quantities involved are vectors. Implement this approach in Python and apply it to the following two-variable problem:

$$\begin{cases} f_0(x_0, x_1) = x_0^2 - 2x_0 - 2x_1^2 + x_1 = 0 \\ f_1(x_0, x_1) = x_0^2 + x_0 - 2x_1^2 - 1.5x_1 - 0.05 = 0 \end{cases} \tag{5.127}$$

This system has two solutions. Do you find them both using the fixed-point iteration method? Do you understand why (not)?

21. In our study of Gaussian quadrature in chapter 7, the brute-force approach will lead to systems of nonlinear equations like the following:

$$\begin{gathered} 2 = c_0 + c_1 + c_2, \quad 0 = c_0x_0 + c_1x_1 + c_2x_2, \quad \frac{2}{3} = c_0x_0^2 + c_1x_1^2 + c_2x_2^2, \\ 0 = c_0x_0^3 + c_1x_1^3 + c_2x_2^3, \quad \frac{2}{5} = c_0x_0^4 + c_1x_1^4 + +c_2x_2^4, \quad 0 = c_0x_0^5 + c_1x_1^5 + +c_2x_2^5 \end{gathered} \tag{5.128}$$

Apply `multi_newton.py` to solve these six equations for the six unknowns.

22. In this problem we solve Eq. (5.5) together with Eq. (5.6), using Newton's method. For simplicity, work in units where $m = M_R = G = 1$ and $M_S = 2$. Assume we have made a measurement at $\mathbf{r}_0 = (2\ 1\ 3)^T$, finding:

$$\mathbf{F}_0 = (-0.016\,859\,96 \quad -0.040\,029\,72 \quad -0.014\,790\,52)^T \tag{5.129}$$

and another measurement at $\mathbf{r}_1 = (4\ 4\ 1)^T$, this time finding

$$\mathbf{F}_1 = (-0.143\,642\,92 \quad 0.120\,962\,46 \quad -0.264\,605\,37)^T \tag{5.130}$$

Set up the problem in a way that allows you to call `multi_newton.py`, with the understanding that the components of **R** are the first three elements of `xs` and the components of **S** are the next three elements of `xs`. If you're having trouble converging, try starting from `xolds = np.array([0.8,-1.1,3.1,3.5,4.5,-2.])`.

23. Implement our basic Broyden method from Eq. (5.89) for our two-variable problem in Eq. (5.72). Separately investigate the convergence for the two cases of $\mathbf{J}(\mathbf{x}^{(0)})$ being the identity matrix or a forward-difference approximation to the Jacobian.

24. Employ the secant method to minimize $f(x) = e^{x-\sqrt{x}} - x$, Eq. (5.46), and Eq. (5.92).

25. Rewrite the function `gradient()` from the code `descent.py` so that it uses an explicit loop. Compare with the original code. Then, study the dependence of the maximum required iteration number on the magnitude of γ; explore values from 0.02 to 0.30, in steps of 0.01.

26. In `descent.py` we studied the simplest case of constant γ. We will now expand our method to also include a "line-search minimization" to determine $\gamma^{(k)}$ at each iteration. If you're currently at position $\mathbf{x}$ and the gradient is $\nabla\phi(\mathbf{x})$, then you can define:

$$u(\gamma) = \phi\left(\mathbf{x} - \gamma\nabla\phi(\mathbf{x})\right) \tag{5.131}$$

Crucially, $u(\gamma)$ is a function of one variable. What this approach does is to take the negative gradient direction, $-\nabla\phi(\mathbf{x})$, and then decide how large a step γ to make in that direction, by minimizing $u(\gamma)$; since this minimization is taking place along the fixed line determined by the negative gradient, this approach is known as line-search minimization. Putting everything together, we have the following algorithm:

$$\begin{aligned} &\text{Minimize } u(\gamma^{(k-1)}) = \phi\left(\mathbf{x}^{(k-1)} - \gamma^{(k-1)}\nabla\phi(\mathbf{x}^{(k-1)})\right) \text{ to find } \gamma^{(k-1)} \\ &\mathbf{x}^{(k)} = \mathbf{x}^{(k-1)} - \gamma^{(k-1)}\nabla\phi(\mathbf{x}^{(k-1)}) \end{aligned} \tag{5.132}$$

To implement the above prescription, combine `descent.py` with (a modified version of) `secant.py`; check your program on the function $\phi(\mathbf{x}) = x_0^2 + 5x_1^2$.

