Numerical Integration

October 19, 2020

Anne Kværnø, revised by **Markus Grasmair**
Date: **Oct 14, 2018** Revision: **Oct 18, 2020**

1 Introduction

In this note, we will discuss numerical methods for the approximation of (finite) integrals of the form

$$I[f](a,b) = \int_a^b f(x)dx.$$

A *numerical quadrature* or a *quadrature rule* is a formula for approximating such integrals. Quadratures are usually of the form

$$Q[f](a,b) = \sum_{i=0}^{n} w_i f(x_i),$$

where x_i, w_i for $i = 0, 1, \ldots, n$ are respectively the *nodes* and the *weights* of the quadrature rule. If the function f is given from the context, we will for simplicity denote the integral and the quadrature simply as $I(a,b)$ and $Q(a,b)$. Examples of numerical quadratures known from previous courses are the trapezoidal rule, the midpoint rule and Simpson's rule.

In practice, we will not use a single (or *simple*) quadrature for the whole interval $[a,b]$, but rather choose a partitioning

$$a = X_0 < X_1 \cdots < X_m = b$$

into m sub-intervals, apply a quadrature on each of the subintervals $[X_j, X_{j+1}]$, and then add the results together. This leads to *composite quadratures* yielding the approximation

$$I[f](a,b) = \sum_{j=0}^{m-1} I[f](X_j, X_{j+1}) \approx \sum_{j=0}^{m-1} Q[f](X_j, X_{j+1}).$$

In this note we will see how quadrature rules can be constructed from integration of interpolation polynomials. We will demonstrate how to do error analysis and how to find error estimates both for simple and composite quadratures. Moreover, we will demonstrate how the partitioning of the integration interval can be chosen automatically based on these error estimates; this idea is called *adaptive integration*.

In the sequel, we will use material from *Preliminaries*, Sections 3.2, 4 and 5.

2 Quadrature based on polynomial interpolation.

This section relies on the content of the note on polynomial interpolation, in particular the section on Lagrange polynomials.

Choose $n+1$ distinct nodes x_i, $i = 0, \dots, n$ in the interval $[a,b]$, and let $p_n(x)$ be the interpolation polynomial satisfying the interpolation condition

$$p_n(x_i) = f(x_i), \qquad i = 0, 1, \dots .$$

We will then use $\int_a^b p_n(x)dx$ as an approximation to $\int_a^b f(x)dx$. By using the Lagrange form of the polynomial

$$p_n(x) = \sum_{i=0}^{n} f(x_i)\ell_i(x)$$

with the cardinal functions $\ell_i(x)$ given by

$$\ell_i(x) = \prod_{j=0, j\neq i}^{n} \frac{x - x_j}{x_i - x_j},$$

the following quadrature formula is obtained:

$$Q[f](a,b) = \int_a^b p_n(x)dx = \sum_{i=0}^{n} f(x_i) \int_a^b \ell_i(x)dx = \sum_{i=0}^{n} w_i f(x_i).$$

The weights in the quadrature are simply the integrals of the cardinal functions over the interval:

$$w_i = \int_a^b \ell_i(x)\, dx.$$

Let us derive two schemes for integration over the interval $[0,1]$, and apply them to the integral

$$I(0,1) = \int_0^1 \cos\left(\frac{\pi}{2}x\right) = \frac{2}{\pi} = 0.636619\dots .$$

Example 1 (trapezoidal rule): Let $x_0 = 0$ and $x_1 = 1$. The cardinal functions and thus the weights are given by

$$\ell_0(x) = 1 - x, \quad w_0 = \int_0^1 (1-x)dx = 1/2,$$
$$\ell_1(x) = x, \qquad w_1 = \int_0^1 x dx = 1/2.$$

The corresponding quadrature rule, better known as the trapezoidal rule and usually denoted by T, is given by

$$T(0,1) = \frac{1}{2}\left[f(0) + f(1)\right].$$

This formula applied to the function $f(x) = \cos(\pi x/2)$ gives

$$T(0,1) = \frac{1}{2}\left[\cos(0) + \cos\left(\frac{\pi}{2}\right)\right] = \frac{1}{2},$$

and the error is

$$I(0,1) - T(0,1) = \frac{2}{\pi} - \frac{1}{2} = 0.138\dots$$

Example 2 (Gauß-Legendre quadrature with two nodes): Let $x_0 = 1/2 - \sqrt{3}/6$ and $x_1 = 1/2 + \sqrt{3}/6$. Then

$$\begin{aligned} \ell_0(x) &= -\sqrt{3}x + \frac{1+\sqrt{3}}{2}, \quad w_0 = \int_0^1 \ell_0(x)dx = 1/2, \\ \ell_1(x) &= \sqrt{3}x + \frac{1-\sqrt{3}}{2}, \qquad w_1 = \int_0^1 \ell_1(x)dx = 1/2. \end{aligned}$$

The quadrature rule is

$$Q(0,1) = \frac{1}{2}\left[f\left(\frac{1}{2} - \frac{\sqrt{3}}{6}\right) + f\left(\frac{1}{2} + \frac{\sqrt{3}}{6}\right)\right].$$

And this quadrature applied to $f(x) = \cos(\pi x/2)$ is given by

$$Q(0,1) = \frac{1}{2}\left[\cos\left(\frac{\pi}{2}x_0\right) + \cos\left(\frac{\pi}{2}x_1\right)\right] = 0.635647\ldots$$

with an error

$$I(0,1) - Q(0,1) = 9.72\ldots \cdot 10^{-4}.$$

So the choice of nodes clearly matters!

