App. A: Sequences and difference equations

Joakim Sundnes[1,2]

[1]Simula Research Laboratory
[2]University of Oslo, Dept. of Informatics

Sep 20, 2019

Sequences is a central topic in mathematics, which has important applications in numerical analysis and scientific computing. For some sequences we can derive a formula that gives the the n-th number x_n as a function of n. However, in many cases it is impossible to derive such a formula, and x_n is instead given by a relation involving x_{n-1} and possibly x_{n-2}. Such relations are called *difference equations*, and we shall see that they are very suitable for solving on a computer. The tools for solving such equations are mainly loops and arrays, which we already know from previous chapters.

1 Sequences in mathematics

In the most general sense, a sequence is simply a collection of numbers:

$$x_0,\ x_1,\ x_2,\ \ldots,\ x_n, \ldots.$$

For instance, all the odd numbers form a sequence

$$1, 3, 5, 7, \ldots$$

and for this sequence we can write a simple formula for the n-th term;

$$x_n = 2n + 1.$$

With this formula at hand, the complete sequence can be written on a compact form;

$$(x_n)_{n=0}^{\infty}, \quad x_n = 2n + 1.$$

Other examples of sequences include

$$1,\ 4,\ 9,\ 16,\ 25,\ \ldots \quad (x_n)_{n=0}^{\infty},\ x_n = n^2,$$
$$1,\ \frac{1}{2},\ \frac{1}{3},\ \frac{1}{4},\ \ldots \quad (x_n)_{n=0}^{\infty},\ x_n = \frac{1}{n+1},$$
$$1,\ 1,\ 2,\ 6,\ 24,\ \ldots \quad (x_n)_{n=0}^{\infty},\ x_n = n!,$$
$$1,\ 1+x,\ 1+x+\frac{1}{2}x^2,\ 1+x+\frac{1}{2}x^2+\frac{1}{6}x^3,\ \ldots \quad (x_n)_{n=0}^{\infty},\ x_n = \sum_{j=0}^{n}\frac{x^j}{j!}.$$

These are all formulated as inifite sequences, which is common in mathematics, but in real-life applications sequences are usually finite: $(x_n)_{n=0}^{N}$. Some familiar examples include the number of approved exersizes every week in IN1900, or the annual value of a loan.

1.1 Difference equations

For all the sequences above we have an explicit formula for the n-th term, but this is not always the case. Many sequences can instead be formulated in terms of a *difference equation*, a relation relating one term in the sequence to one or more of the previous terms. Such equations can be challenging to solve with analytical methods, since in order to compute the n-th term of a sequence we need to compute the entire sequence $x_0, x_1, \ldots, x_{n-1}$. This can be very tedious to do by hand or using a calculator, but on a computer such a problem is easy to solve with a for-loop. Combining sequences and difference equations with programming therefore enables us to consider far more interesting and useful cases.

To start with a simple example, consider the problem of computing how an invested sum of money grows over time. In its simplest form, this problem can be written as putting x_0 money in a bank at year 0, with interest rate p percent per year. What is then the value after N years? You may recall from high school mathematics that the solution to this problem is given by the simple formula $x_n = x_0(1 + p/100)^n$, so there is really no need to formulate and solve the problem as a difference equation. However, very simple problem generalizations, such as a non-constant interest rate, makes the formula difficult to apply, but a difference equation is still applicable. To formulate the problem as a difference equation, the relation between the value at year n, x_n, and the value x_{n-1} for the previous year is given by

$$x_n = x_{n-1} + \frac{p}{100}x_{n-1}$$

If we want to compute x_n, we can now simply start with the known x_0, and compute $x_1, x_2, \ldots, x_n$. The procedure involves repeating a simple calculation many times, which is tedious to do by hand but very well suited for a computer. The complete program for solving this difference equation may look like:

```
import numpy as np
import matplotlib.pyplot as plt
```

```
x0 = 100                        # initial amount
p = 5                           # interest rate
N = 4                           # number of years
index_set = range(N+1)
x = np.zeros(len(index_set))

x[0] = x0
for n in index_set[1:]:
    x[n] = x[n-1] + (p/100.0)*x[n-1]

plt.plot(index_set, x, 'ro')
plt.xlabel('years')
plt.ylabel('amount')
plt.show()
```

