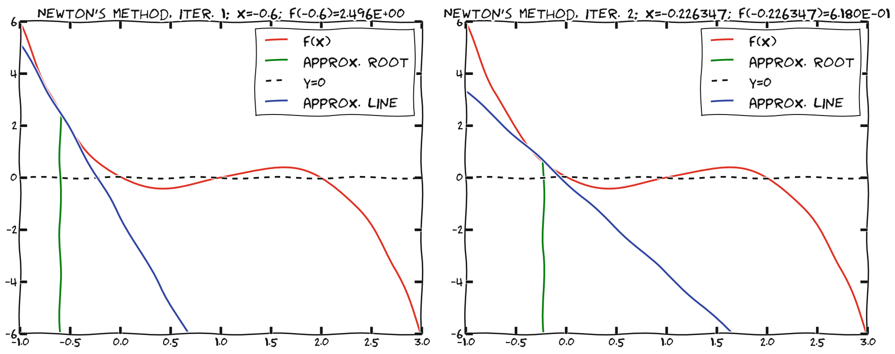
# Solving Nonlinear Algebraic Equations

# 6



As a reader of this book you are probably well into mathematics and often "accused" of being particularly good at "solving equations" (a typical comment at family dinners!). However, is it *really* true that you, with pen and paper, can solve *many* types of equations? Restricting our attention to *algebraic equations in one unknown* $x$, you can certainly do linear equations: $ax + b = 0$, and quadratic ones: $ax^2 + bx + c = 0$. You may also know that there are formulas for the roots of cubic and quartic equations too. Maybe you can do the special trigonometric equation $\sin x + \cos x = 1$ as well, but there it (probably) stops. Equations that are not reducible to one of the mentioned cannot be solved by general analytical techniques, which means that most algebraic equations arising in applications cannot be treated with pen and paper!

If we exchange the traditional idea of finding *exact* solutions to equations with the idea of rather finding *approximate* solutions, a whole new world of possibilities opens up. With such an approach, we can in principle solve *any* algebraic equation.

Let us start by introducing a common generic form for any algebraic equation:

$$f(x) = 0\,.$$

Here, $f(x)$ is some prescribed formula involving $x$. For example, the equation

$$e^{-x}\sin x = \cos x$$

S. Linge, H.P. Langtangen, *Programming for Computations – Python*,
Texts in Computational Science and Engineering 15, DOI 10.1007/978-3-319-32428-9_6

has

$$f(x) = e^{-x}\sin x - \cos x\,.$$

Just move all terms to the left-hand side and then the formula to the left of the equality sign is $f(x)$.

So, when do we really need to solve algebraic equations beyond the simplest types we can treat with pen and paper? There are two major application areas. One is when using *implicit* numerical methods for ordinary differential equations. These give rise to one or a system of algebraic equations. The other major application type is optimization, i.e., finding the maxima or minima of a function. These maxima and minima are normally found by solving the algebraic equation $F'(x) = 0$ if $F(x)$ is the function to be optimized. Differential equations are very much used throughout science and engineering, and actually most engineering problems are optimization problems in the end, because one wants a design that maximizes performance and minimizes cost.

We first consider one algebraic equation in one variable, with our usual emphasis on how to program the algorithms. *Systems* of nonlinear algebraic equations with *many variables* arise from implicit methods for ordinary and partial differential equations as well as in multivariate optimization. Our attention will be restricted to Newton's method for such systems of nonlinear algebraic equations.

**Terminology**

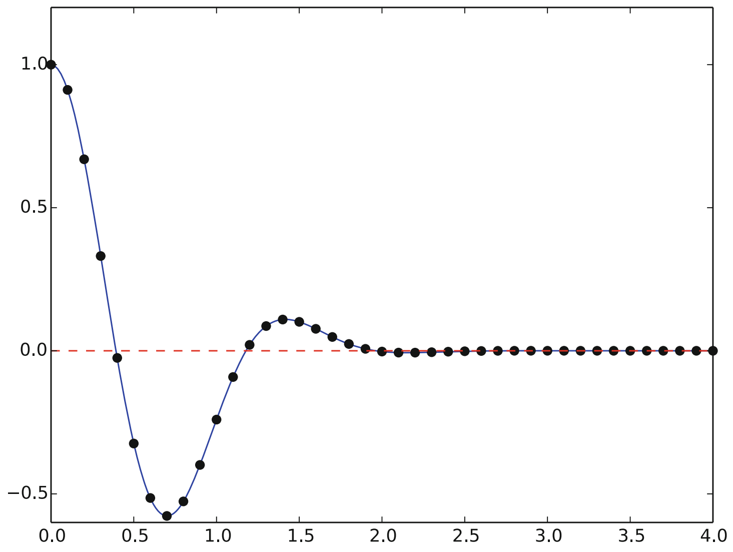
When solving algebraic equations $f(x) = 0$, we often say that the solution $x$ is a *root* of the equation. The solution process itself is thus often called *root finding*.

## 6.1 Brute Force Methods

The representation of a mathematical function $f(x)$ on a computer takes two forms. One is a Python function returning the function value given the argument, while the other is a collection of points $(x, f(x))$ along the function curve. The latter is the representation we use for plotting, together with an assumption of linear variation between the points. This representation is also very suited for equation solving and optimization: we simply go through all points and see if the function crosses the $x$ axis, or for optimization, test for a local maximum or minimum point. Because there is a lot of work to examine a huge number of points, and also because the idea is extremely simple, such approaches are often referred to as *brute force* methods. However, we are not embarrassed of explaining the methods in detail and implementing them.

### 6.1.1 Brute Force Root Finding

Assume that we have a set of points along the curve of a function $f(x)$:



We want to solve $f(x) = 0$, i.e., find the points $x$ where $f$ crosses the $x$ axis. A brute force algorithm is to run through all points on the curve and check if one point is below the $x$ axis and if the next point is above the $x$ axis, or the other way around. If this is found to be the case, we know that $f$ must be zero in between these two $x$ points.

**Numerical algorithm** More precisely, we have a set of $n + 1$ points $(x_i, y_i)$, $y_i = f(x_i)$, $i = 0, \ldots, n$, where $x_0 < \ldots < x_n$. We check if $y_i < 0$ and $y_{i+1} > 0$ (or the other way around). A compact expression for this check is to perform the test $y_i y_{i+1} < 0$. If so, the root of $f(x) = 0$ is in $[x_i, x_{i+1}]$. Assuming a linear variation of $f$ between $x_i$ and $x_{i+1}$, we have the approximation

$$f(x) \approx \frac{f(x_{i+1}) - f(x_i)}{x_{i+1} - x_i}(x - x_i) + f(x_i) = \frac{y_{i+1} - y_i}{x_{i+1} - x_i}(x - x_i) + y_i,$$

which, when set equal to zero, gives the root

$$x = x_i - \frac{x_{i+1} - x_i}{y_{i+1} - y_i} y_i .$$

**Implementation** Given some Python implementation `f(x)` of our mathematical function, a straightforward implementation of the above numerical algorithm looks like

```
x = linspace(0, 4, 10001)
y = f(x)

root = None  # Initialization
for i in range(len(x)-1):
    if y[i]*y[i+1] < 0:
         root = x[i] - (x[i+1] - x[i])/(y[i+1] - y[i])*y[i]
         break  # Jump out of loop

if root is None:
    print 'Could not find any root in [%g, %g]' % (x[0], x[-1])
else:
    print 'Find (the first) root as x=%g' % root
```

(See the file brute_force_root_finder_flat.py.)