Before concluding this section, let us present simple indication on the quality of a method:

Definition (degree of precision). A numerical quadrature has degree of precision d if:

- $Q[p](a,b) = I[p](a,b)$ for all $p \in \mathbb{P}_d$.
- There exists $p \in \mathbb{P}_{d+1}$ with $Q[p](a,b) \neq I[p](a,b)$.

Since both integrals and quadratures are linear in the integrand p, the degree of precision is equal to d if and only if the following conditions hold:

$$\begin{aligned} Q[x^j](a,b) &= I[x^j](a,b), \qquad j = 0,1,\ldots,d, \\ Q[x^{d+1}](a,b) &\neq I[x^{d+1}](a,b). \end{aligned}$$

All quadratures constructed from Lagrange interpolation polynomials in $n+1$ distinct nodes will automatically be of precision at least n. This follows immediately from the way these quadratures are constructed: Indeed, if $p \in \mathbb{P}_n$ is a polynomial of degree at most n, then the interpolation polynomial will simply be equal to p itself, and thus the integration is performed exactly.

Note, however, that the degree of precision can actually be larger than n. It is left to the reader to show that the trapezoidal rule from Example 1 has degree of precision 1, whereas the formula from Example 2 has degree of precision 3.

3 Construction of numerical quadratures

In the following, you will learn the steps on how to construct realistic algorithms for numerical integration, similar to those used in software like Matlab of SciPy. The steps are:

Construction.

1. Choose $n+1$ distinct nodes on a standard interval $[-1,1]$.
2. Let $p_n(x)$ be the polynomial interpolating some general function f in the nodes, and define the quadrature on $[-1,1]$ to be $Q[f](-1,1) := I[p_n](-1,1)$.
3. Transfer the formula Q from $[-1,1]$ to any arbitrary interval $[a,b]$.
4. Find the composite formula by dividing the interval $[a,b]$ into subintervals and applying the quadrature formula on each subinterval.
5. Find an expression for the error $E[f](a,b) = I[f](a,b) - Q[f](a,b)$.
6. Find an expression for an estimate of the error, and use this to create an adaptive algorithm.

3.1 Simpson's rule

We will go through the steps above for one method, Simpson's formula. The strategy is quite generic, so it is more important to understand and remember how results are derived, not exactly what they are. The different algorithms will be implemented and tested, and theoretical results will be verified by numerical experiments.

We will adopt the standard notation and denote this particular quadrature by $S[f](a,b)$.

3.1.1 The quadrature formula on the standard interval [-1,1]

The quadrature rule is defined by the choice of nodes on a standard interval $[-1,1]$. For Simpson's rule, choose the nodes $t_0 = -1$, $t_1 = 0$ and $t_2 = 1$. The corresponding cardinal functions are

$$\ell_0 = \frac{1}{2}(t^2 - t), \qquad \ell_1(t) = 1 - t^2, \qquad \ell_2(t) = \frac{1}{2}(t^2 + t).$$

which gives the weights

$$w_0 = \int_{-1}^{1} \ell_0(t)dt = \frac{1}{3}, \qquad w_1 = \int_{-1}^{1} \ell_1(t)dt = \frac{4}{3}, \qquad w_2 = \int_{-1}^{1} \ell_2(t)dt = \frac{1}{3}$$

such that

$$\int_{-1}^{1} f(t)dt \approx \int_{-1}^{1} p_2(t)dt = \sum_{i=0}^{2} w_i f(t_i) = \frac{1}{3}\left[\, f(-1) + 4f(0) + f(1)\,\right].$$

Simpson's rule has degree of precision 3 (check it yourself).

Example 3: With this quadrature, we obtain

$$\int_{-1}^{1} \cos\left(\frac{\pi t}{2}\right) dt = \frac{4}{\pi} \approx \frac{1}{3}\left[\cos(-\pi/2) + 4\cos(0) + \cos(\pi/2)\right] = \frac{4}{3}.$$

3.1.2 Transfer the integral and the quadrature to the interval $[a, b]$

The integral and the quadrature are transferred to some arbitrary interval $[a, b]$ by the transformation

$$x = \frac{b-a}{2}t + \frac{b+a}{2}, \qquad \text{so} \qquad dx = \frac{b-a}{2}dt.$$

By this transformation, the nodes $t_0 = -1$, $t_1 = 0$, and $t_2 = 1$ in the interval $[-1, 1]$ are mapped to

$$x_0 = a, \qquad x_1 = \frac{a+b}{2}, \qquad x_2 = b.$$

Thus we obtain the quadrature

$$\int_a^b f(x)dx = \frac{b-a}{2}\int_{-1}^1 f\left(\frac{b-a}{2}t + \frac{b+a}{2}\right)dt \approx \frac{b-a}{6}\left[\, f(a) + 4f\left(\frac{b+a}{2}\right) + f(b)\,\right].$$

Simpson's rule over the interval $[a, b]$ becomes therefore

$$S(a,b) = \frac{b-a}{6}\,[\, f(a) + 4f(c) + f(b)\,] \qquad \text{with } c = \frac{b+a}{2}.$$

3.1.3 Composite Simpson's rule

Next we will have to discuss the corresponding composite rule. Here we have to choose a partition of the interval $[a, b]$ into sub-intervals, evaluate the quadrature on each of the sub-intervals, and finally add all results together. The final result will heavily rely on the choice of the sub-intervals, and we will discuss later an automated strategy for their construction. For now, we content ourselves with the simplest construction, where we take sub-intervals of equal lengths.

