The collection $(x_i, z_i)_{i=0}^n$ is the discrete derivative of the discrete version $(x_i, f_i)_{i=0}^n$ of the continuous function $f(x)$. The program below, found in the file `diff_func.py`, takes f, a, b and n as input and computes the discrete derivative of f on the mesh implied by a, b, and h, and then a plot of f and the discrete derivative is made.

```
def diff(f, a, b, n):
    x = linspace(a, b, n+1)
    y = zeros(len(x))
    z = zeros(len(x))
    h = (b-a)/float(n)
    for i in xrange(len(x)):
        y[i] = func(x[i])
    for i in xrange(len(x)-1):
        z[i] = (y[i+1] - y[i])/h
    z[n] = (y[n] - y[n-1])/h
    return y, z

from scitools.std import *
f_formula = sys.argv[1]
a = eval(sys.argv[2])
b = eval(sys.argv[3])
n = int(sys.argv[4])

f = StringFunction(f_formula)
y, z = diff(f, a, b, n)
plot(x, y, 'r-', x, z, 'b-',
     legend=('function', 'derivative'))
```

A.3 Integration Becomes Summation

Some functions can be integrated analytically. You may remember[6] the following cases,

$$\int x^m dx = \frac{1}{m+1} x^{m+1} \text{ for } m \neq -1,$$

$$\int \sin(x) dx = -\cos(x),$$

$$\int \frac{x}{1+x^2} dx = \frac{1}{2} \ln\left(x^2+1\right).$$

These are examples of so-called indefinite integrals. If the function can be integrated analytically, it is straightforward to evaluate an associated definite integral. For example, we have[7]

[6] Actually, we congratulate you if you remember the third one!

[7] Recall, in general, that

$$[f(x)]_a^b = f(b) - f(a).$$

$$\int_0^1 x^m dx = \left[\frac{1}{m+1}x^{m+1}\right]_0^1 = \frac{1}{m+1},$$
$$\int_0^\pi \sin(x)dx = [-\cos(x)]_0^\pi = 2,$$
$$\int_0^1 \frac{x}{1+x^2}dx = \left[\frac{1}{2}\ln\left(x^2+1\right)\right]_0^1 = \frac{1}{2}\ln 2.$$

But lots of functions cannot be integrated analytically and therefore definite integrals must be computed using some sort of numerical approximation. Above, we introduced the discrete version of a function, and we will now use this construction to compute an approximation of a definite integral.

A.3.1 Dividing into Subintervals

Let us start by considering the problem of computing the integral of $\sin(x)$ from $x = 0$ to $x = \pi$. This is not the most exciting or challenging mathematical problem you can think of, but it is good practice to start with a problem you know well when you want to learn a new method. In Chapter A.1.1 we introduce a discrete function $(x_i, s_i)_{i=0}^n$ where $h = \pi/n, s_i = \sin(x_i)$ and $x_i = ih$ for $i = 0, 1, \ldots, n$. Furthermore, in the interval $x_k \leqslant x < x_{k+1}$, we defined the linear function

$$S_k(x) = s_k + \frac{s_{k+1} - s_k}{x_{k+1} - x_k}(x - x_k).$$

We want to compute an approximation of the integral of the function $\sin(x)$ from $x = 0$ to $x = \pi$. The integral

$$\int_0^\pi \sin(x)dx$$

can be divided into subintegrals defined on the intervals $x_k \leqslant x < x_{k+1}$, leading to the following sum of integrals:

$$\int_0^\pi \sin(x)dx = \sum_{k=0}^{n-1} \int_{x_k}^{x_{k+1}} \sin(x)dx\,.$$

To get a feeling for this split of the integral, let us spell the sum out in the case of only four subintervals. Then $n = 4$, $h = \pi/4$,

$$
\begin{aligned}
x_0 &= 0,\\
x_1 &= \pi/4,\\
x_2 &= \pi/2,\\
x_3 &= 3\pi/4\\
x_4 &= \pi.
\end{aligned}
$$

The interval from 0 to π is divided into four intervals of equal length, and we can divide the integral similarily,

$$
\begin{aligned}
\int_0^{\pi} \sin(x)dx = &\int_{x_0}^{x_1} \sin(x)dx + \int_{x_1}^{x_2} \sin(x)dx + \\
&\int_{x_2}^{x_3} \sin(x)dx + \int_{x_3}^{x_4} \sin(x)dx\,. \qquad \text{(A.10)}
\end{aligned}
$$

So far we have changed nothing – the integral can be split in this way – with no approximation at all. But we have reduced the problem of approximating the integral

$$\int_0^{\pi} \sin(x)dx$$

down to approximating integrals on the subintervals, i.e. we need approxiamtions of all the following integrals

$$\int_{x_0}^{x_1} \sin(x)dx, \int_{x_1}^{x_2} \sin(x)dx, \int_{x_2}^{x_3} \sin(x)dx, \int_{x_3}^{x_4} \sin(x)dx\,.$$

The idea is that the function to be integrated changes less over the subintervals than over the whole domain $[0, \pi]$ and it might be reasonable to approximate the sine by a straight line, $S_k(x)$, over each subinterval. The integration over a subinterval will then be very easy.

