Review: differentiation

- We discussed 2 basic methods:
  - Finite-differencing: based on Taylor-series expansions. High-order methods (errors $O(h^2)$) and/or double precision should be used.
  - Interpolation-based methods: differentiate the interpolant to approximate the derivative. Best done on cubic splines.
- Richardson extrapolation: simple method to derive more precise approximations by eliminating the leading-order term in the errors (under certain assumptions).

Numerical integration

For most integrals there is no exact analytical result.

Example: the “error function” erf(x):

$$\operatorname{erf}(x) \equiv \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}\,dt$$

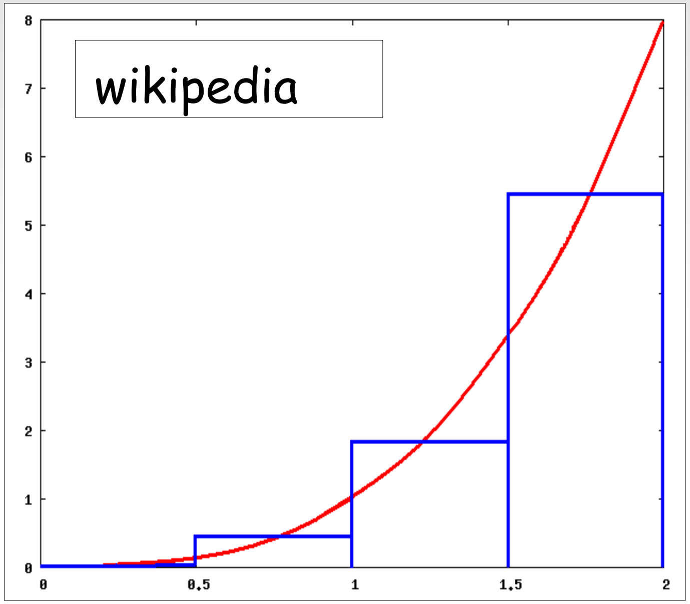
This integral is an important special function, but it cannot be expressed in terms of elementary functions.

Use: the probability for finding an event within n standard deviations from the mean is erf(n/√2) for a normally distributed random variable.

How can we evaluate integrals numerically?

Calculating simple integrals

To start, remember the Riemann sums which approximate the integral by step functions:



Idea: write integral as a weighted sum

$$\int_a^b f(x)dx \approx \sum_{k=0}^{N} c_k f(x_k)$$

choosing weights c_k in some (ideally) optimal way, e.g. from integrating the interpolating polynomial (we certainly can integrate polynomials exactly) → Newton-Cotes formulae.

We could further improve result by choosing nodes x_k better → Gaussian quadrature.

Degree of precision

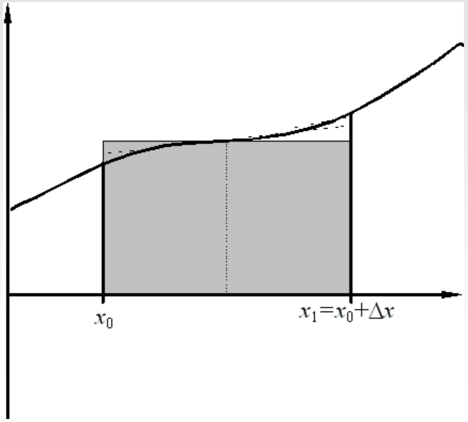
A numerical integration scheme has a degree of precision m if it gives exact results for all polynomials of degree m or less, but not for degree m+1.

The degree of precision can sometimes be higher than is naively expected.

We expect that for a method of degree m the error scales like h^{m+1}, for subdivision size $h=x-x_0$ (Taylor's exp.):

$$f(x) - p(x) = \frac{(x - x_0)^{N+1}}{(N+1)!} f^{N+1}(\xi)$$

Midpoint rule



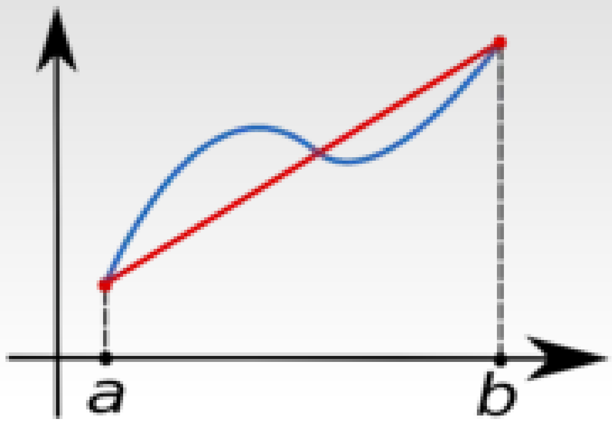
$$\int_a^b f(x)dx \approx (b-a) f\left(\frac{a+b}{2}\right)$$