Divide $[a, b]$ into $2m$ equal intervals of length $h = (b-a)/(2m)$. Let $x_j = a + jh$, $i = 0, \cdots, 2m$, and apply Simpson's rule on each subinterval $[x_{2j}, x_{2j+2}]$. The result is:

$$\begin{aligned}
\int_a^b f(x)dx = \sum_{j=0}^{m-1}\int_{x_{2j}}^{x_{2j+2}} f(x)dx &\approx \sum_{j=0}^{m-1} S(x_{2j}, x_{2j+2}) \\
&= \sum_{j=0}^{m-1}\frac{h}{3}\left[f(x_{2j}) + 4f(x_{2j+1}) + f(x_{2j+2})\right] \\
&= \frac{h}{3}\left[f(x_0) + 4\sum_{j=0}^{m-1} f(x_{2j+1}) + 2\sum_{j=1}^{m-1} f(x_{2j}) + f(x_{2m})\right] =: S_m(a,b).
\end{aligned}$$

We will use the the notation $S_m(a, b)$ for the composite Simpson's rule on m subintervals.

3.1.4 Implementation and testing

It is now time to implement the composite Simpson's method, and see how well it works. Start by calling the necessary modules etc:

[1]:
```
%matplotlib inline

from numpy import *
from matplotlib.pyplot import *
from math import factorial
newparams = {'figure.figsize': (8.0, 4.0), 'axes.grid': True,
             'lines.markersize': 8, 'lines.linewidth': 2,
             'font.size': 14}
rcParams.update(newparams)
```

[2]:
```
def simpson(f, a, b, m=10):
# Find an approximation to an integral by the composite Simpson's method:
# Input:
#   f:    integrand
#   a, b: integration interval
#   m:    number of subintervals
# Output: The approximation to the integral
    n = 2*m
    x_noder = linspace(a, b, n+1)       # equidistributed nodes from a to b
    h = (b-a)/n                         # stepsize
    S1 = f(x_noder[0]) + f(x_noder[n])  # S1 = f(x_0)+f(x_n)
    S2 = sum(f(x_noder[1:n:2]))         # S2 = f(x_1)+f(x_3)+...+f(x_m)
    S3 = sum(f(x_noder[2:n-1:2]))       # S3 = f(x_2)+f(x_4)+...+f(x_{m-1})
    S = h*(S1 + 4*S2 + 2*S3)/3
    return S
```

Test if the code is correct. We know that Simpson's rule has precision 3, thus all third degree polynomials can be integrated exactly. Choose one such polynomial, find the exact integral, and compare.

Numerical experiment 1: Apply the code on the integral, and compare with the exact result:

$$\int_{-1}^{2}(4x^3 + x^2 + 2x - 1)dx = 18.$$

[3]:
```
# Numerical experiment 1
def f(x):                     # Integrand
    return 4*x**3+x**2+2*x-1
a, b = -1, 2                  # Integration interval
I_exact = 18.0                # Exact value of the integral (for comparision)
S = simpson(f, a, b, m=1)     # Numerical solution, using m subintervals
err = I_exact-S               # Error
print('I = {:.8f},  S = {:.8f},  error = {:.3e}'.format(I_exact, S, err))
```

```
I = 18.00000000,  S = 18.00000000,  error = 0.000e+00
```

Numerical experiment 2: We will assume that the error decreases when the number of subintervals m increases. But how much?

Apply the composite method on the integral (again with a known solution):

$$\int_0^1 \cos\left(\frac{\pi x}{2}\right) dx = \frac{2}{\pi}.$$

Use the function `simpson` with $m = 1, 2, 4, 8, 16$ and see how the error changes with m. Comment on the result.

[4]:
```
# Numerical experiment 2
def f(x):
    return cos(0.5*pi*x)
a, b = 0, 1
I_exact = 2/pi
for m in [1,2,4,8,16]:
    S = simpson(f, a, b, m=m)    # Numerical solution, using m subintervals
    err = I_exact-S              # Error
    if m == 1:
        print('m = {:3d},  error = {:.3e}'.format(m, err))
    else:
        print('m = {:3d},  error = {:.3e},  reduction factor = {:.3e}'.format(m,␣
 ↪err, err/err_prev))
    err_prev=err
```

```
m =   1,  error = -1.451e-03
m =   2,  error = -8.568e-05,  reduction factor = 5.903e-02
m =   4,  error = -5.281e-06,  reduction factor = 6.164e-02
m =   8,  error = -3.289e-07,  reduction factor = 6.228e-02
m =  16,  error = -2.054e-08,  reduction factor = 6.245e-02
```

From the experiment we observe that the error is reduced by a factor approximately $0.0625 = 1/16$ whenever the number of subintervals increases with a factor 2. In the following, we will prove that this is in fact what can be expected.