Note the nice use of setting `root` to `None`: we can simply test `if root is None` to see if we found a root and overwrote the `None` value, or if we did not find any root among the tested points.

Running this program with some function, say $f(x) = e^{-x^2}\cos(4x)$ (which has a solution at $x = \frac{\pi}{8}$), gives the root 0.392701, which has an error of $1.9 \cdot 10^{-6}$. Increasing the number of points with a factor of ten gives a root with an error of $2.4 \cdot 10^{-8}$.

After such a quick "flat" implementation of an algorithm, we should always try to offer the algorithm as a Python function, applicable to as wide a problem domain as possible. The function should take $f$ and an associated interval $[a, b]$ as input, as well as a number of points ($n$), and return a list of all the roots in $[a, b]$. Here is our candidate for a good implementation of the brute force rooting finding algorithm:

```
def brute_force_root_finder(f, a, b, n):
    from numpy import linspace
    x = linspace(a, b, n)
    y = f(x)
    roots = []
    for i in range(n-1):
        if y[i]*y[i+1] < 0:
            root = x[i] - (x[i+1] - x[i])/(y[i+1] - y[i])*y[i]
            roots.append(root)
    return roots
```

(See the file brute_force_root_finder_function.py.)

This time we use another elegant technique to indicate if roots were found or not: `roots` is an empty list if the root finding was unsuccessful, otherwise it contains all the roots. Application of the function to the previous example can be coded as

```
def demo():
    from numpy import exp, cos
    roots = brute_force_root_finder(
        lambda x: exp(-x**2)*cos(4*x), 0, 4, 1001)
    if roots:
        print roots
    else:
        print 'Could not find any roots'
```

Note that `if roots` evaluates to `True` if `roots` is non-empty. This is a general test in Python: `if X` evaluates to `True` if `X` is non-empty or has a nonzero value.

## 6.1.2 Brute Force Optimization

**Numerical algorithm** We realize that $x_i$ corresponds to a maximum point if $y_{i-1} < y_i > y_{i+1}$. Similarly, $x_i$ corresponds to a minimum if $y_{i-1} > y_i < y_{i+1}$. We can do this test for all "inner" points $i = 1, \ldots, n-1$ to find all local minima and maxima. In addition, we need to add an end point, $i = 0$ or $i = n$, if the corresponding $y_i$ is a global maximum or minimum.

**Implementation** The algorithm above can be translated to the following Python function (file `brute_force_optimizer.py`):

```python
def brute_force_optimizer(f, a, b, n):
    from numpy import linspace
    x = linspace(a, b, n)
    y = f(x)
    # Let maxima and minima hold the indices corresponding
    # to (local) maxima and minima points
    minima = []
    maxima = []
    for i in range(n-1):
        if y[i-1] < y[i] > y[i+1]:
            maxima.append(i)
        if y[i-1] > y[i] < y[i+1]:
            minima.append(i)

    # What about the end points?
    y_max_inner = max([y[i] for i in maxima])
    y_min_inner = min([y[i] for i in minima])
    if y[0] > y_max_inner:
        maxima.append(0)
    if y[len(x)-1] > y_max_inner:
        maxima.append(len(x)-1)
    if y[0] < y_min_inner:
        minima.append(0)
    if y[len(x)-1] < y_min_inner:
        minima.append(len(x)-1)

    # Return x and y values
    return [(x[i], y[i]) for i in minima], \
           [(x[i], y[i]) for i in maxima]
```

The `max` and `min` functions are standard Python functions for finding the maximum and minimum element of a list or an object that one can iterate over with a for loop.

An application to $f(x) = e^{-x^2}\cos(4x)$ looks like

```
def demo():
    from numpy import exp, cos
    minima, maxima = brute_force_optimizer(
        lambda x: exp(-x**2)*cos(4*x), 0, 4, 1001)
    print 'Minima:', minima
    print 'Maxima:', maxima
```

### 6.1.3 Model Problem for Algebraic Equations

We shall consider the very simple problem of finding the square root of 9, which is the positive solution of $x^2 = 9$. The nice feature of solving an equation whose solution is known beforehand is that we can easily investigate how the numerical method and the implementation perform in the search for the solution. The $f(x)$ function corresponding to the equation $x^2 = 9$ is

$$f(x) = x^2 - 9\,.$$

Our interval of interest for solutions will be [0, 1000] (the upper limit here is chosen somewhat arbitrarily).

In the following, we will present several efficient and accurate methods for solving nonlinear algebraic equations, both single equation and systems of equations. The methods all have in common that they search for *approximate* solutions. The methods differ, however, in the way they perform the search for solutions. The idea for the search influences the efficiency of the search and the reliability of actually finding a solution. For example, Newton's method is very fast, but not reliable, while the bisection method is the slowest, but absolutely reliable. No method is best at all problems, so we need different methods for different problems.

**What is the difference between linear and nonlinear equations?**

You know how to solve linear equations $ax + b = 0$: $x = -b/a$. All other types of equations $f(x) = 0$, i.e., when $f(x)$ is not a linear function of $x$, are called nonlinear. A typical way of recognizing a nonlinear equation is to observe that $x$ is "not alone" as in $ax$, but involved in a product with itself, such as in $x^3 + 2x^2 - 9 = 0$. We say that $x^3$ and $2x^2$ are nonlinear terms. An equation like $\sin x + e^x \cos x = 0$ is also nonlinear although $x$ is not explicitly multiplied by itself, but the Taylor series of $\sin x$, $e^x$, and $\cos x$ all involve polynomials of $x$ where $x$ is multiplied by itself.

## 6.2 Newton's Method

*Newton's method*, also known as *Newton-Raphson's method*, is a very famous and widely used method for solving nonlinear algebraic equations. Compared to the other methods we will consider, it is generally the fastest one (usually by far). It does not guarantee that an existing solution will be found, however.

A fundamental idea of numerical methods for nonlinear equations is to construct a series of linear equations (since we know how to solve linear equations) and hope that the solutions of these linear equations bring us closer and closer to the solution of the nonlinear equation. The idea will be clearer when we present Newton's method and the secant method.