This code only involves tools that we already know. Notice that we store the result of the call to `range` in a variable `index_set`, which will be a list-like object of numbers from 0 to N (representing the years). We could also have called `range` directly everywhere we need it, but since we use it in three places in the code it is convenient to use a variable. The core of the program is the three lines starting with `x[0] = x0`. We here initialize the first element in our solution array with the known `x0`, and then step into the for-loop to compute the rest. The loop variable `n` runs from 1 to N, and the formula inside the loop computes `x[n]` from the known `x[n-1]`. An alternative formulation of the for-loop would be

```
for n in index_set[:-1]:
    x[n+1] = x[n] + (p/100.0)*x[n]
```

Here `n` runs from 0 to 3, and it is easy to verify that the two loops give the same result. However, mixing up these two formulations will easily lead to a loop that runs out of bounds (an `IndexError`) or a loop where some elements are never computed. Such mistakes are probably the most common type of programming error when solving difference equations, and it is a good idea to always examine the loop carefully. If an `IndexError` (or another suspected loop error) occurs, a good debugging strategy is to check the loop definition for the lower and upper value of the loop variable (here `n`), and insert both into the formulas inside the loop to check that they make sense. As an example, consider the deliberately wrong code

```
for n in index_set[1:]:
    x[n+1] = x[n] + (p/100.0)*x[n]
```

Here we see that `n` runs from 1 to 4, and if we insert these bounds into the formula we find that the first pass of the loop will try to compute `x[2]` from `x[1]`. However, we have only initialized `x[0]` so `x[1]` is zero, and therefore `x[2]` and all subsequent values will be set to zero. If we insert the upper bound `n=4` we see that we try to access `x[5]`, but this does not exist and we get an `IndexError`. Performing such simple analysis of a for-loop is often a good way to reveal the source of the error and give an idea of how it can be fixed.

Solving a difference equation without using arrays. The program above stored the sequence as an array, which is a convenient way to program the solver

and enables us to plot the entire sequence. However, if we are only interested in the solution at a single point, i.e., x_n, there is no need to store the entire sequence. Since each x_n only depends on the previous value x_{n-1}, we only need to store the last two values in memory. A complete loop can look like this:

```
x_old = x0
for n in index_set[1:]:
    x_new = x_old + (p/100.)*x_old
    x_old = x_new  # x_new becomes x_old at next step
```

For this simple case we can actually make the code even shorter, since we only use `x_old` in a single line and we can just as well overwrite it once it has been used:

```
x = x0                          #x is here a single number, not array
for n in index_set[1:]:
    x = x + (p/100.)*x
```

We see that these codes store just one or two numbers, and for each pass through the loop we simply update these and overwrite the values we no longer need. Although this approach is quite simple, and we obviously save some memory since we do not store the complete array, programming with an array `x[n]` is usually safer, and we are often interested in plotting the entire sequence. We will therefore mostly use arrays in the subsequent examples.

Extending the solver for the growth of money. Say we are interested in changing our model for interest rate, to a model where the interest is added every day instead of every year. The interest rate per day is $r = p/D$ if p is the annual interest rate and D is the number of days in a year. A common model in business applies $D = 360$, but n counts exact (all) days. The difference equation relating one day's amount to the previous is the same as above

$$x_n = x_{n-1} + \frac{r}{100}x_{n-1},$$

except that the yearly interest rate has been replaced by the daily (r). If we are interested in how much the money grows from a given date to another we also need to find the number of days between those dates. This calculation could of course be done by hand, but Python has a convenient module named `datetime` for this purpose. The following session illustrates how it can be used:

```
>>> import datetime
>>> date1 = datetime.date(2017, 9, 29)  # Sep 29, 2017
>>> date2 = datetime.date(2018, 8, 4)  # Aug 4, 2018
>>> diff = date2 - date1
>>> print diff.days
309
```

