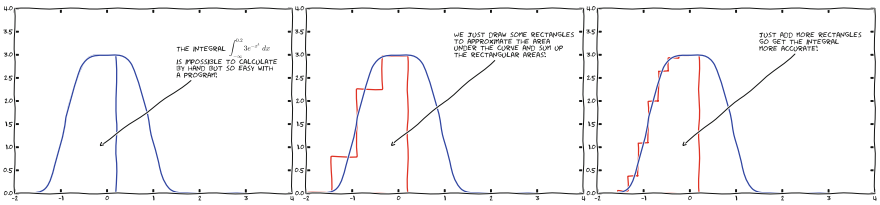
Computing Integrals and Testing Code

6



We now turn our prime attention to the solving of mathematical problems through computer programming. A fundamentally important part of programming, is to check that the written code works as intended. That is, the code must be *tested*. This far, we have checked our coding in rather simple ways, e.g., by comparing to hand calculations. Now, we will look at more powerful test strategies, while addressing numerical computation of integrals.

There are many reasons to choose integration as our first application. Integration is well known already from high school mathematics. Most integrals are not tractable by pen and paper, and a computerized solution approach is both very much simpler and much more powerful—you can essentially treat all integrals $\int_a^b f(x)dx$ in 10 lines of computer code!

Integration also demonstrates the difference between exact mathematics by pen and paper and *numerical mathematics* on a computer. The latter approaches the result of the former without any worries about rounding errors due to finite precision arithmetics in computers (in contrast to differentiation, where such errors prevent us from getting a result as accurate as we desire).

Finally, integration is thought of as a somewhat difficult mathematical concept to grasp, and programming integration should greatly improve the understanding of what integration really *is* and how it works.

Not only shall we understand how to use the computer to integrate, but we shall also learn a series of good habits to ensure your computer work is of the highest scientific quality. In particular, we will have a strong focus on how to write Python code that is free of programming mistakes.

S. Linge, H. P. Langtangen, *Programming for Computations - Python*,
Texts in Computational Science and Engineering 15,
https://doi.org/10.1007/978-3-030-16877-3_6

Calculating an integral is traditionally done by

$$\int_a^b f(x)\,dx = F(b) - F(a), \tag{6.1}$$

where

$$f(x) = \frac{dF}{dx}\,.$$

The major problem with this procedure is that we need to find an *anti-derivative* $F(x)$ corresponding to a given $f(x)$. For some relatively simple integrands $f(x)$, finding $F(x)$ is a doable task. Often, however, it is really challenging, and sometimes even impossible!

The method (6.1) provides an *exact* or *analytical* value of the integral. If we relax the requirement of computing an exact value for the integral, and instead look for *approximate* values, produced by *numerical methods*, integration becomes a very straightforward task for almost any given $f(x)$! In particular, we do not need an anti-derivative $F(x)$ at all, since it is just the known integrand $f(x)$ that enters the calculations.

The (apparent) downside of a numerical method is that it can only find an approximate answer. Leaving the exact for the approximate is a mental barrier in the beginning, but remember that most real applications of integration will involve an $f(x)$ function that contains physical parameters, which are measured with some error. That is, $f(x)$ is very seldom exact, and it does not make sense trying to compute the integral with a smaller error than the one already present in $f(x)$.

Another advantage of numerical methods is that we can easily integrate a function $f(x)$ that is only known as *samples*, i.e., discrete values at some x points, and not as a continuous function of x expressed through a formula. This is highly relevant when f is measured in a physical experiment.

6.1 Basic Ideas of Numerical Integration

We consider the integral

$$\int_a^b f(x)dx\,. \tag{6.2}$$

Most numerical methods for computing this integral split up the original integral into a sum of several integrals, each covering a smaller part of the original integration interval $[a, b]$. This re-writing of the integral is based on a selection of *integration points* x_i, $i = 0, 1, \ldots, n$ that are distributed on the interval $[a, b]$. Integration points may, or may not, be evenly distributed. An even distribution simplifies expressions and is often sufficient, so we will mostly restrict ourselves to that choice. The integration points are then computed as

$$x_i = a + ih, \quad i = 0, 1, \ldots, n, \tag{6.3}$$

where

$$h = \frac{b-a}{n}. \tag{6.4}$$

That is, we get n sub-intervals of the same size h. Given the integration points, the original integral is re-written as a sum of integrals, each integral being computed over the sub-interval between two consecutive integration points. The integral in (6.2) is thus expressed as

$$\int_a^b f(x)dx = \int_{x_0}^{x_1} f(x)dx + \int_{x_1}^{x_2} f(x)dx + \ldots + \int_{x_{n-1}}^{x_n} f(x)dx\,. \tag{6.5}$$

Note that $x_0 = a$ and $x_n = b$.

Proceeding from (6.5), the different integration methods will differ in the way they approximate each integral on the right hand side. The fundamental idea is that each term is an integral over a small interval $[x_i, x_{i+1}]$, and over this small interval, it makes sense to approximate f by a simple shape, say a constant, a straight line, or a parabola, that can be easily integrated. The details will become clear in the coming examples.

Computational Example To understand and compare the numerical integration methods, it is advantageous to use a specific integral for computations and graphical illustrations. In particular, we want to use an integral that we can calculate by hand such that the accuracy of the approximation methods can easily be assessed.

Our specific integral is taken from basic physics. Assume that you speed up your car from rest, on a straight road, and wonder how far you go in T seconds. The displacement is given by the integral $\int_0^T v(t)dt$, where $v(t)$ is the velocity as a function of time. A rapidly increasing velocity function might be

$$v\,(t) = 3t^2 e^{t^3}\,. \tag{6.6}$$

The distance traveled in 1 s is then

$$\int_0^1 v(t)dt, \tag{6.7}$$

which is the integral we aim to compute by numerical methods.

By hand, we get

$$\int_0^1 3t^2 e^{t^3} dt = \left[e^{t^3}\right]_0^1 \approx 1.718, \tag{6.8}$$

which is rounded to 3 decimals for convenience.

6.2 The Composite Trapezoidal Rule

The integral $\int_a^b f(x)dx$ may be interpreted as the area between the x axis and the graph $y = f(x)$ of the integrand. Figure 6.1 illustrates this area for the case in (6.7). Computing the integral $\int_0^1 v(t)dt$ amounts to computing the area of the hatched region.

If we *replace* the true graph in Fig. 6.1 by a set of straight line segments, we may view the area rather as composed of trapezoids, the areas of which are easy to compute. This is illustrated in Fig. 6.2, where four straight line segments give rise to four trapezoids, covering the time intervals $[0, 0.2)$, $[0.2, 0.6)$, $[0.6, 0.8)$ and $[0.8, 1.0]$. Note that we have taken the opportunity here to demonstrate the computations with time intervals that differ in size.

The areas of the four trapezoids shown in Fig. 6.2 now constitute our approximation to the integral (6.7):

$$\int_0^1 v(t)dt \approx h_1(\frac{v(0)+v(0.2)}{2}) + h_2(\frac{v(0.2)+v(0.6)}{2}) + h_3(\frac{v(0.6)+v(0.8)}{2}) + h_4(\frac{v(0.8)+v(1.0)}{2}), \tag{6.9}$$

where

$$h_1 = (0.2 - 0.0), \tag{6.10}$$

$$h_2 = (0.6 - 0.2), \tag{6.11}$$

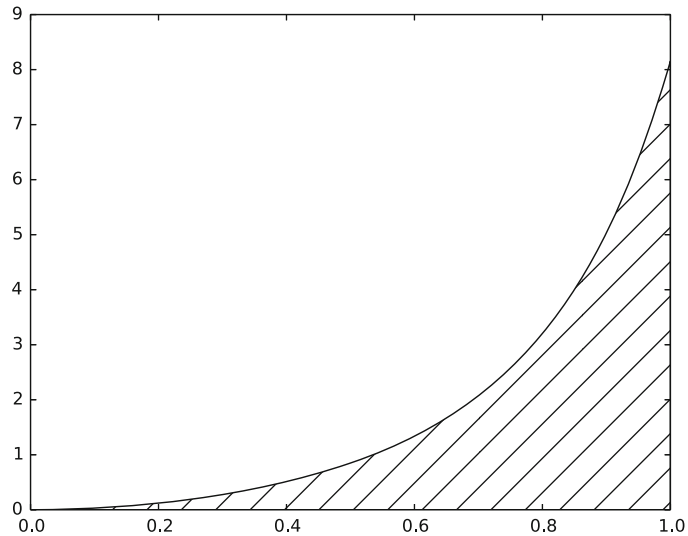


Fig. 6.1 The integral of $v(t)$ interpreted as the area under the graph of v

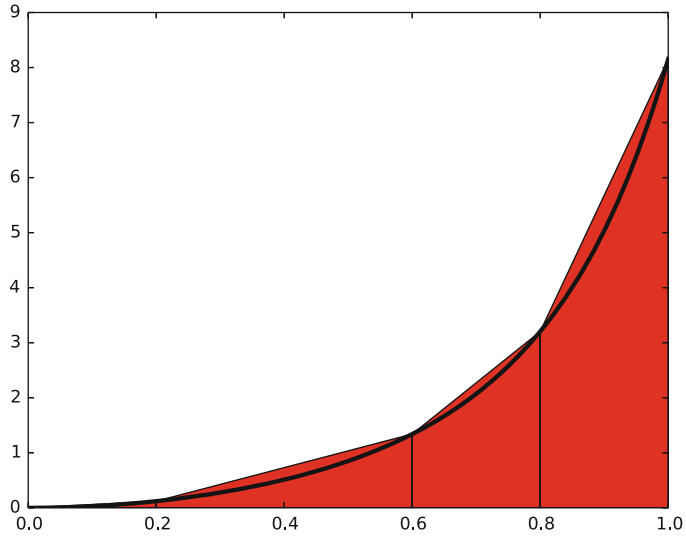


Fig. 6.2 Computing approximately the integral of a function as the sum of the areas of the trapezoids

$$h_3 = (0.8 - 0.6), \tag{6.12}$$

$$h_4 = (1.0 - 0.8) \tag{6.13}$$

With $v(t) = 3t^2 e^{t^3}$, each term in (6.9) is readily computed and our approximate computation gives

$$\int_0^1 v(t)dt \approx 1.895\,. \tag{6.14}$$

Compared to the true answer of 1.718, this is off by about 10%. However, note that we used just four trapezoids to approximate the area. With more trapezoids, the approximation would have become better, since the straight line segments at the upper trapezoid side then would follow the graph more closely. Doing another hand calculation with more trapezoids is not too tempting for a lazy human, though, but it is a perfect job for a computer! Let us therefore derive the expressions for approximating the integral by an arbitrary number of trapezoids.