3.2 Error analysis

First we will find an expression for the error $E(a,b) = I(a,b) - S(a,b)$ over one interval (a,b). This will then be used to find an expression for the composite formula.

Let $c = (a+b)/2$ be the midpoint of the interval, and $h = (b-a)/2$ be the distance between c and the endpoints a and b. Do a Taylor series expansion of the integrand f around the midpoint, and integrate each term in the series:

$$\int_a^b f(x)dx = \int_{-h}^{h} f(c+s)ds$$
$$= \int_{-h}^{h} \left(f(c) + sf'(c) + \frac{1}{2}s^2 f''(c) + \frac{1}{6}s^3 f'''(c) + \frac{1}{24}s^4 f^{(4)}(c) + \cdots \right) ds$$
$$= 2hf(c) + \frac{h^3}{3}f''(c) + \frac{h^5}{60}f^{(4)}(c) + \cdots .$$

Similarly, we compute a Taylor series expansion of the quadrature $S(a,b)$ around c:

$$\begin{aligned}
S(a,b) &= \frac{h}{3}\left(f(c-h) + 4f(c) + f(c+h)\right) \\
&= \frac{h}{3}\Big(f(c) - hf'(c) + \frac{1}{2}h^2 f''(c) - \frac{1}{6}h^3 f'''(c) + \frac{1}{24}h^4 f^{(4)}(c) + \cdots \\
&\qquad + 4f(c) \\
&\qquad + f(c) + hf'(c) + \frac{1}{2}h^2 f''(c) + \frac{1}{6}h^3 f'''(c) + \frac{1}{24}h^5 f^{(4)}(c) + \cdots \Big) \\
&= 2hf(c) + \frac{h^3}{3}f''(c) + \frac{h^5}{36}f^{(4)}(c) + \cdots
\end{aligned}$$

The series expansion of the error becomes therefore:

$$E(a,b) = \int_a^b f(x)dx - S(a,b) = -\frac{h^5}{90}f^{(4)}(c) + \cdots = -\frac{(b-a)^5}{2^5 \cdot 90}f^{(4)}(c) + \cdots ,$$

using $h = (b-a)/2$.

NB! By choosing to do the Taylor-expansions around the midpoint, every second term disappear thanks to symmetry. Choosing another point $\hat{c}$ in the interval will give the same dominant error term (with c replaced by $\hat{c}$), but the calculations will be much more cumbersome.

Usually, we will assume h to be small, such that the first nonzero term in the series dominates the error, and the rest of the series can be ignored. It is however possible, but not trivial, to prove the following result:

Theorem: Error in Simpson's method. Let $f(x) \in C^4[a,b]$. There exist a $\xi \in (a,b)$ such that

$$E(a,b) = \int_a^b f(x)dx - \frac{b-a}{6}\left[f(a) + 4f\left(\frac{b+a}{2}\right) + f(b) \right] = -\frac{(b-a)^5}{2880}f^{(4)}(\xi).$$

NB! : Since $p^{(4)}(x) = 0$ for all $p \in \mathbb{P}_3$, this is in agreement with the observation that degree of precision of Simpson's rule is equal to 3.

We now can use this theorem to find an expression for the error in the composite Simpson's formula $S_m(a,b)$:

$$\begin{aligned}\int_a^b f(x)dx - S_m(a,b) &= \sum_{j=0}^{m-1}\left(\int_{x_{2j}}^{x_{2j+2}} f(x)dx - \frac{h}{3}\left[f(x_{2j}) + 4f(x_{2j+1}) + f(x_{2j+2})\right]\right) \\ &= \sum_{j=0}^{m-1} -\frac{(2h)^5}{2880} f^{(4)}(\xi_j)\end{aligned}$$

where $\xi_j \in (x_{2j}, x_{2j+2})$. We can then use the generalized mean value theorem, see *Preliminaries*, section 5, result 2. According to this, there is some $\xi \in (a,b)$ such that

$$\sum_{j=0}^{m-1} f^{(4)}(\xi_j) = m f^{(4)}(\xi).$$

By inserting the equality $2mh = b - a$, we see that the following theorem has been proved:

Theorem: Error in composite Simpson's method. Let $f(x) \in C^4[a,b]$. There exists $\xi \in (a,b)$ such that

$$\int_a^b f(x)dx - S_m(a,b) = -\frac{(b-a)h^4}{180} f^{(4)}(\xi).$$

Example 4: Find the upper bound for the error when the composite Simpson's rule is applied to the integral $\int_0^1 \cos(\pi x/2)dx$.

In this case $f^{(4)}(x) = (\pi^4/16)\cos(\pi x/2)$, so that $|f^{4)}(x)| \le (\pi/2)^4$. The error bound becomes

$$|I(a,b) - S_m(a,b)| \le \frac{1}{180}\left(\frac{1}{2m}\right)^4 \left(\frac{\pi}{2}\right)^4 = \frac{\pi^4}{46080}\frac{1}{m^4}.$$

If m is increased by a factor 2, the error will be reduced by a factor of 1/16, as indicated by Numerical experiment 2.

Numerical exercise: Include the error bound in the output of Numerical experiment 2, and confirm that it really holds.

Remark: The result above shows that the composite Simpson rule with equidistant nodes converges with convergence order 4 (in terms of the node distance h) to the actual integral. That is, the convergence order is equal to the degree of precision $+1$. This relation between degree of precision and convergence order can be shown to hold in general: If a composite quadrature rule with equidistant nodes is based on a simple quadrature rule, as constructed above, with degree of precision p, then the composite rule will have convergence order $p+1$ for all functions $f \in C^{p+1}[a,b]$.