The Taylor expansion (in h=b-a) yields the error :

$$\left[\int_a^b f(x)dx\right] - \left[(b-a) f\left(\frac{a+b}{2}\right)\right] = \frac{(b-a)h^2}{24} f''(\xi_M)$$

degree 1

Trapezoidal rule



Let $I=c_0 f(x_0)+c_1 f(x_1)$ (here $x_0=a$, $x_1=b$)
assume this is exact for f=1 and f=x:
$x_1-x_0=c_0+c_1 \longrightarrow c_1=c_2=(x_1-x_0)/2$
$(x^2_1-x^2_0)/2=c_0x_0+c_1x_1$ (see book for slightly different derivation)

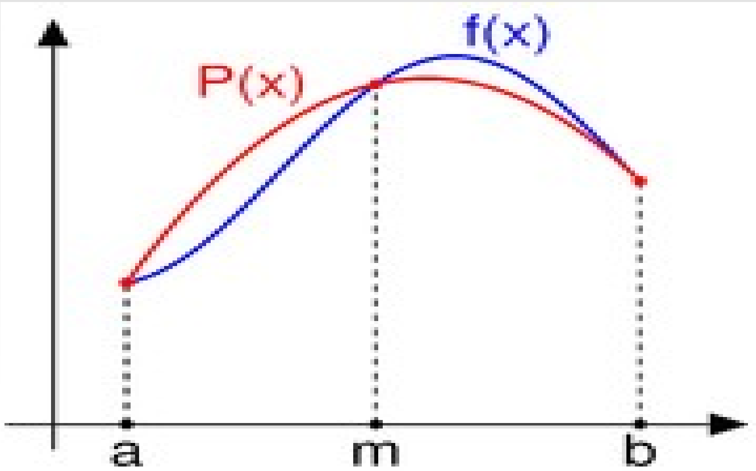
$$\boxed{\int_a^b f(x)dx \approx \frac{(b-a)}{2}\left(f(a)+f(b)\right)}$$

error:

degree 1

$$\left[\int_a^b f(x)dx\right]-\left[\frac{(b-a)}{2}\left(f(a)+f(b)\right)\right]=\frac{(b-a)h^2}{12}f''(\xi_T)$$

Simpsons rule



Derivation – same as trapezoidal rule, but now requiring formula to be exact for 1, x and x^2 (somewhat tedious derivation)

$$\int_a^b f(x)dx \approx \frac{(b-a)}{6}(f(a)+4f(m)+f(b)) \qquad m=(a+b)/2$$

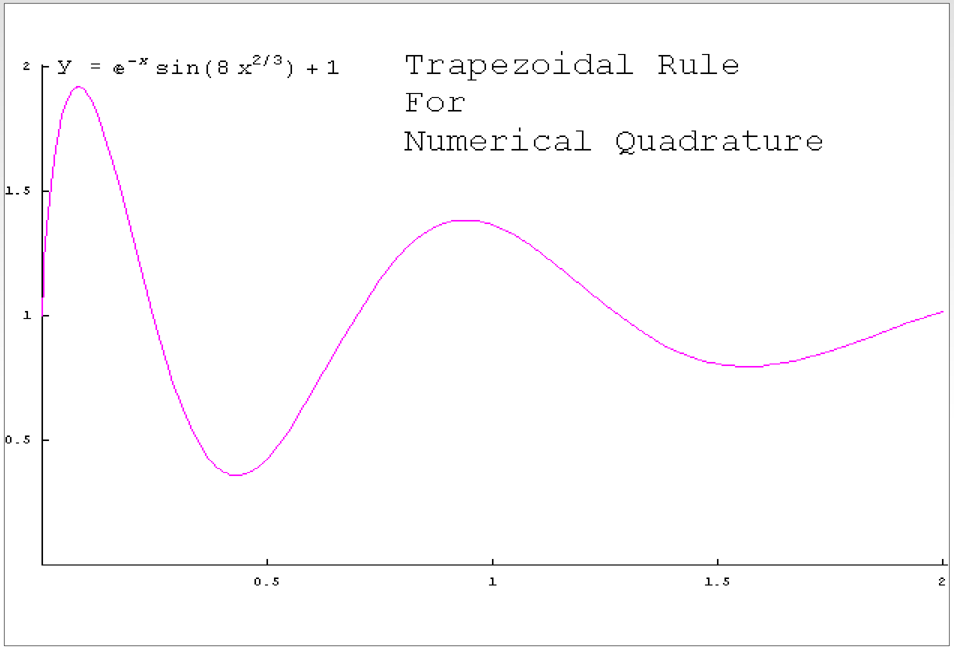
error:

degree 3

$$\left[\int_a^b f(x)dx\right]-\left[\frac{(b-a)}{6}(f(a)+4f(m)+f(b))\right]=-\frac{(b-a)h^4}{180}f^{(4)}(\xi_S)$$