6.2.1 The General Formula

For a given function $f(x)$, we want to approximate the integral $\int_a^b f(x)dx$ by n trapezoids (of equal width). We start out with (6.5) and approximate each integral

on the right hand side with a single trapezoid. In detail,

$$\begin{aligned}\int_a^b f(x)\,dx &= \int_{x_0}^{x_1} f(x)dx + \int_{x_1}^{x_2} f(x)dx + \ldots + \int_{x_{n-1}}^{x_n} f(x)dx, \\ &\approx h\frac{f(x_0)+f(x_1)}{2} + h\frac{f(x_1)+f(x_2)}{2} + \ldots + \\ &\quad h\frac{f(x_{n-1})+f(x_n)}{2}\end{aligned} \tag{6.15}$$

By simplifying the right hand side of (6.15) we get

$$\int_a^b f(x)\,dx \approx \frac{h}{2}\left[f(x_0) + 2f(x_1) + 2f(x_2) + \ldots + 2f(x_{n-1}) + f(x_n)\right] \tag{6.16}$$

which is more compactly written as

$$\int_a^b f(x)\,dx \approx h\left[\frac{1}{2}f(x_0) + \sum_{i=1}^{n-1} f(x_i) + \frac{1}{2}f(x_n)\right]. \tag{6.17}$$

Composite Integration Rules

The word *composite* is often used when a numerical integration method is applied with more than one sub-interval. Strictly speaking then, writing, e.g., "the trapezoidal method", should imply the use of only a single trapezoid, while "the composite trapezoidal method" is the most correct name when several trapezoids are used. However, this naming convention is not always followed, so saying just "the trapezoidal method" may point to a single trapezoid as well as the composite rule with many trapezoids.

6.2.2 A General Implementation

Specific or General Implementation? Suppose we want to compute the specific integral $\int_0^1 v(t)dt$, where $v(t) = 3t^2e^{t^3}$, using the (composite) trapezoidal method in (6.17). Although simple in principle, the practical steps are often confusing to many, because the notation in the abstract formulation in (6.17) differs from the notation in our special problem. Clearly, the f, x, and h in (6.17) correspond to v, t, and perhaps Δt for the trapezoid width in our special problem.

The Programmer's Dilemma

1. **Specific implementation**: Should we write a special program for the particular integral, using the ideas from the general rule (6.17), but replacing f by v, x by t, and h by Δt?
2. **General implementation**: Should we implement the general method (6.17), as written, in a general function `trapezoidal(f, a, b, n)` and solve the particular integral by a specialized call to this function?

A general implementation is always the best choice, not only for integrals, but when programming in general!

The first alternative in the box above sounds less abstract and therefore more attractive to many. Nevertheless, as we hope will be evident from the following, the second alternative is actually the simplest *and* most reliable from both a mathematical and a programming point of view.

These authors will claim that the second alternative is the essence of the power of mathematics, while the first alternative is the source of much confusion about mathematics!

General Implementation For the integral $\int_a^b f(x)dx$, computed by the formula (6.17), we want a corresponding Python function `trapezoidal` to take any f, a, b, and n as input and return the approximation to the integral.

We write `trapezoidal` as close as possible to the formula (6.17), making sure variable names correspond to the mathematical notation:

```
def trapezoidal(f, a, b, n):
    h = (b-a)/n
    f_sum = 0
    for i in range(1, n, 1):
        x = a + i*h
        f_sum = f_sum + f(x)
    return h*(0.5*f(a) + f_sum + 0.5*f(b))
```

Observe how the `for` loop takes care of (only) the sum over $f(x_i)$, and that the x values start with $x = a + h$ (when i is 1), increases with h for each iteration, before ending with $x = a + (n-1)h$. This is consistent with the x values of the sum in (6.17). After the loop, we finalize the computation and return the result. This will be our implementation of choice for the `trapezoidal` function, even though, typically for programming, it could have been implemented in different ways. Which implementation to choose, is sometimes just a matter of personal taste.

One alternative version could be:

```
def trapezoidal(f, a, b, n):
    h = (b-a)/n
    result = 0.5*f(a) + 0.5*f(b)
    for i in range(1, n):
        result += f(a + i*h)
    result *= h
    return result
```

where `0.5*f(a) + 0.5*f(b)` is used in an initialization of `result`, before adding all the function evaluations $f(x_i)$ in the loop. After the loop, the only remaining thing, is to multiply by `h`. In this second alternative, there are also a few other differences to note. The `range` function is called with two parameters only, giving the (default) step of 1. In the loop, instead of computing `x = a + i*h` prior to calling `f(x)`, we combine this into `f(a + i*h)`, which first computes `a + i*h` and then makes the call to `f`. Also, the compact operators `+=` and `*=` have been used.

Using the General Implementation in a Session Having the `trapezoidal` function as the only content of a file `trapezoidal.py` automatically makes that file a module that we can import and test in an interactive session:

```
In [1]: from trapezoidal import trapezoidal

In [2]: from math import exp

In [3]: v = lambda t: 3*(t**2)*exp(t**3)

In [4]: n = 4

In [5]: numerical = trapezoidal(v, 0, 1, n)

In [6]: numerical
Out[6]: 1.9227167504675762
```

Let us compute the exact expression and the error in the approximation. Using V for the anti-derivative, we get:

```
In [7]: V = lambda t: exp(t**3)

In [8]: exact = V(1) - V(0)

In [9]: abs(exact - numerical)   # absolute value of error
Out[9]: 0.20443492200853108
```

Since the sign of the error is irrelevant, we find the absolute value of the error. So, is this error convincing? We can try a larger n:

```
In [10]: numerical = trapezoidal(v, 0, 1, n=400)

In [11]: abs(exact - numerical)
Out[11]: 2.1236490512777095e-05
```

Fortunately, many more trapezoids give a much smaller error.

Using the General Implementation in a Program Instead of computing our integral in an interactive session, we can do it in a program. In that program, we need the (general) function definition of `trapezoidal`, and we need some code that specifies our particular integrand, as well as the other arguments required for calling `trapezoidal`. This code might be placed in a main program. However, it could also be placed in a function that is called from the main program. This is what we will do. A chunk of code doing a particular thing is always best isolated as a function, even if we do not see any future reason to call the function several times, and even if we have no need for arguments to parameterize what goes on inside the

function. Thus, in the present case, we put the statements (otherwise placed a main program) inside a function named `application`.

To achieve flexibility, we proceed to modify `trapezoidal.py`, so that it has a test block and function definitions of `trapezoidal` and `application`:

```
def trapezoidal(f, a, b, n):
    h = (b-a)/n
    f_sum = 0
    for i in range(1, n, 1):
        x = a + i*h
        f_sum = f_sum + f(x)
    return h*(0.5*f(a) + f_sum + 0.5*f(b))

def application():
    from math import exp
    v = lambda t: 3*(t**2)*exp(t**3)
    n = int(input('n: '))
    numerical = trapezoidal(v, 0, 1, n)

    # Compare with exact result
    V = lambda t: exp(t**3)
    exact = V(1) - V(0)
    error = abs(exact - numerical)
    print('n={:d}: {:.16f}, error: {:g}'.format(n, numerical, error))

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

With our newly gained knowledge about the making of modules, we understand that the `if` test becomes true when the module file, `trapezoidal.py`, is run as a program, and false when the module (or part of it) is imported in another program. Consequently, with an import like `from trapezoidal import trapezoidal`, the test fails and `application()` is not called. On the other hand, if we run `trapezoidal.py` as a program, the test condition is positive and `application()` is called. A call to `application` implies that our special problem gets computed. The main program now gets very small, being just a single function call to `application`.

Running the program, e.g., with $n = 4$ gives the output

```
n=4: 1.9227167504675762, error: 0.204435
```

Clearly, with a module like the one shown here, the `trapezoidal` function alone (i.e., without `application`) can easily be imported by other programs to compute other integrals.

6.2.3 A Specific Implementation: What's the Problem?

Let us illustrate the implementation implied by alternative 1 in the *Programmer's dilemma* box in Sect. 6.2.2. That is, we make a special-purpose code, where we adapt the general formula (6.17) to the specific problem $\int_0^1 3t^2 e^{t^3} dt$, in which the integrand is a velocity function $v(t)$.

Implementation Without Function Definitions Basically, we use a `for` loop to compute the sum. Each term with $f(x)$ in the formula (6.17) is replaced by $3t^2e^{t^3}$, x by t, and h by Δt.[1] A first version of this special implementation, without any function definition, could read

```
from math import exp

a = 0.0;  b = 1.0
n = int(input('n: '))
dt = (b - a)/n

# Integral by the trapezoidal method
v_sum = 0
for i in range(1, n, 1):
    t = a + i*dt
    v_sum = v_sum + 3*(t**2)*exp(t**3)

numerical = dt*(0.5*3*(a**2)*exp(a**3) +
                v_sum +
                0.5*3*(b**2)*exp(b**3))

exact_value = exp(1**3) - exp(0**3)
error = abs(exact_value - numerical)
rel_error = (error/exact_value)*100
print('n={:d}: {:.16f}, error: {:g}'.format(n, numerical, error))
```

The problem with the above strategy is at least three-fold:

1. To write the code, we had to reformulate (6.17) for our special problem with a different notation. Errors come easy then.
2. To write the code, we had to insert the integrand $3t^2e^{t^3}$ several places in the code, which quickly leads to errors.
3. If we later want to compute a different integral, the code must be edited in several places. Such edits are likely to introduce errors.

The potential errors related to point 2 serve to illustrate how important it is to define and use appropriate functions.

Implementation with Function Definitions An improved second version of the special implementation, now with functions for the integrand `v` and the anti-derivative `V`, might then read

```
from math import exp

v = lambda t: 3*(t**2)*exp(t**3)      # Define integrand
a = 0.0;  b = 1.0
n = int(input('n: '))
dt = (b - a)/n

# Integral by the trapezoidal method
```

[1] Replacing h by Δt is not strictly required as many use h as interval also along the time axis. Nevertheless, Δt is an even more popular notation for a small time interval, so we use that here.

```
v_sum = 0
for i in range(1, n, 1):
    t = a + i*dt
    v_sum = v_sum + v(t)
numerical = dt*(0.5*v(a) + v_sum + 0.5*v(b))

V = lambda t: exp(t**3)
exact_value = V(b) - V(a)
error = abs(exact_value - numerical)
rel_error = (error/exact_value)*100
print('n={:d}: {:.16f}, error: {:g}'.format(n, numerical, error))
```

Unfortunately, the two other problems (1. and 3.) remain, and they are fundamental.

Computing Another Integral Suppose you next want to compute another integral, say $\int_{-1}^{1.1} e^{-x^2} dx$, using the previous specific implementation as your "starting point". What changes are required in the code then?

First of all, an anti-derivative can not (easily) be found[2,3] for this new integrand, so we drop computing the integration error, and must remove the corresponding code lines. In addition,

- the notation should be changed to fit the new problem. Thus, `t` and `dt` should be replaced by `x` and `h`. Also, the integrand is (most likely) not a velocity any more, so the name `v` should be changed with, e.g., `f`. Similarly, `v_sum` should rather be `f_sum` then.
- the formula for `v` (or `f`) must be replaced by a new formula
- the limits `a` and `b` must be changed

These changes are straightforward to implement, but *they are scattered around in the program*, a fact that requires us to be very careful so we do not introduce new programming errors while we modify the code. It is also very easy to forget one or two of the required changes.

For the sake of comparison, we might see how easy it is to rather use our *general implementation* in `trapezoidal.py` for the task. With the following interactive session, it should be clear that this implementation allows us to compute the new integral $\int_{-1}^{1.1} e^{-x^2} dx$ *without touching the implemented mathematical algorithm*! We can simply do:

```
In [1]: from trapezoidal import trapezoidal  # ...general implementation

In [2]: from math import exp

In [3]: trapezoidal(lambda x: exp(-x**2), -1, 1.1, 400)
Out[3]: 1.5268823686123285
```

[2] You cannot integrate e^{-x^2} by hand, but this particular integral is appearing so often in so many contexts that the integral is a special function, called the Error function and written erf(x). In a code, you can call `erf(x)`. The `erf` function is found in the `math` module.

[3] http://en.wikipedia.org/wiki/Error_function.

Looking back on the two different solutions, the specific implementation and the general implementation, you should realize that *implementing a general mathematical algorithm in a general function* requires somewhat more abstract thinking, but the resulting code can be used over and over again! Essentially, if you apply the special-purpose style, you have to retest the implementation of the algorithm after every change of the program.

The present integral problems result in short code. In more challenging engineering problems, the code quickly grows to hundreds and thousands of lines. Without abstractions, in terms of general algorithms in general reusable functions, the complexity of the program grows so fast that it will be extremely difficult to make sure that the program works properly.

Another advantage of packaging mathematical algorithms in functions, is that a function can be reused by anyone to solve a problem by just calling the function with a proper set of arguments. Understanding the function's inner details is strictly not necessary to compute a new integral. Similarly, you can find libraries of functions on the Internet and use these functions to solve your problems without specific knowledge of every mathematical detail in the functions.

This desirable feature has its downside, of course: the user of a function may misuse it, and the function may contain programming errors and lead to wrong answers. Testing the output of downloaded functions is therefore extremely important before relying on the results.

6.3 The Composite Midpoint Method

The Idea Rather than approximating the area under a curve by trapezoids, we can use plain rectangles. It may sound less accurate to use horizontal lines and not skew lines following the function to be integrated, but an integration method based on rectangles (the *midpoint method*) is in fact slightly more accurate than the one based on trapezoids! In the midpoint method, we construct a rectangle for every sub-interval where the height equals the integrand f at the midpoint of the sub-interval.

For the sake of comparison, we may repeat the hand calculation of $\int_0^1 v(t)dt$ in (6.7), but this time with the midpoint method. With four rectangles (Fig. 6.3) and the same sub-intervals that we used with the trapezoidal method, [0, 0.2), [0.2, 0.6), [0.6, 0.8), and [0.8, 1.0], we get

$$\int_0^1 v(t)dt \approx h_1 v\left(\frac{0+0.2}{2}\right) + h_2 v\left(\frac{0.2+0.6}{2}\right) + h_3 v\left(\frac{0.6+0.8}{2}\right) + h_4 v\left(\frac{0.8+1.0}{2}\right), \tag{6.18}$$

where h_1, h_2, h_3, and h_4 are the widths of the sub-intervals, used previously with the trapezoidal method and defined in (6.10)–(6.13).

With $v(t) = 3t^2 e^{t^3}$, the approximation becomes 1.632. Compared with the true answer (1.718), this is about 5% too small, but it is better than what we got with

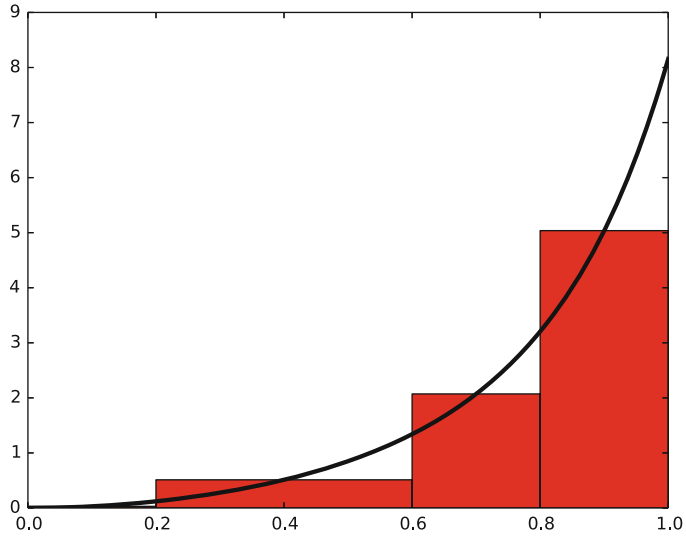


Fig. 6.3 Computing approximately the integral of a function as the sum of the areas of the rectangles

the trapezoidal method (10%) with the same sub-intervals. More rectangles give a better approximation.

6.3.1 The General Formula

Let us derive a formula for the midpoint method based on n rectangles of equal width:

$$\begin{aligned}\int_a^b f(x)\,dx &= \int_{x_0}^{x_1} f(x)dx + \int_{x_1}^{x_2} f(x)dx + \ldots + \int_{x_{n-1}}^{x_n} f(x)dx,\\ &\approx hf\left(\frac{x_0+x_1}{2}\right) + hf\left(\frac{x_1+x_2}{2}\right) + \ldots + hf\left(\frac{x_{n-1}+x_n}{2}\right),\\ &\approx h\left(f\left(\frac{x_0+x_1}{2}\right) + f\left(\frac{x_1+x_2}{2}\right) + \ldots + f\left(\frac{x_{n-1}+x_n}{2}\right)\right).\end{aligned} \tag{6.19}$$

This sum may be written more compactly as

$$\int_a^b f(x)dx \approx h\sum_{i=0}^{n-1} f(x_i), \tag{6.20}$$

where $x_i = \left(a + \frac{h}{2}\right) + ih$.

6.3.2 A General Implementation

We follow the advice and lessons learned from the implementation of the trapezoidal method. Thus, we make a module `midpoint.py` with a general implementation of (6.20) and a function `application`, just like we did with the trapezoidal function:

```
def midpoint(f, a, b, n):
    h = (b-a)/n
    f_sum = 0
    for i in range(0, n, 1):
        x = (a + h/2.0) + i*h
        f_sum = f_sum + f(x)
    return h*f_sum

def application():
    from math import exp
    v = lambda t: 3*(t**2)*exp(t**3)
    n = int(input('n: '))
    numerical = midpoint(v, 0, 1, n)

    # Compare with exact result
    V = lambda t: exp(t**3)
    exact = V(1) - V(0)
    error = abs(exact - numerical)
    print('n={:d}: {:.16f}, error: {:g}'.format(n, numerical, error))

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

In `midpoint`, observe how the x values in the loop start out at $x = a + \frac{h}{2}$ (when i is 0), which is in the middle of the first rectangle. The x values then increase by h for each iteration, meaning that we repeatedly "jump" to the midpoint of the next rectangle as i increases. This is consistent with the formula in (6.20), as is the final x value of $x = a + \frac{h}{2} + (n-1)h$. To convince yourself that the first, intermediate and final x values are correct, look at a case with only three rectangles, for example.

When `application` is called, the particular problem $\int_0^1 3t^2 e^{t^3} dt$ is computed, i.e., the same integral that we handled with the trapezoidal method. Running the program with `n = 4` gives the output

```
n=4: 1.6189751378083810, error: 0.0993067
```

The magnitude of this error is now about 0.1 in contrast to 0.2, which we got with the trapezoidal rule. This is in fact not accidental: one can show mathematically that the error of the midpoint method is a bit smaller than for the trapezoidal method. The differences are seldom of any practical importance, and on a laptop we can easily use $n = 10^6$ and get the answer with an error of about 10^{-12} in a couple of seconds.

6.3.3 Comparing the Trapezoidal and the Midpoint Methods

The next example shows how easy it is to combine the `trapezoidal` and `midpoint` functions to make a comparison of the two methods. The coding is given in `compare_integration_methods.py`:

```
from trapezoidal import trapezoidal
from midpoint import midpoint
from math import exp

g = lambda y: exp(-y**2)
a = 0
b = 2
print('    n        midpoint          trapezoidal')
for i in range(1, 21):
    n = 2**i
    m = midpoint(g, a, b, n)
    t = trapezoidal(g, a, b, n)
    print('{:7d} {:.16f} {:.16f}'.format(n, m, t))
```

Note the efforts put into nice formatting—the output becomes

```
      n        midpoint          trapezoidal
      2 0.8842000076332692 0.8770372606158094
      4 0.8827889485397279 0.8806186341245393
      8 0.8822686991994210 0.8817037913321336
     16 0.8821288703366458 0.8819862452657772
     32 0.8820933014203766 0.8820575578012112
     64 0.8820843709743319 0.8820754296107942
    128 0.8820821359746071 0.8820799002925637
    256 0.8820815770754198 0.8820810181335849
    512 0.8820814373412922 0.8820812976045025
   1024 0.8820814024071774 0.8820813674728968
   2048 0.8820813936736116 0.8820813849400392
   4096 0.8820813914902204 0.8820813893068272
   8192 0.8820813909443684 0.8820813903985197
  16384 0.8820813908079066 0.8820813906714446
  32768 0.8820813907737911 0.8820813907396778
  65536 0.8820813907652575 0.8820813907567422
 131072 0.8820813907631487 0.8820813907610036
 262144 0.8820813907625702 0.8820813907620528
 524288 0.8820813907624605 0.8820813907623183
1048576 0.8820813907624268 0.8820813907623890
```

A visual inspection of the numbers shows how fast the digits stabilize in both methods. It appears that 13 digits have stabilized in the last two rows.

Remark

The trapezoidal and midpoint methods are just two examples in a jungle of numerical integration rules. Other famous methods are Simpson's rule and Gauss quadrature. They all work in the same way:

$$\int_a^b f(x)dx \approx \sum_{i=0}^{n-1} w_i f(x_i) .$$

That is, the integral is approximated by a sum of function evaluations, where each evaluation $f(x_i)$ is given a weight w_i. The different methods differ in the way they construct the evaluation points x_i and the weights w_i. Higher accuracy can be obtained by optimizing the location of x_i.

6.4 Vectorizing the Functions

The functions `midpoint` and `trapezoidal` usually run fast in Python and compute an integral to satisfactory precision within a fraction of a second. However, long loops in Python may run slowly in more complicated implementations. To increase speed, the loops can be replaced by vectorized code. The integration functions offer simple and good examples on how to vectorize loops.

We have already seen simple examples on vectorization in Sect. 1.5, when we evaluated a mathematical function $f(x)$ for a large number of x values stored in an array. Basically, we can write

```
def f(x):
    return exp(-x)*sin(x) + 5*x

from numpy import exp, sin, linspace
x = linspace(0, 4, 101)  # coordinates from 100 intervals on [0, 4]
y = f(x)                 # all points evaluated at once
```

The result `y` is an array that, alternatively, could have been computed by running a `for` loop over the individual `x` values and called `f` for each value. Vectorization essentially eliminates this explicit loop in Python (i.e., the looping over `x` and application of `f` to each `x` value are instead performed in a library with fast, compiled code).

6.4.1 Vectorizing the Midpoint Rule

We start by vectorizing the `midpoint` function, since `trapezoidal` is not equally straightforward to vectorize. In both cases, our vectorization will remove the explicit loop. The fundamental ideas of the vectorized algorithm are to

1. compute and store all the evaluation points in one array `x`
2. call `f(x)` to produce an array of corresponding function values
3. use the `sum` function to sum up the `f(x)` values

The evaluation points in the midpoint method are $x_i = a + \frac{h}{2} + ih, i = 0, \ldots, n-1$. That is, n uniformly distributed coordinates between $a + h/2$ and $b - h/2$. Such coordinates can be calculated by `x = linspace(a+h/2, b-h/2, n)`. Given that the Python implementation `f` of the mathematical function f works with an array argument, which is very often the case in Python, `f(x)` will produce all the function values in an array. The array elements are then summed up by `sum`, when calling `sum(f(x))`. The resulting sum is to be multiplied by the rectangle width h to produce the integral value. The complete function is listed below.

```
from numpy import linspace, sum

def midpoint(f, a, b, n):
    h = (b-a)/n
    x = linspace(a + h/2, b - h/2, n)
    return h*sum(f(x))
```

The code is found in the file integration_methods_vec.py.