A.3.2 Integration on Subintervals

The task now is to approximate integrals on the form

$$\int_{x_k}^{x_{k+1}} \sin(x)dx.$$

Since

$$\sin(x) \approx S_k(x)$$

on the interval (x_k, x_{k+1}), we have

$$\int_{x_k}^{x_{k+1}} \sin(x)dx \approx \int_{x_k}^{x_{k+1}} S_k(x)dx.$$

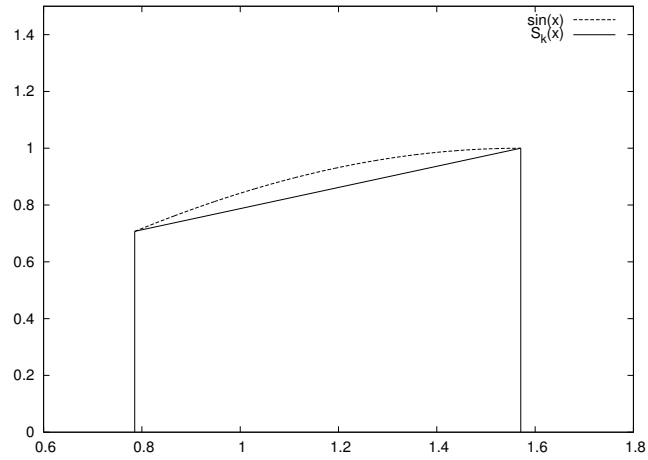


Fig. A.3 $S_k(x)$ and $\sin(x)$ on the interval (x_k, x_{k+1}) for $k = 1$ and $n = 4$.

In Figure A.3 we have graphed $S_k(x)$ and $\sin(x)$ on the interval (x_k, x_{k+1}) for $k = 1$ in the case of $n = 4$. We note that the integral of $S_1(x)$ on this interval equals the area of a trapezoid, and thus we have

$$\int_{x_1}^{x_2} S_1(x)dx = \frac{1}{2}\left(S_1(x_2) + S_1(x_1)\right)(x_2 - x_1),$$

so

$$\int_{x_1}^{x_2} S_1(x)dx = \frac{h}{2}\left(s_2 + s_1\right),$$

and in general we have

$$\begin{aligned}\int_{x_k}^{x_{k+1}} \sin(x)dx &\approx \frac{1}{2}\left(s_{k+1} + s_k\right)(x_{k+1} - x_k) \\ &= \frac{h}{2}\left(s_{k+1} + s_k\right).\end{aligned}$$

A.3.3 Adding the Subintervals

By adding the contributions from each subinterval, we get

$$\begin{aligned}\int_0^{\pi} \sin(x)dx &= \sum_{k=0}^{n-1} \int_{x_k}^{x_{k+1}} \sin(x)dx \\ &\approx \sum_{k=0}^{n-1} \frac{h}{2}\left(s_{k+1} + s_k\right),\end{aligned}$$

so

$$\int_0^{\pi} \sin(x)dx \approx \frac{h}{2}\sum_{k=0}^{n-1}\left(s_{k+1} + s_k\right). \tag{A.11}$$

In the case of $n = 4$, we have

$$\int_0^\pi \sin(x)dx \approx \frac{h}{2}\left[(s_1 + s_0) + (s_2 + s_1) + (s_3 + s_2) + (s_4 + s_3)\right]$$
$$= \frac{h}{2}\left[s_0 + 2\left(s_{1+}s_{2+}s_3\right) + s_4\right] .$$

One can show that (A.11) can be alternatively expressed as[8]

$$\int_0^\pi \sin(x)dx \approx \frac{h}{2}\left[s_0 + 2\sum_{k=1}^{n-1} s_k + s_n\right] . \tag{A.12}$$

This approximation formula is referred to as the Trapezoidal rule of numerical integration. Using the more general program `trapezoidal.py`, presented in the next section, on integrating $\int_0^\pi \sin(x)dx$ with $n = 5, 10, 20$ and 100 yields the numbers 1.5644, 1.8864, 1.9713, and 1.9998 respectively. These numbers are to be compared to the exact value 2. As usual, the approximation becomes better the more points (n) we use.

A.3.4 Generalization

An approximation of the integral

$$\int_a^b f(x)dx$$

can be computed using the discrete version of a continuous function $f(x)$ defined on an interval $[a, b]$. We recall that the discrete version of f is given by $(x_i, y_i)_{i=0}^n$ where

$$x_i = a + ih, \text{ and } y_i = f(x_i)$$

for $i = 0, 1, \ldots, n$. Here, $n \geqslant 1$ is a given integer and $h = (b - a)/n$. The Trapezoidal rule can now be written as

$$\int_a^b f(x)dx \approx \frac{h}{2}\left[y_0 + 2\sum_{k=1}^{n-1} y_k + y_n\right] .$$

The program `trapezoidal.py` implements the Trapezoidal rule for a general function f.

```
def trapezoidal(f, a, b, n):
    h = (b-a)/float(n)
    I = f(a) + f(b)
    for k in xrange(1, n, 1):
```

[8] There are fewer arithmetic operations associated with (A.12) than with (A.11), so the former will lead to faster code.

```
        x = a + k*h
        I += 2*f(x)
    I *= h/2
    return I


from math import *
from scitools.StringFunction import StringFunction
import sys

def test(argv=sys.argv):
    f_formula = argv[1]
    a = eval(argv[2])
    b = eval(argv[3])
    n = int(argv[4])

    f = StringFunction(f_formula)
    I = trapezoidal(f, a, b, n)
    print 'Approximation of the integral: ', I

if __name__ == '__main__':
    test()
```

We have made the file as module such that you can easily import the `trapezoidal` function in another program. Let us do that: We make a table of how the approximation and the associated error of an integral are reduced as n is increased. For this purpose, we want to integrate $\int_{t_1}^{t_2} g(t)dt$, where

$$g(t) = -ae^{-at}\sin(\pi wt) + \pi we^{-at}\cos(\pi wt)\,.$$

The exact integral $G(t) = \int g(t)dt$ equals

$$G(t) = e^{-at}\sin(\pi wt)\,.$$

Here, a and w are real numbers that we set to $1/2$ and 1, respectively, in the program. The integration limits are chosen as $t_1 = 0$ and $t_2 = 4$. The integral then equals zero. The program and its output appear below.

