

From mathematics you probably know the concept of a *sequence*, which is nothing but a collection of numbers with a specific order. A general sequence is written as

$$x_0, x_1, x_2, \dots, x_n, \dots,$$

One example is the sequence of all odd numbers:

$$1, 3, 5, 7, \dots, 2n + 1, \dots$$

For this sequence we have an explicit formula for the n -th term: $2n + 1$, and n takes on the values $0, 1, 2, \dots$. We can write this sequence more compactly as $(x_n)_{n=0}^{\infty}$ with $x_n = 2n + 1$. Other examples of infinite sequences from mathematics are

$$1, 4, 9, 16, 25, \dots \quad (x_n)_{n=0}^{\infty}, \quad x_n = (n + 1)^2, \quad (5.1)$$

$$1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \dots \quad (x_n)_{n=0}^{\infty}, \quad x_n = \frac{1}{n + 1}. \quad (5.2)$$

The former sequences are infinite, because they are generated from all integers ≥ 0 and there are infinitely many such integers. Nevertheless, most sequences from real life applications are finite. If you put an amount x_0 of money in a bank, you will get an interest rate and therefore have an amount x_1 after one year, x_2 after two years, and x_N after N years. This process results in a finite sequence of amounts

$$x_0, x_1, x_2, \dots, x_N, \quad (x_n)_{n=0}^N.$$

Usually we are interested in quite small N values (typically $N \leq 20 - 30$). Anyway, the life of the bank is finite, so the sequence definitely has an end.

For some sequences it is not so easy to set up a general formula for the n -th term. Instead, it is easier to express a relation between two or more consecutive elements. One example where we can do both things is the sequence of odd numbers. This sequence can alternatively be generated by the formula

$$x_{n+1} = x_n + 2. \quad (5.3)$$

To start the sequence, we need an *initial condition* where the value of the first element is specified:

$$x_0 = 1.$$

Relations like (5.3) between consecutive elements in a sequence is called recurrence relations or *difference equations*. Solving a difference equation can be quite challenging in mathematics, but it is almost trivial to solve it on a computer. That is why difference equations are so well suited for computer programming, and the present chapter is devoted to this topic.

The program examples regarding difference equations are found in the folder `src/diffeq`.

5.1 Mathematical Models Based on Difference Equations

The objective of science is to understand complex phenomena. The phenomenon under consideration may be a part of nature, a group of social individuals, the traffic situation in Los Angeles, and so forth. The reason for addressing something in a scientific manner is that it appears to be complex and hard to comprehend. A common scientific approach to gain understanding is to create a model of the phenomenon, and discuss the properties of the model instead of the phenomenon. The basic idea is that the model is easier to understand, but still complex enough to preserve the basic features of the problem at hand¹. Modeling is, indeed, a general idea with applications far beyond science. Suppose, for instance, that you want to invite a friend to your home for the first time. To assist your friend, you may send a map of your neighborhood. Such a map is a model: It exposes the most important landmarks and leave out billions of details that your friend can do very well without. This is the essence of modeling: A good model should be as simple as possible, but still rich enough to include the important structures you are looking for².

¹ “Essentially, all models are wrong, but some are useful.” –George E. P. Box, statistician, 1919-.

² “Everything should be made as simple as possible, but not simpler.” –Albert Einstein, physicist, 1879-1955.

Certainly, the tools we apply to model a certain phenomenon differ a lot in various scientific disciplines. In the natural sciences, mathematics has gained a unique position as the key tool for formulating models. To establish a model, you need to understand the problem at hand and describe it with mathematics. Usually, this process results in a set of equations, i.e., the model consists of equations that must be solved in order to see how realistically the model describes a phenomenon. Difference equations represent one of the simplest yet most effective type of equations arising in mathematical models. The mathematics is simple and the programming is simple, thereby allowing us to focus more on the modeling part. Below we will derive and solve difference equations for diverse applications.

5.1.1 Interest Rates

Our first difference equation model concerns how much money an initial amount x_0 will grow to after n years in a bank with annual interest rate p . You learned in school the formula

$$x_n = x_0 \left(1 + \frac{p}{100}\right)^n. \quad (5.4)$$

Unfortunately, this formula arises after some limiting assumptions, like that of a constant interest rate over all the n years. Moreover, the formula only gives us the amount after each year, not after some months or days. It is much easier to compute with interest rates if we set up a more fundamental model in terms of a difference equation and then solve this equation on a computer.

The fundamental model for interest rates is that an amount x_{n-1} at some point of time t_{n-1} increases its value with p percent to an amount x_n at a new point of time t_n :

$$x_n = x_{n-1} + \frac{p}{100}x_{n-1}. \quad (5.5)$$

If n counts years, p is the annual interest rate, and if p is constant, we can with some arithmetics derive the following solution to (5.5):

$$x_n = \left(1 + \frac{p}{100}\right)x_{n-1} = \left(1 + \frac{p}{100}\right)^2 x_{n-2} = \dots = \left(1 + \frac{p}{100}\right)^n x_0.$$

Instead of first deriving a formula for x_n and then program this formula, we may attack the fundamental model (5.5) in a program (`growth_years.py`) and compute x_1 , x_2 , and so on in a loop:

```
from scitools.std import *
x0 = 100                # initial amount
p = 5                   # interest rate
N = 4                   # number of years
index_set = range(N+1)
```

```
x = zeros(len(index_set))

# solution:
x[0] = x0
for n in index_set[1:]:
    x[n] = x[n-1] + (p/100.0)*x[n-1]
print x
plot(index_set, x, 'ro', xlabel='years', ylabel='amount')
```

The output of `x` is

```
[ 100.      105.      110.25     115.7625    121.550625]
```

Programmers of mathematical software who are trained in making programs more efficient, will notice that it is not necessary to store all the x_n values in an array or use a list with all the indices $0, 1, \dots, N$. Just one integer for the index and two floats for x_n and x_{n-1} are strictly necessary. This can save quite some memory for large values of N . Exercise 5.5 asks you to develop such a memory-efficient program.

Suppose now that we are interested in computing the growth of money after N days instead. The interest rate per day is taken as $r = p/D$ if p is the annual interest rate and D is the number of days in a year. The fundamental model is the same, but now n counts days and p is replaced by r :

$$x_n = x_{n-1} + \frac{r}{100}x_{n-1}. \quad (5.6)$$

A common method in international business is to choose $D = 360$, yet let n count the exact number of days between two dates (see footnote on page 142). Python has a module `datetime` for convenient calculations with dates and times. To find the number of days between two dates, we perform the following operations:

```
>>> import datetime
>>> date1 = datetime.date(2007, 8, 3) # Aug 3, 2007
>>> date2 = datetime.date(2008, 8, 4) # Aug 4, 2008
>>> diff = date2 - date1
>>> print diff.days
367
```

We can modify the previous program to compute with days instead of years:

```
from scitools.std import *
x0 = 100                                # initial amount
p = 5                                  # annual interest rate
r = p/360.0                             # daily interest rate
import datetime
date1 = datetime.date(2007, 8, 3)
date2 = datetime.date(2011, 8, 3)
diff = date2 - date1
N = diff.days
index_set = range(N+1)
x = zeros(len(index_set))

# solution:
x[0] = x0
```

```

for n in index_set[1:]:
    x[n] = x[n-1] + (r/100.0)*x[n-1]
print x
plot(index_set, x, 'ro', xlabel='days', ylabel='amount')

```

Running this program, called `growth_days.py`, prints out 122.5 as the final amount.

It is quite easy to adjust the formula (5.4) to the case where the interest is added every day instead of every year. However, the strength of the model (5.6) and the associated program `growth_days.py` becomes apparent when r varies in time – and this is what happens in real life. In the model we can just write $r(n)$ to explicitly indicate the dependence upon time. The corresponding time-dependent annual interest rate is what is normally specified, and $p(n)$ is usually a piecewise constant function (the interest rate is changed at some specific dates and remains constant between these days). The construction of a corresponding array p in a program, given the dates when p changes, can be a bit tricky since we need to compute the number of days between the dates of changes and index p properly. We do not dive into these details now, but readers who want to compute p and who is ready for some extra brain training and index puzzling can attack Exercise 5.11. For now we assume that an array p holds the time-dependent annual interest rates for each day in the total time period of interest. The `growth_days.py` program then needs a slight modification, typically,

```

p = zeros(len(index_set))
# set up p (might be challenging!)
r = p/360.0                                # daily interest rate
...
for n in index_set[1:]:
    x[n] = x[n-1] + (r[n-1]/100.0)*x[n-1]

```

For the very simple (and not-so-relevant) case where p grows linearly (i.e., daily changes) from 4 to 6 percent over the period of interest, we have made a complete program in the file `growth_days_timedep.py`. You can compare a simulation with linearly varying p between 4 and 6 and a simulation using the average p value 5 throughout the whole time interval.