Putting these tools together, a complete program for daily interest rates may look like

```
import numpy as np
import matplotlib.pyplot as plt
```

```
import datetime

x0 = 100                              # initial amount
p = 5                                 # annual interest rate
r = p/360.0                           # daily interest rate

date1 = datetime.date(2017, 9, 29)
date2 = datetime.date(2018, 8, 4)
diff = date2 - date1
N = diff.days
index_set = range(N+1)
x = np.zeros(len(index_set))

x[0] = x0
for n in index_set[1:]:
    x[n] = x[n-1] + (r/100.0)*x[n-1]

plt.plot(index_set, x, 'ro')
plt.xlabel('days')
plt.ylabel('amount')
plt.show()
```

This program is slightly more sophisticated than the first one, but one may still argue that solving this problem with a difference equation is unnecessarily complex, since we could just apply the well-known formula $x_n = x_0(1 + \frac{r}{100})^n$ to compute any x_n we want. However, we know that the interest rate changes quite often. In this case the formula is no longer valid, but our difference equation can still be used, and only minor changes are needed in the program. The simplest approach is to let `p` be an array of the same length as the number of days, and fill it with the correct interest rates for each day. The modifications to the program above may look like this:

```
p = np.zeros(len(index_set))
# fill p[n] for n in index_set (might be non-trivial...)

r = p/360.0                           # daily interest rate
x = np.zeros(len(index_set))

x[0] = x0
for n in index_set[1:]:
    x[n] = x[n-1] + (r[n-1]/100.0)*x[n-1]
```

The only real difference here is that we initialize `p` as an array, and then `r = p/360.0` becomes an array of the same length. In the formula inside the for-loop we then look up the correct value `r[n-1]` for each iteration of the loop. As noted, filling `p` with the correct values can be non-trivial, but many cases can be handled quite easily. For instance, say the interest rate is piecewise constant and increases from 4.0% to 5.0% on a given date. Code for filling the array with values may then look like this

```
date0 = datetime.date(2017, 9, 29)
date1 = datetime.date(2018, 2, 6)
date2 = datetime.date(2018, 8, 4)
Np = (date1-date0).days
N = (date2-date0).days
```

```
p = np.zeros(len(index_set))
p[:Np] = 4.0
p[Np:] = 5.0
```

1.2 A system of two coupled difference equations

To consider a related example to the one above, assume that we have a fortune F invested with an annual interest rate of p percent. Every year we plan to consume an amount c_n, where n counts years, and we want to compute our fortune x_n at year n. The problem can be formulated as a difference equation, where the fundamental relation from one year to the next is

$$x_n = x_{n-1} + \frac{p}{100}x_{n-1} - c_{n-1}$$

In the simplest case c_n is constant, but inflation demands c_n to increase. To solve this problem, we assume that c_n should grow with a rate of I percent per year, and in the first year we want to consume q percent of first year's interest. The extension of the difference equation above becomes

$$x_n = x_{n-1} + \frac{p}{100}x_{n-1} - c_{n-1}, c_n \qquad\qquad = c_{n-1} + \frac{I}{100}c_{n-1}.$$

with initial conditions $x_0 = F$ and $c_0 = (pF/100)(q/100) = \frac{pFq}{10000}$. This is a coupled system of *two* difference equations, but the programming is not much more difficult than the single equation above. We simply create two arrays, `x` and `c`, initialize both `x[0]` and `c[0]` to the given initial conditions, and then update `x[n]` and `c[n]` inside the loop. A complete code may look like

```
import numpy as np
import matplotlib.pyplot as plt
F = 1e7                        # initial amount
p = 5                           # interest rate
I = 3
q = 75
N = 40                           # number of years
index_set = range(N+1)
x = np.zeros(len(index_set))
c = np.zeros_like(x)

x[0] = F
c[0] = q*p*F*1e-4

for n in index_set[1:]:
    x[n] = x[n-1] + (p/100.0)*x[n-1] - c[n-1]
    c[n] = c[n-1] + (I/100.0)*c[n-1]

plt.plot(index_set, x, 'ro',label = 'Fortune')
plt.plot(index_set, c, 'go', label = 'Yearly consume')
plt.xlabel('years')
plt.ylabel('amounts')
plt.legend()
plt.show()
```