Let us test the code interactively in a Python shell by computing $\int_0^1 3t^2 e^{t^3} dt$. The file with the code above has the name `integration_methods_vec.py` and is a valid module from which we can import the vectorized function:

```
In [1]: from integration_methods_vec import midpoint

In [2]: from numpy import exp

In [3]: v = lambda t: 3*t**2*exp(t**3)

In [4]: midpoint(v, 0, 1, 10)
Out[4]: 1.7014827690091872
```

Note the necessity to use `exp` from `numpy`: our `v` function will be called with `x` as an array, and the `exp` function must be capable of working with an array.

The vectorized code performs all loops very efficiently in compiled code, resulting in much faster execution. Moreover, many readers of the code will also say that the algorithm looks clearer than in the loop-based implementation.

6.4.2 Vectorizing the Trapezoidal Rule

We can use the same approach to vectorize the `trapezoidal` function. However, the trapezoidal rule performs a sum where the end points have different weight. If we do `sum(f(x))`, we get the end points `f(a)` and `f(b)` with a weight of unity instead of one half. A remedy is to subtract the error from `sum(f(x))`: `sum(f(x)) - 0.5*f(a) - 0.5*f(b)`. The vectorized version of the trapezoidal method then becomes (the code is found in integration_methods_vec.py)

```
def trapezoidal(f, a, b, n):
    h = (b-a)/n
    x = linspace(a, b, n+1)
    s = sum(f(x)) - 0.5*f(a) - 0.5*f(b)
    return h*s
```

6.4.3 Speed up Gained with Vectorization

Now that we have created faster, vectorized versions of the functions, it is of interest to measure how much faster they are. Restricting ourselves to the midpoint method, we might proceed as:

```
import timeit
from integration_methods_vec import midpoint as midpoint_vec
from midpoint import midpoint
from numpy import exp

v = lambda t: 3*t**2*exp(t**3)

t = timeit.Timer('midpoint(v, 0, 1, 1000000)', \
                 setup='from __main__ import midpoint, v')
time_midpoint = t.timeit(10)
print('Time, midpoint: {:g} seconds'.format(time_midpoint))

# Vectorized version
t = timeit.Timer('midpoint_vec(v, 0, 1, 1000000)', \
                 setup='from __main__ import midpoint_vec, v')
time_midpoint_vec = t.timeit(10)
print('Time, midpoint vec: {:g} seconds'.format(time_midpoint_vec))

print('Efficiency factor: {:g}'.format(time_midpoint/time_midpoint_vec))
```

Running the program gives

```
Time, midpoint: 19.6083 seconds
Time, midpoint vec: 0.868379 seconds
Efficiency factor: 22.5804
```

We see that the vectorized version is about 20 times faster. The results for the trapezoidal method are very similar, and the factor of about 20 is independent of the number of intervals.

6.5 Rate of Convergence

We have seen that the numerical integration error drops when the number of sub-intervals n is increased (causing each sub-interval to become smaller). This is fine and in line with our expectations, but some important details should be added.

Asymptotic Behavior of the Integration Error It is known that, if only the size h of the sub-intervals is small enough, numerical integration methods typically give an error

$$E = Kh^r\,, \tag{6.21}$$

where K is an unknown constant, while the *convergence rate* r is a known constant that depends on the method. When a method has convergence rate r, it is known as an r-th order method. A large r is beneficial, since E then drops quicker when $h \to 0$.

Clearly, when

$$h = \frac{b-a}{n},$$

(n sub-intervals of equal size h for an integration interval $[a, b]$), an alternative expression for E follows from

$$E = Kh^r, \tag{6.22}$$

$$= K\left(\frac{b-a}{n}\right)^r, \tag{6.23}$$

$$= K\,(b-a)^r\left(\frac{1}{n}\right)^r, \tag{6.24}$$

which, by introducing another constant $C = K\,(b-a)^r$, gives

$$E = Cn^{-r}. \tag{6.25}$$

Convergence Rate for the Trapezoidal and Midpoint Methods Using, for example, the trapezoidal method, we may carry out some experimental runs with our test problem $\int_0^1 3t^2 e^{t^3} dt$, doubling n for each run: $n = 4, 8, 16$. The corresponding errors are then 12%, 3% and 0.78%, respectively. These numbers indicate that the error is reduced by roughly a factor 4 when doubling n. Thus, it seems that the error converges to zero as n^{-2}, which suggests a convergence rate $r = 2$. In fact, it can be shown mathematically that the trapezoidal and the midpoint method both have a convergence rate $r = 2$, i.e., they are both second-order methods. Soon, we will see how this fact (and more) can be exploited in the testing of code.

Remark on the Definition of Convergence Rate

When we later address numerical solution methods for ordinary differential equations (Chap. 8), convergence rate is essentially defined like in (6.21), we just switch (not required) the symbol h with Δt (i.e., the spacing between computed solution values).

However, with iterative methods for the solving of nonlinear algebraic equations (Chap. 7), convergence rate is defined differently. In that case, one usually relates the *error* at an iteration to the *error* at the previous iteration, and the convergence rate appears as a parameter in that relation.

6.6 Testing Code

6.6.1 Problems with Brief Testing Procedures

Previously in this book, our programs have been tested in very simple ways, usually by comparing to hand calculations. For numerical integration, in particular, testing has so far employed two strategies. When the exact solution *was* available, we computed the error and saw that an increase of n gave a decrease in the error. When the exact solution *was not* available, we could (as in the comparison example of the previous section) look at the integral values and see that they stabilized as n grew.

Unfortunately, these are very weak test procedures and not at all satisfactory for claiming that the software we have produced is correctly implemented.

A Deliberate Bug To see this, we can introduce a bug in the `application` function that calls `trapezoidal`: instead of integrating $3t^2e^{t^3}$, we write "accidentally" $3t^3e^{t^3}$, but keep the same anti-derivative $x(t) = e^{t^3}$ for computing the error. With the bug and $n = 4$, the error is 0.1, but without the bug the error is 0.2! It is of course completely impossible to tell if 0.1 is the right value of the error. Fortunately, in this case, increasing n shows that the error stays about 0.3 in the program with the bug, so the test procedure with increasing n (and checking that the error then decreases) points to a problem in the code.

Another Deliberate Bug Let us look at another bug, this time in the mathematical algorithm: instead of computing $\frac{1}{2}(f(a) + f(b))$ as we should, we "forget" the second $\frac{1}{2}$ and write `0.5*f(a) + f(b)`. The error for $n = 440,400$ when computing $\int_{1.1}^{1.9} 3t^2e^{t^3}dt$ goes like 1400, 107, 10, respectively, which looks promising. The problem is that the right errors should be 369, 4.08, and 0.04. That is, the error should be reduced faster in the correct than in the buggy code. The problem, however, is that it is reduced in both codes, and we may stop further testing and believe everything is correctly implemented.

Unit Testing

A good habit is to test small pieces of a larger code individually, one at a time. This is known as *unit testing*: A (small) unit of the code is identified, so that a separate test for this unit can be made. The unit test should be "stand-alone" in the sense that it can be run without the outcome of other tests. Typically, one algorithm in scientific programs is considered a unit. The challenge with unit tests in numerical computing, is to deal with numerical approximation errors. A fortunate side effect of unit testing is that the programmer is forced to use functions to modularize the code into smaller, logical pieces.

6.6.2 Proper Test Procedures

There are three serious ways to test the implementation of numerical methods via unit tests:

1. *Comparing with hand-computed results*. Relevant for problems with few arithmetic operations, i.e., small n.
2. *Solving a problem without numerical errors*. We know, for example, that the trapezoidal rule must be exact for linear integrand functions. The error produced by the program must then be zero (to machine precision).
3. *Demonstrating correct convergence rates*. When exact errors can be computed, a strong test is to let n grow and see if the error approaches zero as fast as theory predicts. As stated previously, for the trapezoidal and midpoint rules it is known that the error depends on n as n^{-2} when $n \to \infty$.

Remark

When testing code, we usually choose computational problems for which the exact solution *is* known. This is obviously a good idea, since it allows the quality of approximate numerical answers to be judged. Do not forget, however, that the exact solution is available because we *deliberately* chose a problem with known exact solution. When we have finished testing (and probably fixing) the code, our belief is that the code will work also for problems with unknown exact solutions. Our strategy then, is to trust the approximate answer from our code.

Hand-Computed Results Let us use two trapezoids and compute the integral $\int_0^1 v(t)dt$, where $v(t) = 3t^2e^{t^3}$:

$$h\frac{(v(0)+v(0.5))}{2} + h\frac{(v(0.5)+v(1))}{2} = 2.463642041244344,$$

when $h = 0.5$ is the width of each trapezoid. Running the program gives exactly the same result.

Note that the exact solution is not involved here. We simply carry out the numerical algorithm by hand, "independent" from the code. We should of course get agreement between these hand calculations and program output when the same n is used. However, assuming we do get agreement, that numerical answer may still differ substantially from the exact solution to the problem. That is of no concern in this test, as the aim is *not* to get as good an answer as possible (potentially achieved with a large n), but rather to check in a simple manner whether the algorithm "seems to be" correctly implemented.

Solving a Problem Without Numerical Errors The best unit tests for numerical algorithms involve mathematical problems where we know the numerical result beforehand. For these unit tests, we choose problems that fulfill two criteria. One

criterion is that the exact solution is known. The other criterion is that the numerical algorithm should produce the exact solution, within machine precision, for any chosen n.

The second criterion may seem strange at first, but take the trapezoidal method, for example, and consider an integral like $\int_a^b (6x - 4)dx$. The integrand is here a straight line, and if you sketch it in a coordinate system along with some relevant trapezoids, you realize that the topmost side of each trapezoid comes exactly on the straight line. This is the case for whatever number n of trapezoids you may choose to use. Thus, the trapezoidal method should solve that problem without any numerical error.[4] We can therefore pick some linear function and construct a test function that checks for equality between the exact solution and the numerical approximation produced by our implementation of the trapezoidal method.

Note that, when testing, numbers should not just be taken out of the air. For example, a specific test case can be $\int_{1.2}^{4.4} (6x - 4)dx$. This integral involves an "arbitrary" interval $[1.2, 4.4]$ and an "arbitrary" linear function $f(x) = 6x - 4$. By "arbitrary", we mean expressions where the special numbers 0 and 1 are avoided, since these have special properties in arithmetic operations (e.g., forgetting to multiply is equivalent to multiplying by 1, and forgetting to add is equivalent to adding 0).

Demonstrating Correct Convergence Rates Also for these unit tests we choose problems for which the exact solution is known. However, contrary to the previous test procedure, we now work with problems for which the numerical algorithm does *not* give zero approximation error. Normally, unit tests must be based on that kind of problems. Thus, the answer we get from our code will contain an approximation error, and since we know the exact solution, we may compute the size of this error. Unfortunately, this is not too helpful, since we have little chance telling if this error is what we *should* have got for the particular n used!

Now the convergence rate comes in handy. If we know (or have reason to assume) that the numerical error has a certain *asymptotic* behavior when $n \to \infty$, we know at what *rate* the numerical error should be reduced. The idea of a corresponding unit test is then to run the algorithm for some n values, compute the error (the absolute value of the difference between the exact solution and the one produced by the numerical method), and check that the error has *approximately* correct asymptotic behavior. For the trapezoidal and midpoint methods in particular, this means that the error should become proportional to n^{-2} when $n \to \infty$.