A difference equation with $r(n)$ is quite difficult to solve mathematically, but the n -dependence in r is easy to deal with in the computerized solution approach.

5.1.2 The Factorial as a Difference Equation

The difference equation

$$x_n = nx_{n-1}, \quad x_0 = 1 \quad (5.7)$$

can quickly be solved recursively:

$$\begin{aligned}
 x_n &= nx_{n-1} \\
 &= n(n-1)x_{n-2} \\
 &= n(n-1)(n-2)x_{n-3} \\
 &= n(n-1)(n-2)\cdots 1.
 \end{aligned}$$

The result x_n is nothing but the factorial of n , denoted as $n!$ (cf. Exercise 2.33). Equation (5.7) then gives a standard recipe to compute $n!$.

5.1.3 Fibonacci Numbers

Every textbook with some material on sequences usually presents a difference equation for generating the famous Fibonacci numbers³:

$$x_n = x_{n-1} + x_{n-2}, \quad x_0 = 1, \quad x_1 = 1, \quad n = 2, 3, \dots \quad (5.8)$$

This equation has a relation between three elements in the sequence, not only two as in the other examples we have seen. We say that this is a difference equation of second order, while the previous examples involving two n levels are said to be difference equations of first order. The precise characterization of (5.8) is a homogeneous difference equation of second order. Such classification is not important when computing the solution in a program, but for mathematical solution methods by pen and paper, the classification helps to determine which mathematical technique to use to solve the problem.

A straightforward program for generating Fibonacci numbers takes the form (`fibonacci1.py`):

```
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]
```

Since x_n is an infinite sequence we could try to run the program for very large N . This causes two problems: The storage requirements of the `x` array may become too large for the computer, but long before this happens, x_n grows in size far beyond the largest integer that can be represented by `int` elements in arrays (the problem appears already for $N = 50$). A possibility is to use array elements of type `int64`, which allows computation of twice as many numbers as with standard `int` elements (see the program `fibonacci1_int64.py`). A better solution is to use `float` elements in the `x` array, despite the fact that the numbers

³ Fibonacci arrived at this equation when modelling rabbit populations.

x_n are integers. With `float96` elements we can compute up to $N = 23600$ (see the program `fibonacci1_float.py`).

The best solution goes as follows. We observe, as mentioned after the `growth_years.py` program and also explained in Exercise 5.5, that we need only three variables to generate the sequence. We can therefore work with just three standard `int` variables in Python:

```
import sys
N = int(sys.argv[1])
xnm1 = 1
xnm2 = 1
n = 2
while n <= N:
    xn = xnm1 + xnm2
    print 'x_%d = %d' % (n, xn)
    xnm2 = xnm1
    xnm1 = xn
    n += 1
```

Here `xnm1` denotes x_{n-1} and `xnm2` denotes x_{n-2} . To prepare for the next pass in the loop, we must shuffle the `xnm1` down to `xnm2` and store the new x_n value in `xnm1`. The nice thing with `int` objects in Python (contrary to `int` elements in NumPy arrays) is that they can hold integers of arbitrary size⁴. We may try a run with `N` set to 250:

```
x_2 = 2
x_3 = 3
x_4 = 5
x_5 = 8
x_6 = 13
x_7 = 21
x_8 = 34
x_9 = 55
x_10 = 89
x_11 = 144
x_12 = 233
x_13 = 377
x_14 = 610
x_15 = 987
x_16 = 1597
...
x_249 = 7896325826131730509282738943634332893686268675876375
x_250 = 12776523572924732586037033894655031898659556447352249
```

In mathematics courses you learn how to derive a formula for the n -th term in a Fibonacci sequence. This derivation is much more complicated than writing a simple program to generate the sequence, but there is a lot of interesting mathematics both in the derivation and the resulting formula!

5.1.4 Growth of a Population

Let x_{n-1} be the number of individuals in a population at time t_{n-1} . The population can consist of humans, animals, cells, or whatever objects where the number of births and deaths is proportional to the number of individuals. Between time levels t_{n-1} and t_n , bx_n individuals are born,

⁴ Note that `int` variables in other computer languages normally has a size limitation like `int` elements in NumPy arrays.

and dx_n individuals die, where b and d are constants. The net growth of the population is then $(b - d)x_n$. Introducing $r = (b - d)100$ for the net growth factor measured in percent, the new number of individuals become

$$x_n = x_{n-1} + \frac{r}{100}x_{n-1}. \quad (5.9)$$

This is the same difference equation as (5.5). It models growth of populations quite well as long as there are optimal growing conditions for each individual. If not, one can adjust the model as explained in Chapter 5.1.5.

To solve (5.9) we need to start out with a known size x_0 of the population. The b and d parameters depend on the time difference $t_n - t_{n-1}$, i.e., the values of b and d are smaller if n counts years than if n counts generations.

5.1.5 Logistic Growth

The model (5.9) for the growth of a population leads to exponential increase in the number of individuals as implied by the solution 5.4. The size of the population increases faster and faster as time n increases, and $x_n \rightarrow \infty$ when $n \rightarrow \infty$. In real life, however, there is an upper limit M of the number of individuals that can exist in the environment at the same time. Lack of space and food, competition between individuals, predators, and spreading of contagious diseases are examples on factors that limit the growth. The number M is usually called the *carrying capacity* of the environment, the maximum population which is sustainable over time. With limited growth, the growth factor r must depend on time:

$$x_n = x_{n-1} + \frac{r(n-1)}{100}x_{n-1}. \quad (5.10)$$

In the beginning of the growth process, there is enough resources and the growth is exponential, but as x_n approaches M , the growth stops and r must tend to zero. A simple function $r(n)$ with these properties is

$$r(n) = \varrho \left(1 - \frac{x_n}{M}\right). \quad (5.11)$$

For small n , $x_n \ll M$ and $r(n) \approx \varrho$, which is the growth rate with unlimited resources. As $n \rightarrow M$, $r(n) \rightarrow 0$ as we want. The model (5.11) is used for *logistic growth*. The corresponding *logistic difference equation* becomes

$$x_n = x_{n-1} + \frac{\varrho}{100}x_{n-1} \left(1 - \frac{x_{n-1}}{M}\right). \quad (5.12)$$

Below is a program (`growth_logistic.py`) for simulating $N = 200$ time intervals in a case where we start with $x_0 = 100$ individuals, a carrying

capacity of $M = 500$, and initial growth of $\varrho = 4$ percent in each time interval:

```
from scitools.std import *
x0 = 100          # initial amount of individuals
M = 500           # carrying capacity
rho = 4           # initial growth rate in percent
N = 200           # number of time intervals
index_set = range(N+1)
x = zeros(len(index_set))

# solution:
x[0] = x0
for n in index_set[1:]:
    x[n] = x[n-1] + (rho/100.0)*x[n-1]*(1 - x[n-1]/float(M))
print x
plot(index_set, x, 'r', xlabel='time units',
      ylabel='no of individuals', hardcopy='tmp.eps')
```

Figure 5.1 shows how the population stabilizes, i.e., that x_n approaches M as N becomes large (of the same magnitude as M).

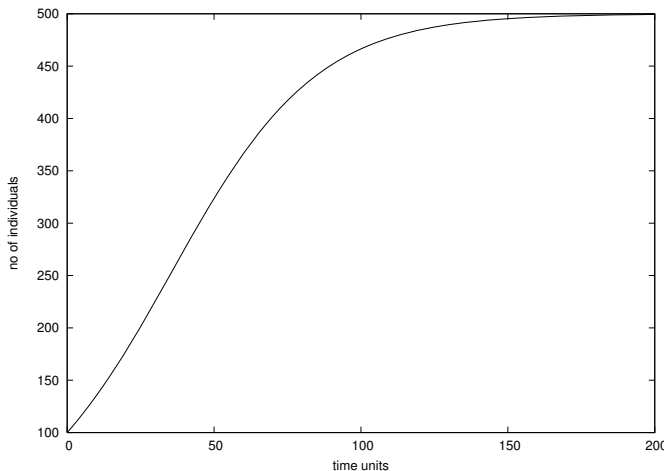


Fig. 5.1 Logistic growth of a population ($\varrho = 4$, $M = 500$, $x_0 = 100$, $N = 200$).

If the equation stabilizes as $n \rightarrow \infty$, it means that $x_n = x_{n-1}$ in this limit. The equation then reduces to

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

By inserting $x_n = M$ we see that this solution fulfills the equation. The same solution technique (i.e., setting $x_n = x_{n-1}$) can be used to check if x_n in a difference equation approaches a limit or not.

