6.6 Essential Skills Learned in Chapter 6

You should know:

- the idea behind the trigonometric approximation;
- what is the Gibbs phenomenon;
- the dicrete Fourier transform and its use for the trigonometric interpolation;
- the idea and the importance of the fast Fourier transform;
- how to use the fast Fourier transform for the efficient trigonometrical interpolation;
- the error behavior for the trionomteric interpolation;
- the aliasing formula and the Sampling-Theorem;
- how to use the fast Fourier transform for the efficient Chebychev interpolation.

7 Numerical Quadrature

Numerical quadrature

= Approximate evaluation of $\int_\Omega f(\mathbf{x})\,\mathrm{d}\mathbf{x}$, integration domain $\Omega \subset \mathbb{R}^d$

Continuous function $f : \Omega \subset \mathbb{R}^d \mapsto \mathbb{R}$ only available as `function y = f(x)` (point evaluation)

Special case $d = 1$: $\Omega = [a, b]$ (interval)

☞ Numerical quadrature methods are key building blocks for methods for the numerical treatment of differential equations.

Remark 7.0.1 (Importance of numerical quadrature).

☞ Numerical quadrature methods are key building blocks for methods for the numerical treatment of partial differential equations (➢ Course "Numerical treatment of partial differential equations") ⚠

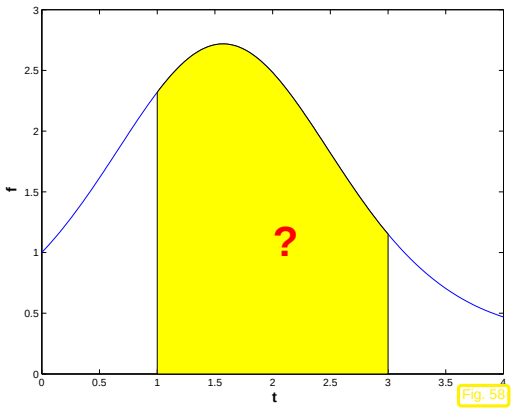


Fig. 58

Numerical quadrature methods

approximate

$$\int_a^b f(t)\,\mathrm{d}t$$

Example 7.0.2 (Heating production in electrical circuits).

Time-harmonic excitation:

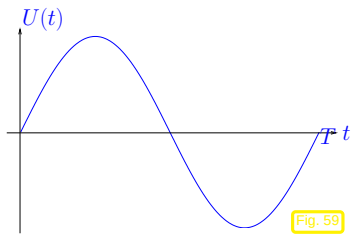


Fig. 59

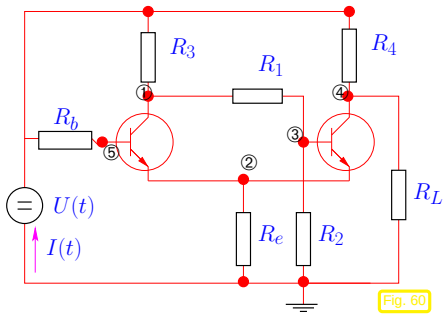


Fig. 60

Integrating power $P = UI$ over period $[0, T]$ yields heat production per period:

$$W_{\text{therm}} = \int_0^T U(t)I(t)\,\mathrm{d}t\,, \quad \text{where} \quad I = I(U)\,.$$

`function I = current(U)` involves solving non-linear system of equations! ◇

7.1 Quadrature Formulas

n-point quadrature formula on $[a,b]$: (n-point quadrature rule)

$$\int_a^b f(t)\,\mathrm{d}t \approx Q_n(f) := \sum_{j=1}^{n} w_j^n\, f(c_j^n)\,. \qquad (7.1.1)$$

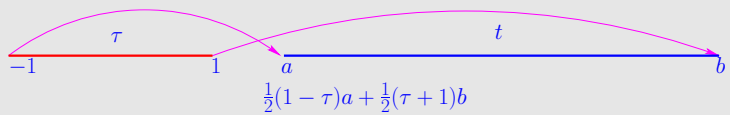
w_j^n : quadrature weights $\in \mathbb{R}$ (*ger.:* Quadraturgewichte)
c_j^n : quadrature nodes $\in [a,b]$ (*ger.:* Quadraturknoten)

Remark 7.1.1 (Transformation of quadrature rules).

Given: quadrature formula $(\widehat{c}_j, \widehat{w}_j)_{j=1}^n$ on reference interval $[-1,1]$

Idea: transformation formula for integrals

$$\int_a^b f(t)\,\mathrm{d}t = \tfrac{1}{2}(b-a)\int_{-1}^{1} \widehat{f}(\tau)\,\mathrm{d}\tau\,, \quad \widehat{f}(\tau) := f(\tfrac{1}{2}(1-\tau)a + \tfrac{1}{2}(\tau+1)b)\,. \qquad (7.1.2)$$



- quadrature formula for general interval $[a,b]$, $a,b \in \mathbb{R}$:

$$\int_a^b f(t)\,\mathrm{d}t \approx \tfrac{1}{2}(b-a)\sum_{j=1}^{n} \widehat{w}_j \widehat{f}(\widehat{c}_j) = \sum_{j=1}^{n} w_j f(c_j) \quad \text{with} \quad \begin{aligned} c_j &= \tfrac{1}{2}(1-\widehat{c}_j)a + \tfrac{1}{2}(1+\widehat{c}_j)b\,, \\ w_j &= \tfrac{1}{2}(b-a)\widehat{w}_j\,. \end{aligned}$$

- A 1D quadrature formula on arbitrary intervals can be specified by providing its weights $\widehat{w}_j$ /nodes $\widehat{c}_j$ for integration domain $[-1,1]$. Then the above transformation is assumed.

Other common choice of reference interval: $[0,1]$

△

Inevitable for generic integrand:

quadrature error $\quad E(n) := \left| \int_a^b f(t)\,\mathrm{d}t - Q_n(f) \right|$

Given families of quadrature rules $\{Q_n\}_n$ with quadrature weights $\left\{w_j^n,\ j=1,\dots,n\right\}_{n\in\mathbb{N}}$ and quadrature nodes $\left\{c_j^n,\ j=1,\dots,n\right\}_{n\in\mathbb{N}}$ we

should be aware of the *asymptotic* behavior of quadrature error $E(n)$ for $n \to \infty$

Qualitative distinction:
- ▷ algebraic convergence $E(n) = O(n^{-p})$, $p > 0$
- ▷ exponential convergence $E(n) = O(q^n)$, $0 \le q < 1$

Note that the number n of nodes agrees with the number of f-evaluations required for evaluation of the quadrature formula. This is usually used as a *measure for the cost* of computing $Q_n(f)$.