3.3 Error estimate

From a practical point of view, the error expression derived above has some limitations, the main difficulty being that it depends on the unknown value $f^{(4)}(\xi)$. In practice, we can at best use an error estimate of the form

$$|I(a,b) - S_m(a,b)| \leq \frac{(b-a)h^4}{180}\|f^{(4)}\|_\infty.$$

This bound, however, often vastly overestimates the actual error. In addition, we do not always know (or want to find) $\|f^{(4)}\|_\infty$. So the question arises: How can we find an estimate of the error, without any extra analytical calculations?

This is the idea: Let the interval (a,b) chosen small, such that $f^{(4)}(x)$ can be assumed to be almost constant over the interval. Let $H = b-a$ be the length of the interval. Let $S_1(a,b)$ and $S_2(a,b)$ be the results from Simpson's formula over one and two subintervals respectively. Further, let $C = -f^{(4)}(x)/2880$ for some $x \in [a,b]$ — which x does not matter, as $f^{(4)}$ is assumed almost constant anyway. The errors of the two approximations are then given by

$$\begin{aligned} I(a,b) - S_1(a,b) &\approx CH^5, \\ I(a,b) - S_2(a,b) &\approx 2C\left(\frac{H}{2}\right)^5. \end{aligned}$$

Subtract the two expressions to eliminate $I(a,b)$:

$$S_2(a,b) - S_1(a,b) \approx \frac{15}{16}CH^5 \qquad \Rightarrow \qquad CH^5 \approx \frac{16}{15}(S_2(a,b) - S_1(a,b)).$$

Insert this in the expression for the error:

$$\begin{aligned} E_1(a,b) = I(a,b) - S_1(a,b) &\approx \frac{16}{15}(\,S_2(a,b) - S_1(a,b)\,) = \mathcal{E}_1(a,b), \\ E_2(a,b) = I(a,b) - S_2(a,b) &\approx \frac{1}{15}(\,S_2(a,b) - S_1(a,b)\,) = \mathcal{E}_2(a,b). \end{aligned}$$

This gives us a computable estimate for the error, both in S_1 and S_2. As the error in $S_2(a,b)$ is about $1/16$ of the error in $S_1(a,b)$, and we anyway need to compute both, we will use $S_2(a,b)$ as our approximation. An even better approximation to the integral is given for free by just adding the error estimate:

$$I(a,b) \approx S_2(a,b) + \mathcal{E}_2(a,b) = \frac{16}{15}S_2(a,b) - \frac{1}{15}S_1(a,b).$$

Example 5: Find an approximation to the integral $\int_0^1 \cos(x)dx = \sin(1)$ by Simpson's rule over one and two subintervals. Find the error estimates $\mathcal{E}_m$, $m = 1,2$ and compare with the exact error.

Solution:

$$\begin{aligned} S_1(0,1) &= \frac{1}{6}\big[\cos(0.0) + 4\cos(0.5) + \cos(1.0)\big] = 0.8417720923, \\ S_2(0,1) &= \frac{1}{12}\big[\cos(0.0) + 4\cos(0.25) + 2\cos(0.5) + 4\cos(0.75) + \cos(1.0)\big] = 0.8414893826. \end{aligned}$$

The exact error and the error estimate become:

$$E_1(0,1) = \sin(1) - S_1(0,1) = -3.011 \cdot 10^{-4}, \quad \mathcal{E}_1(0,1) = \frac{16}{15}(S_2 - S_1) = -3.016 \cdot 10^{-4},$$
$$E_2(0,1) = \sin(1) - S_2(0,1) = -1.840 \cdot 10^{-5}, \quad \mathcal{E}_2(0,1) = \frac{1}{16}(S_2 - S_1) = -1.885 \cdot 10^{-5}.$$

In this case, ther is a very good correspondence between the error estimate and the exact error. An even better approximation is obtained by adding the error estimate to S_2:

$$Q = S_2(0,1) + \mathcal{E}_2(0,1) = 0.8414705353607151$$

with an error $\sin(1) - Q = 4.4945 \cdot 10^{-7}$. This gives a lot of additional accuracy without any extra work.

3.3.1 Implementation of Simpson's method with an error estimate

The function `simpson_basic` returns

$$S_2(a,b) \approx \int_a^b f(x)dx$$

including an error estimate.

[5]:
```
def simpson_basic(f, a, b):
    # Simpson's method with error estimate
    # Input:
    #   f:     integrand
    #   a, b: integration interval
    # Output:
    #   S_2(a,b) and the error estimate.

    # The nodes
    c = 0.5*(a+b)
    d = 0.5*(a+c)
    e = 0.5*(c+b)

    # Calculate S1=S_1(a,b), S2=S_2(a,b)
    H = b-a
    S1 = H*(f(a)+4*f(c)+f(b))/6
    S2 = 0.5*H*(f(a)+4*f(d)+2*f(c)+4*f(e)+f(b))/6

    error_estimate = (S2-S1)/15    # Error estimate for S2
    return S2, error_estimate
```