### 6.2.1 Deriving and Implementing Newton's Method

Figure 6.1 shows the $f(x)$ function in our model equation $x^2 - 9 = 0$. Numerical methods for algebraic equations require us to guess at a solution first. Here, this guess is called $x_0$. The fundamental idea of Newton's method is to approximate the original function $f(x)$ by a straight line, i.e., a linear function, since it is straightforward to solve linear equations. There are infinitely many choices of how to approximate $f(x)$ by a straight line. Newton's method applies the tangent of $f(x)$ at $x_0$, see the rightmost tangent in Fig. 6.1. This linear tangent function crosses the $x$ axis at a point we call $x_1$. This is (hopefully) a better approximation to the solution of $f(x) = 0$ than $x_0$. The next fundamental idea is to repeat this process. We find the tangent of $f$ at $x_1$, compute where it crosses the $x$ axis, at a point called $x_2$, and repeat the process again. Figure 6.1 shows that the process brings us closer and closer to the left. It remains, however, to see if we hit $x = 3$ or come sufficiently close to this solution.

How do we compute the tangent of a function $f(x)$ at a point $x_0$? The tangent function, here called $\tilde{f}(x)$, is linear and has two properties:

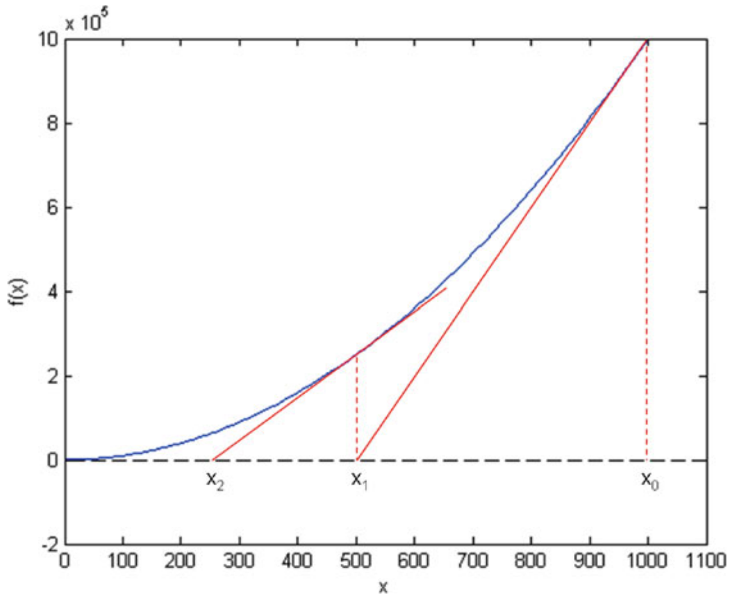


**Fig. 6.1** Illustrates the idea of Newton's method with $f(x) = x^2 - 9$, repeatedly solving for crossing of tangent lines with the $x$ axis

1. the slope equals to $f'(x_0)$
2. the tangent touches the $f(x)$ curve at $x_0$

So, if we write the tangent function as $\tilde{f}(x) = ax + b$, we must require $\tilde{f}'(x_0) = f'(x_0)$ and $\tilde{f}(x_0) = f(x_0)$, resulting in

$$\tilde{f}(x) = f(x_0) + f'(x_0)(x - x_0)\,.$$

The key step in Newton's method is to find where the tangent crosses the $x$ axis, which means solving $\tilde{f}(x) = 0$:

$$\tilde{f}(x) = 0 \quad \Rightarrow \quad x = x_0 - \frac{f(x_0)}{f'(x_0)}\,.$$

This is our new candidate point, which we call $x_1$:

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}\,.$$

With $x_0 = 1000$, we get $x_1 \approx 500$, which is in accordance with the graph in Fig. 6.1. Repeating the process, we get

$$x_2 = x_1 - \frac{f(x_1)}{f'(x_1)} \approx 250\,.$$

The general scheme of Newton's method may be written as

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}, \quad n = 0, 1, 2, \ldots \tag{6.1}$$

The computation in (6.1) is repeated until $f(x_n)$ is close enough to zero. More precisely, we test if $|f(x_n)| < \epsilon$, with $\epsilon$ being a small number.

We moved from 1000 to 250 in two iterations, so it is exciting to see how fast we can approach the solution $x = 3$. A computer program can automate the calculations. Our first try at implementing Newton's method is in a function `naive_Newton`:

```python
def naive_Newton(f, dfdx, x, eps):
    while abs(f(x)) > eps:
        x = x - float(f(x))/dfdx(x)
    return x
```

The argument `x` is the starting value, called $x_0$ in our previous mathematical description. We use `float(f(x))` to ensure that an integer division does not happen by accident if `f(x)` and `dfdx(x)` both are integers for some `x`.

To solve the problem $x^2 = 9$ we also need to implement

```python
def f(x):
    return x**2 - 9

def dfdx(x):
    return 2*x

print naive_Newton(f, dfdx, 1000, 0.001)
```

**Why not use an array for the $x$ approximations?**

Newton's method is normally formulated with an *iteration index* $n$,

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}\,.$$

Seeing such an index, many would implement this as

```
x[n+1] = x[n] - f(x[n])/dfdx(x[n])
```

Such an array is fine, but requires storage of all the approximations. In large industrial applications, where Newton's method solves millions of equations at once, one cannot afford to store all the intermediate approximations in memory, so then it is important to understand that the algorithm in Newton's method has no more need for $x_n$ when $x_{n+1}$ is computed. Therefore, we can work with one variable `x` and overwrite the previous value:

```
x = x - f(x)/dfdx(x)
```

Running `naive_Newton(f, dfdx, 1000, eps=0.001)` results in the approximate solution 3.000027639. A smaller value of `eps` will produce a more accurate solution. Unfortunately, the plain `naive_Newton` function does not return how many iterations it used, nor does it print out all the approximations $x_0, x_1, x_2, \ldots$, which would indeed be a nice feature. If we insert such a printout, a rerun results in

```
500.0045
250.011249919
125.02362415
62.5478052723
31.3458476066
15.816483488
8.1927550496
4.64564330569
3.2914711388
3.01290538807
3.00002763928
```

We clearly see that the iterations approach the solution quickly. This speed of the search for the solution is the primary strength of Newton's method compared to other methods.