Therefore we consider the quadrature error as a function of n.

Idea: Equidistant quadrature nodes $t_j := a + hj$, $h := \frac{b-a}{n}$, $j = 0,\dots,n$: choose the n weights such that the error $E(n) = 0$ for all polynomials f of degree $n-1$.

Example 7.1.2 (Newton-Cotes formulas).

- $n = 1$: Trapezoidal rule

$$\widehat{Q}_{\mathrm{trp}}(f) := \frac{1}{2}\left(f(0) + f(1)\right) \qquad (7.1.3)$$

$$\left(\int_a^b f(t)\,\mathrm{d}t \approx \frac{b-a}{2}(f(a)+f(b)) \right)$$

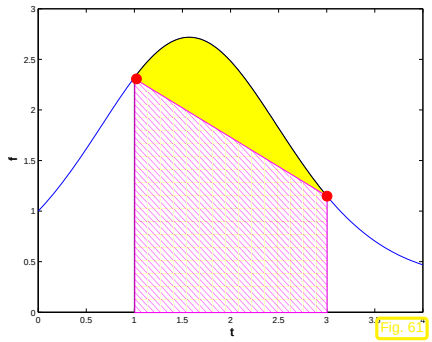


Fig. 61

• $n = 2$: Simpson rule

$$\frac{h}{6}\left(f(0) + 4\,f(\tfrac{1}{2}) + f(1)\right) \quad \left(\int_a^b f(t)\,\mathrm{d}t \approx \frac{b-a}{6}\left(f(a) + 4\,f\left(\frac{a+b}{2}\right) + f(b)\right)\right) \tag{7.1.4}$$

◇

Remark 7.1.3 (Error estimates for polynomial quadrature).

Quadrature error estimates directly from L^∞-interpolation error estimates for Lagrangian interpolation with polynomial of degree $n-1$:

$$f \in C^n([a,b]) \quad \Rightarrow \quad \left| \int_a^b f(t)\,\mathrm{d}t - Q_n(f) \right| \le \frac{1}{n!}(b-a)^{n+1} \left\| f^{(n)} \right\|_{L^\infty([a,b])} . \tag{7.1.5}$$

△

Idea: Gaussian quadrature: Choose the n weights and the n points such that the error $E(n) = 0$ for all polynomials f of degree $2n-1$.

Example 7.1.4 (2-point quadrature rule of order 4).

Necessary & sufficient conditions for order 4 (first wrong integral is $\int_a^b x^4 dx$):

$$Q_n(p) = \int_a^b p(t)\,\mathrm{d}t \;\forall p \in \mathcal{P}_3 \quad \Leftrightarrow \quad Q_n(t^q) = \frac{1}{q+1}(b^{q+1} - a^{q+1}) , \quad q = 0,1,2,3 .$$

4 equations for weights w_j and nodes c_j, $j = 1,2$ $(a = -1, b = 1)$, *cf.* Rem. **??**

$$\begin{aligned} \int_{-1}^1 1\,\mathrm{d}t &= 2 = 1w_1 + 1w_2 \quad , \quad \int_{-1}^1 t\,\mathrm{d}t = 0 = c_1 w_1 + c_2 w_2 \\ \int_{-1}^1 t^2\,\mathrm{d}t &= \frac{2}{3} = c_1^2 w_1 + c_2^2 w_2 \quad , \quad \int_{-1}^1 t^3\,\mathrm{d}t = 0 = c_1^3 w_1 + c_2^3 w_2 . \end{aligned} \tag{7.1.6}$$

Solve using MAPLE:

```
> eqns := seq(int(x^k, x=-1..1) = w[1]*xi[1]^k+w[2]*xi[2]^k,k=0..3);
> sols := solve(eqns, indets(eqns, name)):
> convert(sols, radical);
```


➢ weights & nodes: $\left\{ w_2 = 1, w_1 = 1, c_1 = 1/3\sqrt{3}, c_2 = -1/3\sqrt{3} \right\}$

▶ quadrature formula: $$\int_{-1}^1 f(x)\,dx \approx f\left(\frac{1}{\sqrt{3}}\right) + f\left(-\frac{1}{\sqrt{3}}\right) \tag{7.1.7}$$

◇

Remark 7.1.5 (Computing Gauss nodes and weights).

Compute nodes/weights of Gaussian quadrature by solving an eigenvalue problem! (Golub-Welsch algorithm [18, Sect. 3.5.4])

In codes: c_j, w_j from tables!

Code 7.1.6: Golub-Welsch algorithm

```
from numpy import zeros, diag, sqrt, size
from numpy.linalg import eigh

def gaussquad(n):
        b = zeros(n-1);
        for i in xrange(size(b)):
          b[i]=(i+1)/sqrt(4*(i+1)*(i+1)-1)
        J=diag(b,-1)+diag(b,1)
        x,ev=eigh(J); w=2*ev[0]*ev[0]
        return (x,w)
```

△

Idea: Clenshaw-Curtis quadrature:

$$\int_{-1}^1 f(x)\,dx = \int_0^\pi f(\cos\theta)\,\sin\theta d\theta = \sum_{\text{even } k} \frac{2a_k}{1-k^2} \tag{7.1.8}$$

with a_k the Fourier coefficients of $F(\theta) = f(\cos\theta) = \sum_{k=0}^\infty a_k \cos(k\theta)$.

Advantage for the Clenshaw-Curtis is the speed and stability of the fast Fourier transform.

Code 7.1.7: Clensahw-Curtis: direct implementation

```
def cc2(func,a,b,N):
    """
    Clenshaw-Curtis quadrature rule
    by FFT with the function values
    """
    bma = 0.5*(b-a)
    x = np.cos(np.pi * np.linspace(0,N,N+1)/N) # Chebyshev points
    x *= bma
    x += 0.5*(a+b)
    fx = func(x)*0.5/N
    vx = np.hstack((fx,fx[-2:0:-1]))
    g = np.real(np.fft.fft(vx))
    A = np.zeros(N+1)
    A[0] = g[0]; A[N] = g[N]
    A[1:N] = g[1:N] + np.flipud(g[N+1:])
```

```
    w = 0.*x
    w[::2] = 2./(1.-np.r_[:N+1:2]**2)
    return np.dot(w,A)*bma
```

Code 7.1.8: Clensahw-Curtis: weights and points

```
def cc1(func, a, b, N):
    """
    Clenshaw-Curtis quadrature rule
    by constructing the points and the weights
    """
    bma = b-a
    c = np.zeros([2,2*N-2])
    c[0][0] = 2.0
    c[1][1] = 1
    c[1][-1]= 1
    for i in np.arange(2.,N,2):
        val = 2.0/(1-pow(i,2))
        c[0][i] = val
        c[0][2*N-2-i] = val

    f = np.real(np.fft.ifft(c))
    w = f[0][:N]; w[0] *= 0.5; w[-1] *= 0.5 # weights
```

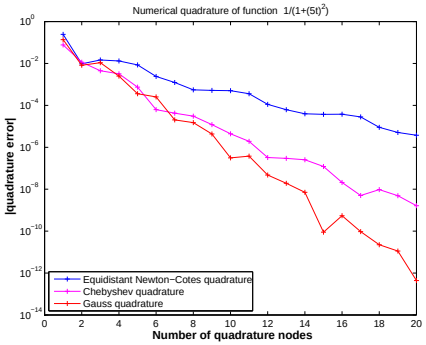
```
    x = 0.5*((b+a)+(N-1)*bma*f[1][:N]) # points
    return np.dot(w,func(x))*bma
```

Example 7.1.9 (Error of (non-composite) quadratures).