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

def g(t):
    return -a*exp(-a*t)*sin(pi*w*t) + pi*w*exp(-a*t)*cos(pi*w*t)

def G(t):  # integral of g(t)
    return exp(-a*t)*sin(pi*w*t)

a = 0.5
w = 1.0
t1 = 0
t2 = 4
exact = G(t2) - G(t1)
for n in 2, 4, 8, 16, 32, 64, 128, 256, 512:
    approx = trapezoidal(g, t1, t2, n)
    print 'n=%3d approximation=%12.5e  error=%12.5e' % \
          (n, approx, exact-approx)
```

```
n=  2 approximation= 5.87822e+00  error=-5.87822e+00
n=  4 approximation= 3.32652e-01  error=-3.32652e-01
n=  8 approximation= 6.15345e-02  error=-6.15345e-02
n= 16 approximation= 1.44376e-02  error=-1.44376e-02
n= 32 approximation= 3.55482e-03  error=-3.55482e-03
n= 64 approximation= 8.85362e-04  error=-8.85362e-04
n=128 approximation= 2.21132e-04  error=-2.21132e-04
n=256 approximation= 5.52701e-05  error=-5.52701e-05
n=512 approximation= 1.38167e-05  error=-1.38167e-05
```

We see that the error is reduced as we increase n. In fact, as n is doubled we realize that the error is roughly reduced by a factor of 4, at least when $n > 8$. This is an important property of the Trapezoidal rule, and checking that a program reproduces this property is an important check of the validity of the implementation.

A.4 Taylor Series

The single most important mathematical tool in computational science is the Taylor series. It is used to derive new methods and also for the analysis of the accuracy of approximations. We will use the series many times in this text. Right here, we just introduce it and present a few applications.

A.4.1 Approximating Functions Close to One Point

Suppose you know the value of a function f at some point x_0, and you are interested in the value of f close to x. More precisely, suppose we know $f(x_0)$ and we want an approximation of $f(x_0 + h)$ where h is a small number. If the function is smooth and h is really small, our first approximation reads

$$f(x_0 + h) \approx f(x_0). \tag{A.13}$$

That approximation is, of course, not very accurate. In order to derive a more accurate approximation, we have to know more about f at x_0. Suppose that we know the value of $f(x_0)$ and $f'(x_0)$, then we can find a better approximation of $f(x_0 + h)$ by recalling that

$$f'(x_0) \approx \frac{f(x_0 + h) - f(x_0)}{h}.$$

Hence, we have

$$f(x_0 + h) \approx f(x_0) + hf'(x_0). \tag{A.14}$$

A.4.2 Approximating the Exponential Function

Let us be a bit more specific and consider the case of

$$f(x) = e^x$$

around

$$x_0 = 0.$$

Since $f'(x) = e^x$, we have $f'(0) = 1$, and then it follows from $(A.14)$ that

$$e^h \approx 1 + h.$$

The little program below (found in `taylor1.py`) prints e^h and $1+h$ for a range of h values.

```
from math import exp
for h in 1, 0.5, 1/20.0, 1/100.0, 1/1000.0:
    print 'h=%8.6f exp(h)=%11.5e  1+h=%g' % (h, exp(h), 1+h)
```

```
h=1.000000 exp(h)=2.71828e+00  1+h=2
h=0.500000 exp(h)=1.64872e+00  1+h=1.5
h=0.050000 exp(h)=1.05127e+00  1+h=1.05
h=0.010000 exp(h)=1.01005e+00  1+h=1.01
h=0.001000 exp(h)=1.00100e+00  1+h=1.001
```

As expected, $1 + h$ is a good approximation to e^h the smaller h is.

A.4.3 More Accurate Expansions

The approximations given by $(A.13)$ and $(A.14)$ are referred to as Taylor series. You can read much more about Taylor series in any Calculus book. More specifically, (A.13) and (A.14) are known as the zeroth- and first-order Taylor series, respectively. The second-order Taylor series is given by

$$f(x_0 + h) \approx f(x_0) + hf'(x_0) + \frac{h^2}{2} f''(x_0), \tag{A.15}$$

the third-order series is given by

$$f(x_0 + h) \approx f(x_0) + hf'(x_0) + \frac{h^2}{2} f''(x_0) + \frac{h^3}{6} f'''(x_0), \tag{A.16}$$

and the fourth-order series reads

$$f(x_0 + h) \approx f(x_0) + hf'(x_0) + \frac{h^2}{2} f''(x_0) + \frac{h^3}{6} f'''(x_0) + \frac{h^4}{24} f''''(x_0). \tag{A.17}$$

In general, the n-th order Taylor series is given by

$$f(x_0 + h) \approx \sum_{k=0}^{n} \frac{h^k}{k!} f^{(k)}(x_0), \tag{A.18}$$

where we recall that $f^{(k)}$ denotes the $k-th$ derivative of f, and

$$k! = 1 \cdot 2 \cdot 3 \cdot 4 \cdots (k-1) \cdot k$$

is the factorial (cf. Exercise 2.33). By again considering $f(x) = e^x$ and $x_0 = 0$, we have

$$f(x_0) = f'(x_0) = f''(x_0) = f'''(x_0) = f''''(x_0) = 1$$

which gives the following Taylor series:

$$\begin{aligned}
e^h &\approx 1, && \text{zeroth-order,}\\
e^h &\approx 1 + h, && \text{first-order,}\\
e^h &\approx 1 + h + \tfrac{1}{2}h^2, && \text{second-order,}\\
e^h &\approx 1 + h + \tfrac{1}{2}h^2 + \tfrac{1}{6}h^3, && \text{third-order,}\\
e^h &\approx 1 + h + \tfrac{1}{2}h^2 + \tfrac{1}{6}h^3 + \tfrac{1}{24}h^4, && \text{fourth-order.}
\end{aligned}$$

The program below, called `taylor2.py`, prints the error of these approximations for a given value of h (note that we can easily build up a Taylor series in a list by adding a new term to the last computed term in the list).