Let us develop a more precise method for such unit tests based on convergence rates. Consider a set of $q + 1$ experiments with various n: $n_0, n_1, n_2, \ldots, n_q$. We compute the corresponding errors $E_0, \ldots, E_q$. For two consecutive experiments, number i and $i - 1$, we have the error model

$$E_i = Cn_i^{-r}, \tag{6.26}$$

$$E_{i-1} = Cn_{i-1}^{-r}. \tag{6.27}$$

[4] In fact, so would the midpoint method! This is because, for each rectangle, the error to each side of the midpoint is equally large with opposite signs, meaning that they cancel each other.

These are two equations for two unknowns C and r. We can easily eliminate C by dividing, e.g., (6.26) by (6.27). Doing so, and proceeding to solve for r, followed by also introducing a subscript $i-1$ for r, gives

$$r_{i-1} = -\frac{\ln(E_i/E_{i-1})}{\ln(n_i/n_{i-1})}. \tag{6.28}$$

The subscript is introduced, since the estimated value for r will vary with i. Hopefully, r_{i-1} approaches the correct convergence rate as the number of intervals increases and $i \to q$.

6.6.3 Finite Precision of Floating-Point Numbers

The test procedures above lead to comparison of numbers for checking that calculations were correct. Such comparison is more complicated than what a newcomer might think. Suppose we have a calculation `a + b` and want to check that the result is what we expect.

Adding Integers We start with 1 + 2:

```
In [1]: a = 1; b = 2; expected = 3

In [2]: a + b == expected
Out[2]: True
```

Adding Real Numbers Then we proceed with 0.1 + 0.2:

```
In [3]: a = 0.1; b = 0.2; expected = 0.3

In [4]: a + b == expected
Out[4]: False
```

Approximate Representation of Real Numbers on a Computer So why is $0.1 + 0.2 \neq 0.3$? The reason is that, generally, real numbers cannot be represented exactly on a computer. They must instead be approximated by a floating-point number[5] that can only store a finite amount of information, usually about 17 digits of a real number. Let us print 0.1, 0.2, 0.1 + 0.2, and 0.3 with 17 decimals:

```
In [5]: print('{:.17f}\n{:.17f}\n{:.17f}\n{:.17f}'\
              .format(0.1, 0.2, 0.1 + 0.2, 0.3))
0.10000000000000001
0.20000000000000001
0.30000000000000004
0.29999999999999999
```

We see that all of the numbers have an inaccurate digit in the 17th decimal place. Because 0.1 + 0.2 evaluates to 0.30000000000000004 and 0.3 is represented as 0.29999999999999999, these two numbers are not equal. In general, real numbers in Python have (at most) 16 correct decimals.

[5] `https://en.wikipedia.org/wiki/Floating_point`.

Rounding Errors When we compute with real numbers, these numbers are inaccurately represented on the computer, and arithmetic operations with inaccurate numbers lead to small rounding errors in the final results. Depending on the type of numerical algorithm, the rounding errors may or may not accumulate.

Testing with a Tolerance If we cannot make tests like `0.1 + 0.2 == 0.3`, what should we then do? The answer is that we must accept some small inaccuracy and make a test with a *tolerance*. Here is the recipe:

```
In [1]: a = 0.1; b = 0.2; expected = 0.3

In [2]: computed = a + b

In [3]: diff = abs(expected - computed)

In [4]: tol = 1E-15

In [5]: diff < tol
Out[5]: True
```

Here we have set the tolerance for comparison to 10^{-15}, but calculating `0.3 - (0.1 + 0.2)` shows that it equals `-5.55e-17`, so a lower tolerance could be used in this particular example. However, in other calculations we have little idea about how accurate the answer is (there could be accumulation of rounding errors in more complicated algorithms), so 10^{-15} or 10^{-14} are robust values. As we demonstrate below, these tolerances depend on the magnitude of the numbers in the calculations.

Absolute and Relative Differences Doing an experiment with $10^k + 0.3 - (10^k + 0.1 + 0.2)$ for $k = 1, \ldots, 10$ shows that the answer (which should be zero) is around 10^{16-k}. This means that the tolerance must be larger if we compute with larger numbers. Setting a proper tolerance therefore requires some experiments to see what level of accuracy one can expect. A way out of this difficulty is to work with *relative* instead of *absolute* differences. In a relative difference we divide by one of the operands, e.g.,

$$a = 10^k + 0.3, \quad b = (10^k + 0.1 + 0.2), \quad c = \frac{a-b}{a}.$$

Computing this c for various k shows a value around 10^{-16}. A safer procedure is thus to use *relative differences*.

We may exemplify this in a quick session, using $k = 10$,

```
In [1]: a = 10**10 + 0.3

In [2]: b = 10**10 + 0.1 + 0.2

In [3]: diff = a-b

In [4]: diff
Out[4]: -1.9073486328125e-06

In [5]: rel_diff = (a-b)/a
```

```
In [6]: rel_diff
Out[6]: -1.9073486327552798e-16
```

Clearly, `diff` here would certainly not be smaller than a tolerance of $1E-15$ (as we used above). So, to check whether a and b are equal, we should rather proceed as

```
In [7]: tol = 1E-15

In [8]: rel_diff < tol
Out[8]: True
```

6.6.4 Constructing Unit Tests and Writing Test Functions

Python has several frameworks for automatic testing of software. By just one command, you can get Python to run through a (potentially) very large number of tests for various parts of your software. This is an extremely useful feature during program development: whenever you have done some changes to one or more files, launch the test command and make sure nothing is broken because of your edits.

The test frameworks `nose` and `py.test` are particularly attractive, since they are very easy to use. Tests are placed in special *test functions* that the frameworks can recognize and run for you. The requirements to a test function are simple:

- the name must start with `test_`
- the test function cannot have any arguments
- the tests inside test functions must be boolean expressions
- a boolean expression `b` must be tested in an *assert statement* of the form `assert b, msg`. When `b` is true, nothing happens. However, if `b` is false, `msg` is written out, where `msg` is an optional object (string or number).

Suppose we have written a function

```
def add(a, b):
    return a + b
```

A corresponding test function can then be

```
def test_add():
    expected = 2
    computed = add(1, 1)
    assert computed == expected, '1+1={:g}'.format(computed)
```

Test functions can be in any program file or in separate files, typically with names starting with `test`. You can also collect tests in subdirectories: running `py.test -s -v` will actually run all tests in all `test*.py` files in all subdirectories, while `nosetests -s -v` restricts the attention to subdirectories whose names start with `test` or end with `_test` or `_tests`.

As long as we add integers, the equality test in the `test_add` function is appropriate, but if we try to call `add(0.1, 0.2)` instead, we will face the rounding

error problems explained in Sect. 6.6.3, and we must use a test with tolerance instead:

```
def test_add():
    expected = 0.3
    computed = add(0.1, 0.2)
    tol = 1E-14
    diff = abs(expected - computed)
    assert diff < tol, 'diff={:g}'.format(diff)
```

Below we shall write test functions for each of the three test procedures we suggested: comparison with hand calculations, checking problems that can be exactly solved, and checking convergence rates. We stick to testing the trapezoidal integration code and collect all test functions in one common file by the name `test_trapezoidal.py`.

Hand-Computed Numerical Results Our previous hand calculations for two trapezoids can be utilized in a test function like this:

```
from trapezoidal import trapezoidal

def test_trapezoidal_one_exact_result():
    """Compare one hand-computed result."""
    from math import exp
    v = lambda t: 3*(t**2)*exp(t**3)
    n = 2
    computed = trapezoidal(v, 0, 1, n)
    expected = 2.463642041244344
    error = abs(expected - computed)
    tol = 1E-14
    success = error < tol
    msg = 'error={:g} > tol={:g}'.format(error, tol)
    assert success, msg
```

Note the importance of checking `computed` against `expected` with a tolerance: rounding errors from the arithmetics inside `trapezoidal` will not make the result exactly like the hand-computed one.

Solving a Problem Without Numerical Errors We know that the trapezoidal rule is exact for linear integrands. Choosing the integral $\int_{1.2}^{4.4}(6x-4)dx$ as a test case, the corresponding test function could, for example, check with three different n values, and may look like

```
def test_trapezoidal_linear():
    """Check that linear functions are integrated exactly."""
    f = lambda x: 6*x - 4
    F = lambda x: 3*x**2 - 4*x # Anti-derivative
    a = 1.2; b = 4.4
    expected = F(b) - F(a)
    tol = 1E-14
    for n in 2, 20, 21:
        computed = trapezoidal(f, a, b, n)
        error = abs(expected - computed)
        success = error < tol
        msg = 'n={:d}, err={:g}'.format(n, error)
        assert success, msg
```

Demonstrating Correct Convergence Rates Computing convergence rates requires somewhat more tedious programming than for the previous tests, but it can be applied to more general integrands. The algorithm typically goes like

- for $i = 0, 1, 2, \ldots, q$
  - $n_i = 2^{i+1}$
  - Compute integral with n_i intervals
  - Compute the error E_i
  - Estimate r_i from (6.28) if $i > 0$

The corresponding code may look like

```
def convergence_rates(f, F, a, b, num_experiments=14):
    from math import log
    from numpy import zeros
    expected = F(b) - F(a)
    n = zeros(num_experiments, dtype=int)
    E = zeros(num_experiments)
    r = zeros(num_experiments-1)
    for i in range(num_experiments):
        n[i] = 2**(i+1)
        computed = trapezoidal(f, a, b, n[i])
        E[i] = abs(expected - computed)
        if i > 0:
            r_im1 = -log(E[i]/E[i-1])/log(n[i]/n[i-1])
            # Truncate to two decimals:
            r[i-1] = float('{:.2f}'.format(r_im1))
    return r
```

Making a test function is a matter of choosing `f`, `F`, `a`, and `b`, and then checking the value of r_i for the largest i:

```
def test_trapezoidal_conv_rate():
    """Check empirical convergence rates against the expected value 2."""
    from math import exp
    v = lambda t: 3*(t**2)*exp(t**3)
    V = lambda t: exp(t**3)
    a = 1.1; b = 1.9
    r = convergence_rates(v, V, a, b, 14)
    print(r)
    tol = 0.01
    msg = str(r[-4:])  # show last 4 estimated rates
    assert (abs(r[-1]) - 2) < tol, msg
```

Running the test shows that all r_i, except the first one, equal the target limit 2 within two decimals. This observation suggests a tolerance of 10^{-2}.

6.7 Double and Triple Integrals

6.7.1 The Midpoint Rule for a Double Integral

Given a double integral over a rectangular domain $[a, b] \times [c, d]$,

$$\int_a^b \int_c^d f(x, y) dy dx,$$

how can we approximate this integral by numerical methods?

Derivation via One-Dimensional Integrals Since we know how to deal with integrals in one variable, a fruitful approach is to view the double integral as two integrals, each in one variable, which can be approximated numerically by previous one-dimensional formulas. To this end, we introduce a help function $g(x)$ and write

$$\int_a^b \int_c^d f(x, y)dydx = \int_a^b g(x)dx, \quad g(x) = \int_c^d f(x, y)dy\,.$$

Each of the integrals

$$\int_a^b g(x)dx, \quad g(x) = \int_c^d f(x, y)dy$$

can be discretized by any numerical integration rule for an integral in one variable. Let us use the midpoint method (6.20) and start with $g(x) = \int_c^d f(x, y)dy$. We introduce n_y intervals on $[c, d]$ with length h_y. The midpoint rule for this integral then becomes

$$g(x) = \int_c^d f(x, y)dy \approx h_y \sum_{j=0}^{n_y-1} f(x, y_j), \quad y_j = c + \frac{1}{2}h_y + jh_y\,.$$

The expression looks somewhat different from (6.20), but that is because of the notation: since we integrate in the y direction and will have to work with both x and y as coordinates, we must use n_y for n, h_y for h, and the counter i is more naturally called j when integrating in y. Integrals in the x direction will use h_x and n_x for h and n, and i as counter.