### 6.2.2 Making a More Efficient and Robust Implementation

The `naive_Newton` function works fine for the example we are considering here. However, for more general use, there are some pitfalls that should be fixed in an improved version of the code. An example may illustrate what the problem is: let us solve $\tanh(x) = 0$, which has solution $x = 0$. With $|x_0| \le 1.08$ everything works fine. For example, $x_0$ leads to six iterations if $\epsilon = 0.001$:

```
-1.05895313436
0.989404207298
-0.784566773086
0.36399816111
-0.0330146961372
2.3995252668e-05
```

Adjusting $x_0$ slightly to 1.09 gives division by zero! The approximations computed by Newton's method become

```
-1.09331618202
1.10490354324
-1.14615550788
1.30303261823
-2.06492300238
13.4731428006
-1.26055913647e+11
```

The division by zero is caused by $x_7 = -1.26055913647 \cdot 10^{11}$, because $\tanh(x_7)$ is 1.0 to machine precision, and then $f'(x) = 1 - \tanh(x)^2$ becomes zero in the denominator in Newton's method.

The underlying problem, leading to the division by zero in the above example, is that Newton's method *diverges*: the approximations move further and further away from $x = 0$. If it had not been for the division by zero, the condition in the `while` loop would always be true and the loop would run forever. Divergence of Newton's method occasionally happens, and the remedy is to abort the method when a maximum number of iterations is reached.

Another disadvantage of the `naive_Newton` function is that it calls the $f(x)$ function twice as many times as necessary. This extra work is of no concern when $f(x)$ is fast to evaluate, but in large-scale industrial software, one call to $f(x)$ might take hours or days, and then removing unnecessary calls is important. The solution in our function is to store the call `f(x)` in a variable (`f_value`) and reuse the value instead of making a new call `f(x)`.

To summarize, we want to write an improved function for implementing Newton's method where we

- avoid division by zero
- allow a maximum number of iterations
- avoid the extra evaluation of $f(x)$

A more robust and efficient version of the function, inserted in a complete program `Newtons_method.py` for solving $x^2 - 9 = 0$, is listed below.

```python
def Newton(f, dfdx, x, eps):
    f_value = f(x)
    iteration_counter = 0
    while abs(f_value) > eps and iteration_counter < 100:
        try:
            x = x - float(f_value)/dfdx(x)
        except ZeroDivisionError:
            print "Error! - derivative zero for x = ", x
            sys.exit(1)     # Abort with error

        f_value = f(x)
        iteration_counter += 1

    # Here, either a solution is found, or too many iterations
    if abs(f_value) > eps:
        iteration_counter = -1
    return x, iteration_counter

def f(x):
    return x**2 - 9

def dfdx(x):
    return 2*x

solution, no_iterations = Newton(f, dfdx, x=1000, eps=1.0e-6)

if no_iterations > 0:    # Solution found
    print "Number of function calls: %d" % (1 + 2*no_iterations)
    print "A solution is: %f" % (solution)
else:
    print "Solution not found!"
```

Handling of the potential division by zero is done by a `try-except` construction. Python *tries* to run the code in the `try` block. If anything goes wrong here, or more precisely, if Python raises an *exception* caused by a problem (such as division by zero, array index out of bounds, use of undefined variable, etc.), the execution jumps immediately to the `except` block. Here, the programmer can take appropriate actions. In the present case, we simply stop the program. (Professional programmers would avoid calling `sys.exit` inside a function. Instead, they would raise a new exception with an informative error message, and let the calling code have another `try-except` construction to stop the program.)

The division by zero will always be detected and the program will be stopped. The main purpose of our way of treating the division by zero is to give the user a more informative error message and stop the program in a gentler way.

Calling `sys.exit` with an argument different from zero (here `1`) signifies that the program stopped because of an error. It is a good habit to supply the value `1`, because tools in the operating system can then be used by other programs to detect that our program failed.

To prevent an infinite loop because of divergent iterations, we have introduced the integer variable `iteration_counter` to count the number of iterations in Newton's method. With `iteration_counter` we can easily extend the condition in the `while` such that no more iterations take place when the number of iterations reaches 100. We could easily let this limit be an argument to the function rather than a fixed constant.

The `Newton` function returns the approximate solution and the number of iterations. The latter equals $-1$ if the convergence criterion $|f(x)| < \epsilon$ was not reached within the maximum number of iterations. In the calling code, we print out the solution and the number of function calls. The main cost of a method for solving $f(x) = 0$ equations is usually the evaluation of $f(x)$ and $f'(x)$, so the total number of calls to these functions is an interesting measure of the computational work. Note that in function `Newton` there is an initial call to $f(x)$ and then one call to $f$ and one to $f'$ in each iteration.

Running `Newtons_method.py`, we get the following printout on the screen:

```
Number of function calls: 25
A solution is: 3.000000
```

As we did with the integration methods in Chapter 3, we will collect our solvers for nonlinear algebraic equations in a separate file named `nonlinear_solvers.py` for easy import and use. The first function placed in this file is then `Newton`.

The Newton scheme will work better if the starting value is close to the solution. A good starting value may often make the difference as to whether the code actually *finds* a solution or not. Because of its speed, Newton's method is often the method of first choice for solving nonlinear algebraic equations, even if the scheme is not guaranteed to work. In cases where the initial guess may be far from the solution, a good strategy is to run a few iterations with the bisection method (see Chapter 6.4) to narrow down the region where $f$ is close to zero and then switch to Newton's method for fast convergence to the solution.

Newton's method requires the analytical expression for the derivative $f'(x)$. Derivation of $f'(x)$ is not always a reliable process by hand if $f(x)$ is a complicated function. However, Python has the symbolic package SymPy, which we may use to create the required `dfdx` function. In our sample problem, the recipe goes as follows:

```python
from sympy import *
x = symbols('x')               # define x as a mathematical symbol
f_expr = x**2 - 9              # symbolic expression for f(x)
dfdx_expr = diff(f_expr, x)   # compute f'(x) symbolically
# Turn f_expr and dfdx_expr into plain Python functions
f = lambdify([x],      # argument to f
             f_expr)   # symbolic expression to be evaluated
dfdx = lambdify([x], dfdx_expr)
print dfdx(5)          # will print 10
```

The nice feature of this code snippet is that `dfdx_expr` is the exact analytical expression for the derivative, `2*x`, if you print it out. This is a symbolic expression so we cannot do numerical computing with it, but the `lambdify` constructions turn symbolic expressions into callable Python functions.

The next method is the secant method, which is usually slower than Newton's method, but it does not require an expression for $f'(x)$, and it has only one function call per iteration.

## 6.3 The Secant Method

When finding the derivative $f'(x)$ in Newton's method is problematic, or when function evaluations take too long; we may adjust the method slightly. Instead of using tangent lines to the graph we may use secants[1]. The approach is referred to as the *secant method*, and the idea is illustrated graphically in Fig. 6.2 for our example problem $x^2 - 9 = 0$.