1.3 More examples of difference equations

As noted above, sequences and difference have countless applications in mathematics, science, and engineering. Here we present a selection of well known examples.

Fibonacci numbers as a difference equation. No programming or math course is complete without an example on Fibonacci numbers:

$$x_n = x_{n-1} + x_{n-2}, \quad x_0 = 1,\ x_1 = 1$$

This sequence was originally derived by Fibnacci for modeling rat populations, but it has a range of peculiar mathematical properties and has therefore attracted much attention from mathematicians. The equation differs from the ones above since x_n depends on the two previous values $(n-1,\ n-2)$, which makes this a *second order difference equation*. This classification is important for mathematical solution techniques, but in a program there is not much difference between first and second order equations. The complete code to solve this difference equation and generate the Fibonacci numbers can be written as

```
import sys
from numpy import zeros

N = int(sys.argv[1])
x = zeros(N+1, int)
x[0] = 1
x[1] = 1
for n in range(2, N+1):
    x[n] = x[n-1] + x[n-2]
    print(n, x[n])
```

The Fibonacci numbers grow quickly and running this program for large N (try for instance $N = 100$) will lead to problems with overflow. The NumPy `int` type supports up to 9223372036854775807 (almost 10^{19}, so it is very rarely a problem). We can fix the problem by avoiding NumPy arrays and instead use the standard Python `int` type, but we will not go into these details here.

Logistic growth. If we return to the initial problem of calculating growth of money in a bank, the solution formula for this problem can be written on the form

$$x_n = x_0 C^n \quad (= x_0 e^{n \ln C})$$

Since n counts years, this is an example of exponential growth in time, with the general formula $x = x_0 e^{\lambda t}$. Populations of humans, animals, and cells also exhibit the same type of growth when there are unlimited resources (space and food), and the model for exponential growth therefore has many applications in biology.[1] However, most environments can only support a finite number M of

[1] The formula $x = x_0 e^{\lambda t}$ is the solution of the *differential equation* $dx/dt = \lambda x$, and this formulation may be more familiar to some readers. Differential equations and difference equations are closely related, and these relations are explored in more detail in Chapter 9.

individuals, while in the exponential growth model the population continues to grow. How can we alter the equation to be a more realistic model for growing populations?

Initially, when resources are abundant, growth is exponential:

$$x_n = x_{n-1} + \frac{r}{100} x_{n-1}$$

To enforce the limit M, the growth rate r must decay to zero as x_n approaches M. The simplest variation of $r(n)$ is linear:

$$r(n) = \varrho \left(1 - \frac{x_n}{M}\right)$$

We observe: $r(n) \approx \varrho$ for small n, when $x_n \ll M$, and $r(n) \to 0$ as n grows and $x_n \to M$. This formulation of r leads to the logistic growth model:

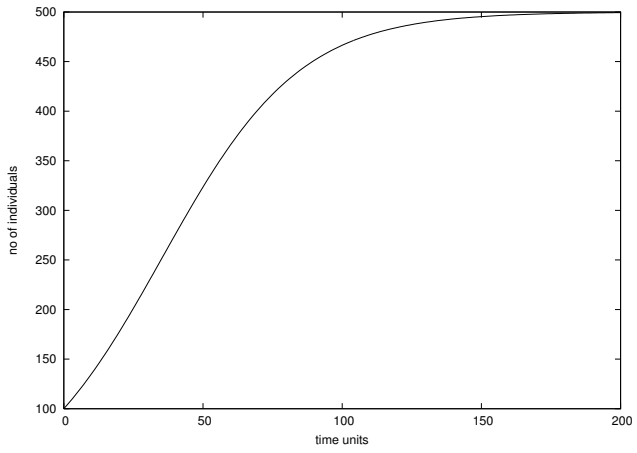
$$x_n = x_{n-1} + \frac{\varrho}{100} x_{n-1} \left(1 - \frac{x_{n-1}}{M}\right).$$