Test: As a first check of the implementation, use the example above, and make sure that the results are the same:

```
[6]: # Test of simpson_basic

     def f(x):                  # Integrand
         return cos(x)

     a, b = 0, 1                # Integration interval

     I_exact = sin(1)           # Exact solution for comparision

     # Simpson's method over two intervals, with error estimate
     S, error_estimate = simpson_basic(f, a, b)

     # Print the result and the exact solution
     print('Numerical solution = {:.8f}, exact solution = {:.8f}'.format(S, I_exact))

     # Compare the error and the error estimate
     print('Error in S2 = {:.3e},  error estimate for S2 = {:.3e}'.format(I_exact-S,␣
      ↪error_estimate))
```

```
Numerical solution = 0.84148938, exact solution = 0.84147098
Error in S2 = -1.840e-05,  error estimate for S2 = -1.885e-05
```

Next, let us see how reliable the quadrature and the error estimates are for another example, which you have to do yourself:

Numerical experiment 3: Given the integral (with solution)

$$I(a,b) = \int_a^b \frac{1}{1+16x^2}dx = \left.\frac{\arctan(4x)}{4}\right|_a^b$$

1. Use `simpson_basic` to find an approximation to the integral over the interval $[0,8]$. Print out $S_2(0,8)$, the error estimate $\mathcal{E}_2(0,8)$ and the real error $E_2(0,8)$. How reliable are the error estimates?
2. Repeat the experiment over the intervals $[0,1]$ and $[4,8]$. Notice the difference between exact error of the two intervals.
3. Repeat the experiment over the interval $[0,0.1]$.

This is what you should observe from the experiment:

1. Interval $[0,8]$: The error is large, and the error estimate is significantly smaller than the real error (the error is *under-estimated*).
2. Interval $[0,1]$: As for the interval $[0,8]$.
3. Interval $[4,8]$: Small error, and a reasonable error estimate.
4. Interval $[0,0.1]$: Small error, reasonable error estimate.

Why is it so, and how can we deal with it? Obviously, we need small subintervals near $x = 0$, while large subintervals are acceptable in the last half of the interval.

Explanation: The error in Simpson's method is given by

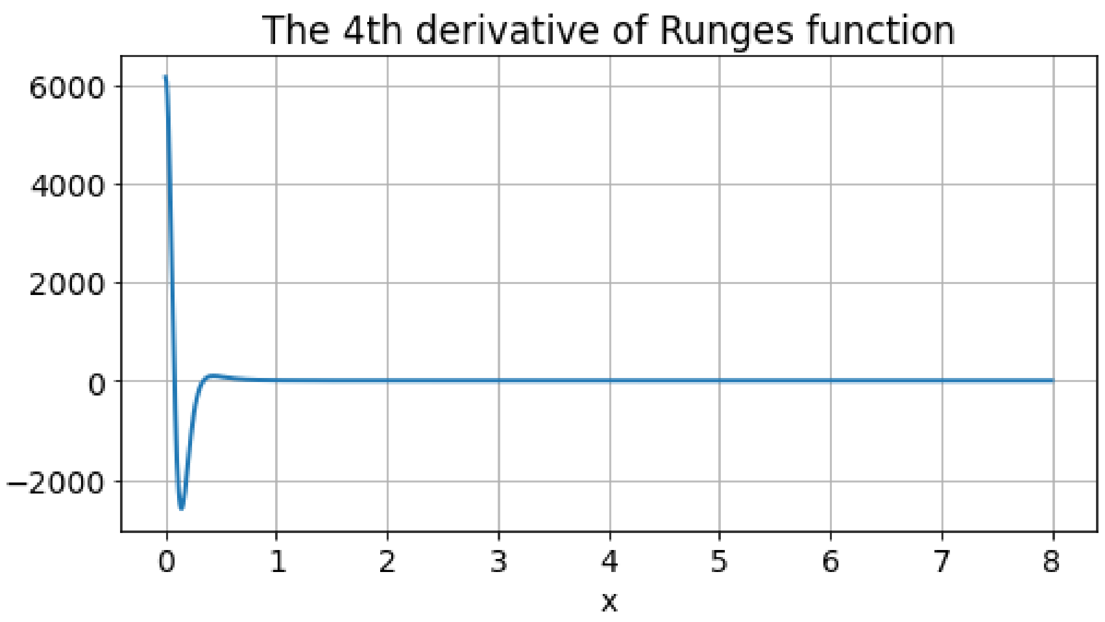
$$E(a,b) = -\frac{(b-a)^5}{2880} f^{(4)}(\xi).$$

So let us take a look at $f^{(4)}(x)$:

$$f(x) = \frac{1}{1+16x^2} \qquad \Rightarrow \qquad f^{(4)}(x) = 6144\frac{1280x^4 - 160x^2 + 1}{(1-16x^2)^5}$$

```
[7]: # Plot the 4th derivate of Runge's function:
     def df4(x):
         return 6144*(1280*x**4-160*x**2+1)/((1+16*x**2)**5)
     x = linspace(0, 8, 1001)
     plot(x, df4(x))
     title('The 4th derivative of Runges function');
     xlabel('x')
```

```
[7]: Text(0.5, 0, 'x')
```



It is no surprise that the error is large and the error estimates fail (we have assumed $f^{(4)}$ almost constant for the estimates) over the interval $[0,1]$. The part of the interval where $f^{(4)}(x)$ is large has to be partitioned in significantly smaller subintervals to get an acceptable result. But how, as $f^{(4)}$ is in general not known? This is the topic of the next section.