The idea of the secant method is to think as in Newton's method, but instead of using $f'(x_n)$, we approximate this derivative by a finite difference or the *secant*, i.e., the slope of the straight line that goes through the points $(x_n, f(x_n))$ and $(x_{n-1}, f(x_{n-1}))$ on the graph, given by the two most recent approximations $x_n$ and $x_{n-1}$. This slope reads

$$\frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}\,. \tag{6.2}$$

Inserting this expression for $f'(x_n)$ in Newton's method simply gives us the secant method:

$$x_{n+1} = x_n - \frac{f(x_n)}{\frac{f(x_n)-f(x_{n-1})}{x_n-x_{n-1}}},$$

or

$$x_{n+1} = x_n - f(x_n)\frac{x_n - x_{n-1}}{f(x_n) - f(x_{n-1})}\,. \tag{6.3}$$

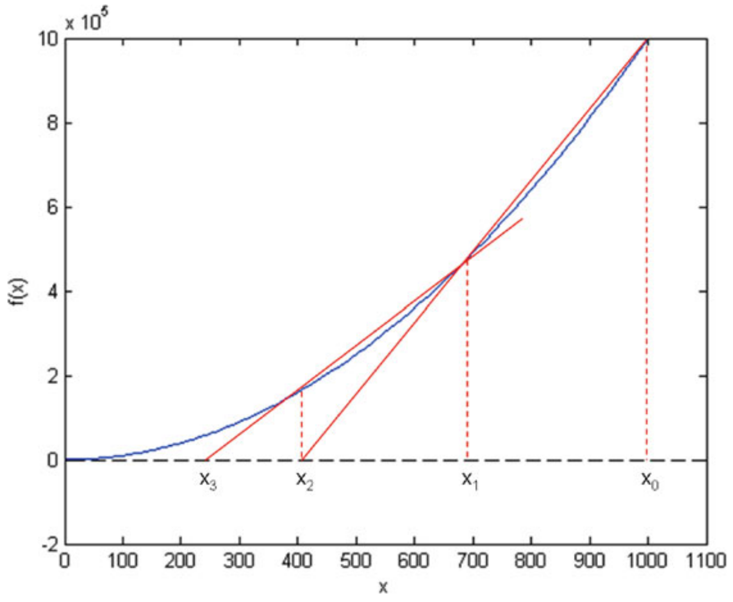


**Fig. 6.2** Illustrates the use of secants in the secant method when solving $x^2 - 9 = 0$, $x \in [0, 1000]$. From two chosen starting values, $x_0 = 1000$ and $x_1 = 700$ the crossing $x_2$ of the corresponding secant with the $x$ axis is computed, followed by a similar computation of $x_3$ from $x_1$ and $x_2$

[1] https://en.wikipedia.org/wiki/Secant_line

Comparing (6.3) to the graph in Fig. 6.2, we see how *two* chosen starting points ($x_0 = 1000$, $x_1 = 700$, and corresponding function values) are used to compute $x_2$. Once we have $x_2$, we similarly use $x_1$ and $x_2$ to compute $x_3$. As with Newton's method, the procedure is repeated until $f(x_n)$ is below some chosen limit value, or some limit on the number of iterations has been reached. We use an iteration counter here too, based on the same thinking as in the implementation of Newton's method.

We can store the approximations $x_n$ in an array, but as in Newton's method, we notice that the computation of $x_{n+1}$ only needs knowledge of $x_n$ and $x_{n-1}$, not "older" approximations. Therefore, we can make use of only three variables: `x` for $x_{n+1}$, `x1` for $x_n$, and `x0` for $x_{n-1}$. Note that `x0` and `x1` must be given (guessed) for the algorithm to start.

A program `secant_method.py` that solves our example problem may be written as:

```
def secant(f, x0, x1, eps):
    f_x0 = f(x0)
    f_x1 = f(x1)
    iteration_counter = 0
    while abs(f_x1) > eps and iteration_counter < 100:
        try:
            denominator = float(f_x1 - f_x0)/(x1 - x0)
            x = x1 - float(f_x1)/denominator
        except ZeroDivisionError:
            print "Error! - denominator zero for x = ", x
            sys.exit(1)     # Abort with error
        x0 = x1
        x1 = x
        f_x0 = f_x1
        f_x1 = f(x1)
        iteration_counter += 1
    # Here, either a solution is found, or too many iterations
    if abs(f_x1) > eps:
        iteration_counter = -1
    return x, iteration_counter

def f(x):
    return x**2 - 9

x0 = 1000;   x1 = x0 - 1

solution, no_iterations = secant(f, x0, x1, eps=1.0e-6)

if no_iterations > 0:    # Solution found
    print "Number of function calls: %d" % (2 + no_iterations)
    print "A solution is: %f" % (solution)
else:
    print "Solution not found!"
```

The number of function calls is now related to `no_iterations`, i.e., the number of iterations, as `2 + no_iterations`, since we need two function calls before entering the `while` loop, and then one function call per loop iteration. Note that, even though we need two points on the graph to compute each updated estimate, only a *single* function call (`f(x1)`) is required in each iteration since `f(x0)` becomes the "old" `f(x1)` and may simply be copied as `f_x0 = f_x1` (the exception is the very first iteration where two function evaluations are needed).

Running `secant_method.py`, gives the following printout on the screen:

```
Number of function calls: 19
A solution is: 3.000000
```

As with the function `Newton`, we place `secant` in the file `nonlinear_solvers.py` for easy import and use later.

## 6.4 The Bisection Method

Neither Newton's method nor the secant method can guarantee that an existing solution will be found (see Exercises 6.1 and 6.2). The bisection method, however, does that. However, if there are several solutions present, it finds only one of them, just as Newton's method and the secant method. The bisection method is slower than the other two methods, so reliability comes with a cost of speed.

To solve $x^2 - 9 = 0$, $x \in [0, 1000]$, with the bisection method, we reason as follows. The first key idea is that if $f(x) = x^2 - 9$ is *continuous* on the interval and the function values for the interval endpoints ($x_L = 0$, $x_R = 1000$) have *opposite signs*, $f(x)$ *must* cross the $x$ axis at least once on the interval. That is, we know there is at least one solution.

The second key idea comes from dividing the interval in two equal parts, one to the left and one to the right of the midpoint $x_M = 500$. By evaluating the sign of $f(x_M)$, we will immediately know whether a solution must exist to the left or right of $x_M$. This is so, since if $f(x_M) \geq 0$, we know that $f(x)$ has to cross the $x$ axis between $x_L$ and $x_M$ at least once (using the same argument as for the original interval). Likewise, if instead $f(x_M) \leq 0$, we know that $f(x)$ has to cross the $x$ axis between $x_M$ and $x_R$ at least once.

In any case, we may proceed with half the interval only. The exception is if $f(x_M) \approx 0$, in which case a solution is found. Such interval halving can be continued until a solution is found. A "solution" in this case, is when $|f(x_M)|$ is sufficiently close to zero, more precisely (as before): $|f(x_M)| < \epsilon$, where $\epsilon$ is a small number specified by the user.