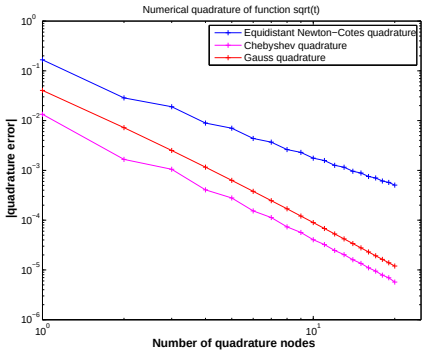
Code 7.1.10: important polynomial quadrature rules

```
from gaussquad import gaussquad
from numpy import *

def numquad(f,a,b,N,mode='equidistant'):
        """Numerical quadrature on [a,b] by polynomial quadrature formula
        f -> function to be integrated (handle)
        a,b -> integration interval [a,b] (endpoints included)
        N -> Maximal degree of polynomial
        mode (equidistant, Chebychev = Clenshaw-Curtis, Gauss) selects quadrature rule
        """
        # use a dictionary as "switch" statement:
        quadrule = {'gauss':quad_gauss, 'equidistant':quad_equidistant, 'chebychev':quad_chebychev}
        nvals = range(1,N+1); res = []
        try:
                for n in nvals:
                        res.append(quadrule[mode.lower()](f,a,b,n))
        except KeyError:
                print "invalid quadrature type!"
        else:
                return (nvals,res)

def quad_gauss(f,a,b,deg):
        # get Gauss points for [-1,1]
        [gx,w] = gaussquad(deg);
        # transform to [a,b]
        x = 0.5*(b-a)*gx+0.5*(a+b)
        y = f(x)
        return 0.5*(b-a)*dot(w,y)

def quad_equidistant(f,a,b,deg):
        p = arange(deg+1.0,0.0,-1.0)
        w = (power(b,p) - power(a,p))/p
        x = linspace(a,b,deg+1)
        y = f(x)
        # "Quick and dirty" implementation through polynomial interpolation
        poly = polyfit(x,y,deg)
        return dot(w,poly)
```

```

def quad_chebychev(f,a,b,deg):
        p = arange(deg+1.0,0.0,-1.0)
        w = (power(b,p) - power(a,p))/p
        x = 0.5*(b-a)*cos((arange(0,deg+1)+0.5)/(deg+1)*pi)+0.5*(a+b);
        y = f(x)
        # "Quick and dirty" implementation through polynomial interpolation
        poly = polyfit(x,y,deg)
        return dot(w,poly)
```



quadrature error, $f_1(t) := \frac{1}{1+(5t)^2}$ on $[0,1]$



quadrature error, $f_2(t) := \sqrt{t}$ on $[0,1]$

Asymptotic behavior of quadrature error $\epsilon_n := \left| \int_0^1 f(t)\,\mathrm{d}t - Q_n(f) \right|$ for "$n \to \infty$":

➢ exponential convergence $\epsilon_n \approx O(q^n)$, $0 < q < 1$, for C^∞-integrand f_1 $\rightsquigarrow$: Newton-Cotes quadrature : $q \approx 0.61$, Clenshaw-Curtis quadrature : $q \approx 0.40$, Gauss-Legendre quadrature : $q \approx 0.27$

➢ algebraic convergence $\epsilon_n \approx O(n^{-\alpha})$, $\alpha > 0$, for integrand f_2 with singularity at $t = 0$ $\rightsquigarrow$ Newton-Cotes quadrature : $\alpha \approx 1.8$, Clenshaw-Curtis quadrature : $\alpha \approx 2.5$, Gauss-Legendre quadrature : $\alpha \approx 2.7$

Code 7.1.11: tracking errors on quadrature rules

```
from numquad import numquad
import matplotlib.pyplot as plt
from numpy import *

def numquaderrs():
        """Numerical quadrature on [0,1]"""
        N = 20;

        plt.figure()
        exact = arctan(5)/5;
        f = lambda x: 1./(1+power(5.0*x,2))
        nvals,eqdres = numquad(f,0,1,N,'equidistant')
        nvals,chbres = numquad(f,0,1,N,'Chebychev')
        nvals,gaures = numquad(f,0,1,N,'Gauss')
        plt.semilogy(nvals,abs(eqdres-exact),'b+-',label='Equidistant Newton-Cotes quadrature')
        plt.semilogy(nvals,abs(chbres-exact),'m+-',label='Clenshaw-Curtis quadrature')
        plt.semilogy(nvals,abs(gaures-exact),'r+-',label='Gauss quadrature')
        plt.title('Numerical quadrature of function 1/(1+(5t)^2)')
        plt.xlabel('f Number of quadrature nodes')
        plt.ylabel('f |quadrature error|')
        plt.legend(loc="lower left")
        plt.show()
# eqdp1 = polyfit(nvals,log(abs(eqdres-exact)),1)
# chbp1 = polyfit(nvals,log(abs(chbres-exact)),1)
# gaup1 = polyfit(nvals,log(abs(gaures-exact)),1)
# plt.savefig("../PICTURES/numquaderr1.eps")

        plt.figure()
        exact = array(2./3.);
        f = lambda x: sqrt(x)
        nvals,eqdres = numquad(f,0,1,N,'equidistant')
        nvals,chbres = numquad(f,0,1,N,'Chebychev')
        nvals,gaures = numquad(f,0,1,N,'Gauss')
        plt.loglog(nvals,abs(eqdres-exact),'b+-',label='Equidistant Newton-Cotes quadrature')
        plt.loglog(nvals,abs(chbres-exact),'m+-',label='Clenshaw-Curtis quadrature')
        plt.loglog(nvals,abs(gaures-exact),'r+-',label='Gauss quadrature')
        plt.axis([1, 25, 0.000001, 1]);
        plt.title('Numerical quadrature of function sqrt(t)')
        plt.xlabel('f Number of quadrature nodes')
        plt.ylabel('f |quadrature error|')
        plt.legend(loc="lower left")
        plt.show()
# eqdp1 = polyfit(nvals,log(abs(eqdres-exact)),1)
# chbp1 = polyfit(nvals,log(abs(chbres-exact)),1)
# gaup1 = polyfit(nvals,log(abs(gaures-exact)),1)
# plt.savefig("../PICTURES/numquaderr2.eps")

if "__name__" == "__main__":
        numquaderrs()
```


◇

Equal spacing is a disaster for high-order interpolation and integration !