Composite formulas

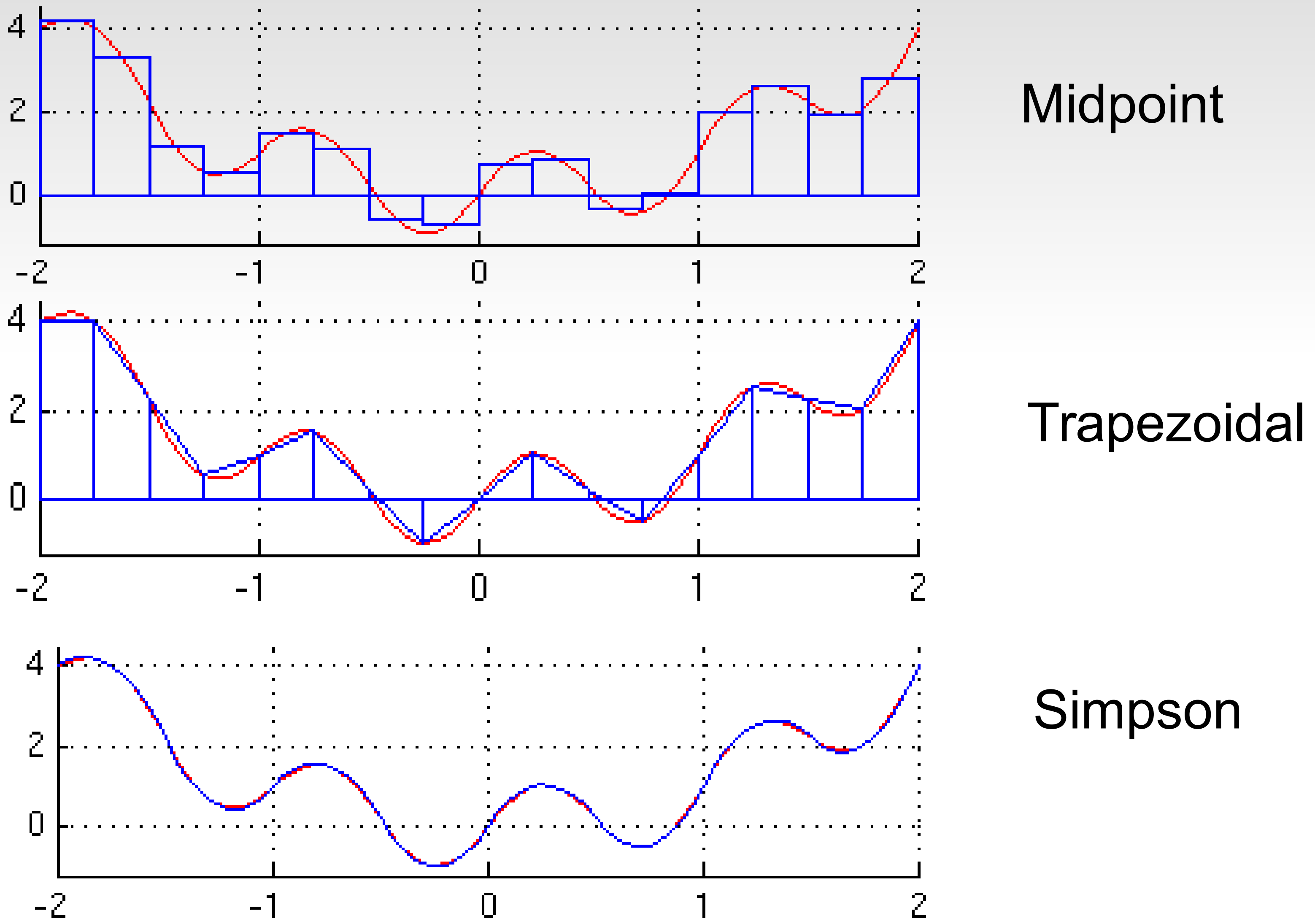
Instead of just applying each rule over the whole interval, we divide the interval into subintervals and sum over them → scaling with h determines how quickly the error shrinks e.g. for trapezoid rule:



$$\int_a^b f(x)dx \approx \frac{h}{2}[f(a)+f(m)]+\frac{h}{2}[f(m)+f(b)]$$

$$=\frac{h}{2}[f(a)+2f(m)+f(b)]$$

Composite rules: illustrations



Composite trapezoid rule

for N subdivisions:

h = (b-a)/N

and for fixed [a, b] the error scales like $h^2 = 1/N^2$

```
## module trapezoid
''' Inew = trapezoid(f,a,b,Iold,k).
    Recursive trapezoidal rule:
    Iold = Integral of f(x) from x = a to b computed by
    trapezoidal rule with 2^(k-1) panels.
    Inew = Same integral computed with 2^k panels.
'''
def trapezoid(f,a,b,Iold,k):
    if k == 1:Inew = (f(a) + f(b))*(b - a)/2.0
    else:
        n = 2**(k -2 )       # Number of new points
        h = (b - a)/n        # Spacing of new points
        x = a + h/2.0
        sum = 0.0
        for i in range(n):
            sum = sum + f(x)
            x = x + h
        Inew = (Iold + h*sum)/2.0
    return Inew
```

$$\int_a^b f(x) \approx T_N = \frac{h}{2}\left[f(a) + 2\sum_{k=1}^{N-1} f(a+kh) + f(b) \right]$$

Composite Simpson rule

The trapezoid rule uses x_k=a+k*h, k = 1,2,...,N-1 (& a,b) the mid-point rule we use the points in between the x_k:

$y_k = (x_{k-1}+x_k)/2 = a+(k-1/2)*h$, k = 1,2,...,N

the Simpson rule uses all these points, with weight 4 for the y_k and weight 2 for the (doubled) points x_k.

setting h = (b-a)/(2N) and adjusting the points accordingly: x_k=a+2*k*h, y_k = a+(2*k-1)*h, we find:

$$S_{2N} = \frac{h}{3}\left[f(a) + 4\sum_{k=1}^{N} f(y_k) + 2\sum_{k=1}^{N-1} f(x_k) + f(b)\right] = \frac{T_N + 2M_N}{3}$$

There are better, more modern methods, however ...

Romberg integration

Trapezoid method with N sub-divisions: error on $T_N \sim 1/N^2$

So error on T_N is about 4x error T_{2N} (Richardson extrapolation)

$$4(I - T_{2N}) \approx (I - T_N) \longrightarrow I \approx (4T_{2N} - T_N)/3$$

The latter should be a more accurate estimate (recall our discussion of Richardson's extrapolation – this eliminates the leading error term). What formula is this?

$$T_1 = \frac{(b-a)}{2}[f(a) + f(b)]\,;\ T_2 = \frac{(b-a)}{4}[f(a) + 2f(m) + f(b)]$$

$$\frac{4T_2 - T_1}{3} = \frac{(b-a)}{6}[f(a) + 4f(m) + f(b)] = S_2$$

This is Simpsons rule! $\longrightarrow$ a second order method

Romberg integration

Similarly, for Simpsons rule with N subdivisions: error~$1/N^4$, thus we can apply Richardson's extrapolation again, to eliminate the h^4 order of the error:

$$16(I - S_{2N}) \approx (I - S_N) \longrightarrow I \approx (16S_{2N} - S_N)/15$$

surprise: it's the 5-node rule, error ~ $1/N^6$

This can then be repeated! → Romberg integration

Notation: $R_{n,1} = T_{2^{N-1}},\ R_{n,2} = S_{2^{N-1}}$

$$4^k(I - R_{n,k}) \approx (I - R_{n-1,k}) \Rightarrow R_{n,k+1} = \frac{(4^k R_{n,k} - R_{n-1,k})}{4^k - 1}$$

build higher $R_{n,k}$ recursively from $R_{n-1,k}$ and $R_{n,k-1}$, starting always from $R_{n,1}$ → lower triangular matrix (see book).

$$\begin{bmatrix} R_{1,1} & & & \\ R_{2,1} & R_{2,2} & & \\ R_{3,1} & R_{3,2} & R_{3,3} & \\ R_{4,1} & R_{4,2} & R_{4,3} & R_{4,4} \end{bmatrix}$$