```
from math import exp
import sys
h = float(sys.argv[1])

Taylor_series = []
Taylor_series.append(1)
Taylor_series.append(Taylor_series[-1] + h)
Taylor_series.append(Taylor_series[-1] + (1/2.0)*h**2)
Taylor_series.append(Taylor_series[-1] + (1/6.0)*h**3)
Taylor_series.append(Taylor_series[-1] + (1/24.0)*h**4)

print 'h =', h
for order in range(len(Taylor_series)):
    print 'order=%d, error=%g' % \
          (order, exp(h) - Taylor_series[order])
```

By running the program with $h = 0.2$, we have the following output:

```
h = 0.2
order=0, error=0.221403
order=1, error=0.0214028
order=2, error=0.00140276
order=3, error=6.94248e-05
order=4, error=2.75816e-06
```

We see how much the approximation is improved by adding more terms. For $h = 3$ all these approximations are useless:

```
h = 3.0
order=0, error=19.0855
order=1, error=16.0855
order=2, error=11.5855
order=3, error=7.08554
order=4, error=3.71054
```

However, by adding more terms we can get accurate results for any h. The method from Chapter 5.1.7 computes the Taylor series for e^x with n terms in general. Running the associated program `exp_Taylor_series_diffeq.py` for various values of h shows how much is gained by adding more terms to the Taylor series. For $h = 3$,

$e^3 = 20.086$ and we have

$n+1$	Taylor series
2	4
4	13
8	19.846
16	20.086

For $h = 50$, $e^{50} = 5.1847 \cdot 10^{21}$ and we have

$n+1$	Taylor series
2	51
4	$2.2134 \cdot 10^4$
8	$1.7960 \cdot 10^8$
16	$3.2964 \cdot 10^{13}$
32	$1.3928 \cdot 10^{19}$
64	$5.0196 \cdot 10^{21}$
128	$5.1847 \cdot 10^{21}$

Here, the evolution of the series as more terms are added is quite dramatic (and impressive!).

A.4.4 Accuracy of the Approximation

Recall that the Taylor series is given by

$$f(x_0 + h) \approx \sum_{k=0}^{n} \frac{h^k}{k!} f^{(k)}(x_0). \tag{A.19}$$

This can be rewritten as an equality by introducing an error term,

$$f(x_0 + h) = \sum_{k=0}^{n} \frac{h^k}{k!} f^{(k)}(x_0) + O(h^{n+1}). \tag{A.20}$$

Let's look a bit closer at this for $f(x) = e^x$. In the case of $n = 1$, we have

$$e^h = 1 + h + O(h^2). \tag{A.21}$$

This means that there is a constant c that does not depend on h such that

$$\left| e^h - (1 + h) \right| \leqslant ch^2, \tag{A.22}$$

so the error is reduced quadratically in h. This means that if we compute the fraction

$$q_h^1 = \frac{\left| e^h - (1 + h) \right|}{h^2},$$

we expect it to be bounded as h is reduced. The program `taylor_err1.py` prints q_h^1 for $h = 1/10, 1/20, 1/100$ and $1/1000$.

```
from numpy import exp, abs

def q_h(h):
    return abs(exp(h) - (1+h))/h**2
```

```
print "  h     q_h"
for h in 0.1, 0.05, 0.01, 0.001:
    print "%5.3f %f" %(h, q_h(h))
```

We can run the program and watch the output:

Terminal

```
taylor_err1.py
  h     q_h
0.100 0.517092
0.050 0.508439
0.010 0.501671
0.001 0.500167
```

We observe that $q_h \approx 1/2$ and it is definitely bounded independent of h. We can now rewrite all the approximations of e^h defined above in term of equalities:

$$
\begin{aligned}
e^h &= 1 + O(h), && \text{zeroth-order},\\
e^h &= 1 + h + O(h^2), && \text{first-order},\\
e^h &= 1 + h + \tfrac{1}{2}h^2 + O(h^3), && \text{second-order},\\
e^h &= 1 + h + \tfrac{1}{2}h^2 + \tfrac{1}{6}h^3 + O(h^4), && \text{third-order},\\
e^h &= 1 + h + \tfrac{1}{2}h^2 + \tfrac{1}{6}h^3 + \tfrac{1}{24}h^4 + O(h^5), && \text{fourth-order}.
\end{aligned}
$$

The program `taylor_err2.py` prints

$$
\begin{aligned}
q_h^0 &= \frac{\left|e^h - 1\right|}{h},\\
q_h^1 &= \frac{\left|e^h - (1+h)\right|}{h^2},\\
q_h^2 &= \frac{\left|e^h - \left(1 + h + \frac{h^2}{2}\right)\right|}{h^3},\\
q_h^3 &= \frac{\left|e^h - \left(1 + h + \frac{h^2}{2} + \frac{h^3}{6}\right)\right|}{h^4},\\
q_h^4 &= \frac{\left|e^h - \left(1 + h + \frac{h^2}{2} + \frac{h^3}{6} + \frac{h^4}{24}\right)\right|}{h^5},
\end{aligned}
$$

for $h = 1/5, 1/10, 1/20$ and $1/100$.