Mathematical models like (5.12) are often easier to work with if we *scale* the variables, as briefly described in Chapter 4.7.2. Basically, this means that we divide each variable by a characteristic size of that variable such that the value of the new variable is typically 1. In the present case we can scale x_n by M and introduce a new variable,

$$y_n = \frac{x_n}{M}.$$

Similarly, x_0 is replaced by $y_0 = x_0/M$. Inserting $x_n = My_n$ in (5.12) and dividing by M gives

$$y_n = y_{n-1} + qy_{n-1}(1 - y_{n-1}), \quad (5.13)$$

where $q = \varrho/100$ is introduced to save typing. Equation (5.13) is simpler than (5.12) in that the solution lies approximately between⁵ y_0 and 1, and there are only two dimensionless input parameters to care about: q and y_0 . To solve (5.12) we need knowledge of three parameters: x_0 , ϱ , and M .

5.1.6 Payback of a Loan

A loan L is to be paid back over N months. The payback in a month consists of the fraction L/N plus the interest increase of the loan. Let the annual interest rate for the loan be p percent. The monthly interest rate is then $\frac{p}{12}$. The value of the loan after month n is x_n , and the change from x_{n-1} can be modeled as

$$x_n = x_{n-1} + \frac{p}{12 \cdot 100} x_{n-1} - \left(\frac{p}{12 \cdot 100} x_{n-1} + \frac{L}{N} \right), \quad (5.14)$$

$$= x_{n-1} - \frac{L}{N}, \quad (5.15)$$

for $n = 1, \dots, N$. The initial condition is $x_0 = L$. A major difference between (5.15) and (5.6) is that all terms in the latter are proportional to x_n or x_{n-1} , while (5.15) also contains a constant term (L/N). We say that (5.6) is homogeneous and linear, while (5.15) is inhomogeneous (because of the constant term) and linear. The mathematical solution of inhomogeneous equations are more difficult to find than the solution of homogeneous equations, but in a program there is no big difference: We just add the extra term $-L/N$ in the formula for the difference equation.

The solution of (5.15) is not particularly exciting⁶. What is more interesting, is what we pay each month, y_n . We can keep track of both y_n and x_n in a variant of the previous model:

$$y_n = \frac{p}{12 \cdot 100} x_{n-1} + \frac{L}{N}, \quad (5.16)$$

$$x_n = x_{n-1} + \frac{p}{12 \cdot 100} x_{n-1} - y_n. \quad (5.17)$$

⁵ Values larger than 1 can occur, see Exercise 5.21.

⁶ Use (5.15) repeatedly to derive the solution $x_n = L - nL/N$.

Equations (5.16)–(5.17) is a system of difference equations. In a computer code, we simply update y_n first, and then we update x_n , inside a loop over n . Exercise 5.6 asks you to do this.

5.1.7 Taylor Series as a Difference Equation

Consider the following system of two difference equations

$$e_n = e_{n-1} + a_{n-1}, \quad (5.18)$$

$$a_n = \frac{x}{n} a_{n-1}, \quad (5.19)$$

with initial conditions $e_0 = 0$ and $a_0 = 1$. We can start to nest the solution:

$$\begin{aligned} e_1 &= 0 + a_0 = 0 + 1 = 1, \\ a_1 &= x, \\ e_2 &= e_1 + a_1 = 1 + x, \\ a_2 &= \frac{x}{2} a_1 = \frac{x^2}{2}, \\ e_3 &= e_2 + a_2 = 1 + x + \frac{x^2}{2}, \\ e_4 &= 1 + x + \frac{x^2}{2} + \frac{x^3}{3 \cdot 2}, \\ e_5 &= 1 + x + \frac{x^2}{2} + \frac{x^3}{3 \cdot 2} + \frac{x^4}{4 \cdot 3 \cdot 2} \end{aligned}$$

The observant reader who has heard about Taylor series (see Chapter A.4) will recognize this as the Taylor series of e^x :

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!}. \quad (5.20)$$

How do we derive a system like (5.18)–(5.19) for computing the Taylor polynomial approximation to e^x ? The starting point is the sum $\sum_{n=0}^{\infty} \frac{x^n}{n!}$. This sum is coded by adding new terms to an accumulation variable in a loop. The mathematical counterpart to this code is a difference equation

$$e_{n+1} = e_n + \frac{x^n}{n!}, \quad e_0 = 0, \quad n = 0, 1, 2, \dots \quad (5.21)$$

or equivalently (just replace n by $n - 1$):

$$e_n = e_{n-1} + \frac{x^{n-1}}{(n-1)!}, \quad e_0 = 0, \quad n = 1, 2, 3, \dots \quad (5.22)$$

Now comes the important observation: the term $x^n/n!$ contains many of the computations we already performed for the previous term $x^{n-1}/(n-1)!$ because

$$x^n = \frac{x \cdot x \cdots x}{n(n-1)(n-2) \cdots 1}, \quad x^{n-1} = \frac{x \cdot x \cdots x}{(n-1)(n-2)(n-3) \cdots 1}.$$

Let $a_n = x^n/n!$. We see that we can go from a_{n-1} to a_n by multiplying a_{n-1} by x/n :

$$\frac{x}{n}a_{n-1} = \frac{x}{n} \frac{x^{n-1}}{(n-1)!} \frac{x}{n} = \frac{x^n}{n!} = a_n, \quad (5.23)$$

which is nothing but (5.19). We also realize that $a_0 = 1$ is the initial condition for this difference equation. In other words, (5.18) sums the Taylor polynomial, and (5.19) updates each term in the sum.

The system (5.18)–(5.19) is very easy to implement in a program and constitutes an efficient way to compute (5.20). The function `exp_diffeq` does the work⁷:

```
def exp_diffeq(x, N):
    n = 1
    an_prev = 1.0 # a_0
    en_prev = 0.0 # e_0
    while n <= N:
        en = en_prev + an_prev
        an = x/n*an_prev
        en_prev = en
        an_prev = an
        n += 1
    return en
```

This function along with a direct evaluation of the Taylor series for e^x and a comparison with the exact result for various N values can be found in the file `exp_Taylor_series_diffeq.py`.

5.1.8 Making a Living from a Fortune

Suppose you want to live on a fortune F . You have invested the money in a safe way that gives an annual interest of p percent. Every year you plan to consume an amount c_n , where n counts years. The development of your fortune x_n from one year to the other can then be modeled by

$$x_n = x_{n-1} + \frac{p}{100}x_{n-1} - c_{n-1}, \quad x_0 = F. \quad (5.24)$$

A simple example is to keep c constant, say q percent of the interest the first year:

$$x_n = x_{n-1} + \frac{p}{100}x_{n-1} - \frac{pq}{10^4}F, \quad x_0 = F. \quad (5.25)$$

⁷ Observe that we do not store the sequences in arrays, but make use of the fact that only the most recent sequence element is needed to calculate a new element.

A more realistic model is to assume some inflation of I percent per year. You will then like to increase c_n by the inflation. We can extend the model in two ways. The simplest and clearest way, in the author's opinion, is to track the evolution of two sequences x_n and c_n :

$$x_n = x_{n-1} + \frac{p}{100}x_{n-1} - c_{n-1}, \quad x_0 = F, \quad c_0 = \frac{pq}{10^4}F, \quad (5.26)$$

$$(5.27)$$

$$c_n = c_{n-1} + \frac{I}{100}c_{n-1}. \quad (5.28)$$

This is a system of two difference equations with two unknowns. The solution method is, nevertheless, not much more complicated than the method for a difference equation in one unknown, since we can first compute x_n from (5.27) and then update the c_n value from (5.28). You are encouraged to write the program (see Exercise 5.7).

Another way of making a difference equation for the case with inflation, is to use an explicit formula for c_{n-1} , i.e., solve (5.27) and end up with a formula like (5.4). Then we can insert the explicit formula

$$c_{n-1} = \left(1 + \frac{I}{100}\right)^{n-1} \frac{pq}{10^4}F$$

in (5.24), resulting in only one difference equation to solve.

5.1.9 Newton's Method

The difference equation

$$x_n = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}, \quad x_0 \text{ given}, \quad (5.29)$$

generates a sequence x_n where, if the sequence converges (i.e., if $x_n - x_{n-1} \rightarrow 0$), x_n approaches a root of $f(x)$. That is, $x_n \rightarrow x$, where x solves the equation $f(x) = 0$. Equation (5.29) is the famous Newton's method for solving nonlinear algebraic equations $f(x) = 0$. When $f(x)$ is not linear, i.e., $f(x)$ is not on the form $ax + b$ with constant a and b , (5.29) becomes a *nonlinear difference equation*. This complicates analytical treatment of difference equations, but poses no extra difficulties for numerical solution.