Romberg integration: implementation

This Romberg integration method leads to a very efficient integration for smooth integrands.

```
## module romberg
''' I,nPanels = romberg(f,a,b,tol=1.0e-6).
    Romberg intergration of f(x) from x = a to b.
    Returns the integral and the number of panels used.
'''
from numpy import zeros
from trapezoid import *
def romberg(f,a,b,tol=1.0e-6):

    def richardson(r,k):
        for j in range(k-1,0,-1):
            const = 4.0**(k-j)
            r[j] = (const*r[j+1] - r[j])/(const - 1.0)
        return r

    r = zeros(21)
    r[1] = trapezoid(f,a,b,0.0,1)
    r_old = r[1]
    for k in range(2,21):
        r[k] = trapezoid(f,a,b,r[k-1],k)
        r = richardson(r,k)
        if abs(r[1]-r_old) < tol*max(abs(r[1]),1.0):
            return r[1],2**(k-1)
        r_old = r[1]
    print "Romberg quadrature did not converge"
```

Romberg integration: example

Example: evaluate the integral:

$$\int_0^{\sqrt{\pi}} 2x^2 \cos x^2$$

This method is far more efficient than any of the previously discussed ones.

```
#!/usr/bin/python
## example6_7
from math import cos,sqrt,pi
from romberg import *

def f(x): return 2.0*(x**2)*cos(x**2)

I,n = romberg(f,trapezoid,0,sqrt(pi))
print ''Integral ='',I
print ''nPanels ='',n
raw_input(''\nPress return to exit'')
```

The results of running the program are:

```
Integral = -0.894831469504
nPanels = 64
```

Python in-build methods

Integration methods are contained in scipy.integrate, e.g.:

scipy.integrate.quad(fcn,a,b): uses various methods, with default absolute and relative tolerances of 1.49e-8 (can be chosen as optional arguments, see help). One or both ends of the interval could be infinite.

scipy.integrate.newton_cotes(rn): returns weights and error coefficient for Newton-Cotes integration.

scipy.integrate.romberg(fcn,a,b): Romberg method with default absolute and relative tolerances of 1.49e-8 (can be chosen as optional arguments, see help).

dblquad, tplquad : evaluate a double or triple integral, resp.

As usual, check the help for details: 'help(command)'

Quad usage example: the error function

```
import scipy as sci
from scipy import integrate
import numpy as np
import math
def f(t):
    return 2.0/math.sqrt(math.pi)*math.exp(-t**2/2)
h,errh=sci.integrate.quad(f,0,inf)
print h,errh
→ 1.41421356237 1.43941363721e-08
```

Comments

Limits $\rightarrow \infty$: either bound tail or re-map range (quad handles infinities directly), e.g.

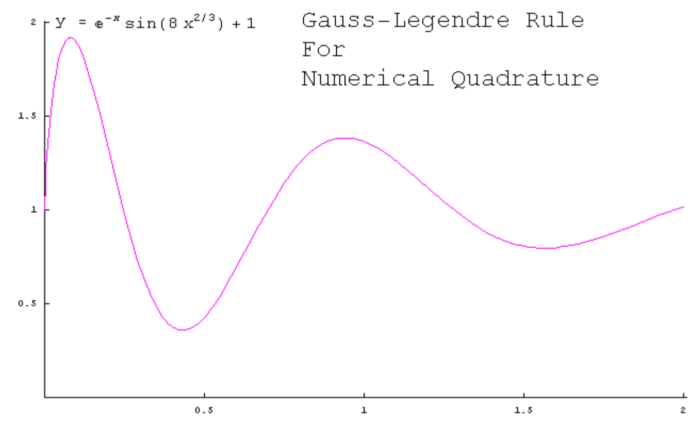
$$\int_a^b f(x)dx = \int_{1/b}^{1/a} \frac{1}{t^2} f\left(\frac{1}{t}\right) dt \qquad ab > 0$$

multi-dimension integrals: very hard! N^m for m dimensions

simplify analytically when possible, then do 1D integrations over dimensions

Monte-Carlo approaches

Gaussian quadrature $\rightarrow$ choosing knot locations, or integrate f(x)*W(x) exactly for f polynomial, W known.



More comments

adaptive subdivision: divide initial interval into smaller intervals in region where integral has large error; useful e.g. for step-functions, strongly peaked integrands, etc.

integration vs differential equations: → can compute an integral also via a differential equation:dy/dx=f(x), f(a)=0 → I=y(b)

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

→ may be better for sharply peak integrands, as adaptive step-size control is easier for differential equations (our next topic)

beware also of spurious convergence, e.g. for under-sampled oscillations (→ know your integrand!), program robustly, add tests, etc, as always!