➢Divide the integration domain in small pieces and use low-order rule on each piece (composite quadrature)

➢Take into account the eventual non-smoothness of f when dividing the integration domain

7.2 Composite Quadrature

With $a = x_0 < x_1 < \cdots < x_{m-1} < x_m = b$

$$\int_a^b f(t)\,\mathrm{d}t = \sum_{j=1}^{m} \int_{x_{j-1}}^{x_j} f(t)\,\mathrm{d}t\,. \tag{7.2.1}$$

Recall (7.1.5): for polynomial quadrature rule and $f \in C^n([a,b])$ quadrature error shrinks with $n+1$st power of length of integration interval.

► Reduction of quadrature error can be achieved by

- splitting of the integration interval according to (7.2.1),
- using the intended quadrature formula on each sub-interval $[x_{j-1}, x_j]$.

Note: Increasse in total no. of f-evaluations incurred, which has to be balanced with the gain in accuracy to achieve optimal efficiency,

Idea:

- Partition integration domain $[a,b]$ by mesh (grid) $\mathcal{M} := \{a = x_0 < x_1 < \ldots < x_m = b\}$
- Apply quadrature formulas on sub-intervals $I_j := [x_{j-1}, x_j]$, $j = 1, \ldots, m$, and sum up.

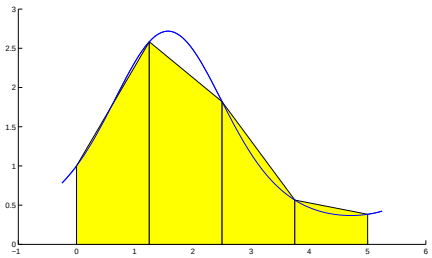
► composite quadrature rule

Note: Here we only consider one and the same quadrature formula (local quadrature formula) applied on all sub-intervals.

Example 7.2.1 (Simple composite polynomial quadrature rules).

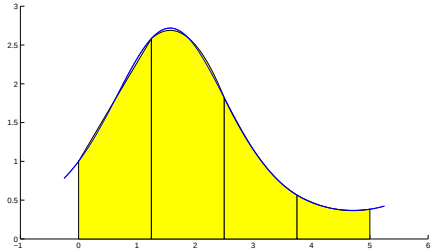
Composite trapezoidal rule, *cf.* (8.6.2)

$$\int_a^b f(t)\mathrm{d}t = \tfrac{1}{2}(x_1 - x_0)f(a) + \sum_{j=1}^{m-1} \tfrac{1}{2}(x_{j+1} - x_{j-1})f(x_j) + \tfrac{1}{2}(x_m - x_{m-1})f(b)\,. \tag{7.2.2}$$



Composite Simpson rule, *cf.* (7.1.4)

$$\int_a^b f(t)\mathrm{d}t = \tfrac{1}{6}(x_1 - x_0)f(a) + \sum_{j=1}^{m-1} \tfrac{1}{6}(x_{j+1} - x_{j-1})f(x_j) + \sum_{j=1}^{m} \tfrac{2}{3}(x_j - x_{j-1})f(\tfrac{1}{2}(x_j + x_{j-1})) + \tfrac{1}{6}(x_m - x_{m-1})f(b)\,. \tag{7.2.3}$$



Formulas (7.2.2), (7.2.3) directly suggest efficient implementation with minimal number of f-evaluations.

◇

Focus: *asymptotic* behavior of quadrature error for

mesh width $\quad h := \max_{j=1,\ldots,m} |x_j - x_{j-1}| \to 0$

For *fixed* local n-point quadrature rule: $O(mn)$ f-evaluations for composite quadrature ("total cost")

➢ If mesh equidistant ($|x_j - x_{j-1}| = h$ for all j), then total cost for composite numerical quadrature $= O(h^{-1})$.

Theorem 7.2.1 (Convergence of composite quadrature formulas)**.**
For a composite quadrature formula Q based on a local quadrature formula of order $p \in \mathbb{N}$ holds

$$\exists C > 0: \quad \left| \int_I f(t)\,\mathrm{d}t - Q(f) \right| \le C h^p \left\| f^{(p)} \right\|_{L^\infty(I)} \quad \forall f \in C^p(I), \forall \mathcal{M}\,.$$

Proof. Apply interpolation error estimate . □

Example 7.2.2 (Quadrature errors for composite quadrature rules).

Composite quadrature rules based on

- trapezoidal rule (8.6.2) ➢ local order 2 (exact for linear functions),
- Simpson rule (7.1.4) ➢ local order 3 (exact for quadratic polynomials)

on equidistant mesh $\mathcal{M} := \{jh\}_{j=0}^{n}$, $h = 1/n$, $n \in \mathbb{N}$.

Code 7.2.3: composite trapezoidal rule (7.2.2)

```
def trapezoidal(func,a,b,N):
        """
        Numerical quadrature based on trapezoidal rule
        func: handle to y = f(x)
        a,b: bounds of integration interval
        N+1: number of equidistant quadrature points
        """

        from numpy import linspace, sum
        # quadrature nodes
        x = linspace(a,b,N+1); h = x[1]-x[0]
        # quadrature weights: internal nodes: w=1, boundary nodes: w=0.5
        I = sum(func(x[1:-1])) + 0.5*(func(x[0])+func(x[-1]))
        return I*h

if __name__ == "__main__":
        import matplotlib.pyplot as plt
```


Code 7.2.4: composite Simpson rule (7.2.3)

```
from numpy import linspace, sum, size

def simpson(func,a,b,N):
        """
        Numerical quadrature based on Simpson rule
        func: handle to y = f(x)
        a,b: bounds of integration interval
        N+1: number of equidistant quadrature points
        """

        # ensure that we have an even number of subintervals
        if N%2 == 1: N = N+1
        # quadrature nodes
        x = linspace(a,b,N+1); h = x[1]-x[0]
        # quadrature weights:
        # internal nodes: even: w=2/3, odd: w=4/3
        # boundary nodes: w=1/6
        I = h*sum(func(x[0:-2:2]) + 4*func(x[1:-1:2]) +
          func(x[2::2]))/3.0
        return I

if __name__ == "__main__":
        import matplotlib.pyplot as plt
```