The sketched strategy seems reasonable, so let us write a reusable function that can solve a general algebraic equation $f(x) = 0$ (`bisection_method.py`):

```
def bisection(f, x_L, x_R, eps, return_x_list=False):
    f_L = f(x_L)
    if f_L*f(x_R) > 0:
        print "Error! Function does not have opposite \
                 signs at interval endpoints!"
        sys.exit(1)
    x_M = float(x_L + x_R)/2.0
    f_M = f(x_M)
    iteration_counter = 1
```

```python
    if return_x_list:
        x_list = []

    while abs(f_M) > eps:
        if f_L*f_M > 0:   # i.e. same sign
            x_L = x_M
            f_L = f_M
        else:
            x_R = x_M
        x_M = float(x_L + x_R)/2
        f_M = f(x_M)
        iteration_counter += 1
        if return_x_list:
            x_list.append(x_M)
    if return_x_list:
        return x_list, iteration_counter
    else:
        return x_M, iteration_counter

def f(x):
    return x**2 - 9

a = 0;   b = 1000

solution, no_iterations = bisection(f, a, b, eps=1.0e-6)

print "Number of function calls: %d" % (1 + 2*no_iterations)
print "A solution is: %f" % (solution)
```

Note that we first check if $f$ changes sign in $[a, b]$, because that is a requirement for the algorithm to work. The algorithm also relies on a continuous $f(x)$ function, but this is very challenging for a computer code to check.

We get the following printout to the screen when `bisection_method.py` is run:

```
Number of function calls: 61
A solution is: 3.000000
```

We notice that the number of function calls is much higher than with the previous methods.

**Required work in the bisection method**

If the starting interval of the bisection method is bounded by $a$ and $b$, and the solution at step $n$ is taken to be the middle value, the error is bounded as

$$\frac{|b-a|}{2^n}, \tag{6.4}$$

because the initial interval has been halved $n$ times. Therefore, to meet a tolerance $\epsilon$, we need $n$ iterations such that the length of the current interval equals $\epsilon$:

$$\frac{|b-a|}{2^n} = \epsilon \quad \Rightarrow \quad n = \frac{\ln((b-a)/\epsilon)}{\ln 2}\,.$$

This is a great advantage of the bisection method: we know beforehand how many iterations $n$ it takes to meet a certain accuracy $\epsilon$ in the solution.

As with the two previous methods, the function `bisection` is placed in the file `nonlinear_solvers.py` for easy import and use.

## 6.5 Rate of Convergence

With the methods above, we noticed that the number of iterations or function calls could differ quite substantially. The number of iterations needed to find a solution is closely related to the *rate of convergence*, which dictates the speed of error reduction as we approach the root. More precisely, we introduce the error in iteration $n$ as $e_n = |x - x_n|$, and define the *convergence rate* $q$ as

$$e_{n+1} = Ce_n^q, \tag{6.5}$$

where $C$ is a constant. The exponent $q$ measures how fast the error is reduced from one iteration to the next. The larger $q$ is, the faster the error goes to zero, and the fewer iterations we need to meet the stopping criterion $|f(x)| < \epsilon$.

A single $q$ in (6.5) is defined in the limit $n \to \infty$. For finite $n$, and especially smaller $n$, $q$ will vary with $n$. To estimate $q$, we can compute all the errors $e_n$ and set up (6.5) for three consecutive experiments $n-1$, $n$, and $n+1$:

$$\begin{aligned} e_n &= Ce_{n-1}^q, \\ e_{n+1} &= Ce_n^q . \end{aligned}$$

Dividing these two equations by each other and solving with respect to $q$ gives

$$q = \frac{\ln(e_{n+1}/e_n)}{\ln(e_n/e_{n-1})} .$$

Since this $q$ will vary somewhat with $n$, we call it $q_n$. As $n$ grows, we expect $q_n$ to approach a limit ($q_n \to q$). To compute all the $q_n$ values, we need all the $x_n$ approximations. However, our previous implementations of Newton's method, the secant method, and the bisection method returned just the final approximation.

Therefore, we have extended the implementations in the module file `nonlinear_solvers.py` such that the user can choose whether the final value or the whole history of solutions is to be returned. Each of the extended implementations now takes an extra parameter `return_x_list`. This parameter is a boolean, set to `True` if the function is supposed to return all the root approximations, or `False`, if the function should only return the final approximation. As an example, let us take a closer look at `Newton`:

```python
def Newton(f, dfdx, x, eps, return_x_list=False):
    f_value = f(x)
    iteration_counter = 0
    if return_x_list:
        x_list = []
    while abs(f_value) > eps and iteration_counter < 100:
        try:
            x = x - float(f_value)/dfdx(x)
        except ZeroDivisionError:
            print "Error! - derivative zero for x = ", x
            sys.exit(1)     # Abort with error

        f_value = f(x)
        iteration_counter += 1
        if return_x_list:
            x_list.append(x)

    # Here, either a solution is found, or too many iterations
    if abs(f_value) > eps:
        iteration_counter = -1  # i.e., lack of convergence

    if return_x_list:
        return x_list, iteration_counter
    else:
        return x, iteration_counter
```

The function is found in the file `nonlinear_solvers.py`.
We can now make a call

```
x, iter = Newton(f, dfdx, x=1000, eps=1e-6, return_x_list=True)
```

and get a list `x` returned. With knowledge of the exact solution $x$ of $f(x) = 0$ we can compute all the errors $e_n$ and all the associated $q_n$ values with the compact function

```
def rate(x, x_exact):
    e = [abs(x_ - x_exact) for x_ in x]
    q = [log(e[n+1]/e[n])/log(e[n]/e[n-1])
         for n in range(1, len(e)-1, 1)]
    return q
```

The error model (6.5) works well for Newton's method and the secant method. For the bisection method, however, it works well in the beginning, but not when the solution is approached.
We can compute the rates $q_n$ and print them nicely,

```
def print_rates(method, x, x_exact):
    q = ['%.2f' % q_ for q_ in rate(x, x_exact)]
    print method + ':'
    for q_ in q:
        print q_,
    print
```

The result for `print_rates('Newton', x, 3)` is

```
Newton:
1.01 1.02 1.03 1.07 1.14 1.27 1.51 1.80 1.97 2.00
```

indicating that $q = 2$ is the rate for Newton's method. A similar computation using the secant method, gives the rates

```
secant:
1.26 0.93 1.05 1.01 1.04 1.05 1.08 1.13 1.20 1.30 1.43
1.54 1.60 1.62 1.62
```

Here it seems that $q \approx 1.6$ is the limit.