```
from numpy import exp, abs

def q_0(h):
    return abs(exp(h) - 1) / h
def q_1(h):
    return abs(exp(h) - (1 + h)) / h**2
def q_2(h):
    return abs(exp(h) - (1 + h + (1/2.0)*h**2)) / h**3
def q_3(h):
    return abs(exp(h) - (1 + h + (1/2.0)*h**2 + \
                         (1/6.0)*h**3)) / h**4
def q_4(h):
    return abs(exp(h) - (1 + h + (1/2.0)*h**2 + (1/6.0)*h**3 + \
```

```
                          (1/24.0)*h**4)) / h**5
hlist = [0.2, 0.1, 0.05, 0.01]
print "%-05s %-09s %-09s %-09s %-09s %-09s" \
      %("h", "q_0", "q_1", "q_2", "q_3", "q_4")
for h in hlist:
    print "%.02f  %04f  %04f  %04f  %04f  %04f" \
          %(h, q_0(h), q_1(h), q_2(h), q_3(h), q_4(h))
```

By using the program, we get the following table:

```
h     q_0       q_1       q_2       q_3       q_4
0.20  1.107014  0.535069  0.175345  0.043391  0.008619
0.10  1.051709  0.517092  0.170918  0.042514  0.008474
0.05  1.025422  0.508439  0.168771  0.042087  0.008403
0.01  1.005017  0.501671  0.167084  0.041750  0.008344
```

Again we observe that the error of the approximation behaves as indicated in $(A.20)$.

A.4.5 Derivatives Revisited

We observed aboved that

$$f'(x) \approx \frac{f(x+h) - f(x)}{h}.$$

By using the Taylor series, we can obtain this approximation directly, and also get an indication of the error of the approximation. From $(A.20)$ it follows that

$$f(x+h) = f(x) + hf'(x) + O(h^2),$$

and thus

$$f'(x) = \frac{f(x+h) - f(x)}{h} + O(h), \tag{A.23}$$

so the error is proportional to h. We can investigate if this is the case through some computer experiments. Take $f(x) = \ln(x)$, so that $f'(x) = 1/x$. The program `diff_ln_err.py` prints h and

$$\frac{1}{h}\left| f'(x) - \frac{f(x+h) - f(x)}{h} \right| \tag{A.24}$$

at $x = 10$ for a range of h values.

```
def error(h):
    return (1.0/h)*abs(df(x) - (f(x+h)-f(x))/h)

from math import log as ln

def f(x):
   return ln(x)

def df(x):
    return 1.0/x

x = 10
hlist = []
```

```
for h in 0.2, 0.1, 0.05, 0.01, 0.001:
    print "%.4f   %4f" % (h, error(h))
```

From the output

```
0.2000   0.004934
0.1000   0.004967
0.0500   0.004983
0.0100   0.004997
0.0010   0.005000
```

we observe that the quantity in (A.24) is constant (≈ 0.5) independent of h, which indicates that the error is proportional to h.

A.4.6 More Accurate Difference Approximations

We can also use the Taylor series to derive more accurate approximations of the derivatives. From $(A.20)$, we have

$$f(x+h) \approx f(x) + hf'(x) + \frac{h^2}{2}f''(x) + O(h^3). \tag{A.25}$$

By using $-h$ insted of h, we get

$$f(x-h) \approx f(x) - hf'(x) + \frac{h^2}{2}f''(x) + O(h^3). \tag{A.26}$$

By subtracting $(A.26)$ from $(A.25)$, we have

$$f(x+h) - f(x-h) = 2hf'(x) + O(h^3),$$

and consequently

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + O(h^2). \tag{A.27}$$

Note that the error is now $O(h^2)$ whereas the error term of $(A.23)$ is $O(h)$. In order to see if the error is actually reduced, let us compare the following two approximations

$$f'(x) \approx \frac{f(x+h) - f(x)}{h} \text{ and } f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

by applying them to the discrete version of $\sin(x)$ on the interval $(0, \pi)$. As usual, we let $n \geqslant 1$ be a given integer, and define the mesh

$$x_i = ih \text{ for } i = 0, 1, \ldots, n,$$

where $h = \pi/n$. At the nodes, we have the functional values

$$s_i = \sin(x_i) \text{ for } i = 0, 1, \ldots, n,$$

and at the inner nodes we define the first (F) and second (S) order approximations of the derivatives given by

$$d_i^F = \frac{s_{i+1} - s_i}{h},$$

and

$$d_i^S = \frac{s_{i+1} - s_{i-1}}{2h},$$

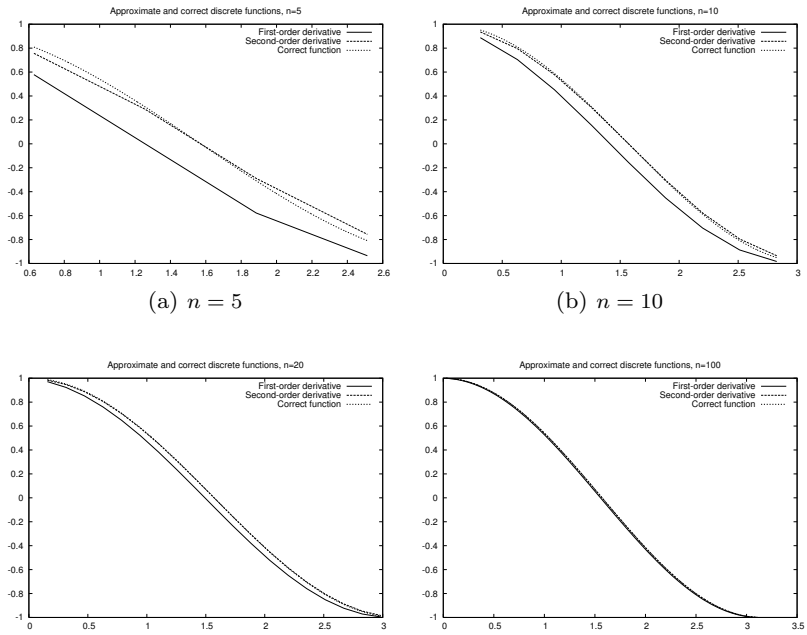
respectively for $i = 1, 2, \ldots, n-1$. These values should be compared to the exact derivative given by

$$d_i = \cos(x_i) \text{ for } i = 1, 2, \ldots, n-1.$$

The following program, found in `diff_1st2nd_order.py`, plots the discrete functions $(x_i, d_i)_{i=1}^{n-1}$, $(x_i, d_i^F)_{i=1}^{n-1}$, and $(x_i, d_i^S)_{i=1}^{n-1}$ for a given n. Note that the first three functions in this program are completely general in that they can be used for any $f(x)$ on any mesh. The special case of $f(x) = \sin(x)$ and comparing first- and second-order formulas is implemented in the `example` function. This latter function is called in the test block of the file. That is, the file is a module and we can reuse the first three functions in other programs (in particular, we can use the third function in the next example).