We can quickly sketch the derivation of (5.29). Suppose we want to solve the equation

$$f(x) = 0$$

and that we already have an approximate solution x_{n-1} . If $f(x)$ were linear, $f(x) = ax + b$, it would be very easy to solve $f(x) = 0$: $x = -b/a$. The idea is therefore to approximate $f(x)$ in the vicinity of $x = x_{n-1}$ by a linear function, i.e., a straight line $f(x) \approx \tilde{f}(x) = ax + b$. This

line should have the same slope as $f(x)$, i.e., $a = f'(x_{n-1})$, and both the line and f should have the same value at $x = x_{n-1}$. From this condition one can find $b = f(x_{n-1}) - x_{n-1}f'(x_{n-1})$. The approximate function (line) is then

$$\tilde{f}(x) = f(x_{n-1}) + f'(x_{n-1})(x - x_{n-1}). \quad (5.30)$$

This expression is just the two first terms of a Taylor series approximation to $f(x)$ at $x = x_{n-1}$. It is now easy to solve $\tilde{f}(x) = 0$ with respect to x , and we get

$$x = x_{n-1} - \frac{f(x_{n-1})}{f'(x_{n-1})}. \quad (5.31)$$

Since \tilde{f} is only an approximation to f , x in (5.31) is only an approximation to a root of $f(x) = 0$. Hopefully, the approximation is better than x_{n-1} so we set $x_n = x$ as the next term in a sequence that we hope converges to the correct root. However, convergence depends highly on the shape of $f(x)$, and there is no guarantee that the method will work.

The previous programs for solving difference equations have typically calculated a sequence x_n up to $n = N$, where N is given. When using (5.29) to find roots of nonlinear equations, we do not know a suitable N in advance that leads to an x_n where $f(x_n)$ is sufficiently close to zero. We therefore have to keep on increasing n until $f(x_n) < \epsilon$ for some small ϵ . Of course, the sequence diverges, we will keep on forever, so there must be some maximum allowable limit on n , which we may take as N .

It can be convenient to have the solution of (5.29) as a function for easy reuse. Here is a first rough implementation:

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

This function might well work, but $f(x)/dfdx(x)$ can imply integer division, so we should ensure that the numerator or denominator is of float type. There are also two function evaluations of $f(x)$ in every pass in the loop (one in the loop body and one in the while condition). We can get away with only one evaluation if we store the $f(x)$ in a local variable. In the small examples with $f(x)$ in the present course, twice as many function evaluations of f as necessary does not matter, but the same `Newton` function can in fact be used for much more complicated functions, and in those cases twice as much work can be noticeable. As a programmer, you should therefore learn to optimize the code by removing unnecessary computations.

Another, more serious, problem is the possibility dividing by zero. Almost as serious, is dividing by a very small number that creates a large value, which might cause Newton's method to diverge. Therefore, we should test for small values of $f'(x)$ and write a warning or raise an exception.

Another improvement is to add a boolean argument `store` to indicate whether we want the $(x, f(x))$ values during the iterations to be stored in a list or not. These intermediate values can be handy if we want to print out or plot the convergence behavior of Newton's method.

An improved Newton function can now be coded as

```
def Newton(f, x, dfdx, epsilon=1.0E-7, N=100, store=False):
    f_value = f(x)
    n = 0
    if store: info = [(x, f_value)]
    while abs(f_value) > epsilon and n <= N:
        dfdx_value = float(dfdx(x))
        if abs(dfdx_value) < 1E-14:
            raise ValueError("Newton: f'(%g)=%g" % (x, dfdx_value))

        x = x - f_value/dfdx_value

        n += 1
        f_value = f(x)
        if store: info.append((x, f_value))
    if store:
        return x, info
    else:
        return x, n, f_value
```

Note that to use the Newton function, we need to calculate the derivative $f'(x)$ and implement it as a Python function and provide it as the `dfdx` argument. Also note that what we return depends on whether we store $(x, f(x))$ information during the iterations or not.

It is quite common to test if $dfdx(x)$ is zero in an implementation of Newton's method, but this is not strictly necessary in Python since an exception `ZeroDivisionError` is always raised when dividing by zero.

We can apply the Newton function to solve the equation⁸ $e^{-0.1x^2} \sin(\frac{\pi}{2}x) = 0$:

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

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)
```

⁸ Fortunately you realize that the exponential function can never be zero, so the solutions of the equation must be the zeros of the sine function, i.e., $\frac{\pi}{2}x = i\pi$ for all integers $i = \dots, -2, -1, 0, 1, 2, \dots$. This gives $x = 2i$ as the solutions.

```
print 'root:', x
for i in range(len(info)):
    print 'Iteration %3d: f(%g)=%g' % \
        (i, info[i][0], info[i][1])
```

The Newton function and this program can be found in the file `Newton.py`. Running this program with an initial x value of 1.7 results in the output

```
root: 1.99999999768449
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
```

The convergence is fast towards the solution $x = 2$. The error is of the order 10^{-10} even though we stop the iterations when $f(x) \leq 10^{-7}$.

Trying a start value of 3 we would expect the method to find either nearby solution $x = 2$ or $x = 4$, but now we get

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

We have definitely solved $f(x) = 0$ in the sense that $|f(x)| \leq \epsilon$, where ϵ is a small value (here $\epsilon \sim 10^{-79}$). However, the solution $x \approx 42.5$ is *not* close to the solution ($x = 42$ and $x = 44$ are the solutions closest to the computed x). Can you use your knowledge of how the Newton method works and figure out why we get such strange behavior?

The demo program `Newton_movie.py` can be used to investigate the strange behavior. This program takes five command-line arguments: a formula for $f(x)$, a formula for $f'(x)$ (or the word `numeric`, which indicates a numerical approximation of $f'(x)$), a guess at the root, and the minimum and maximum x values in the plots. We try the following case with the program:

Terminal

```
Newton_movie.py 'exp(-0.1*x**2)*sin(pi/2*x)' numeric 3 -3 43
```

As seen, we start with $x = 3$ as the initial guess. In the first step of the method, we compute a new value of the root, now $x = 4.66667$. As we see in Figure 5.2, this root is near an extreme point of $f(x)$ so that the derivative is small, and the resulting straight line approximation to $f(x)$ at this root becomes quite flat. The result is a new guess at the root: $x42.5$. This root is far away from the last root, but the second problem is that $f(x)$ is quickly damped as we move to increasing x values, and at $x = 42.5$ f is small enough to fulfill the convergence criterion. Any guess at the root out in this region would satisfy that criterion.

You can run the `Newton_movie.py` program with other values of the initial root and observe that the method usually finds the nearest roots.

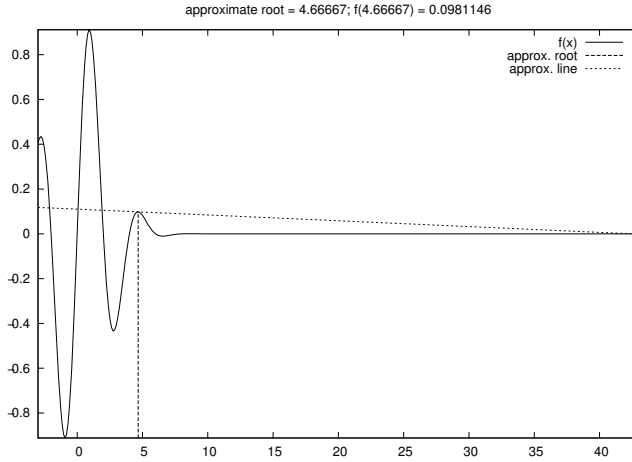


Fig. 5.2 Failure of Newton's method to solve $e^{-0.1x^2} \sin(\frac{\pi}{2}x) = 0$. The plot corresponds to the second root found (starting with $x = 3$).

5.1.10 The Inverse of a Function

Given a function $f(x)$, the inverse function of f , say we call it $g(x)$, has the property that if we apply g to the value $f(x)$, we get x back:

$$g(f(x)) = x.$$

Similarly, if we apply f to the value $g(x)$, we get x :

$$f(g(x)) = x. \quad (5.32)$$

By hand, you substitute $g(x)$ by (say) y in (5.32) and solve (5.32) with respect to y to find some x expression for the inverse function. For example, given $f(x) = x^2 - 1$, we must solve $y^2 - 1 = x$ with respect to y . To ensure a unique solution for y , the x values have to be limited to an interval where $f(x)$ is monotone, say $x \in [0, 1]$ in the present example. Solving for y gives $y = \sqrt{1+x}$, therefore and $g(x) = \sqrt{1+x}$. It is easy to check that $f(g(x)) = (\sqrt{1+x})^2 - 1 = x$.

Numerically, we can use the “definition” (5.32) of the inverse function g at one point at a time. Suppose we have a sequence of points $x_0 < x_1 < \dots < x_N$ along the x axis such that f is monotone in $[x_0, x_N]$: $f(x_0) > f(x_1) > \dots > f(x_N)$ or $f(x_0) < f(x_1) < \dots < f(x_N)$. For each point x_i , we have

$$f(g(x_i)) = x_i.$$

The value $g(x_i)$ is unknown, so let us call it γ . The equation

$$f(\gamma) = x_i \quad (5.33)$$

can be solved by respect γ . However, (5.33) is in general nonlinear if f is a nonlinear function of x . We must then use, e.g., Newton's method

to solve (5.33). Newton's method works for an equation phrased as " $f(x) = 0$ ", which in our case is $f(\gamma) - x_i = 0$, i.e., we seek the roots of the function $F(\gamma) \equiv f(\gamma) - x_i$. Also the derivative $F'(\gamma)$ is needed in Newton's method. For simplicity we may use an approximate finite difference:

$$\frac{dF}{d\gamma} \approx \frac{F(\gamma + h) - F(\gamma - h)}{2h}.$$

As start value γ_0 , we can use the previously computed g value: g_{i-1} . We introduce the short notation $\gamma = \text{Newton}(F, \gamma_0)$ to indicate the solution of $F(\gamma) = 0$ with initial guess γ_0 .

The computation of all the g_0, \dots, g_N values can now be expressed by

$$g_i = \text{Newton}(F, g_{i-1}), \quad i = 1, \dots, N, \quad (5.34)$$

and for the first point we may use x_0 as start value (for instance):

$$g_0 = \text{Newton}(F, x_0). \quad (5.35)$$

Equations (5.34)–(5.35) constitute a difference equation for g_i , since given g_{i-1} , we can compute the next element of the sequence by (5.34). Because (5.34) is a nonlinear equation in the new value g_i , and (5.34) is therefore an example of a *nonlinear difference equation*.

The following program computes the inverse function $g(x)$ of $f(x)$ at some discrete points x_0, \dots, x_N . Our sample function is $f(x) = x^2 - 1$:

```
from Newton import Newton
from scitools.std import *

def f(x):
    return x**2 - 1

def F(gamma):
    return f(gamma) - xi

def dFdx(gamma):
    return (F(gamma+h) - F(gamma-h))/(2*h)

h = 1E-6
x = linspace(0.01, 3, 21)
g = zeros(len(x))

for i in range(len(x)):
    xi = x[i]

    # compute start value (use last g[i-1] if possible):
    if i == 0:
        gamma0 = x[0]
    else:
        gamma0 = g[i-1]

    gamma, n, F_value = Newton(F, gamma0, dFdx)
    g[i] = gamma

plot(x, f(x), 'r-', x, g, 'b-',
     title='f1', legend=('original', 'inverse'))
```

Note that with $f(x) = x^2 - 1$, $f'(0) = 0$, so Newton's method divides by zero and breaks down unless with let $x_0 > 0$, so here we set $x_0 = 0.01$. The `f` function can easily be edited to let the program compute the inverse of another function. The `F` function can remain the same since it applies a general finite difference to approximate the derivative of the `f(x)` function. The complete program is found in the file `inverse_function.py`. A better implementation is suggested in Exercise 7.20.

5.2 Programming with Sound

Sound on a computer is nothing but a sequence of numbers. As an example, consider the famous A tone at 440 Hz. Physically, this is an oscillation of a tunefork, loudspeaker, string or another mechanical medium that makes the surrounding air also oscillate and transport the sound as a compression wave. This wave may hit our ears and through complicated physiological processes be transformed to an electrical signal that the brain can recognize as sound. Mathematically, the oscillations are described by a sine function of time:

$$s(t) = A \sin(2\pi ft), \quad (5.36)$$

where A is the amplitude or strength of the sound and f is the frequency (440 Hz for the A in our example). In a computer, $s(t)$ is represented at discrete points of time. CD quality means 44100 samples per second. Other sample rates are also possible, so we introduce r as the sample rate. An f Hz tone lasting for m seconds with sample rate r can then be computed as the sequence

$$s_n = A \sin\left(2\pi f \frac{n}{r}\right), \quad n = 0, 1, \dots, m \cdot r. \quad (5.37)$$

With Numerical Python this computation is straightforward and very efficient. Introducing some more explanatory variable names than r , A , and m , we can write a function for generating a note:

```
import numpy
def note(frequency, length, amplitude=1, sample_rate=44100):
    time_points = numpy.linspace(0, length, length*sample_rate)
    data = numpy.sin(2*numpy.pi*frequency*time_points)
    data = amplitude*data
    return data
```

5.2.1 Writing Sound to File

The `note` function above generates an array of `float` data representing a note. The sound card in the computer cannot play these data, because

the card assumes that the information about the oscillations appears as a sequence of two-byte integers. With an array's `astype` method we can easily convert our data to two-byte integers instead of `floats`:

```
data = data.astype(numpy.int16)
```

That is, the name of the two-byte integer data type in `numpy` is `int16` (two bytes are 16 bits). The maximum value of a two-byte integer is $2^{15} - 1$, so this is also the maximum amplitude. Assuming that `amplitude` in the `note` function is a relative measure of intensity, such that the value lies between 0 and 1, we must adjust this amplitude to the scale of two-byte integers:

```
max_amplitude = 2**15 - 1  
data = max_amplitude*data
```

The `data` array of `int16` numbers can be written to a file and played as an ordinary file in CD quality. Such a file is known as a wave file or simply a WAV file since the extension is `.wav`. Python has a module `wave` for creating such files. Given an array of sound, `data`, we have in SciTools a module `sound` with a function `write` for writing the data to a WAV file (using functionality from the `wave` module):

```
import scitools.sound  
scitools.sound.write(data, 'Atone.wav')
```

You can now use your favorite music player to play the `Atone.wav` file, or you can play it from within a Python program using

```
scitools.sound.play('Atone.wav')
```

The `write` function can take more arguments and write, e.g., a stereo file with two channels, but we do not dive into these details here.

5.2.2 Reading Sound from File

Given a sound signal in a WAV file, we can easily read this signal into an array and mathematically manipulate the data in the array to change the flavor of the sound, e.g., add echo, treble, or bass. The recipe for reading a WAV file with name `filename` is

```
data = scitools.sound.read(filename)
```

The `data` array has elements of type `int16`. Often we want to compute with this array, and then we need elements of `float` type, obtained by the conversion

```
data = data.astype(float)
```

The `write` function automatically transforms the element type back to `int16` if we have not done this explicitly.

One operation that we can easily do is adding an echo. Mathematically this means that we add a damped delayed sound, where the original sound has weight β and the delayed part has weight $1 - \beta$, such that the overall amplitude is not altered. Let d be the delay in seconds. With a sampling rate r the number of indices in the delay becomes dr , which we denote by b . Given an original sound sequence s_n , the sound with echo is the sequence

$$e_n = \beta s_n + (1 - \beta) s_{n-b}. \quad (5.38)$$

We cannot start n at 0 since $e_0 = s_{0-b} = s_{-b}$ which is a value outside the sound data. Therefore we define $e_n = s_n$ for $n = 0, 1, \dots, b$, and add the echo thereafter. A simple loop can do this (again we use descriptive variable names instead of the mathematical symbols introduced):

```
def add_echo(data, beta=0.8, delay=0.002, sample_rate=44100):
    newdata = data.copy()
    shift = int(delay*sample_rate) # b (math symbol)
    for i in range(shift, len(data)):
        newdata[i] = beta*data[i] + (1-beta)*data[i-shift]
    return newdata
```

The problem with this function is that it runs slowly, especially when we have sound clips lasting several seconds (recall that for CD quality we need 44100 numbers per second). It is therefore necessary to vectorize the implementation of the difference equation for adding echo. The update is then based on adding slices:

```
newdata[shift:] = beta*data[shift:] + \
    (1-beta)*data[:len(data)-shift]
```

5.2.3 Playing Many Notes

How do we generate a melody mathematically in a computer program? With the `note` function we can generate a note with a certain amplitude, frequency, and duration. The note is represented as an array. Putting sound arrays for different notes after each other will make up a melody. If we have several sound arrays `data1`, `data2`, `data3`, ..., we can make a new array consisting of the elements in the first array followed by the elements of the next array followed by the elements in the next array and so forth:

```
data = numpy.concatenate((data1, data2, data3, ...))
```

Here is an example of creating a little melody (start of “Nothing Else Matters” by Metallica) using constant (max) amplitude of all the notes:

```
E1 = note(164.81, .5)
G = note(392, .5)
B = note(493.88, .5)
E2 = note(659.26, .5)
intro = numpy.concatenate((E1, G, B, E2, B, G))
high1_long = note(987.77, 1)
high1_short = note(987.77, .5)
high2 = note(1046.50, .5)
high3 = note(880, .5)
high4_long = note(659.26, 1)
high4_medium = note(659.26, .5)
high4_short = note(659.26, .25)
high5 = note(739.99, .25)
pause_long = note(0, .5)
pause_short = note(0, .25)
song = numpy.concatenate(
    (intro, intro, high1_long, pause_long, high1_long,
     pause_long, pause_long,
     high1_short, high2, high1_short, high3, high1_short,
     high3, high4_short, pause_short, high4_long, pause_short,
     high4_medium, high5, high4_short))
scitools.sound.play(song)
scitools.sound.write(song, 'tmp.wav')
```

We could send `song` to the `add_echo` function to get some echo, and we could also vary the amplitudes to get more dynamics into the song. You can find the generation of notes above as the function `Nothing_Else_Matters(echo=False)` in the `scitools.sound` module.

5.3 Summary

5.3.1 Chapter Topics

Sequences. In general, a finite sequence can be written as

$$(x_n)_{n=0}^N, \quad x_n = f(n),$$

where $f(n)$ is some expression involving n and possibly other parameters. The coding of such a sequence takes the form

```
x = zeros(N+1)
for n in range(N+1):
    x[n] = f(n)
```

Here, we store the whole sequence in an array, which is convenient if we want to plot the evolution of the sequence (i.e., x_n versus n). Occasionally, this can require too much memory. Especially when checking for convergence of x_n toward some limit as $N \rightarrow \infty$, N may be large

if the sequence converges slowly. The array can be skipped if we print the sequence elements as they are computed:

```
for n in range(N+1):
    print f(n)
```

Difference Equations. Equations involving more than one element of a sequence are called difference equations. We have typically looked at difference equations relating x_n to the previous element x_{n-1} :

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

where f is some specified function. Any difference equation must have an initial condition prescribing x_0 , say $x_0 = X$. The solution of a difference equation is a sequence. Very often, n corresponds to a time parameter in applications.

We have also looked at systems of difference equations of the form

$$\begin{aligned} x_n &= f(x_{n-1}, y_{n-1}), & x_0 &= X \\ y_n &= g(y_{n-1}, x_{n-1}, x_n), & y_0 &= Y \end{aligned}$$

where f and g denote formulas involving already computed quantities. Note that x_n does not depend on y_n , which means that we can first compute x_n and then y_n :

```
index_set = range(N+1)
x = zeros(len(index_set))
y = zeros(len(index_set))
x[0] = X
y[0] = Y
for n in index_set[1:]:
    x[n] = f(x[n-1], y[n-1])
    y[n] = g(y[n-1], x[n], x[n-1])
```

Sound. Sound on a computer is a sequence of 2-byte integers. These can be stored in an array. Creating the sound of a tone consist of sampling a sine function and storing the values in an array (and converting to 2-byte integers). If we want to manipulate a given sound, say add some echo, we convert the array elements to ordinary floating-point numbers and perform mathematical operations on the array elements.

5.3.2 Summarizing Example: Music of a Sequence

Problem. The purpose of this summarizing example is to listen to the sound generated by two mathematical sequences. The first one is given by an explicit formula, constructed to oscillate around 0 with decreasing amplitude:

$$x_n = e^{-4n/N} \sin(8\pi n/N). \quad (5.39)$$

The other sequence is generated by the difference equation (5.13) for logistic growth, repeated here for convenience:

$$x_n = x_{n-1} + qx_{n-1}(1 - x_{n-1}), \quad x = x_0. \quad (5.40)$$

We let $x_0 = 0.01$ and $q = 2$. This leads to fast initial growth toward the limit 1, and then oscillations around this limit (this problem is studied in Exercise 5.21).

The absolute value of the sequence elements x_n are of size between 0 and 1, approximately. We want to transform these sequence elements to tones, using the techniques of Chapter 5.2. First we convert x_n to a frequency the human ear can hear. The transformation

$$y_n = 440 + 200x_n \quad (5.41)$$

will make a standard A reference tone out of $x_n = 0$, and for the maximum value of x_n around 1 we get a tone of 640 Hz. Elements of the sequence generated by (5.39) lie between -1 and 1, so the corresponding frequencies lie between 240 Hz and 640 Hz. The task now is to make a program that can generate and play the sounds.

Solution. Tones can be generated by the `note` function from the `scitools.sound` module. We collect all tones corresponding to all the y_n frequencies in a list `tones`. Letting `N` denote the number of sequence elements, the relevant code segment reads

```
from scitools.sound import *
freqs = 440 + x*200
tones = []
duration = 30.0/N      # 30 sec sound in total
for n in range(N+1):
    tones.append(max_amplitude*note(freqs[n], duration, 1))
data = concatenate(tones)
write(data, filename)
data = read(filename)
play(filename)
```

It is illustrating to plot the sequences too,

```
plot(range(N+1), freqs, 'ro')
```

To generate the sequences (5.39) and (5.40), we make two functions, `oscillations` and `logistic`, respectively. These functions take the number of sequence elements (`N`) as input and return the sequence stored in an array.

In another function `make_sound` we compute the sequence, transform the elements to frequencies, generate tones, write the tones to file, and play the sound file.

As always, we collect the functions in a module and include a test block where we can read the choice of sequence and the sequence length from the command line. The complete module file look as follows:

```
from scitools.sound import *
from scitools.std import *

def oscillations(N):
    x = zeros(N+1)
    for n in range(N+1):
        x[n] = exp(-4*n/float(N))*sin(8*pi*n/float(N))
    return x

def logistic(N):
    x = zeros(N+1)
    x[0] = 0.01
    q = 2
    for n in range(1, N+1):
        x[n] = x[n-1] + q*x[n-1]*(1 - x[n-1])
    return x

def make_sound(N, seqtype):
    filename = 'tmp.wav'
    x = eval(seqtype)(N)
    # convert x values to frequencies around 440:
    freqs = 440 + x*200
    plot(range(N+1), freqs, 'ro')
    # generate tones:
    tones = []
    duration = 30.0/N      # 30 sec sound in total
    for n in range(N+1):
        tones.append(max_amplitude*note(freqs[n], duration, 1))
    data = concatenate(tones)
    write(data, filename)
    data = read(filename)
    play(filename)

if __name__ == '__main__':
    try:
        seqtype = sys.argv[1]
        N = int(sys.argv[2])
    except IndexError:
        print 'Usage: %s oscillations|logistic N' % sys.argv[0]
        sys.exit(1)
    make_sound(N, seqtype)
```

This code should be quite easy to read at the present stage in the book. However, there is one statement that deserves a comment:

```
x = eval(seqtype)(N)
```

The `seqtype` argument reflects the type of sequence and is a string that the user provides on the command line. The values of the string equal the function names `oscillations` and `logistic`. With `eval(seqtype)` we turn the string into a function name. For example, if `seqtype` is `'logistic'`, performing an `eval(seqtype)(N)` is the same as if we had written `logistic(N)`. This technique allows the user of the program to choose a function call inside the code. Without `eval` we would need to explicitly test on values:

```

if seqtype == 'logistic':
    x = logistic(N)
elif seqtype == 'oscillations':
    x = oscillations(N)

```

This is not much extra code to write in the present example, but if we have a large number of functions generating sequences, we can save a lot of boring if-else code by using the `eval` construction.

The next step, as a reader who have understood the problem and the implementation above, is to run the program for two cases: the `oscillations` sequence with $N = 40$ and the `logistic` sequence with $N = 100$. By altering the q parameter to lower values, you get other sounds, typically quite boring sounds for non-oscillating logistic growth ($q < 1$). You can also experiment with other transformations of the form (5.41), e.g., increasing the frequency variation from 200 to 400.

5.4 Exercises

Exercise 5.1. *Determine the limit of a sequence.*

Given the sequence

$$a_n = \frac{7 + 1/n}{3 - 1/n^2},$$

make a program that computes and prints out a_n for $n = 1, 2, \dots, N$. Read N from the command line. Does a_n approach a finite limit when $n \rightarrow \infty$? Name of program file: `sequence_limit1.py`. \diamond

Exercise 5.2. *Determine the limit of a sequence.*

Solve Exercise 5.1 when the sequence of interest is given by

$$D_n = \frac{\sin(2^{-n})}{2^{-n}}.$$

Name of program file: `sequence_limit2.py`. \diamond

Exercise 5.3. *Experience convergence problems.*

Given the sequence

$$D_n = \frac{f(x+h) - f(x)}{h}, \quad h = 2^{-n} \quad (5.42)$$

make a function `D(f, x, N)` that takes a function $f(x)$, a value x , and the number N of terms in the sequence as arguments, and returns an array with the D_n values for $n = 0, 1, \dots, N-1$. Make a call to the `D` function with $f(x) = \sin x$, $x = 0$, and $N = 80$. Plot the evolution of the computed D_n values, using small circles for the data points.

Make another call to `D` where $x = \pi$ and plot this sequence in a separate figure. What would be your expected limit? Why do the com-

putations go wrong for large N ? (Hint: Print out the numerator and denominator in D_n .) Name of program file: `sequence_limits3.py`. \diamond

Exercise 5.4. *Convergence of sequences with π as limit.*

The following sequences all converge to π :

$$\begin{aligned}(a_n)_{n=1}^\infty, \quad a_n &= 4 \sum_{k=1}^n \frac{(-1)^{k+1}}{2k-1}, \\(b_n)_{n=1}^\infty, \quad b_n &= \left(6 \sum_{k=1}^n k^{-2} \right)^{1/2}, \\(c_n)_{n=1}^\infty, \quad c_n &= \left(90 \sum_{k=1}^n k^{-4} \right)^{1/4}, \\(d_n)_{n=1}^\infty, \quad d_n &= \frac{6}{\sqrt{3}} \sum_{k=0}^n \frac{(-1)^k}{3^k(2k+1)}, \\(e_n)_{n=1}^\infty, \quad e_n &= 16 \sum_{k=0}^n \frac{(-1)^k}{5^{2k+1}(2k+1)} - 4 \sum_{k=0}^n \frac{(-1)^k}{239^{2k+1}(2k+1)}.\end{aligned}$$

Make a function for each sequence that returns an array with the elements in the sequence. Plot all the sequences, and find the one that converges fastest toward the limit π . Name of program file: `pi.py`. \diamond

Exercise 5.5. *Reduce memory usage of difference equations.*

Consider the program `growth_years.py` from Chapter 5.1.1. Since x_n depends on x_{n-1} only, we do not need to store all the $N+1$ x_n values. We actually only need to store x_n and its previous value x_{n-1} . Modify the program to use two variables for x_n and not an array. Also avoid the `index_set` list and use an integer counter for n and a `while` instead. (Of course, without the arrays it is not possible to plot the development of x_n , so you have to remove the `plot` call.) Name of program file: `growth_years_efficient.py`. \diamond

Exercise 5.6. *Development of a loan over N months.*

Solve (5.16)–(5.17) for $n = 1, 2, \dots, N$ in a Python function. Name of program file: `loan.py`. \diamond

Exercise 5.7. *Solve a system of difference equations.*

Solve (5.27)–(5.28) by generating the x_n and c_n sequences in a Python function. Let the function return the computed sequences as arrays. Plot the x_n sequence. Name of program file: `fortune_and_inflation1.py`. \diamond

Exercise 5.8. *Extend the model (5.27)–(5.28).*

In the model (5.27)–(5.28) the new fortune is the old one, plus the interest, minus the consumption. During year n , x_n is normally also

reduced with t percent tax on the earnings $x_{n-1} - x_{n-2}$ in year $n - 1$. Extend the model with an appropriate tax term, modify the program from Exercise 5.7, and plot x_n with tax ($t = 28$) and without tax ($t = 0$). Name of program file: `fortune_and_inflation2.py`. \diamond

Exercise 5.9. *Experiment with the program from Exer. 5.8.*

Suppose you expect to live for N years and can accept that the fortune x_n vanishes after N years. Experiment with the program from Exercise 5.8 for how large the initial c_0 can be in this case. Choose some appropriate values for p , q , I , and t . Name of program file: `fortune_and_inflation3.py`. \diamond

Exercise 5.10. *Change index in a difference equation.*

A mathematically equivalent equation to (5.5) is

$$x_{i+1} = x_i + \frac{p}{100}x_i, \quad (5.43)$$

since the name of the index can be chosen arbitrarily. Suppose someone has made the following program for solving (5.43) by a slight editing of the program `growth1.py`:

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

# solution:
x[0] = x0
for i in index_set[1:]:
    x[i+1] = x[i] + (p/100.0)*x[i]
print x
plot(index_set, x, 'ro', xlabel='years', ylabel='amount')
```

This program does not work. Make a correct version, but keep the difference equations in its present form with the indices $i+1$ and i . Name of program file: `growth1_index_ip1.py`. \diamond

Exercise 5.11. *Construct time points from dates.*

A certain quantity p (which may be an interest rate) is piecewise constant and undergoes changes at some specific dates, e.g.,

$$p \text{ changes to } \begin{cases} 4.5 & \text{on Jan 4, 2009} \\ 4.75 & \text{on March 21, 2009} \\ 6.0 & \text{on April 1, 2009} \\ 5.0 & \text{on June 30, 2009} \\ 4.5 & \text{on Nov 1, 2009} \\ 2.0 & \text{on April 1, 2010} \end{cases} \quad (5.44)$$

Given a start date d_1 and an end date d_2 , fill an array p with the right p values, where the array index counts days. Use the `datetime` module to

compute the number of days between dates and store the information about changes in p in a list of 2-tuples, where each 2-tuple holds the date of change and the corresponding value of p . For the changes in p given above, the list becomes

```
change_in_p = [(datetime.date(2009, 1, 4), 4.5),
               (datetime.date(2009, 3, 21), 4.75),
               (datetime.date(2009, 4, 1), 6.0),
               (datetime.date(2009, 6, 30), 5.0),
               (datetime.date(2009, 11, 1), 4.5),
               (datetime.date(2010, 4, 1), 2.0)]
```

Name of program file: `dates2days.py`. \diamond

Exercise 5.12. *Solve nonlinear equations by Newton's method.*

Import the `Newton` function from the `Newton.py` file from Chapter 5.1.9 to solve the following nonlinear algebraic equations:

$$\sin x = 0, \quad (5.45)$$

$$x = \sin x, \quad (5.46)$$

$$x^5 = \sin x, \quad (5.47)$$

$$x^4 \sin x = 0, \quad (5.48)$$

$$x^4 = 0, \quad (5.49)$$

$$x^{10} = 0, \quad (5.50)$$

$$\tanh x = x^{10}. \quad (5.51)$$

Read the starting point x_0 and the equation to be solved from the command line (use `StringFunction` from Chapter 3.1.4 to convert the formula for $f(x)$ to a Python function). Print out the evolution of the roots (based on the `info` list). You will need to carefully plot the $f(x)$ function to understand how Newton's method will behave in each case for different starting values x_0 . Find an x_0 value for each equation so that Newton's method will converge toward the root $x = 0$. Name of program file: `Newton_examples.py`. \diamond

Exercise 5.13. *Visualize the convergence of Newton's method.*

Let x_0, x_1, \dots, x_N be the sequence of roots generated by Newton's method applied to a nonlinear algebraic equation $f(x) = 0$ (cf. Chapter 5.1.9). In this exercise, the purpose is to plot the sequences $(x_n)_{n=0}^N$ and $(f(x_n))_{n=0}^N$. Make a general function `Newton_plot(f, x, dfdx, epsilon=1E-7)` for this purpose. The first two arguments, `f` and `dfdx`, are Python functions representing the $f(x)$ function in the equation and its derivative $f'(x)$, respectively. Newton's method is run until $|f(x_N)| \leq \epsilon$, and the ϵ value is the third argument (`epsilon`). The `Newton_plot` function should make one plot of $(x_n)_{n=0}^N$ and $(f(x_n))_{n=0}^N$ on the screen and one save to a PNG file. (Hint: You can save quite some coding by calling the improved `Newton`

function from Chapter 5.1.9, which is available in the `Newton` module in `src/diffeq/Newton.py`.)

Demonstrate the function on the equation $x^6 \sin \pi x = 0$, with $\epsilon = 10^{-13}$. Try different starting values for Newton's method: $x_0 = -2.6, -1.2, 1.5, 1.7, 0.6$. Compare the results with the exact solutions $x = \dots, -2, -1, 0, 1, 2, \dots$. Name of program file: `Newton2.py`. \diamond

Exercise 5.14. *Implement the Secant method.*

Newton's method (5.29) for solving $f(x) = 0$ requires the derivative of the function $f(x)$. Sometimes this is difficult or inconvenient. The derivative can be approximated using the last two approximations to the root, x_{n-2} and x_{n-1} :

$$f'(x_{n-1}) \approx \frac{f(x_{n-1}) - f(x_{n-2})}{x_{n-1} - x_{n-2}}.$$

Using this approximation in (5.29) leads to the Secant method:

$$x_n = x_{n-1} - \frac{f(x_{n-1})(x_{n-1} - x_{n-2})}{f(x_{n-1}) - f(x_{n-2})}, \quad x_0, x_1 \text{ given.} \quad (5.52)$$

Here $n = 2, 3, \dots$. Make a program that applies the Secant method to solve $x^5 = \sin x$. Name of program file: `Secant.py`. \diamond

Exercise 5.15. *Test different methods for root finding.*

Make a program for solving $f(x) = 0$ by Newton's method (Chapter 5.1.9), the Bisection method (Chapter 3.6.2), and the Secant method (Exercise 5.14). For each method, the sequence of root approximations should be written out (nicely formatted) on the screen. Read $f(x)$, a , b , x_0 , and x_1 from the command line. Newton's method starts with x_0 , the Bisection method starts with the interval $[a, b]$, whereas the Secant method starts with x_0 and x_1 .