This is a *nonlinear* difference equation, while all the examples considered above were linear. This distinction is important in mathematical analysis of the equations, but it does not make much difference for solving the equation in a program. To change the program above from exponential to logistic growt, we can simply replace

```
x[n] = x[n-1] + p/100.0)*x[n-1]
```

by

```
x[n] = x[n-1] + (rho/100.0)*x[n-1]*(1 - x[n-1]/float(M))
```



The factorial as a difference equation. The factorial $n!$ is defined as

$$n(n-1)(n-2)\cdots 1, \quad 0! = 1$$

The following difference equation has $x_n = n!$ as solution and can be used to compute the factorial:

$$x_n = nx_{n-1}, \quad x_0 = 1$$

As above, we may ask the question of whether such a difference equation is very useful, when we have an explicit formula for computing any x_n we want. One answer to this question is that in many applications we actually need to compute the entire sequence of factorials $x_n = n!$ for $n = 0, \ldots N$. We could apply the standard formula to compute these, but it involves a lot of redundant computations, since we perform n multiplications for each x_n. When solving the difference equation, each new x_n requires a single multiplication, and for large values of n this may speed up the program considerably.

1.4 Sequences and series as approximations

Have you ever though about how $\sin x, \ln x$, or e^x is really calculated? These functions have been defined to have some desired mathematical properties, but we need some kind of algorithm to evaluate the function values. A convenient approach will is to approximate $\sin x$, etc. by polynomials, since they are easy to calculate. It turns out that such approximations exist, for example this result by Grerory in 1667:

$$\sin x = \sum_{k=0}^{\infty} (-1)^k \frac{x^{2k+1}}{(2k+1)!}$$

and an even more amazing result by Taylor in 1715:

$$f(x) = \sum_{k=0}^{\infty} \frac{1}{k!} (\frac{d^k}{dx^k} f(0)) x^k.$$

The latter result means that for any function $f(x)$, if we can compute the function value and its derivatives for $x = 0$, we can approximate the function value at any x by evaluating a polynomial. For practical applications, we work with a truncated sum

$$f(x) \approx \sum_{k=0}^{N} \frac{1}{k!} (\frac{d^k}{dx^k} f(0)) x^k.$$

The most popular choice is actually $N = 1$ which has been essential in developing physics and technology. As an example, consider these approximations to the exponential function

$$\begin{aligned} e^x &= \sum_{k=0}^{\infty} \frac{x^k}{k!} \\ &\approx 1 + x + \frac{1}{2}x^2 + \frac{1}{6}x^3 \\ &\approx 1 + x. \end{aligned}$$

These are obviously not very good approximations for large x, but around $x = 0$ they may be sufficiently accurate for many applications. We can also make the Taylor polynomials accurate around any point $x = a$:

$$f(x) \approx \sum_{k=0}^{N} \frac{1}{k!}(\frac{d^k}{dx^k} f(a))(x-a)^k.$$

Taylor polynomial formulated as a difference equation. The Taylor series for e^x around $x = 0$ is given by