**Remark** If we in the bisection method think of the length of the current interval containing the solution as the error $e_n$, then (6.5) works perfectly since $e_{n+1} = \frac{1}{2}e_n$, i.e., $q = 1$ and $C = \frac{1}{2}$, but if $e_n$ is the true error $|x - x_n|$, it is easily seen from a sketch that this error can oscillate between the current interval length and a potentially very small value as we approach the exact solution. The corresponding rates $q_n$ fluctuate widely and are of no interest.

## 6.6 Solving Multiple Nonlinear Algebraic Equations

So far in this chapter, we have considered a single nonlinear algebraic equation. However, *systems* of such equations arise in a number of applications, foremost nonlinear ordinary and partial differential equations. Of the previous algorithms, only Newton's method is suitable for extension to systems of nonlinear equations.

### 6.6.1 Abstract Notation

Suppose we have $n$ nonlinear equations, written in the following abstract form:

$$F_0(x_0, x_1, \ldots, x_n) = 0, \tag{6.6}$$
$$F_1(x_0, x_1, \ldots, x_n) = 0, \tag{6.7}$$
$$\vdots = \vdots \tag{6.8}$$
$$F_n(x_0, x_1, \ldots, x_n) = 0\,. \tag{6.9}$$
$$\tag{6.10}$$

It will be convenient to introduce a *vector notation*

$$\boldsymbol{F} = (F_0, \ldots, F_1), \quad \boldsymbol{x} = (x_0, \ldots, x_n)\,.$$

The system can now be written as $\boldsymbol{F}(\boldsymbol{x}) = \boldsymbol{0}$.

As a specific example on the notation above, the system

$$x^2 = y - x\cos(\pi x) \tag{6.11}$$
$$yx + e^{-y} = x^{-1} \tag{6.12}$$

can be written in our abstract form by introducing $x_0 = x$ and $x_1 = y$. Then

$$F_0(x_0, x_1) = x^2 - y + x\cos(\pi x) = 0,$$
$$F_1(x_0, x_1) = yx + e^{-y} - x^{-1} = 0\,.$$

## 6.6.2 Taylor Expansions for Multi-Variable Functions

We follow the ideas of Newton's method for one equation in one variable: approximate the nonlinear $f$ by a linear function and find the root of that function. When $n$ variables are involved, we need to approximate a *vector function* $\boldsymbol{F}(\boldsymbol{x})$ by some linear function $\tilde{\boldsymbol{F}} = \boldsymbol{J}\boldsymbol{x} + \boldsymbol{c}$, where $\boldsymbol{J}$ is an $n \times n$ matrix and $\boldsymbol{c}$ is some vector of length $n$.

The technique for approximating $\boldsymbol{F}$ by a linear function is to use the first two terms in a Taylor series expansion. Given the value of $\boldsymbol{F}$ and its partial derivatives with respect to $\boldsymbol{x}$ at some point $\boldsymbol{x}_i$, we can approximate the value at some point $\boldsymbol{x}_{i+1}$ by the two first term in a Taylor series expansion around $\boldsymbol{x}_i$:

$$\boldsymbol{F}(\boldsymbol{x}_{i+1}) \approx \boldsymbol{F}(\boldsymbol{x}_i) + \nabla\boldsymbol{F}(\boldsymbol{x}_i)(\boldsymbol{x}_{i+1} - \boldsymbol{x}_i)\,.$$

The next terms in the expansions are omitted here and of size $||\boldsymbol{x}_{i+1} - \boldsymbol{x}_i||^2$, which are assumed to be small compared with the two terms above.

The expression $\nabla\boldsymbol{F}$ is the matrix of all the partial derivatives of $\boldsymbol{F}$. Component $(i, j)$ in $\nabla\boldsymbol{F}$ is

$$\frac{\partial F_i}{\partial x_j}\,.$$

For example, in our $2 \times 2$ system (6.11)-(6.12) we can use SymPy to compute the Jacobian:

```
>>> from sympy import *
>>> x0, x1 = symbols('x0 x1')
>>> F0 = x0**2 - x1 + x0*cos(pi*x0)
>>> F1 = x0*x1 + exp(-x1) - x0**(-1)
>>> diff(F0, x0)
-pi*x0*sin(pi*x0) + 2*x0 + cos(pi*x0)
>>> diff(F0, x1)
-1
>>> diff(F1, x0)
x1 + x0**(-2)
>>> diff(F1, x1)
x0 - exp(-x1)
```

We can then write

$$\nabla\boldsymbol{F} = \begin{pmatrix} \frac{\partial F_0}{\partial x_0} & \frac{\partial F_0}{\partial x_1} \\ \frac{\partial F_1}{\partial x_0} & \frac{\partial F_1}{\partial x_1} \end{pmatrix} = \begin{pmatrix} 2x_0 + \cos(\pi x_0) - \pi x_0 \sin(\pi x_0) & -1 \\ x_1 + x_0^{-2} & x_0 - e^{-x_1} \end{pmatrix}$$

The matrix $\nabla\boldsymbol{F}$ is called the *Jacobian* of $\boldsymbol{F}$ and often denoted by $\boldsymbol{J}$.

### 6.6.3 Newton's Method

The idea of Newton's method is that we have some approximation $\boldsymbol{x}_i$ to the root and seek a new (and hopefully better) approximation $\boldsymbol{x}_{i+1}$ by approximating $\boldsymbol{F}(\boldsymbol{x}_{i+1})$ by a linear function and solve the corresponding linear system of algebraic equations. We approximate the nonlinear problem $\boldsymbol{F}(\boldsymbol{x}_{i+1}) = 0$ by the linear problem

$$\boldsymbol{F}(\boldsymbol{x}_i) + \boldsymbol{J}(\boldsymbol{x}_i)(\boldsymbol{x}_{i+1} - \boldsymbol{x}_i) = \boldsymbol{0}, \tag{6.13}$$

where $\boldsymbol{J}(\boldsymbol{x}_i)$ is just another notation for $\nabla\boldsymbol{F}(\boldsymbol{x}_i)$. The equation (6.13) is a linear system with coefficient matrix $\boldsymbol{J}$ and right-hand side vector $\boldsymbol{F}(\boldsymbol{x}_i)$. We therefore write this system in the more familiar form

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

where we have introduce a symbol $\boldsymbol{\delta}$ for the unknown vector $\boldsymbol{x}_{i+1} - \boldsymbol{x}_i$ that multiplies the Jacobian $\boldsymbol{J}$.

The $i$-th iteration of Newton's method for systems of algebraic equations consists of two steps:

1. Solve the linear system $\boldsymbol{J}(\boldsymbol{x}_i)\boldsymbol{\delta} = -\boldsymbol{F}(\boldsymbol{x}_i)$ with respect to $\boldsymbol{\delta}$.
2. Set $\boldsymbol{x}_{i+1} = \boldsymbol{x}_i + \boldsymbol{\delta}$.