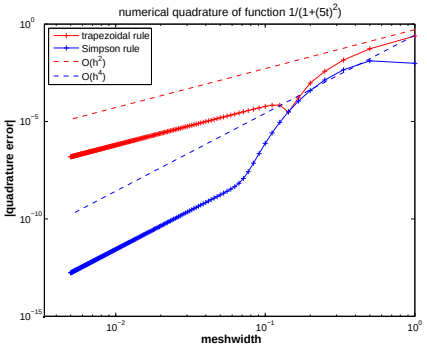


```
        from scipy import integrate
        from numpy import array, linspace, size, log, polyfit

        # define a function and an interval:
        f = lambda x: x**2
        left = 0.0; right = 1.0

        # exact integration with scipy.integrate.quad:
        exact,e = integrate.quad(f,left,right)
        # trapezoid rule for different number of quadrature points
        N = linspace(2,101,100)
        res = array(N) # preallocate same amount of space as N uses
        for i in xrange(size(N)):
                res[i] = trapezoidal(f,left,right,N[i])
        err = abs(res - exact)
        #plt.loglog(N,err,'o')
        #plt.show()

        # linear fit to determine convergence order
        p = polyfit(log(N),log(err),1)
        # output the convergence order
        print "convergence order:",-p[0]
```

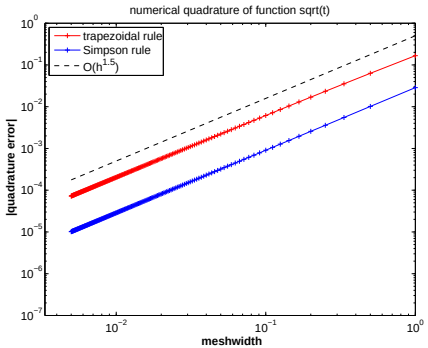


```
        from scipy import integrate
        from numpy import array, linspace, size, log, polyfit

        # define a function and an interval:
        f = lambda x: 1./(1+(5*x)**2)
        left = 0.0; right = 1.0

        # exact integration with scipy.integrate.quad:
        exact,e = integrate.quad(f,left,right)
        # trapezoid rule for different number of quadrature points
        N = linspace(2,101,100)
        res = array(N) # preallocate same ammount of space as N uses
        for i in xrange(size(N)):
                res[i] = simpson(f,left,right,N[i])
        err = abs(res - exact)
        #plt.loglog(N,err,'o')
        #plt.show()

        # linear fit to determine convergence order
        p = polyfit(log(N[-20:]),log(err[-20:]),1)
        # output the convergence order
        print "convergence order:",-p[0]
```

Note: `fnct` is supposed to accept vector arguments and return the function value for each vector component!



quadrature error, $f_1(t) := \frac{1}{1+(5t)^2}$ on $[0,1]$



quadrature error, $f_2(t) := \sqrt{t}$ on $[0,1]$

Asymptotic behavior of quadrature error $E(n) := \left|\int_0^1 f(t)\,\mathrm{d}t - Q_n(f)\right|$ for meshwidth "$h \to 0$"

☛ algebraic convergence $E(n) = O(h^\alpha)$ of order $\alpha > 0$, $n = h^{-1}$

➢ Sufficiently smooth integrand f_1: trapezoidal rule $\to \alpha = 2$, Simpson rule $\to \alpha = 4$!?

➢ singular integrand f_2: $\alpha = 3/2$ for trapezoidal rule **&** Simpson rule **!**

(lack of) smoothness of integrand limits convergence !

Simpson rule: order = 4 **?** investigate with MAPLE

```
> rule := 1/3*h*(f(2*h)+4*f(h)+f(0))
> err := taylor(rule - int(f(x),x=0..2*h),h=0,6);
```

$$err := \left(\frac{1}{90}\left(D^{(4)}\right)(f)(0)\,h^5 + O\left(h^6\right), h, 6\right)$$

➢ Simpson rule is of order 4, indeed **!**

Code 7.2.5: errors of composite trapezoidal and Simpson rule

```
#!/usr/bin/env python

import numpy as np
import matplotlib.pyplot as plt
from scipy import integrate
# own integrators:
from trapezoidal import trapezoidal
from simpson import simpson

# integration functions:
integrators = [trapezoidal, simpson]
intNames = ('trapezoidal','simpson')

# define a few different Ns
N = np.linspace(2,201,200)

# define a function...
f = lambda x: 1./(1+(5*x)**2)
#f = lambda x: x**2
# ...and an interval
left=0.0; right=1.0

# "exact" integration with scipy function:
exact,e = integrate.quad(f,left,right,epsabs=1e-12)

# our versions
err = []; res = []
for int in integrators:
        for i in xrange(np.size(N)):
                res.append(int(f,left,right,N[i]))
        err.append(np.abs(np.array(res)-exact))
        res = []

plt.figure()
# evaluation
logN=np.log(N)
for i in xrange(np.size(integrators)):
        # linear fit to determine convergence orders
        p = np.polyfit(logN[-20:],np.log(err[i][-20:]),1) # only look at last 20 entries- asymptotic!
        # plot errors
        plt.loglog(N,err[i],'o', label=intNames[i])
        # plot linear fitting
        x = np.linspace(min(logN),max(logN),10)
        y = np.polyval(p,x)
        plt.loglog(np.exp(x),np.exp(y),label="linear fit: m="+str(-p[0])[:4])
        # output the convergence order
        print "convergence order of "+intNames[i]+":",-p[0]

plt.xlabel("log(N)"); plt.ylabel("log(err)")
```

```
plt.legend(loc="lower left")
plt.show()
```

◇

Remark 7.2.6 (Removing a singularity by transformation).

Ex. 7.2.2 ➢ lack of smoothness of integrand limits rate of algebraic convergence of composite quadrature rule for meshwidth $h \to 0$.

Idea: recover integral with smooth integrand by "analytic preprocessing"

Here is an example:

For $f \in C^\infty([0,b])$ compute $\int_0^b \sqrt{t} f(t)\,\mathrm{d}t$ via quadrature rule (→ Ex. 7.2.2)

substitution $s = \sqrt{t}$: $$\int_0^b \sqrt{t} f(t)\,\mathrm{d}t = \int_0^{\sqrt{b}} \boxed{2s^2 f(s^2)}\,\mathrm{d}s\,. \tag{7.2.4}$$

Then: Apply quadrature rule to smooth integrand

△

Example 7.2.7 (Convergence of equidistant trapezoidal rule).