3.4 Adaptive integration

Given a basic function, for example `simpson_basic`, returning an approximation $Q(a,b)$ to the integral, as well as an error estimate $\mathcal{E}(a,b)$. Based on this, we want to find a partitioning of the interval:

$$a = X_0 < X_1 \cdots < X_m = b$$

such that

$$|\mathcal{E}(X_j, X_{j+1})| \approx \frac{X_{k+1} - X_k}{b-a} \cdot Tol$$

where Tol is a tolerance given by the user. In this case

$$\text{Accumulated error over } (a,b) \approx \sum_{j=0}^{m-1} \mathcal{E}(X_k, X_{k+1}) \leq \text{Tol}.$$

Such a partitioning can be found by an recursive algorithm:

Algorithm: Adaptive quadrature. Given f, a, b and a user defined tolerance Tol. * Calculate $Q(a,b)$ and $\mathcal{E}(a,b)$.

- **if** $|\mathcal{E}(a,b)| \leq \text{Tol}$:
  - Accept the result, return $Q(a,b) + \mathcal{E}(a,b)$ as an approximation to $I(a,b)$.
- **else**:
  - Let $c = (a+b)/2$, and repeat the process on each of the subintervals $[a,c]$ and $[c,b]$, with tolerance $\text{Tol}/2$.
- Sum up the accepted results from each subinterval.

3.4.1 Implementation

The adaptive algorithm is implemented below with `simpson_basic` as the basic quadrature routine. The function `simpson_adaptive` is a recursive function, that is a function that calls itself. To avoid it to do so infinitely many times, an extra variable `level` is introduced, this will increase by one for each time the function calls itself. If `level` is over some maximum value, the result is returned, and a warning printed.

[8]:
```python
def simpson_basic(f, a, b):
    # Simpson's method with error estimate
    # Input:
    #   f:    integrand
    #   a, b: integration interval
    # Output:
    #   S_2(a,b) and the error estimate.

    # The nodes
    c = 0.5*(a+b)
    d = 0.5*(a+c)
    e = 0.5*(c+b)

    # Calculate S1=S_1(a,b), S2=S_2(a,b)
    H = b-a
    S1 = H*(f(a)+4*f(c)+f(b))/6
    S2 = 0.5*H*(f(a)+4*f(d)+2*f(c)+4*f(e)+f(b))/6

    error_estimate = (S2-S1)/15    # Error estimate for S2
    return S2, error_estimate
```

[9]:
```
def simpson_adaptive(f, a, b, tol = 1.e-6, level = 0, maks_level=15):
    # Simpson's adaptive method
    # Input:
    #   f:    integrand
    #   a, b: integration interval
    #   tol:  tolerance
    #   level, maks_level: For the recursion. Just ignore them.
    # Output:
    #   The approximation to the integral


    Q, error_estimate = simpson_basic(f, a, b)    # The quadrature and the error␣
↪estimate

    # -------------------------------------------------
    # Write the output, and plot the nodes.
    # This part is only for illustration.
    if level == 0:
        print(' l   a           b         feil_est   tol')
        print('==============================================')
    print('{:2d}   {:.6f}   {:.6f}   {:.2e}   {:.2e}'.format(
            level, a, b, abs(error_estimate), tol))

    x = linspace(a, b, 101)
    plot(x, f(x), [a, b], [f(a), f(b)], '.r')
    title('The integrand and the subintervals')
    # -------------------------------------------------

    if level >= maks_level:
        print('Warning: Maximum number of levels used.')
        return Q

    if abs(error_estimate) < tol:         # Accept the result, and return
        result = Q + error_estimate
    else:
        # Divide the interval in two, and apply the algorithm to each interval.
        c = 0.5*(b+a)
        result_left  = simpson_adaptive(f, a, c, tol = 0.5*tol, level = level+1)
        result_right = simpson_adaptive(f, c, b, tol = 0.5*tol, level = level+1)
        result = result_right + result_left
    return result
```