27. Carry out multidimensional minimization using Newton's method for our example in Eq. (5.101). After you locate both minima (how would you know they're minima?), try starting from `xolds = np.array([1.5,0.])`. Then, use that same initial guess vector in the gradient-descent method and compare the two results.

28. An intriguing generalization of Newton's method for multidimensional root-finding borrows ideas from multidimensional minimization. The goal is to ensure that Newton's method can do a good job of finding the solution to a system of equations, even if the initial guess was poor; in other words, we wish to make sure that each step we make decreases the function-component values.[55]

To do so, we now examine the quantity $(\mathbf{f}(\mathbf{x}^{(k-1)}))^T\mathbf{f}(\mathbf{x}^{(k-1)})$: it makes sense to require that after the step the magnitude of the functions is reduced, i.e., that $(\mathbf{f}(\mathbf{x}^{(k)}))^T\mathbf{f}(\mathbf{x}^{(k)})$ is smaller. It's easy to show that the Newton step is a descent direction; in other words, the step prescribed by Newton's method picks a direction that reduces the functions' values. Here's how to see this:

$$\begin{aligned} \frac{1}{2}\left\{\nabla\left[\left(\mathbf{f}(\mathbf{x}^{(k-1)})\right)^T\mathbf{f}(\mathbf{x}^{(k-1)})\right]\right\}^T\mathbf{q}^{(k)} &= \left(\mathbf{f}(\mathbf{x}^{(k-1)})\right)^T\mathbf{J}(\mathbf{x}^{(k-1)})\left(-\left(\mathbf{J}(\mathbf{x}^{(k-1)})\right)^{-1}\mathbf{f}(\mathbf{x}^{(k-1)})\right) \\ &= -\left(\mathbf{f}(\mathbf{x}^{(k-1)})\right)^T\mathbf{f}(\mathbf{x}^{(k-1)}) < 0 \end{aligned} \tag{5.133}$$

On the first line, $\mathbf{q}^{(k)}$ is the Newton step as per Eq. (5.83). We also took the gradient of our $\mathbf{f}^T\mathbf{f}$ quantity using a standard vector identity, and also re-expressed the Newton step from Eq. (5.80). From there, we cancelled the Jacobians and were left with a negative value. Indeed, the Newton step is a descent direction.

Now the question arises how to pick the magnitude of our step in that direction (similarly to what we saw when studying the gradient-descent method for minimization). We could, at this point, employ a line-search approach as we did in an earlier problem. Instead, we will try the empirical approach of multiplying the full Newton step with $\lambda = 0.25$, if the function value would have otherwise increased. Try out this

55 In jargon, we are looking for a *globally* convergent method.

backtracking strategy for our example system from Eq. (5.72): starting from `xolds = np.array([-100.,100.])`, compare the unmodified and modified Newton's runs. Make sure to print out a message each time the step is/would have been bad.

29. In this problem we will employ the same discretization of the action as in `action.py`, still for the case of the quartic oscillator. Imagine you made a mistake when trying to approach the fixed-endpoints in your initial guess, leading you to start with $q^{(0)}(t) = 0.9375t^2 + 1.0625t - 2$. Plot the initial guess and converged solution together. Are you getting the same converged solution as in the main text? Why (or why not)? Compute the value of the action from Eq. (5.111) as well as the eigenvalues of the Jacobian for the two converged solutions and use those results to justify your conclusions.
30. Modify `action.py` to address the physical context of the *harmonic* oscillator. Take $x_0 = 0$, $x_{n-1} = 1$ and $T = 1$. Plot the coordinate of the particle as a function of time; also show the (analytically known) answer. Incidentally, what happens if you take T to be larger?
    Then, try to solve the same problem using the gradient-descent method. You will have to tune `n`, `gamma`, `kmax`, and `tol` (as well as your patience levels) to do so.
31. Modify `action.py` to address the physical context of a vertically thrown ball; start by thinking what force this implies. Take $x_0 = x_{n-1} = 0$ and $T = 10$. Plot your solution for the height of the particle as a function of time.
32. Repeat the study of the quartic oscillator, but this time instead of using Eq. (5.111) for the discretization, employ a central-difference formula. This means you should analytically calculate the new gradient and Hessian, before implementing things in Python.