```
def first_order(f, x, h):
    return (f(x+h) - f(x))/h

def second_order(f, x, h):
    return (f(x+h) - f(x-h))/(2*h)

def derivative_on_mesh(formula, f, a, b, n):
    """
    Differentiate f(x) at all internal points in a mesh
    on [a,b] with n+1 equally spaced points.
    The differentiation formula is given by formula(f, x, h).
    """
    h = (b-a)/float(n)
    x = linspace(a, b, n+1)
    df = zeros(len(x))
    for i in xrange(1, len(x)-1):
        df[i] = formula(f, x[i], h)
    # return x and values at internal points only
    return x[1:-1], df[1:-1]

def example(n):
    a = 0; b = pi;
    x, dF = derivative_on_mesh(first_order,  sin, a, b, n)
    x, dS = derivative_on_mesh(second_order, sin, a, b, n)
    # accurate plot of the exact derivative at internal points:
    h = (b-a)/float(n)
    xfine = linspace(a+h, b-h, 1001)
    exact = cos(xfine)
    plot(x, dF, 'r-', x, dS, 'b-', xfine, exact, 'y-',
         legend=('First-order derivative',
                 'Second-order derivative',
                 'Correct function'),
         title='Approximate and correct discrete '\
               'functions, n=%d' % n)

# main program:
from scitools.std import *
try:
    n = int(sys.argv[1])
```

```
except:
    print "usage: %s n" %sys.argv[0]
    sys.exit(1)

example(n)
```



(a) $n = 5$ (b) $n = 10$

(c) $n = 20$ (d) $n = 100$

Fig. A.4 Plots of exact and approximate derivatives with various number of mesh points n.

The result of running the program with four different n values is presented in Figure A.4. Observe that d_i^S is a better approximation to d_i than d_i^F, and note that both approximations become very good as n is getting large.

A.4.7 Second-Order Derivatives

We have seen that the Taylor series can be used to derive approximations of the derivative. But what about higher order derivatives? Next we shall look at second order derivatives. From $(A.20)$ we have

$$f(x_0 + h) = f(x_0) + hf'(x_0) + \frac{h^2}{2}f''(x_0) + \frac{h^3}{6}f'''(x_0) + O(h^4),$$

and by using $-h$, we have

$$f(x_0 - h) = f(x_0) - hf'(x_0) + \frac{h^2}{2}f''(x_0) - \frac{h^3}{6}f'''(x_0) + O(h^4)$$

By adding these equations, we have

$$f(x_0 + h) + f(x_0 - h) = 2f(x_0) + h^2 f''(x_0) + O(h^4),$$

and thus

$$f''(x_0) = \frac{f(x_0 - h) - 2f(x_0) + f(x_0 + h)}{h^2} + O(h^2). \tag{A.28}$$

For a discrete function $(x_i, y_i)_{i=0}^n$, $y_i = f(x_i)$, we can define the following approximation of the second derivative,

$$d_i = \frac{y_{i-1} - 2y_i + y_{i+1}}{h^2}\,. \tag{A.29}$$

We can make a function, found in the file `diff2nd.py`, that evaluates (A.29) on a mesh. As an example, we apply the function to

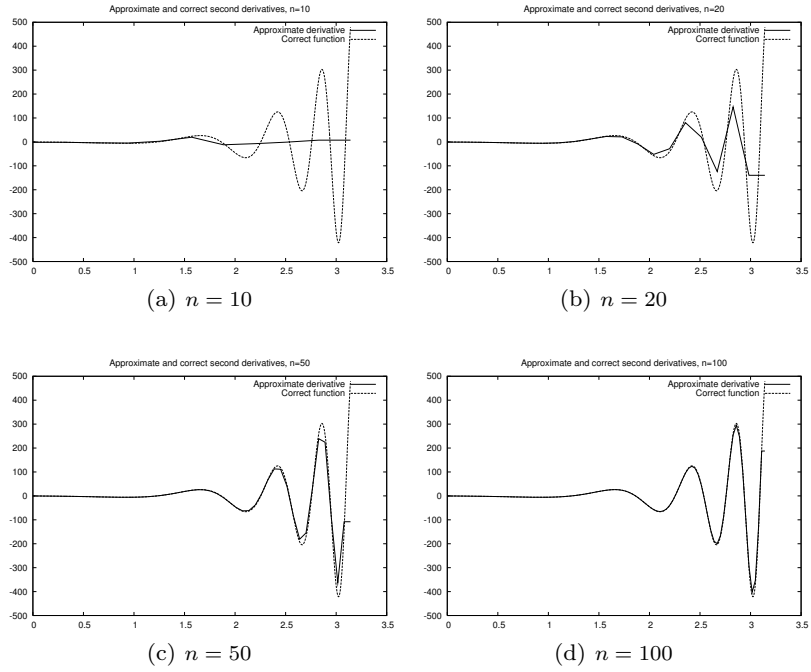
$$f(x) = \sin(e^x),$$

where the exact second-order derivative is given by

$$f''(x) = e^x \cos\left(e^x\right) - \left(\sin\left(e^x\right)\right) e^{2x}\,.$$