The double integral is $\int_a^b g(x)dx$, which can be approximated by the midpoint method:

$$\int_a^b g(x)dx \approx h_x \sum_{i=0}^{n_x-1} g(x_i), \quad x_i = a + \frac{1}{2}h_x + ih_x\,.$$

Putting the formulas together, we arrive at the *composite midpoint method for a double integral*:

$$\begin{aligned}\int_a^b \int_c^d f(x, y)dydx &\approx h_x \sum_{i=0}^{n_x-1} h_y \sum_{j=0}^{n_y-1} f(x_i, y_j) \\ &= h_x h_y \sum_{i=0}^{n_x-1} \sum_{j=0}^{n_y-1} f(a + \frac{h_x}{2} + ih_x, c + \frac{h_y}{2} + jh_y)\,.\end{aligned} \tag{6.29}$$

Direct Derivation The formula (6.29) can also be derived directly in the two-dimensional case by applying the idea of the midpoint method. We divide the rectangle $[a, b] \times [c, d]$ into $n_x \times n_y$ equal-sized parts called *cells*. The idea of the midpoint method is to approximate f by a constant over each cell, and evaluate the constant at the midpoint. Cell (i, j) occupies the area

$$[a + ih_x, a + (i + 1)h_x] \times [c + jh_y, c + (j + 1)h_y],$$

and the midpoint is (x_i, y_j) with

$$x_i = a + ih_x + \frac{1}{2}h_x, \quad y_j = c + jh_y + \frac{1}{2}h_y\,.$$

The integral over the cell is therefore $h_x h_y f(x_i, y_j)$, and the total double integral is the sum over all cells, which is nothing but formula (6.29).

Programming a Double Sum The formula (6.29) involves a double sum, which is normally implemented as a double for loop. A Python function implementing (6.29) may look like

```
def midpoint_double1(f, a, b, c, d, nx, ny):
    hx = (b - a)/nx
    hy = (d - c)/ny
    I = 0
    for i in range(nx):
        for j in range(ny):
            xi = a + hx/2 + i*hx
            yj = c + hy/2 + j*hy
            I = I + hx*hy*f(xi, yj)
    return I
```

If this function is stored in a module file `midpoint_double.py`, we can compute some integral, e.g., $\int_2^3 \int_0^2 (2x + y)dxdy = 9$ in an interactive shell and demonstrate that the function computes the right number:

```
In [1]: from midpoint_double import midpoint_double1

In [2]: def f(x, y):
   ...:     return 2*x + y
   ...:

In [3]: midpoint_double1(f, 0, 2, 2, 3, 5, 5)
Out[3]: 9.000000000000005
```

Reusing Code for One-Dimensional Integrals It is very natural to write a two-dimensional midpoint method as we did in function `midpoint_double1` when we have the formula (6.29). However, we could alternatively ask, much as we did in the mathematics, can we reuse a well-tested implementation for one-dimensional integrals to compute double integrals? That is, can we use function `midpoint`

```
def midpoint(f, a, b, n):
    h = (b-a)/n
    f_sum = 0
```

```
    for i in range(0, n, 1):
        x = (a + h/2.0) + i*h
        f_sum = f_sum + f(x)
    return h*f_sum
```

from Sect. 6.3.2 "twice"? The answer is yes, if we think as we did in the mathematics: compute the double integral as a midpoint rule for integrating $g(x)$ and define $g(x_i)$ in terms of a midpoint rule over f in the y coordinate. The corresponding function has very short code:

```
def midpoint_double2(f, a, b, c, d, nx, ny):
    def g(x):
        return midpoint(lambda y: f(x, y), c, d, ny)

    return midpoint(g, a, b, nx)
```

The important advantage of this implementation is that we reuse a well-tested function for the standard one-dimensional midpoint rule and that we apply the one-dimensional rule exactly as in the mathematics.

Verification via Test Functions How can we test that our functions for the double integral work? The best unit test is to find a problem where the numerical approximation error vanishes because then we know exactly what the numerical answer should be. The midpoint rule is exact for linear functions, regardless of how many subinterval we use. Also, any linear two-dimensional function $f(x, y) = px + qy + r$ will be integrated exactly by the two-dimensional midpoint rule. We may pick $f(x, y) = 2x + y$ and create a proper *test function* that can automatically verify our two alternative implementations of the two-dimensional midpoint rule. To compute the integral of $f(x, y)$ we take advantage of SymPy to eliminate the possibility of errors in hand calculations. The test function becomes

```
def test_midpoint_double():
    """Test that a linear function is integrated exactly."""
    def f(x, y):
        return 2*x + y

    a = 0;  b = 2;  c = 2;  d = 3
    import sympy
    x, y = sympy.symbols('x  y')
    I_expected = sympy.integrate(f(x, y), (x, a, b), (y, c, d))
    # Test three cases: nx < ny, nx = ny, nx > ny
    for nx, ny in (3, 5), (4, 4), (5, 3):
        I_computed1 = midpoint_double1(f, a, b, c, d, nx, ny)
        I_computed2 = midpoint_double2(f, a, b, c, d, nx, ny)
        tol = 1E-14
        #print I_expected, I_computed1, I_computed2
        assert abs(I_computed1 - I_expected) < tol
        assert abs(I_computed2 - I_expected) < tol
```

Let Test Functions Speak Up?

If we call the above `test_midpoint_double` function and nothing happens, our implementations are correct. However, it is somewhat annoying to have a function that is completely silent when it works—are we sure all things are properly computed? During development it is therefore highly recommended to insert a print command such that we can monitor the calculations and be convinced that the test function does what we want. Since a test function should not have any print command, we simply comment it out as we have done in the function listed above.

The trapezoidal method can be used as alternative for the midpoint method. The derivation of a formula for the double integral and the implementations follow exactly the same ideas as we explained with the midpoint method, but there are more terms to write in the formulas. Exercise 6.13 asks you to carry out the details. That exercise is a very good test on your understanding of the mathematical and programming ideas in the present section.

6.7.2 The Midpoint Rule for a Triple Integral

Theory Once a method that works for a one-dimensional problem is generalized to two dimensions, it is usually quite straightforward to extend the method to three dimensions. This will now be demonstrated for integrals. We have the triple integral

$$\int_a^b \int_c^d \int_e^f g(x, y, z)dzdydx$$

and want to approximate the integral by a midpoint rule. Following the ideas for the double integral, we split this integral into one-dimensional integrals:

$$p(x, y) = \int_e^f g(x, y, z)dz$$

$$q(x) = \int_c^d p(x, y)dy$$

$$\int_a^b \int_c^d \int_e^f g(x, y, z)dzdydx = \int_a^b q(x)dx$$

For each of these one-dimensional integrals we apply the midpoint rule:

$$p(x,y) = \int_e^f g(x,y,z)dz \approx \sum_{k=0}^{n_z-1} g(x,y,z_k),$$

$$q(x) = \int_c^d p(x,y)dy \approx \sum_{j=0}^{n_y-1} p(x,y_j),$$

$$\int_a^b \int_c^d \int_e^f g(x,y,z)dzdydx = \int_a^b q(x)dx \approx \sum_{i=0}^{n_x-1} q(x_i),$$

where

$$z_k = e + \frac{1}{2}h_z + kh_z, \quad y_j = c + \frac{1}{2}h_y + jh_y \quad x_i = a + \frac{1}{2}h_x + ih_x\,.$$

Starting with the formula for $\int_a^b \int_c^d \int_e^f g(x,y,z)dzdydx$ and inserting the two previous formulas gives

$$\int_a^b \int_c^d \int_e^f g(x,y,z)\,dzdydx \approx$$
$$h_x h_y h_z \sum_{i=0}^{n_x-1} \sum_{j=0}^{n_y-1} \sum_{k=0}^{n_z-1} g(a + \frac{1}{2}h_x + ih_x, c + \frac{1}{2}h_y + jh_y, e + \frac{1}{2}h_z + kh_z)\,. \tag{6.30}$$

Note that we may apply the ideas under *Direct derivation* at the end of Sect. 6.7.1 to arrive at (6.30) directly: divide the domain into $n_x \times n_y \times n_z$ cells of volumes $h_x h_y h_z$; approximate g by a constant, evaluated at the midpoint (x_i, y_j, z_k), in each cell; and sum the cell integrals $h_x h_y h_z g(x_i, y_j, z_k)$.