Sometimes there are surprises: convergence of a composite quadrature rule is much better than predicted by the order of the local quadrature formula:

Equidistant trapezoidal rule (order 2), see (7.2.2)

$$\int_a^b f(t)\,\mathrm{d}t \approx T_m(f) := h\left(\tfrac{1}{2}f(a) + \sum_{k=1}^{m-1} f(kh) + \tfrac{1}{2}f(b)\right)\,, \quad h := \frac{b-a}{m}\,. \tag{7.2.5}$$

Code 7.2.8: equidistant trapezoidal quadrature formula

```
def trapezoidal(func,a,b,N):
        """
        Numerical quadrature based on trapezoidal rule
        func: handle to y = f(x)
        a,b: bounds of integration interval
        N+1: number of equidistant quadrature points
        """

        from numpy import linspace, sum
        # quadrature nodes
        x = linspace(a,b,N+1); h = x[1]-x[0]
        # quadrature weights: internal nodes: w=1, boundary nodes: w=0.5
        I = sum(func(x[1:-1])) + 0.5*(func(x[0])+func(x[-1]))
        return I*h

if __name__ == "__main__":
        import matplotlib.pyplot as plt
        from scipy import integrate
        from numpy import array, linspace, size, log, polyfit

        # define a function and an interval:
        f = lambda x: x**2
        left = 0.0; right = 1.0

        # exact integration with scipy.integrate.quad:
        exact,e = integrate.quad(f,left,right)
        # trapezoid rule for different number of quadrature points
        N = linspace(2,101,100)
        res = array(N) # preallocate same amount of space as N uses

        for i in xrange(size(N)):
                res[i] = trapezoidal(f,left,right,N[i])
        err = abs(res - exact)
        #plt.loglog(N,err,'o')
        #plt.show()

        # linear fit to determine convergence order
        p = polyfit(log(N),log(err),1)
        # output the convergence order
        print "convergence order:",-p[0]
```

1-periodic smooth (analytic) integrand

$$f(t) = \frac{1}{\sqrt{1 - a\sin(2\pi t - 1)}}\,, \quad 0 < a < 1\,.$$

("exact value of integral": use T_{500})

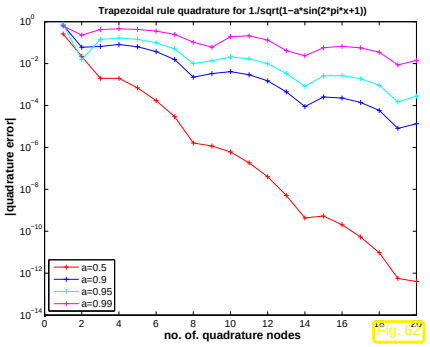


Fig. 62

quadrature error for $T_n(f)$ on $[0,1]$

exponential convergence !!

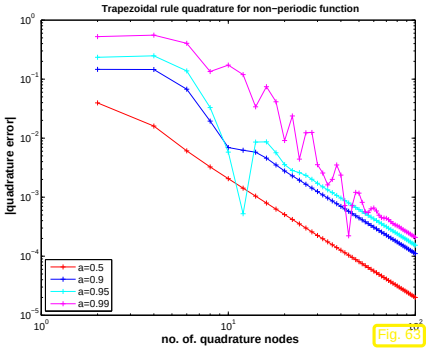


Fig. 63

quadrature error for $T_n(f)$ on $[0,\frac{1}{2}]$

merely algebraic convergence

Code 7.2.9: tracking error of equidistant trapezoidal quadrature formula

```
import matplotlib.pyplot as plt
from scipy import integrate
from trapezoidal import trapezoidal
from numpy import *

# define the function: (0 < a < 1)
f = lambda x: 1./sqrt(1-a*sin(2*pi*x+1))
avals = [0.5, 0.9, 0.95, 0.99];

left = 0
# first interval: [0, 0.5]
err = []; right = 0.5
# loop over different values of a:
for a in avals:
        # exact integration with scipy.integrate.quad:
        exact,e = integrate.quad(f,left,right)
        # trapezoid rule for different number of quadrature points
        N = linspace(2,50,49)
        res = array(N) # preallocate same amount of space as N uses
        for i in xrange(size(N)):
                res[i] = trapezoidal(f,left,right,N[i])
        err.append( abs(res - exact) )
plt.figure()
plt.xlabel('N')
plt.ylabel('error')
plt.title(u'Trapezoidal rule quadrature for non-periodic function')
for i in xrange(size(avals)): plt.loglog(N,err[i],'-o',label='a=
```



```
  '+str(avals[i]))
plt.legend(loc="lower left")
plt.show()
#plt.savefig("../PICTURES/traperr2.eps")

# second interval: [0, 1]
err = []; right = 1
# loop over different values of a:
for a in avals:
        # exact integration with scipy.integrate.quad:
        exact,e = integrate.quad(f,left,right)
        # trapezoid rule for different number of quadrature points
        N = linspace(2,50,49)
        res = array(N) # preallocate same amount of space as N uses
        for i in xrange(size(N)):
                res[i] = trapezoidal(f,left,right,N[i])
        err.append( abs(res - exact) )
plt.figure()
plt.xlabel('N')
plt.ylabel('error')
plt.title(r"Trapezoidal rule quadrature for $(1-a\cdot\sin(2\pi x+1))^{\frac{-1}{2}}$")
for i in xrange(size(avals)): plt.loglog(N,err[i],'-o',label='a=
  '+str(avals[i]))
plt.legend(loc="lower left")
plt.show()
#plt.savefig("../PICTURES/traperr1.eps")
```


Explanation:

$$f(t) = e^{2\pi i k t} \quad \blacktriangleright \quad \begin{cases} \int_0^1 f(t)\,\mathrm{d}t = \begin{cases} 0 & \text{, if } k \neq 0\,, \\ 1 & \text{, if } k = 0\,. \end{cases} \\ T_m(f) = \frac{1}{m}\sum_{l=0}^{m-1} e^{\frac{2\pi i}{m}lk} \overset{(6.1.2)}{=} \begin{cases} 0 & \text{, if } k \notin m\mathbb{Z}\,, \\ 1 & \text{, if } k \in m\mathbb{Z}\,. \end{cases} \end{cases}$$

$\blacktriangleright$

Equidistant trapezoidal rule T_m is exact for trigonometric polynomials of degree $< 2m$!

It takes sophisticated tools from complex analysis to conclude exponential convergence for analytic integrands from the above observation.

$\diamond$

7.3 Adaptive Quadrature

Example 7.3.1 (Rationale for adaptive quadrature).