Solving systems of linear equations must make use of appropriate software. Gaussian elimination is the most common, and in general the most robust, method for this purpose. Python's `numpy` package has a module `linalg` that interfaces the well-known LAPACK package with high-quality and very well tested subroutines for linear algebra. The statement `x = numpy.linalg.solve(A, b)` solves a system $Ax = b$ with a LAPACK method based on Gaussian elimination.

When nonlinear systems of algebraic equations arise from discretization of partial differential equations, the Jacobian is very often sparse, i.e., most of its elements are zero. In such cases it is important to use algorithms that can take advantage of the many zeros. Gaussian elimination is then a slow method, and (much) faster methods are based on iterative techniques.

### 6.6.4 Implementation

Here is a very simple implementation of Newton's method for systems of nonlinear algebraic equations:

```python
import numpy as np

def Newton_system(F, J, x, eps):
    """
    Solve nonlinear system F=0 by Newton's method.
    J is the Jacobian of F. Both F and J must be functions of x.
    At input, x holds the start value. The iteration continues
    until ||F|| < eps.
    """
    F_value = F(x)
    F_norm = np.linalg.norm(F_value, ord=2)  # l2 norm of vector
    iteration_counter = 0
    while abs(F_norm) > eps and iteration_counter < 100:
        delta = np.linalg.solve(J(x), -F_value)
        x = x + delta
        F_value = F(x)
        F_norm = np.linalg.norm(F_value, ord=2)
        iteration_counter += 1

    # Here, either a solution is found, or too many iterations
    if abs(F_norm) > eps:
        iteration_counter = -1
    return x, iteration_counter
```

We can test the function `Newton_system` with the $2 \times 2$ system (6.11)-(6.12):

```
def test_Newton_system1():
    from numpy import cos, sin, pi, exp

    def F(x):
        return np.array(
            [x[0]**2 - x[1] + x[0]*cos(pi*x[0]),
             x[0]*x[1] + exp(-x[1]) - x[0]**(-1)])

    def J(x):
        return np.array(
            [[2*x[0] + cos(pi*x[0]) - pi*x[0]*sin(pi*x[0]), -1],
             [x[1] + x[0]**(-2), x[0] - exp(-x[1])]])

    expected = np.array([1, 0])
    tol = 1e-4
    x, n = Newton_system(F, J, x=np.array([2, -1]), eps=0.0001)
    print n, x
    error_norm = np.linalg.norm(expected - x, ord=2)
    assert error_norm < tol, 'norm of error =%g' % error_norm
    print 'norm of error =%g' % error_norm
```

Here, the testing is based on the L2 norm of the error vector. Alternatively, we could test against the values of `x` that the algorithm finds, with appropriate tolerances. For example, as chosen for the error norm, if `eps=0.0001`, a tolerance of $10^{-4}$ can be used for `x[0]` and `x[1]`.

## 6.7 Exercises

**Exercise 6.1: Understand why Newton's method can fail**
The purpose of this exercise is to understand when Newton's method works and fails. To this end, solve $\tanh x = 0$ by Newton's method and study the intermediate details of the algorithm. Start with $x_0 = 1.08$. Plot the tangent in each iteration of Newton's method. Then repeat the calculations and the plotting when $x_0 = 1.09$. Explain what you observe.
Filename: `Newton_failure.*`.

**Exercise 6.2: See if the secant method fails**
Does the secant method behave better than Newton's method in the problem described in Exercise 6.1? Try the initial guesses

1. $x_0 = 1.08$ and $x_1 = 1.09$
2. $x_0 = 1.09$ and $x_1 = 1.1$
3. $x_0 = 1$ and $x_1 = 2.3$
4. $x_0 = 1$ and $x_1 = 2.4$

Filename: `secant_failure.*`.

**Exercise 6.3: Understand why the bisection method cannot fail**
Solve the same problem as in Exercise 6.1, using the bisection method, but let the initial interval be $[-5, 3]$. Report how the interval containing the solution evolves during the iterations.
Filename: `bisection_nonfailure.*`.

**Exercise 6.4: Combine the bisection method with Newton's method**
An attractive idea is to combine the reliability of the bisection method with the speed of Newton's method. Such a combination is implemented by running the bisection method until we have a narrow interval, and then switch to Newton's method for speed.

Write a function that implements this idea. Start with an interval $[a, b]$ and switch to Newton's method when the current interval in the bisection method is a fraction $s$ of the initial interval (i.e., when the interval has length $s(b - a)$). Potential divergence of Newton's method is still an issue, so if the approximate root jumps out of the narrowed interval (where the solution is known to lie), one can switch back to the bisection method. The value of $s$ must be given as an argument to the function, but it may have a default value of 0.1.

Try the new method on $\tanh(x) = 0$ with an initial interval $[-10, 15]$.
Filename: `bisection_Newton.py`.

**Exercise 6.5: Write a test function for Newton's method**
The purpose of this function is to verify the implementation of Newton's method in the `Newton` function in the file `nonlinear_solvers.py`. Construct an algebraic equation and perform two iterations of Newton's method by hand or with the aid of SymPy. Find the corresponding size of $|f(x)|$ and use this as value for `eps` when calling `Newton`. The function should then also perform two iterations and return the same approximation to the root as you calculated manually. Implement this idea for a unit test as a test function `test_Newton()`.
Filename: `test_Newton.py`.

**Exercise 6.6: Solve nonlinear equation for a vibrating beam**
An important engineering problem that arises in a lot of applications is the vibrations of a clamped beam where the other end is free. This problem can be analyzed analytically, but the calculations boil down to solving the following nonlinear algebraic equation:

$$\cosh \beta \cos \beta = -1,$$

where $\beta$ is related to important beam parameters through

$$\beta^4 = \omega^2 \frac{\varrho A}{EI},$$

where $\varrho$ is the density of the beam, $A$ is the area of the cross section, $E$ is Young's modulus, and $I$ is the moment of the inertia of the cross section. The most important parameter of interest is $\omega$, which is the frequency of the beam. We want to compute the frequencies of a vibrating steel beam with a rectangular cross section having width $b = 25$ mm and height $h = 8$ mm. The density of steel is 7850 kg/m$^3$, and $E = 2\cdot 10^{11}$ Pa. The moment of inertia of a rectangular cross section is $I = bh^3/12$.

a) Plot the equation to be solved so that one can inspect where the zero crossings occur.

*Hint* When writing the equation as $f(\beta) = 0$, the $f$ function increases its amplitude dramatically with $\beta$. It is therefore wise to look at an equation with damped amplitude, $g(\beta) = e^{-\beta} f(\beta) = 0$. Plot $g$ instead.

b) Compute the first three frequencies.

Filename: `beam_vib.py`.