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}.$$

If we now define

$$e_n = \sum_{k=0}^{n-1} \frac{x^k}{k!} = \sum_{k=0}^{n-2} \frac{x^k}{k!} + \frac{x^{n-1}}{(n-1)!}$$

we can formulate the sum in e_n as the difference equation

$$e_n = e_{n-1} + \frac{x^{n-1}}{(n-1)!}, \quad e_0 = 0.$$

We see that this difference equation involves $(n-1)!$, and above we introduced a difference equation for the factorial. We can utilize this idea to formulate a more efficient computation of the Taylor polynomial. Observe that

$$\frac{x^n}{n!} = \frac{x^{n-1}}{(n-1)!} \cdot \frac{x}{n},$$

and if we let $a_n = x^n/n!$ it can be computed efficiently by solving

$$a_n = a_{n-1}\frac{x}{n}, \quad a_0 = 1.$$

Now we can formulate a system of two difference equations for the Taylor polynomial, where we update each term via the a_n equation and sum the terms via the e_n equation:

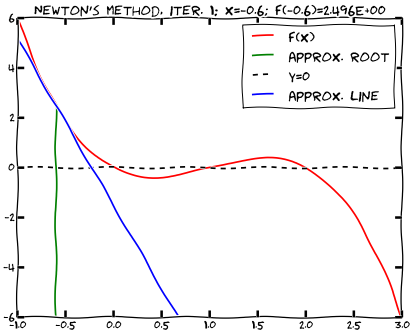
$$\begin{aligned} e_n &= e_{n-1} + a_{n-1}, \quad e_0 = 0,\ a_0 = 1 \\ a_n &= \frac{x}{n} a_{n-1}. \end{aligned}$$

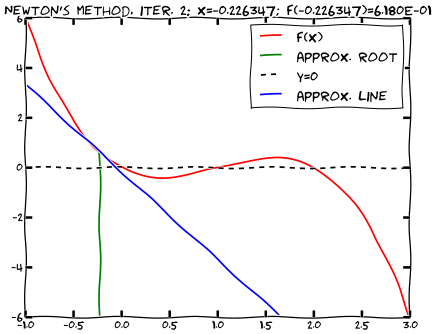
Although we are here solving a system of two difference equations, the computation is far more efficient than the single equation above, since we avoid the repeated multiplications involved in the factorial computation.

Difference equations for solving nonlinear algebraic equations. We often need to solve algbraic equations on the form

$$f(x) = 0$$

We have simple formulas for solving some such equations, for instance for $f(x) = ax + b$ (linear), or $f(x) = ax^2 + bx + c$ (quadratic). Other special cases such as $\sin x + \cos x = 1$ can also be solved analytically, but general nonlinear equations cannot be solved by hand and need to be approximated. Well known numerical methods to solve such equations include the robust but slow bisection method, and the usually much faster alternatives Newton's method and the secant method. Both Newton's method and the secant method are examples of difference equations!





Simpson (1740) came up with the following general method for solving $f(x) = 0$, based on ideas by Newton. Starting from some initial guess x_0, gradually improve the approximation by iterations

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

This is a nonlinear first-order difference equation. As $n \to \infty$, we hope that $x_n \to x_s$, where x_s is the solution to $f(x_s) = 0$. But of course we don't really want to let n approach infinity, so how do we choose N so that x_N is sufficiently close to x_s? Since we want to solve $f(x) = 0$, the best approach is to solve the equation until $f(x) \leq \epsilon$, where ϵ is a small tolerance. In practice, Newton's method will usually converge rather quickly, or not converge at all, so setting some upper bound on the number of iterations is a good idea. A simple program implementing Newton's method may look like

```
def Newton(f, dfdx, x, epsilon=1.0E-7, max_n=100):
    n = 0
    while abs(f(x)) > epsilon and n <= max_n:
        x = x - f(x)/dfdx(x)
        n += 1
    return x, n, f(x)
```

The arguments to the function are Python functions `f` and `dfdx` implementing $f(x)$ and its derivative. Both of these arguments are called inside the function and both must therefore be callable. The `x` argument is the initial guess for the solution x, and the two optional arguments at the end are the tolerance and the maximum number of iteration. The method itself is implemented as a standard while-loop. Note that this is a very simple implementation, and by no means the most elegant or efficient. For instance we evaluate $f(x)$ twice for each pass of the loop, while only one evaluation is strictly necessary, since we could store the value in a variable. It could be useful to store the `x` and `f(x)` values in each iteration in arrays, instead of simply overwriting `x` with the new solution, to be able to plot or analyze the convergence of the method.