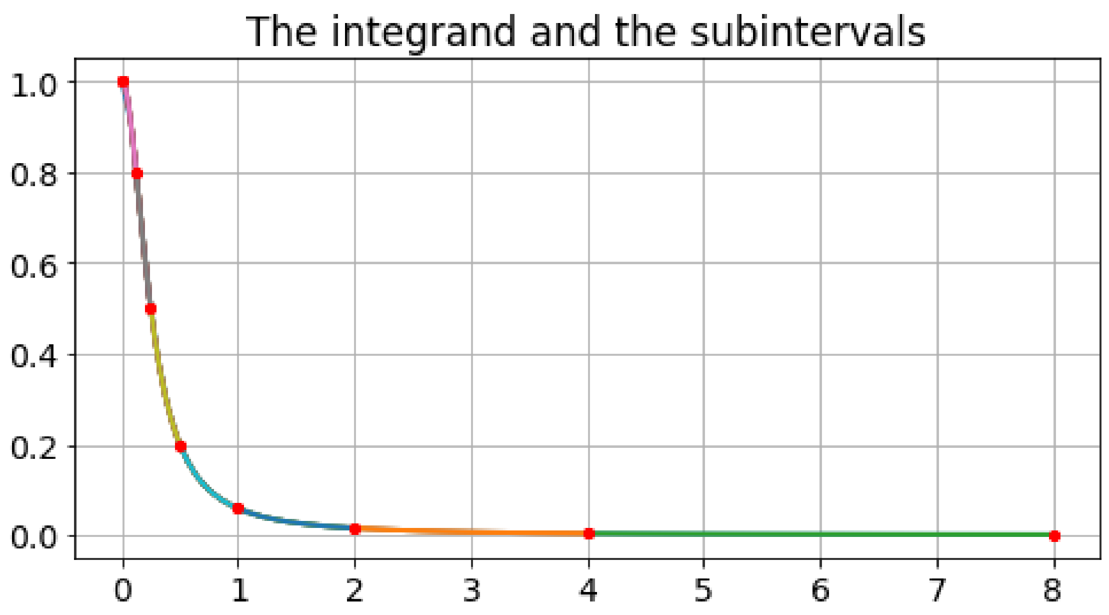
Numerical experiment 4: Use adaptive Simpson to find an approximation to the integral $\int_0^5 1/(1+16x^2)dx$ using the tolerances Tol=$10^{-3}, 10^{-5}, 10^{-7}$. Compare the numerical result with the exact one.

[10]:
```
# Test: The adaptive Simpson's method
def f(x):                                       # Integrand
    return 1/(1+(4*x)**2)
a, b = 0, 8                                     # Integration interval
I_exact = 0.25*(arctan(4*b)-arctan(4*a))  # Exact integral
tol = 1.e-3                                     # Tolerance
# Apply the algorithm
result = simpson_adaptive(f, a, b, tol=tol)
# Print the result and the exact solution
print('\nNumerical solution = {:8f}, exact solution = {:8f}'
      .format(result, I_exact))
# Compare the measured error and the tolerance
err = I_exact - result
print('\nTolerance = {:.1e}, error = {:.3e}'.format(tol, abs(err)))
```

```
 l   a           b           feil_est   tol
==============================================
 0   0.000000    8.000000    4.25e-02   1.00e-03
 1   0.000000    4.000000    1.85e-02   5.00e-04
 2   0.000000    2.000000    5.11e-03   2.50e-04
 3   0.000000    1.000000    7.84e-04   1.25e-04
 4   0.000000    0.500000    6.41e-04   6.25e-05
 5   0.000000    0.250000    3.43e-05   3.13e-05
 6   0.000000    0.125000    1.21e-06   1.56e-05
 6   0.125000    0.250000    1.31e-06   1.56e-05
 5   0.250000    0.500000    7.82e-07   3.13e-05
 4   0.500000    1.000000    1.45e-05   6.25e-05
 3   1.000000    2.000000    1.40e-05   1.25e-04
 2   2.000000    4.000000    8.29e-06   2.50e-04
 1   4.000000    8.000000    4.33e-06   5.00e-04

Numerical solution = 0.384903, exact solution = 0.384889

Tolerance = 1.0e-03, error = 1.343e-05
```



3.5 Other quadrature formulas

Simpson's rule is only one example of quadrature rules derived from polynomial interpolations. There are many others, and the whole process of deriving the methods, do error analysis, develop error estimates and adaptive algorithms can be repeated.

Let us just conclude with a few other popular classes of methods:

3.5.1 Newton-Cotes formulas

These are based on equidistributed nodes. The simplest choices here — the *closed* Newton-Cotes methods — use the nodes $x_i = a + ih$ with $h = (b-a)/n$. Examples of these are the Trapezoidal rule and Simpson's rule. The main appeal of these rules is the simple definition of the nodes.

If n is odd, the Newton-Cotes method with $n+1$ nodes has degree of precision n; if n is even, it has degree of precision $n+1$. The corresponding convergence order for the composite rule is, as for all such rules, one larger than the degree of precision, provided that the function f is sufficiently smooth.

However, for $n \geq 8$ negative weights begin to appear in the definitions. This has the undesired effect that the numerical integral of a positive function can be negative. In addition, this can lead to cancellation errors in the numerical evaluation, which may result in a lower practical accuracy. Since the rules with $n = 6$ and $n = 7$ yield the same convergence order, this mean that it is mostly the rules with $n \leq 6$ that are used in practice.

The *open* Newton-Cotes methods, in contrast, use the nodes $x_i = a + (i + 1/2)h$ with $h = (b - a)/(n+1)$. The simplest example here is the midpoint rule. Here negative weights appear already for $n \geq 2$. Thus the midpoint rule is the only such rule that is commonly used in applications.

3.5.2 Gauß quadratures

For the standard interval $[-1, 1]$ choose the nodes as the zeros of the Legendre polynomial L_n of degree n:

$$L_n(t) = \frac{d^n}{dt^n}(t^2 - 1)^n.$$

The resulting quadrature rules have a degree of precision $d = 2n - 1$, and the corresponding composite rules have a convergence order of $2n$. It is possible to show that this is the highest achievable degree of precision with n nodes.

For $n = 1$, one obtains the midpoint rule. For $n = 2$ one obtains the method discussed near the beginning of these notes in example 2, which has the nodes $x_0 = 1/2 + \sqrt{3}/6$ and $x_1 = 1/2 - \sqrt{3}/6$ and the corresponding weights $w_0 = w_1 = 1/2$.