Consider composite trapezoidal rule (7.2.2) on mesh $\mathcal{M} := \{a = x_0 < x_1 < \cdots < x_m = b\}$:

Local quadrature error (for $f \in C^2([a,b])$):

$$\int_{x_{k-1}}^{x_k} f(t)\,\mathrm{d}t - \frac{1}{2}(f(x_{k-1}) + f(x_k)) \leq (x_k - x_{k-1})^3 \left\|f''\right\|_{L^\infty([x_{k-1},x_k])} .$$

➢ Do not use equidistant mesh !

Refine $\mathcal{M}$, where $|f''|$ large !

Makes sense, e.g., for "spike function" ▷

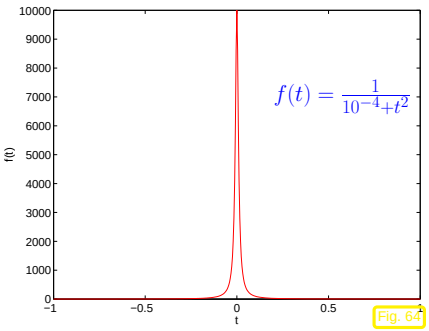


Fig. 64

◇

Goal:	*Equilibrate error contributions of all mesh intervals*
Tool:	Local a posteriori error estimation
	(Estimate contributions of mesh intervals from intermediate results)
Policy:	Local mesh refinement

▶ *Adaptive multigrid quadrature* → [14, Sect. 9.7]

Idea: local error estimation by comparing local results of two quadrature formulas Q_1, Q_2 of *different* order → local error estimates

heuristics: error$(Q_2) \ll$ error(Q_1) ⇒ error$(Q_1) \approx Q_2(f) - Q_1(f)$.

Now: Q_1 = trapezoidal rule (order 2) ↔ Q_2 = Simpson rule (order 4)

Given: mesh $\mathcal{M} := \{a = x_0 < x_1 < \cdots < x_m = b\}$

❶ (error estimation)

For $I_k = [x_{k-1}, x_k]$, $k = 1, \ldots, m$ (midpoints $p_k := \frac{1}{2}(x_{k-1} + x_k)$)

$$\mathtt{EST}_k := \Bigg|\underbrace{\frac{h_k}{6}(f(x_{k-1}) + 4f(p_k) + f(x_k))}_{\text{Simpson rule}} - \underbrace{\frac{h_k}{4}(f(x_{k-1}) + 2f(p_k) + f(x_k))}_{\text{trapezoidal rule on split mesh interval}}\Bigg| . \tag{7.3.1}$$

❷ (Termination)

Simpson rule on $\mathcal{M}$ ⇒ preliminary result I

If $\displaystyle\sum_{k=1}^{m} \mathtt{EST}_k \leq \mathtt{RTOL} \cdot I$ ($RTOL$:= prescribed tolerance) ⇒ **STOP** (7.3.2)

❸ (local mesh refinement)

$$\mathcal{S} := \{k \in \{1,\ldots,m\} : \mathtt{EST}_k \geq \eta \cdot \frac{1}{m}\sum_{j=1}^{m} \mathtt{EST}_j\} , \quad \eta \approx 0.9 . \tag{7.3.3}$$

▶ new mesh: $\mathcal{M}^* := \mathcal{M} \cup \{p_k : k \in \mathcal{S}\}$.

Then continue with step ❶ and mesh $\mathcal{M} \leftarrow \mathcal{M}^*$.

Non-optimal recursive implementation:

Code 7.3.2: h-adaptive numerical quadrature

```
from numpy import *
from scipy import integrate

def adaptquad(f,M,rtol,abstol):
        """
        adaptive quadrature using trapezoid and simpson rules
        Arguments:
        f            handle to function f
        M            initial mesh
        rtol         relative tolerance for termination
        abstol       absolute tolerance for termination, necessary in case the exact integral value = 0, which renders a relative tolerance meaningless.
        """
        h = diff(M)                    # compute lengths of mesh intervals
        mp = 0.5*( M[:-1]+M[1:] )      # compute midpoint positions
        fx = f(M); fm = f(mp)          # evaluate function at positions and midpoints
        trp_loc = h*( fx[:-1]+2*fm+fx[1:] )/4    # local trapezoid rule
        simp_loc= h*( fx[:-1]+4*fm+fx[1:] )/6    # local simpson rule
        I = sum(simp_loc)              # use simpson rule value as intermediate approximation for integral value
        est_loc = abs(simp_loc - trp_loc)        # difference of values
```

```
20          obtained from local composite trapezoidal rule and local simpson rule is used as an
            estimate for the local quadrature error.
            err_tot  = sum(est_loc)                                    # estimate
            for global error (sum moduli of local error contributions)
21          # if estimated total error not below relative or absolute threshold, refine mesh
22          if err_tot > rtol*abs(I) and err_tot > abstol:
23                  refcells = nonzero( est_loc >
                      0.9*sum(est_loc)/size(est_loc) )[0]
24                  I =
                      adaptquad(f,sort(append(M,mp[refcells])),rtol,abstol)
                      # add midpoints of intervalls with large error contributions,
                      recurse.
25          return I
26
27  if __name__ == '__main__':
28          f = lambda x: exp(6*sin(2*pi*x))
29          #f = lambda x: 1.0/(1e-4+x*x)
30          M = arange(11.)/10  # 0, 0.1, ... 0.9, 1
31          rtol = 1e-6; abstol = 1e-10
32          I = adaptquad(f,M,rtol,abstol)
33          exact,e = integrate.quad(f,M[0],M[-1])
34          print 'adaptquad:',I, '"exact":',exact
35          print 'error:',abs(I-exact)
```


Comments on Code 7.3.1:

- Arguments: `f` $\hat{=}$ *handle* to function f, `M` $\hat{=}$ initial mesh, `rtol` $\hat{=}$ relative tolerance for termination, `abstol` $\hat{=}$ absolute tolerance for termination, necessary in case the exact integral value $= 0$, which renders a relative tolerance meaningless.
- line 13: compute lengths of mesh-intervals $[x_{j-1}, x_j]$,
- line 14: store positions of midpoints p_j,
- line 15: evaluate function (vector arguments!),
- line 16: local composite trapezoidal rule (7.2.2),
- line 17: local simpson rule (7.1.4),
- line 18: value obtained from composite simpson rule is used as intermediate approximation for integral value,
- line 19: difference of values obtained from local composite trapezoidal rule ($\sim Q_1$) and ocal simpson rule ($\sim Q_2$) is used as an estimate for the local quadrature error.
- line 20: estimate for global error by summing up moduli of local error contributions,
- line 21: terminate, once the estimated total error is below the relative or absolute error threshold,
- line 24 otherwise, add midpoints of mesh intervals with large error contributions according to (7.3.3) to the mesh and continue.