Run the program for each of the equations listed in Exercise 5.12. You should first plot the $f(x)$ functions as suggested in that exercise so you know how to choose x_0 , x_1 , a , and b in each case. Name of program file: `root_finder_examples.py`. \diamond

Exercise 5.16. *Difference equations for computing $\sin x$.*

The purpose of this exercise is to derive and implement difference equations for computing a Taylor polynomial approximation to $\sin x$, using the same ideas as in (5.18)–(5.19) for a Taylor polynomial approximation to e^x in Chapter 5.1.7.

The Taylor series for $\sin x$ is presented in Exercise 4.16, Equation (4.19) on page 228. To compute $S(x; n)$ efficiently, we try to compute a new term from the last computed term. Let $S(x; n) = \sum_{j=0}^n a_j$, where the expression for a term a_j follows from the formula (4.19). Derive the following relation between two consecutive terms in the series,

$$a_j = -\frac{x^2}{(2j+1)2j}a_{j-1}. \quad (5.53)$$

Introduce $s_j = S(x; j-1)$ and define $s_0 = 0$. We use s_j to accumulate terms in the sum. For the first term we have $a_0 = x$. Formulate a system of two difference equations for s_j and a_j in the spirit of (5.18)–(5.19). Implement this system in a function `S(x, n)`, which returns s_{n+1} and a_{n+1} . The latter is the first neglected term in the sum (since $s_{n+1} = \sum_{j=0}^n a_j$) and may act as a rough measure of the size of the error in the approximation. Suggest how to test that the `S(x, n)` function works correctly. Name of program file: `sin_Taylor_series_diffeq.py`.

◇

Exercise 5.17. *Difference equations for computing $\cos x$.*

Carry out the steps in Exercise 5.16, but do it for the Taylor series of $\cos x$ instead of $\sin x$ (look up the Taylor series for $\cos x$ in a mathematics textbook or search on the Internet). Name of program file: `cos_Taylor_series_diffeq.py`.

◇

Exercise 5.18. *Make a guitar-like sound.*

Given start values x_0, x_1, \dots, x_p , the following difference equation is known to create guitar-like sound:

$$x_n = \frac{1}{2}(x_{n-p} + x_{n-p-1}), \quad n = p+1, \dots, N. \quad (5.54)$$

With a sampling rate r , the frequency of this sound is given by r/p . Make a program with a function `solve(x, p)` which returns the solution array `x` of (5.54). To initialize the array `x[0:p+1]` we look at two methods, which can be implemented in two alternative functions:

1. $x_0 = 1, x_1 = x_2 = \dots = x_p = 0$
2. x_0, \dots, x_p are uniformly distributed random numbers in $[-1, 1]$

Import `max_amplitude`, `write`, and `play` from the `scitools.sound` module. Choose a sampling rate r and set $p = r/440$ to create a 440 Hz tone (A). Create an array `x1` of zeros with length $3r$ such that the tone will last for 3 seconds. Initialize `x1` according to method 1 above and solve (5.54). Multiply the `x1` array by `max_amplitude`. Repeat this process for an array `x2` of length $2r$, but use method 2 for the initial values and choose p such that the tone is 392 Hz (G). Concatenate `x1` and `x2`, call `write` and then `play` to play the sound. As you will experience, this sound is amazingly similar to the sound of a guitar string, first playing A for 3 seconds and then playing G for 2 seconds. (The method (5.54) is called the Karplus-Strong algorithm and was discovered in 1979 by a researcher, Kevin Karplus, and his student Alexander Strong, at Stanford University.) Name of program file: `guitar_sound.py`.

◇

Exercise 5.19. *Damp the bass in a sound file.*

Given a sequence x_0, \dots, x_{N-1} , the following *filter* transforms the sequence to a new sequence y_0, \dots, y_{N-1} :

$$y_n = \begin{cases} x_n, & n = 0 \\ -\frac{1}{4}(x_{n-1} - 2x_n + x_{n+1}), & 1 \leq n \leq N-2 \\ x_n, & n = N-1 \end{cases} \quad (5.55)$$

If x_n represents sound, y_n is the same sound but with the bass damped. Load some sound file (e.g., the one from Exercise 5.18) or call

```
x = scitools.sound.Nothing_Else_Matters(echo=True)
```

to get a sound sequence. Apply the filter (5.55) and play the resulting sound. Plot the first 300 values in the x_n and y_n signals to see graphically what the filter does with the signal. Name of program file: `damp_bass.py`. \diamond

Exercise 5.20. *Damp the treble in a sound file.*

Solve Exercise 5.19 to get some experience with coding a filter and trying it out on a sound. The purpose of this exercise is to explore some other filters that reduce the treble instead of the bass. Smoothing the sound signal will in general damp the treble, and smoothing is typically obtained by letting the values in the new filtered sound sequence be an average of the neighboring values in the original sequence.

The simplest smoothing filter can apply a standard average of three neighboring values:

$$y_n = \begin{cases} x_n, & n = 0 \\ \frac{1}{3}(x_{n-1} + x_n + x_{n+1}), & 1 \leq n \leq N-2 \\ x_n, & n = N-1 \end{cases} \quad (5.56)$$

Two other filters put less emphasis on the surrounding values:

$$y_n = \begin{cases} x_n, & n = 0 \\ \frac{1}{4}(x_{n-1} + 2x_n + x_{n+1}), & 1 \leq n \leq N-2 \\ x_n, & n = N-1 \end{cases} \quad (5.57)$$

$$y_n = \begin{cases} x_n, & n = 0 \\ \frac{1}{16}(x_{n-2} + 4x_{n-1} + 6x_n + 4x_{n+1} + x_{n+2}), & 1 \leq n \leq N-2 \\ x_n, & n = N-1 \end{cases} \quad (5.58)$$

Apply all these three filters to a sound file and listen to the result. Plot the first 300 values in the x_n and y_n signals for each of the three filters to see graphically what the filter does with the signal. Name of program file: `damp_treble.py`. \diamond

Exercise 5.21. *Demonstrate oscillatory solutions of (5.13).*

Modify the `growth_logistic.py` program from Chapter 5.1.5 to solve the equation (5.13) on page 244. Read the input parameters y_0 , q , and N from the command line.

Equation (5.13) has the solution $y_n = 1$ as $n \rightarrow \infty$. Demonstrate, by running the program, that this is the case when $y_0 = 0.3$, $q = 1$, and $N = 50$.

For larger q values, y_n does not approach a constant limit, but y_n oscillates instead around the limiting value. Such oscillations are sometimes observed in wildlife populations. Demonstrate oscillatory solutions when q is changed to 2 and 3.

It could happen that y_n stabilizes at a constant level for larger N . Demonstrate that this is not the case by running the program with $N = 1000$. Name of program file: `growth_logistic2.py`. \diamond

Exercise 5.22. *Make the program from Exer. 5.21 more flexible.*

It is tedious to run a program like the one from Exercise 5.21 repeatedly for a wide range of input parameters. A better approach is to let the computer do the manual work. Modify the program from Exercise 5.21 such that the computation of y_n and the plot is made in a function. Let the title in the plot contain the parameters y_0 and q (N is easily visible from the x axis). Also let the name of the hardcopy reflect the values of y_0 , q , and N . Then make loops over y_0 and q to perform the following more comprehensive set of experiments:

- $y = 0.01, 0.3$
- $q = 0.1, 1, 1.5, 1.8, 2, 2.5, 3$
- $N = 50$

How does the initial condition (the value y_0) seem to influence the solution?

The keyword argument `show=False` can be used in the plot call if you do not want all the plot windows to appear on the screen. Name of program file: `growth_logistic3.py`. \diamond

Exercise 5.23. *Simulate the price of wheat.*

The demand for wheat in year t is given by

$$D_t = ap_t + b,$$

where $a < 0$, > 0 , and p_t is the price of wheat. Let the supply of wheat be

$$S_t = Ap_{t-1} + B + \ln(1 + p_{t-1}),$$

where A and B are given constants. We assume that the price p_t adjusts such that all the produced wheat is sold. That is, $D_t = S_t$.

For $A = 1$, $a = -3$, $b = 5$, $B = 0$, find from numerical computations, a stable price such that the production of wheat from year to year is constant. That is, find p such that $ap + b = Ap + B + \ln(1 + p)$.

Assume that in a very dry year the production of wheat is much less than planned. Given that price this year, p_0 , is 4.5 and $D_t = S_t$, compute in a program how the prices p_1, p_2, \dots, p_N develop. This implies solving the difference equation

$$ap_t + b = Ap_{t-1} + B + \ln(1 + p_{t-1}).$$

From the p_t values, compute S_t and plot the points (p_t, S_t) for $t = 0, 1, 2, \dots, N$. How do the prices move when $N \rightarrow \infty$? Name of program file: `wheat.py`. \diamond