```
from diff_1st2nd_order import derivative_on_mesh
from scitools.std import *

def diff2nd(f, x, h):
    return (f(x+h) - 2*f(x) + f(x-h))/(h**2)

def example(n):
    a = 0;  b = pi

    def f(x):
        return sin(exp(x))

    def exact_d2f(x):
        e_x = exp(x)
        return e_x*cos(e_x) - sin(e_x)*exp(2*x)

    x, d2f = derivative_on_mesh(diff2nd, f, a, b, n)
    h = (b-a)/float(n)
    xfine = linspace(a+h, b-h, 1001)  # fine mesh for comparison
    exact = exact_d2f(xfine)
    plot(x, d2f, 'r-', xfine, exact, 'b-',
         legend=('Approximate derivative',
                 'Correct function'),
         title='Approximate and correct second order '\
               'derivatives, n=%d' % n,
         hardcopy='tmp.eps')

n = int(sys.argv[1])

example(n)
```

In Figure A.5 we compare the exact and the approximate derivatives for $n = 10, 20, 50$, and 100. As usual, the error descreases when n becomes larger, but note here that the error is very large for small values of n.



(a) $n = 10$ (b) $n = 20$

(c) $n = 50$ (d) $n = 100$

Fig. A.5 Plots of exact and approximate second-order derivatives with various mesh resolution n.

A.5 Exercises

Exercise A.1. *Interpolate a discrete function.*

In a Python function, represent the mathematical function

$$f(x) = \exp(-x^2)\cos(2\pi x)$$

on a mesh consisting of $q + 1$ equally spaced points on $[-1, 1]$, and return 1) the interpolated function value at $x = -0.45$ and 2) the error in the interpolated value. Call the function and write out the error for $q = 2, 4, 8, 16$. Name of program file: `interpolate_exp_cos.py` ◇

Exercise A.2. *Study a function for different parameter values.*

Develop a program that creates a plot of the function $f(x) = \sin(\frac{1}{x+\varepsilon})$ for x in the unit interval, where $\varepsilon > 0$ is a given input parameter. Use $n + 1$ nodes in the plot.

(a) Test the program using $n = 10$ and $\varepsilon = 1/5$.

(b) Refine the program such that it plots the function for two values of n; say n and $n + 10$.

(c) How large do you have to choose n in order for the difference between these two functions to be less than 0.1? Hint: Each function gives an array. Create a `while` loop and use the `max` function of the arrays to retrieve the maximum value and compare these.

(d) Let $\varepsilon = 1/10$, and repeat (c).
(e) Let $\varepsilon = 1/20$, and repeat (c).
(f) Try to find a formula for how large n needs to be for a given value of ε such that increasing n further does not change the plot so much that it is visible on the screen. Note that there is no exact answer to this question.

Name of program file: `plot_sin_eps.py` ◇

Exercise A.3. *Study a function and its derivative.*
Consider the function

$$f(x) = \sin\left(\frac{1}{x+\varepsilon}\right)$$

for x ranging from 0 to 1, and the derivative

$$f'(x) = \frac{-\cos\left(\frac{1}{x+\varepsilon}\right)}{(x+\varepsilon)^2}.$$

Here, ε is a given input parameter.

(a) Develop a program that creates a plot of the derivative of $f = f(x)$ based on a finite difference approximation using n computational nodes. The program should also graph the exact derivative given by $f' = f'(x)$ above.
(b) Test the program using $n = 10$ and $\varepsilon = 1/5$.
(c) How large do you have to choose n in order for the difference between these two functions to be less than 0.1? Hint: Each function gives an array. Create a `while` loop and use the `max` function of the arrays to retrieve the maximum value and compare these.
(d) Let $\varepsilon = 1/10$, and repeat (c).
(e) Let $\varepsilon = 1/20$, and repeat (c).
(f) Try determine experimentally how large n needs to be for a given value of ε such that increasing n further does not change the plot so much that you can view it on the screen. Note, again, that there is no exact solution to this problem.

Name of program file: `sin_deriv.py` ◇

Exercise A.4. *Use the Trapezoidal method.*
The purpose of this excercise is to test the program `trapezoidal.py`.

(a) Let

$$\overline{a} = \int_0^1 e^{4x} dx = \frac{1}{4}e^4 - \frac{1}{4}.$$

Compute the integral using the program `trapezoidal.py` and, for a given n, let $a(n)$ denote the result. Try to find, experimentally, how large you have to choose n in order for

$$|\overline{a} - a(n)| \leqslant \varepsilon$$

where $\varepsilon = 1/100$.

(b) Repeat (a) with $\varepsilon = 1/1000$.

(c) Repeat (a) with $\varepsilon = 1/10000$.

(d) Try to figure out, in general, how large n has to be in order for

$$|\overline{a} - a(n)| \leqslant \varepsilon$$