An improved implementation of Newton's method may look like this

```
def Newton(f, x, dfdx, epsilon=1.0E-7, max_n=100,
           store=False):
    f_value = f(x)
    n = 0
    if store: info = [(x, f_value)]
    while abs(f_value) > epsilon and n <= max_n:
        x = x - f_value/dfdx(x)
        n += 1
        f_value = f(x)
        if store: info.append((x, f_value))
    if store:
        return x, info
    else:
        return x, n, f_value
```

Here we only do one $f(x)$ call for each iteration and we have included the option of storing $(x, f(x))$ values during the iterations.

How good is Newton's method? A large family of methods have been derived as modifications of the original Newton's method, and methods of this kind are the standard tools to use for solving non-linear algebraic equations. However, the methods are not without faults, and it is important to be aware of their limitations. As a general rule, Newton's method is fast but not very robust. It usually converges quickly to the correct solution if the initial guess is good, but will often diverge if the initial guess is far from the true solution. We can illustrate the behavior by applying the code above to solve the non-linear equation

$$e^{-0.1x^2}\sin(\frac{\pi}{2}x) = 0.$$

This equation has an infinite number of solutions, including $x = 0, \pm 2, \pm 4, \pm 6, \ldots$, since $\sin(n\pi) = 0$ for all integers n. A program for solving this equation, using the `Newton` function shown above, may look like this:

```
from math import sin, cos, exp, pi
import sys

def g(x):
    return exp(-0.1*x**2)*sin(pi/2*x)
```

```
def dg(x):
    return -2*0.1*x*exp(-0.1*x**2)*sin(pi/2*x) + \
           pi/2*exp(-0.1*x**2)*cos(pi/2*x)

x0 = float(sys.argv[1])
x, info = Newton(g, x0, dg, store=True)
print('Computed root:', x)

# Print the evolution of the difference equation
# (i.e., the search for the root)
for i in range(len(info)):
    print(f'Iteration {i:3d}: f({info[i][0]})={info[i][1]}')
```

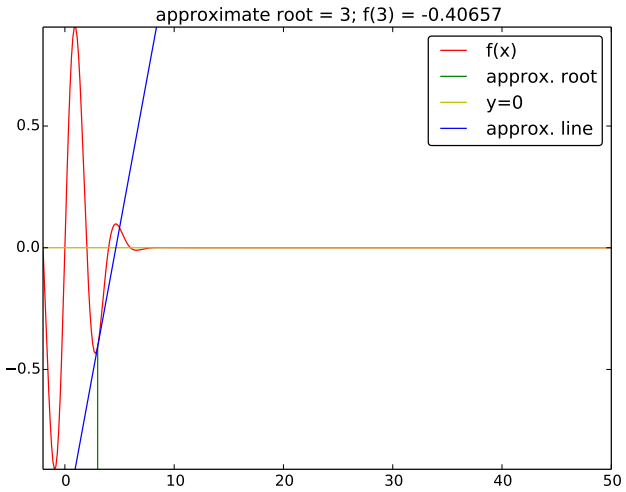
We can run this program with different starting values for x to see how the method works. Choosing $x_0 = 1.7$ gives quick convergence towards the closest root $x = 2$

```
zero: 1.999999999768449
Iteration   0: f(1.7)=0.340044
Iteration   1: f(1.99215)=0.00828786
Iteration   2: f(1.99998)=2.53347e-05
Iteration   3: f(2)=2.43808e-10
```