Implementation We follow the ideas for the implementations of the midpoint rule for a double integral. The corresponding functions are shown below and found in the file `midpoint_triple.py`.

```
def midpoint_triple1(g, a, b, c, d, e, f, nx, ny, nz):
    hx = (b - a)/nx
    hy = (d - c)/ny
    hz = (f - e)/nz
    I = 0
    for i in range(nx):
        for j in range(ny):
            for k in range(nz):
                xi = a + hx/2 + i*hx
                yj = c + hy/2 + j*hy
                zk = e + hz/2 + k*hz
                I = I + hx*hy*hz*g(xi, yj, zk)
    return I
```

```
def midpoint(f, a, b, n):
    h = (b-a)/n
    f_sum = 0
    for i in range(0, n, 1):
        x = (a + h/2.0) + i*h
        f_sum = f_sum + f(x)
    return h*f_sum

def midpoint_triple2(g, a, b, c, d, e, f, nx, ny, nz):
    def p(x, y):
        return midpoint(lambda z: g(x, y, z), e, f, nz)

    def q(x):
        return midpoint(lambda y: p(x, y), c, d, ny)

    return midpoint(q, a, b, nx)

def test_midpoint_triple():
    """Test that a linear function is integrated exactly."""
    def g(x, y, z):
        return 2*x + y - 4*z

    a = 0;  b = 2;  c = 2;  d = 3;  e = -1;  f = 2
    import sympy
    x, y, z = sympy.symbols('x y z')
    I_expected = sympy.integrate(
        g(x, y, z), (x, a, b), (y, c, d), (z, e, f))
    for nx, ny, nz in (3, 5, 2), (4, 4, 4), (5, 3, 6):
        I_computed1 = midpoint_triple1(
            g, a, b, c, d, e, f, nx, ny, nz)
        I_computed2 = midpoint_triple2(
            g, a, b, c, d, e, f, nx, ny, nz)
        tol = 1E-14
        print(I_expected, I_computed1, I_computed2)
        assert abs(I_computed1 - I_expected) < tol
        assert abs(I_computed2 - I_expected) < tol

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

6.7.3 Monte Carlo Integration for Complex-Shaped Domains

Repeated use of one-dimensional integration rules to handle double and triple integrals constitute a working strategy only if the integration domain is a rectangle or box. For any other shape of domain, completely different methods must be used. A common approach for two- and three-dimensional domains is to divide the domain into many small triangles or tetrahedra and use numerical integration methods for each triangle or tetrahedron. The overall algorithm and implementation is too complicated to be addressed in this book. Instead, we shall employ an alternative, very simple and general method, called Monte Carlo integration. It can

be implemented in half a page of code, but requires orders of magnitude more function evaluations in double integrals compared to the midpoint rule.

Monte Carlo integration, however, is much more computationally efficient than the midpoint rule when computing higher-dimensional integrals in more than three variables over hypercube domains. Our ideas for double and triple integrals can easily be generalized to handle an integral in m variables. A midpoint formula then involves m sums. With n cells in each coordinate direction, the formula requires n^m function evaluations. That is, the computational work explodes as an exponential function of the number of space dimensions. Monte Carlo integration, on the other hand, does not suffer from this explosion of computational work and is the preferred method for computing higher-dimensional integrals. So, it makes sense in a chapter on numerical integration to address Monte Carlo methods, both for handling complex domains and for handling integrals with many variables.

The Monte Carlo Integration Algorithm The idea of Monte Carlo integration of $\int_a^b f(x)dx$ is to use the mean-value theorem from calculus, which states that the integral $\int_a^b f(x)dx$ equals the length of the integration domain, here $b-a$, times the *average* value of f, $\bar{f}$, in $[a,b]$. The average value can be computed by sampling f at a set of *random* points inside the domain and take the mean of the function values. In higher dimensions, an integral is estimated as the area/volume of the domain times the average value, and again one can evaluate the integrand at a set of random points in the domain and compute the mean value of those evaluations.

Let us introduce some quantities to help us make the specification of the integration algorithm more precise. Suppose we have some two-dimensional integral

$$\int_\Omega f(x,y)dxdx,$$

where Ω is a two-dimensional domain defined via a help function $g(x,y)$:

$$\Omega = \{(x,y) \mid g(x,y) \geq 0\}$$

That is, points (x,y) for which $g(x,y) \geq 0$ lie inside Ω, and points for which $g(x,y) < \Omega$ are outside Ω. The boundary of the domain $\partial\Omega$ is given by the implicit curve $g(x,y) = 0$. Such formulations of geometries have been very common during the last couple of decades, and one refers to g as a *level-set function* and the boundary $g = 0$ as the zero-level contour of the level-set function. For simple geometries one can easily construct g by hand, while in more complicated industrial applications one must resort to mathematical models for constructing g.

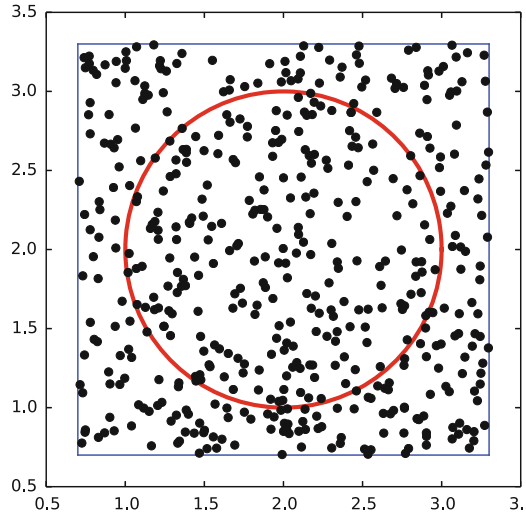
Let $A(\Omega)$ be the area of a domain Ω. We can estimate the integral by this Monte Carlo integration method:

1. embed the geometry Ω in a rectangular area R
2. draw a large number of *random* points (x,y) in R

3. count the fraction q of points that are inside Ω
4. approximate $A(\Omega)/A(R)$ by q, i.e., set $A(\Omega) = qA(R)$
5. evaluate the mean of f, $\bar{f}$, at the points inside Ω
6. estimate the integral as $A(\Omega)\bar{f}$

Note that $A(R)$ is trivial to compute since R is a rectangle, while $A(\Omega)$ is unknown. However, if we assume that the fraction of $A(R)$ occupied by $A(\Omega)$ is the same as the fraction of random points inside Ω, we get a simple estimate for $A(\Omega)$.

To get an idea of the method, consider a circular domain Ω embedded in a rectangle as shown below. A collection of random points is illustrated by black dots.



Implementation A Python function implementing $\int_\Omega f(x, y)dxdy$ can be written like this:

```
import numpy as np

def MonteCarlo_double(f, g, x0, x1, y0, y1, n):
    """
    Monte Carlo integration of f over a domain g>=0, embedded
    in a rectangle [x0,x1]x[y0,y1]. n^2 is the number of
    random points.
    """
    # Draw n**2 random points in the rectangle
    x = np.random.uniform(x0, x1, n)
    y = np.random.uniform(y0, y1, n)
    # Compute sum of f values inside the integration domain
    f_mean = 0
    num_inside = 0   # number of x,y points inside domain (g>=0)
    for i in range(len(x)):
        for j in range(len(y)):
            if g(x[i], y[j]) >= 0:
                num_inside = num_inside + 1
```

```
                f_mean = f_mean + f(x[i], y[j])
    f_mean = f_mean/num_inside
    area = num_inside/(n**2)*(x1 - x0)*(y1 - y0)
    return area*f_mean
```

(See the file MC_double.py.)

Verification A simple test case is to check the area of a rectangle $[0, 2] \times [3, 4.5]$ embedded in a rectangle $[0, 3] \times [2, 5]$. The right answer is 3, but Monte Carlo integration is, unfortunately, never exact so it is impossible to predict the output of the algorithm. All we know is that the estimated integral should approach 3 as the number of random points goes to infinity. Also, for a fixed number of points, we can run the algorithm several times and get different numbers that fluctuate around the exact value, since different sample points are used in different calls to the Monte Carlo integration algorithm.

The area of the rectangle can be computed by the integral $\int_0^2 \int_3^{4.5} dydx$, so in this case we identify $f(x, y) = 1$, and the g function can be specified as (e.g.) 1 if (x, y) is inside $[0, 2] \times [3, 4.5]$ and -1 otherwise. Here is an example, using samples of different sizes, on how we can utilize the `MonteCarlo_double` function to compute the area:

```
In [1]: from MC_double import MonteCarlo_double

In [2]: def g(x, y):
   ...:     return (1 if (0 <= x <= 2 and 3 <= y <= 4.5) else -1)
   ...:

In [3]: MonteCarlo_double(lambda x, y: 1, g, 0, 3, 2, 5, 100)
Out[3]: 2.9484

In [4]: MonteCarlo_double(lambda x, y: 1, g, 0, 3, 2, 5, 1000)
Out[4]: 2.947032

In [5]: MonteCarlo_double(lambda x, y: 1, g, 0, 3, 2, 5, 1000)
Out[5]: 3.0234600000000005

In [6]: MonteCarlo_double(lambda x, y: 1, g, 0, 3, 2, 5, 2000)
Out[6]: 2.9984580000000003

In [7]: MonteCarlo_double(lambda x, y: 1, g, 0, 3, 2, 5, 2000)
Out[7]: 3.1903469999999996

In [8]: MonteCarlo_double(lambda x, y: 1, g, 0, 3, 2, 5, 5000)
Out[8]: 2.986515
```

Note the compact `if-else` construction in the definition of g. It is a one-line alternative to, for example,

```
if (0 <= x <= 2) and (3 <= y <= 4.5):
    return 1
else:
    return -1
```

From the output, we see that the values fluctuate around 3, a fact that supports a correct implementation, but in principle, bugs could be hidden behind the inaccurate answers.

It is mathematically known that the standard deviation of the Monte Carlo estimate of an integral converges as $n^{-1/2}$, where n is the number of samples. This kind of convergence rate estimate could be used to verify the implementation, but the topic is beyond the scope of this book.

Test Function for Function with Random Numbers To make a test function, we need a unit test that has identical behavior each time we run the test. Thus, since the algorithm generates pseudo-random numbers, we apply the standard technique of fixing the seed (of the random number generator), so that the sequence of numbers generated is the same every time we run the algorithm. Assuming that the `MonteCarlo_double` function works, we fix the seed, observe a certain result, and take this result as the correct result. Provided the test function always uses this seed, we should get exactly this result every time the `MonteCarlo_double` function is called. Of course, this procedure does not test whether the `MonteCarlo_double` function works right now (but we hope our assumption of correctness is well founded!). Still, it is nevertheless useful when future changes are made, since at any time we can confirm that `MonteCarlo_double` gives the same answer as before. The test function can be written as shown below.

```
def test_MonteCarlo_double_rectangle_area():
    """Check the area of a rectangle."""
    def g(x, y):
        return (1 if (0 <= x <= 2 and 3 <= y <= 4.5) else -1)

    x0 = 0;  x1 = 3;  y0 = 2;  y1 = 5  # embedded rectangle
    n = 1000
    np.random.seed(8)       # must fix the seed!
    I_expected = 3.121092  # computed with this seed
    I_computed = MonteCarlo_double(
        lambda x, y: 1, g, x0, x1, y0, y1, n)
    assert abs(I_expected - I_computed) < 1E-14
```

(See the file MC_double.py.)

Integral Over a Circle The test above involves a trivial function $f(x, y) = 1$. We should also test a non-constant f function and a more complicated domain. Let Ω be a circle at the origin with radius 2, and let $f = \sqrt{x^2 + y^2}$. This choice makes it possible to compute an exact result: in polar coordinates, $\int_\Omega f(x, y)dxdy$ simplifies to $2\pi \int_0^2 r^2 dr = 16\pi/3$. We must be prepared for quite crude approximations that fluctuate around this exact result. As in the test case above, we experience better results with larger number of points. When we have such evidence for a working implementation, we can turn the test into a proper test function. Here is an example:

```
def test_MonteCarlo_double_circle_r():
    """Check the integral of r over a circle with radius 2."""
    def g(x, y):
        xc, yc = 0, 0  # center
        R = 2          # radius
        return  R**2 - ((x-xc)**2 + (y-yc)**2)