for a given value of ε.

Name of program file: `trapezoidal_test_exp.py` ◇

Exercise A.5. *Compute a sequence of integrals.*

(a) Let

$$\overline{b}_k = \int_0^1 x^k dx = \frac{1}{k+1},$$

and let $b_k(n)$ denote the result of using the program `trapezoidal.py` to compute $\int_0^1 x^k dx$. For $k = 4, 6$ and 8, try to figure out, by doing numerical experiments, how large n needs to be in order for $b_k(n)$ to satisfy

$$\left|\overline{b}_k - b_k(n)\right| \leqslant 0.0001.$$

Note that n will depend on k. Hint: Run the program for each k, look at the output, and calculate $\left|\overline{b}_k - b_k(n)\right|$ manually.

(b) Try to generalize the result in (a) to arbitrary $k \geqslant 2$.

(c) Generate a plot of x^k on the unit interval for $k = 2, 4, 6, 8$, and 10, and try to figure out if the results obtained in (a) and (b) are reasonable taking into account that the program `trapezoidal.py` was developed using a piecewise linear approximation of the function.

Name of program file: `trapezoidal_test_power.py` ◇

Exercise A.6. *Use the Trapezoidal method.*

The purpose of this excercise is to compute an approximation of the integral[9]

$$I = \int_{-\infty}^{\infty} e^{-x^2} dx$$

using the Trapezoidal method.

(a) Plot the function e^{-x^2} for x ranging from -10 to 10 and use the plot to argue that

$$\int_{-\infty}^{\infty} e^{-x^2} dx = 2 \int_0^{\infty} e^{-x^2} dx.$$

(b) Let $T(n, L)$ be the approximation of the integral

[9] You may consult your Calculus book to verify that the exact solution is $\sqrt{\pi}$.

$$2\int_0^L e^{-x^2}dx$$

computed by the Trapezoidal method using n computational points. Develop a program that computes the value of T for a given n and L.

(c) Extend the program developed in (b) to write out values of $T(n, L)$ in a table with rows corresponding to $n = 100, 200, \ldots, 500$ and columns corresponding to $L = 2, 4, 6, 8, 10$.

(d) Extend the program to also print a table of the errors in $T(n, L)$ for the same n and L values as in (c). The exact value of the integral is $\sqrt{\pi}$.

Comment. Numerical integration of integrals with finite limits requires a choice of n, while with infinite limits we also need to truncate the domain, i.e., choose L in the present example. The accuracy depends on both n and L. Name of program file: `integrate_exp.py` ◇

Exercise A.7. *Trigonometric integrals.*

The purpose of this excercise is to demonstrate a property of trigonometric functions that you will meet in later courses. In this excercise, you may compute the integrals using the program `trapezoidal.py` with $n = 100$.

(a) Consider the integrals

$$I_{p,q} = 2\int_0^1 \sin(p\pi x)\sin(q\pi x)dx$$

and fill in values of the integral $I_{p,q}$ in a table with rows corresponding to $q = 0, 1, \ldots, 4$ and columns corresponding to $p = 0, 1, \ldots, 4$.

(b) Repeat (a) for the integrals

$$I_{p,q} = 2\int_0^1 \cos(p\pi x)\cos(q\pi x)dx.$$

(c) Repeat (a) for the integrals

$$I_{p,q} = 2\int_0^1 \cos(p\pi x)\sin(q\pi x)dx.$$

Name of program file: `ortho_trig_funcs.py` ◇

Exercise A.8. *Plot functions and their derivatives.*

(a) Use the program `diff_func.py` to plot approximations of the derivative for the following functions defined on the interval ranging from $x = 1/1000$ to $x = 1$:

$$f(x) = \ln\left(x + \frac{1}{100}\right),$$
$$g(x) = \cos(e^{10x}),$$
$$h(x) = x^x.$$

(b) Extend the program such that both the discrete approximation and the correct (analytical) derivative can be plotted. The analytical derivative should be evaluated in the same computational points as the numerical approximation. Test the program by comparing the discrete and analytical derivative of x^3.

(c) Use the program develped in (b) to compare the analytical and discrete derivatives of the functions given in (a). How large do you have to choose n in each case in order for the plots to become indistinguishable on your screen. Note that the analytical derivatives are given by:

$$f'(x) = \frac{1}{x + \frac{1}{100}},$$
$$g'(x) = -10e^{10x}\sin\left(e^{10x}\right)$$
$$h'(x) = (\ln x)\, x^x + x x^{x-1}$$

Name of program file: `diff_functions.py` ◇

Exercise A.9. *Use the Trapezoidal method.*

Develop an efficient program that creates a plot of the function

$$f(x) = \frac{1}{2} + \frac{1}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$$

for $x \in [0, 10]$. The integral should be approximated using the Trapezoidal method and use as few function evaluations of e^{-t^2} as possible.
Name of program file: `plot_integral.py` ◇