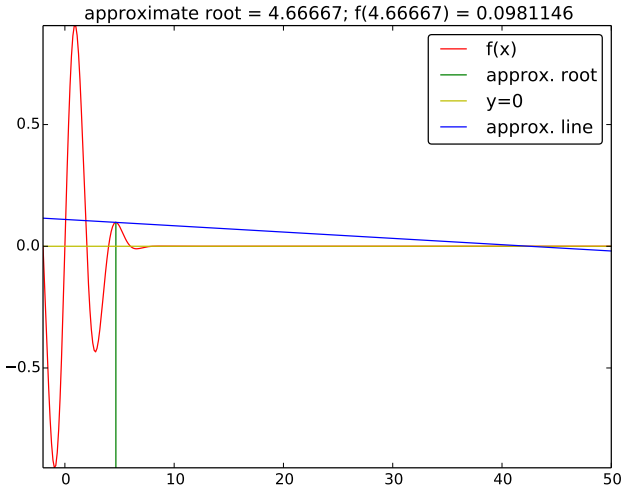
Next, we try $x_0 = 3$, for which the closest roots are $x = 2$ and $x = 4$:

```
zero: 42.49723316011362
Iteration   0: f(3)=-0.40657
Iteration   1: f(4.66667)=0.0981146
Iteration   2: f(42.4972)=-2.59037e-79
```

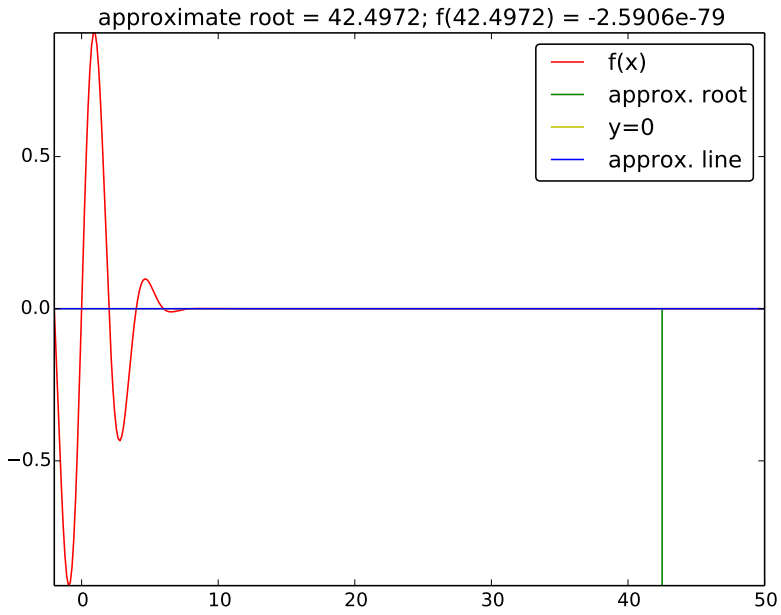
Although we see from the final value of $f(x)$ that the equation is in fact "solved", we did not find the expected root. We can analyze the steps in more detail to get an idea of what went wrong. The method is based on a local *linearization* of the non-linear equation. One way to view it in each step we use take the derivative in the point we are at, and use this to construct a linear equation $ax + b = 0$. Then we solve this equation and hope it improves our approximate solution of the original equation. We can visualize each step to see how the solution evolved in this latter case. Starting at $x = 3$, we step to the right, to where the blue line crosses the x-axis in this figure:



We see that this is a a bit to the right of the root we are looking for ($x = 4$), and $f(x)$ is clearly not zero in this point, so we do another step:



This step brings us to $x = 42.4972$, and we are done since $f(x) \approx 0$ at this point.



The main lesson learnt from this is that Newton's method does not always work, and to avoid potential serious errors it is important to understand the method.

1.5 Summary of difference equations

- A sequence where x_n is expressed by x_{n-1}, x_{n-2} etc is a difference equation
- In general no explicit formula for x_n, so hard to solve on paper for large n
- Easy to solve in Python:
  - Start with x_0
  - Compute x_n from x_{n-1} i a for loop
- Easily extended to systems of difference equations
  - Just update all the sequences in the same for loop