    # Exact: integral of r*r*dr over circle with radius R becomes
    # 2*pi*1/3*R**3
    import sympy
```

```
r = sympy.symbols('r')
I_exact = sympy.integrate(2*sympy.pi*r*r, (r, 0, 2))
print('Exact integral: {:g}'.format(I_exact.evalf()))
x0 = -2;  x1 = 2;  y0 = -2;  y1 = 2
n = 1000
np.random.seed(6)
I_expected = 16.7970837117376384  # Computed with this seed
I_computed = MonteCarlo_double(
    lambda x, y: np.sqrt(x**2 + y**2),
    g, x0, x1, y0, y1, n)
print('MC approximation, {:d} samples: {:.16f}'\
      .format(n**2, I_computed))
assert abs(I_expected - I_computed) < 1E-15
```

(See the file `MC_double.py`.)

Remark About Version Control of Files

Having a suite of test functions for automatically checking that your software works is considered as a fundamental requirement for reliable computing. Equally important is a system that can keep track of different versions of the files and the tests, known as a *version control system*. Today's most popular version control system is Git,[a] which the authors strongly recommend the reader to use for programming and writing reports. The combination of Git and cloud storage such as GitHub is a very common way of organizing scientific or engineering work. We have a quick intro[b] to Git and GitHub that gets you up and running within minutes.

The typical workflow with Git goes as follows.

1. Before you start working with files, make sure you have the latest version of them by running `git pull`.
2. Edit files, remove or create files (new files must be registered by `git add`).
3. When a natural piece of work is done, *commit* your changes by the `git commit` command.
4. Implement your changes also in the cloud by doing `git push`.

A nice feature of Git is that people can edit the same file at the same time and very often Git will be able to automatically merge the changes (!). Therefore, version control is crucial when you work with others or when you do your work on different types of computers. Another key feature is that anyone can at any time view the history of a file, see who did what when, and roll back the entire file collection to a previous commit. This feature is, of course, fundamental for reliable work.

[a] https://en.wikipedia.org/wiki/Git_(software).
[b] http://hplgit.github.io/teamods/bitgit/Langtangen_bitgit-bootstrap.html.

6.8 Exercises

Exercise 6.1: Hand Calculations for the Trapezoidal Method
Compute by hand the area composed of two trapezoids (of equal width) that approximates the integral $\int_1^3 2x^3 dx$. Make a test function that calls the `trapezoidal` function in `trapezoidal.py` and compares the return value with the hand-calculated value.
Filename: `trapezoidal_test_func.py`.

Exercise 6.2: Hand Calculations for the Midpoint Method
Compute by hand the area composed of two rectangles (of equal width) that approximates the integral $\int_1^3 2x^3 dx$. Make a test function that calls the `midpoint` function in `midpoint.py` and compares the return value with the hand-calculated value.
Filename: `midpoint_test_func.py`.

Exercise 6.3: Compute a Simple Integral
Apply the `trapezoidal` and `midpoint` functions to compute the integral $\int_2^6 x(x-1)dx$ with 2 and 100 subintervals. Compute the error too.
Filename: `integrate_parabola.py`.

Exercise 6.4: Hand-Calculations with Sine Integrals
We consider integrating the sine function: $\int_0^b \sin(x)dx$.

a) Let $b = \pi$ and use two intervals in the trapezoidal and midpoint method. Compute the integral by hand and illustrate how the two numerical methods approximate the integral. Compare with the exact value.
b) Do a) when $b = 2\pi$.

Filename: `integrate_sine.py`.

Exercise 6.5: Make Test Functions for the Midpoint Method
Modify the file `test_trapezoidal.py` such that the three tests are applied to the function `midpoint` implementing the midpoint method for integration.
Filename: `test_midpoint.py`.

Exercise 6.6: Explore Rounding Errors with Large Numbers
The trapezoidal method integrates linear functions exactly, and this property was used in the test function `test_trapezoidal_linear` in the file `test_trapezoidal.py`. Change the function used in Sect. 6.6.2 to $f(x) = 6 \cdot 10^8 x - 4 \cdot 10^6$ and rerun the test. What happens? How must you change the test to make it useful? How does the convergence rate test behave? Any need for adjustment?
Filename: `test_trapezoidal2.py`.

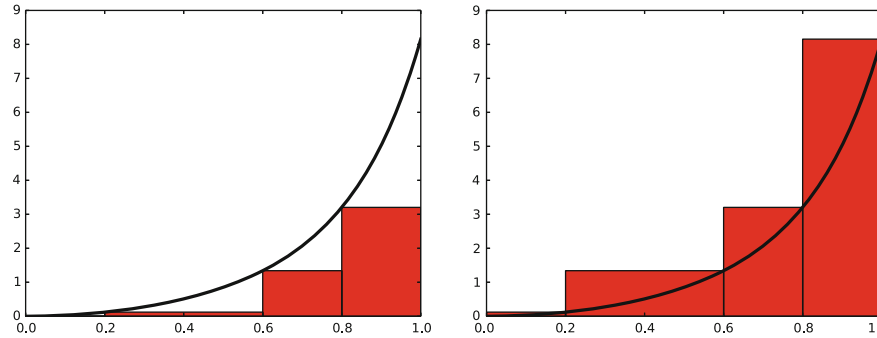


Fig. 6.4 Illustration of the rectangle method with evaluating the rectangle height by either the left or right point

Exercise 6.7: Write Test Functions for $\int_0^4 \sqrt{x}dx$

We want to test how the `trapezoidal` function works for the integral $\int_0^4 \sqrt{x}dx$. Two of the tests in `test_trapezoidal.py` are meaningful for this integral. Compute by hand the result of using two or three trapezoids and modify the `test_trapezoidal_one_exact_result` function accordingly. Then modify `test_trapezoidal_conv_rate` to handle the square root integral.
Filename: `test_trapezoidal3.py`.

Remarks The convergence rate test fails. Printing out `r` shows that the actual convergence rate for this integral is 1.5 and not 2. The reason is that the error in the trapezoidal method[6] is $-(b-a)^3 n^{-2} f''(\xi)$ for some (unknown) $\xi \in [a,b]$. With $f(x) = \sqrt{x}$, $f''(\xi) \to -\infty$ as $\xi \to 0$, pointing to a potential problem in the size of the error. Running a test with $a > 0$, say $\int_{0.1}^4 \sqrt{x}dx$ shows that the convergence rate is indeed restored to 2.

Exercise 6.8: Rectangle Methods

The midpoint method divides the interval of integration into equal-sized subintervals and approximates the integral in each subinterval by a rectangle whose height equals the function value at the midpoint of the subinterval. Instead, one might use either the left or right end of the subinterval as illustrated in Fig. 6.4. This defines a *rectangle method* of integration. The height of the rectangle can be based on the left or right end or the midpoint.

a) Write a function `rectangle(f, a, b, n, height='left')` for computing an integral $\int_a^b f(x)dx$ by the rectangle method with height computed based on the value of `height`, which is either `left`, `right`, or `mid`.
b) Write three test functions for the three unit test procedures described in Sect. 6.6.2. Make sure you test for `height` equal to `left`, `right`, and `mid`. You may call the `midpoint` function for checking the result when `height=mid`.

[6] http://en.wikipedia.org/wiki/Trapezoidal_rule#Error_analysis.

Hint Edit `test_trapezoidal.py`.
Filename: `rectangle_methods.py`.

Exercise 6.9: Adaptive Integration
Suppose we want to use the trapezoidal or midpoint method to compute an integral $\int_a^b f(x)dx$ with an error less than a prescribed tolerance ϵ. What is the appropriate size of n?

To answer this question, we may enter an iterative procedure where we compare the results produced by n and $2n$ intervals, and if the difference is smaller than ϵ, the value corresponding to $2n$ is returned. Otherwise, we halve n and repeat the procedure.

Hint It may be a good idea to organize your code so that the function `adaptive_integration` can be used easily in future programs you write.

a) Write a function

```
adaptive_integration(f, a, b, eps, method=midpoint)
```

that implements the idea above (`eps` corresponds to the tolerance ϵ, and `method` can be `midpoint` or `trapezoidal`).

b) Test the method on $\int_0^2 x^2 dx$ and $\int_0^2 \sqrt{x}dx$ for $\epsilon = 10^{-1}, 10^{-10}$ and write out the exact error.
c) Make a plot of n versus $\epsilon \in [10^{-1}, 10^{-10}]$ for $\int_0^2 \sqrt{x}dx$. Use logarithmic scale for ϵ.

Filename: `adaptive_integration.py`.

Remarks The type of method explored in this exercise is called *adaptive*, because it tries to adapt the value of n to meet a given error criterion. The true error can very seldom be computed (since we do not know the exact answer to the computational problem), so one has to find other indicators of the error, such as the one here where the changes in the integral value, as the number of intervals is doubled, is taken to reflect the error.

Exercise 6.10: Integrating x Raised to x
Consider the integral

$$I = \int_0^4 x^x \, dx \, .$$

The integrand x^x does not have an anti-derivative that can be expressed in terms of standard functions (visit http://wolframalpha.com and type `integral(x**x,x)` to convince yourself that our claim is right. Note that Wolfram alpha does give you an answer, but that answer is an approximation, it is *not* exact. This is because Wolfram alpha too uses numerical methods to arrive at the answer, just as you will in this

exercise). Therefore, we are forced to compute the integral by numerical methods. Compute a result that is right to four digits.

Hint Use ideas from Exercise 6.9.
Filename: `integrate_x2x.py`.

Exercise 6.11: Integrate Products of Sine Functions
In this exercise we shall integrate

$$I_{j,k} = \int_{-\pi}^{\pi} \sin(jx)\sin(kx)dx,$$

where j and k are integers.

a) Plot $\sin(x)\sin(2x)$ and $\sin(2x)\sin(3x)$ for $x \in [-\pi, \pi]$ in separate plots. Explain why you expect $\int_{-\pi}^{\pi} \sin x \sin 2x\, dx = 0$ and $\int_{-\pi}^{\pi} \sin 2x \sin 3x\, dx = 0$.
b) Use the trapezoidal rule to compute $I_{j,k}$ for $j = 1, \ldots, 10$ and $k = 1, \ldots, 10$.

Filename: `products_sines.py`.

Exercise 6.12: Revisit Fit of Sines to a Function
This is a continuation of Exercise 4.13. The task is to approximate a given function $f(t)$ on $[-\pi, \pi]$ by a sum of sines,

$$S_N(t) = \sum_{n=1}^{N} b_n \sin(nt)\,. \tag{6.31}$$

We are now interested in computing the unknown coefficients b_n such that $S_N(t)$ is in some sense the *best approximation* to $f(t)$. One common way of doing this is to first set up a general expression for the *approximation error*, measured by "summing up" the squared deviation of S_N from f:

$$E = \int_{-\pi}^{\pi} (S_N(t) - f(t))^2 dt\,.$$

We may view E as a function of $b_1, \ldots, b_N$. Minimizing E with respect to $b_1, \ldots, b_N$ will give us a *best approximation*, in the sense that we adjust $b_1, \ldots, b_N$ such that S_N deviates as little as possible from f.

Minimization of a function of N variables, $E(b_1, \ldots, b_N)$ is mathematically performed by requiring all the partial derivatives to be zero:

$$\frac{\partial E}{\partial b_1} = 0,$$
$$\frac{\partial E}{\partial b_2} = 0,$$
$$\vdots$$
$$\frac{\partial E}{\partial b_N} = 0\,.$$

a) Compute the partial derivative $\partial E/\partial b_1$ and generalize to the arbitrary case $\partial E/\partial b_n$, $1 \leq n \leq N$.
b) Show that

$$b_n = \frac{1}{\pi}\int_{-\pi}^{\pi} f(t)\sin(nt)\,dt\,.$$

c) Write a function `integrate_coeffs(f, N, M)` that computes $b_1, \ldots, b_N$ by numerical integration, using M intervals in the trapezoidal rule.
d) A remarkable property of the trapezoidal rule is that it is exact for integrals $\int_{-\pi}^{\pi} \sin nt\,dt$ (when subintervals are of equal size). Use this property to create a function `test_integrate_coeff` to verify the implementation of `integrate_coeffs`.
e) Implement the choice $f(t) = \frac{1}{\pi}t$ as a Python function `f(t)` and call `integrate_coeffs(f, 3, 100)` to see what the optimal choice of b_1, b_2, b_3 is.
f) Make a function `plot_approx(f, N, M, filename)` where you plot `f(t)` together with the best approximation S_N as computed above, using M intervals for numerical integration. Save the plot to a file with name `filename`.
g) Run `plot_approx(f, N, M, filename)` for $f(t) = \frac{1}{\pi}t$ for $N = 3, 6, 12, 24$. Observe how the approximation improves.
h) Run `plot_approx` for $f(t) = e^{-(t-\pi)}$ and $N = 100$. Observe a fundamental problem: regardless of N, $S_N(-\pi) = 0$, not $e^{2\pi} \approx 535$. (There are ways to fix this issue.)

Filename: `autofit_sines.py`.

Exercise 6.13: Derive the Trapezoidal Rule for a Double Integral
Use ideas in Sect. 6.7.1 to derive a formula for computing a double integral $\int_a^b \int_c^d f(x, y)dydx$ by the trapezoidal rule. Implement and test this rule.
Filename: `trapezoidal_double.py`.

Exercise 6.14: Compute the Area of a Triangle by Monte Carlo Integration
Use the Monte Carlo method from Sect. 6.7.3 to compute the area of a triangle with vertices at $(-1, 0)$, $(1, 0)$, and $(3, 0)$.
Filename: `MC_triangle.py`.