Example 7.3.3 (h-adaptive numerical quadrature).

- approximate $\displaystyle\int_0^1 \exp(6\sin(2\pi t))\,\mathrm{d}t$, initial mesh $\mathcal{M}_0 = \{j/10\}_{j=0}^{10}$

Algorithm: adaptive quadrature, Code 7.3.1

Tolerances: `rtol` $= 10^{-6}$, `abstol` $= 10^{-10}$

We monitor the distribution of quadrature points during the adaptive quadrature and the true and estimated quadrature errors. The "exact" value for the integral is computed by composite Simpson rule on an equidistant mesh with 10^7 intervals.

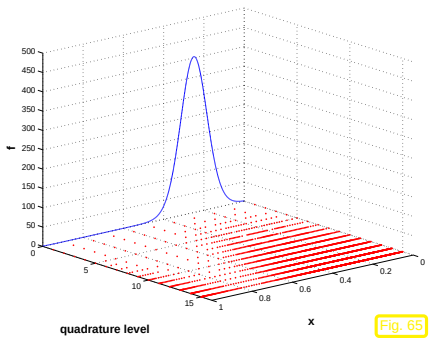


Fig. 65

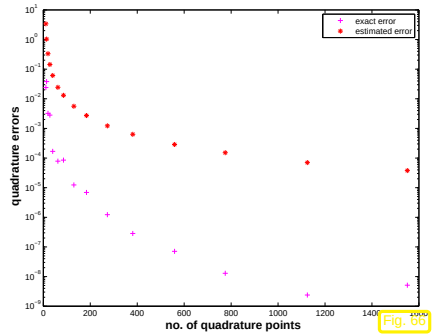


Fig. 66

- approximate $\displaystyle\int_0^1 \min\{\exp(6\sin(2\pi t)), 100\}\,\mathrm{d}t$, initial mesh as above

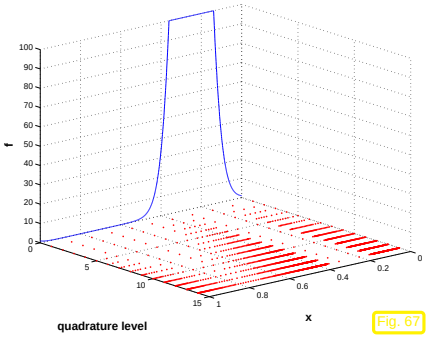


Fig. 67

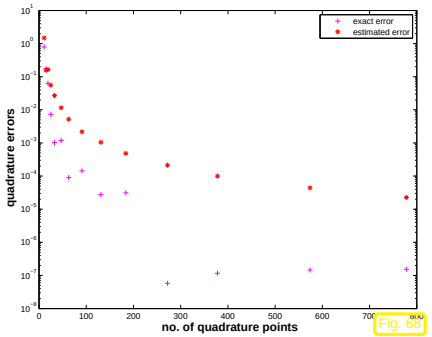


Fig. 68

Observation:

- Adaptive quadrature locally decreases meshwidth where integrand features variations or kinks.
- Trend for estimated error mirrors behavior of true error.
- Overestimation may be due to taking the modulus in (7.3.1)

However, the important information we want to glean from EST_k is about the *distribution* of the quadrature error.

Num. Meth. Phys. Gradinaru D-MATH 7.3 p. 405

◇

Remark 7.3.4 (Adaptive quadrature in Python).

`scipy.integrate.quad`:	adaptive multigrid quadrature wrapper to the Fortran library Q
see help and `scipy.integrate.explain_quad()` for details	
`scipy.integrate.quadrature`:	adaptive Gaussian quadrature

△

7.4 Multidimensional Quadrature

The cases of dimension $d = 2$ or $d = 3$ are easely treated by the iteration of the previous quadrature rules, e.g:

$$I = \int_a^b \int_c^d f(x,y)dxdy = \int_a^b F(y)dy,$$

where

$$F(y) = \int_c^d f(x,y)dx \approx \sum_{j_1=1}^{n_1} w_{j_1}^1 f(c_{j_1}^1, y),$$

with the corresponding quadrature rule in the x-direction.
The integration of $F(y)$ requires then the quadrature rule in the y-direction:

$$I \approx \sum_{j_1=1}^{n_1} \sum_{j_2=1}^{n_2} w_{j_1}^1 w_{j_2}^2 f(c_{j_1}^1, c_{j_2}^2).$$

We obtain hence the tensor product quadrature:
given the 1-dimensional quadrature rules

$$\left(w_{j_k}^k, c_{j_k}^k\right)_{1 \le j_k \le n_k}, \qquad k = 1, \dots, d,$$

the d-dimensional integral is approximated by

$$I \approx \sum_{j_1=1}^{n_1} \dots \sum_{j_d=1}^{n_d} w_{j_1}^1 \dots w_{j_d}^d f(c_{j_1}^1, \dots, c_{j_d}^d).$$

☹ Drawback: the number of d-dimensional quadrature points N increases exponentially with dimension d: with n quadrature points in each direction, we have $N = n^d$ quadrature points!

Num. Meth. Phys. Gradinaru D-MATH 7.4 p. 40

➢ Note: the convergence speed depends essentially on the dimension and smoothness of the function to integrate: $O(N^{-r/d})$ for a function $f \in C^r$.

7.4.1 Quadrature on Classical Sparse Grids

Consider first a quadrature formula in 1-dimension:

$$I = \int_a^b f(x)dx \approx Q_n^1(f).$$

Increasing the number of points used by the quadrature is expected to improve the result:
$Q_{n+1}^1(f) - Q_n^1(f)$ is expected to decrease. Here a simple example using the trapezoidal rule:

$$T_1(f,a,b) = \frac{f(a)+f(b)}{2}(b-a),$$

and let us denote the error by

$$S_1(f,a,b) = \int_a^b f(x)dx - T_1(f,a,b),$$

Num. Meth. Phys. Gradinaru D-MATH 7.4 p. 406

Num. Meth. Phys. Gradinaru D-MATH 7.4 p. 40

which describe in Figure 7.4.1 the area uncovered by the first trapez T_1. Here we used only the values $f(a)$ and $f(b)$. Clearly, the approximation is quite bad. We may improve it by using two shorter trapezes, i.e. using $f((a+b)/2)$ as supplementary information:

$$T_2(f,a,b) = \frac{f(a)+f((a+b)/2)}{2} + \frac{f((a+b)/2)+f(b)}{2}.$$

Clearly, both terms in the last sum are large.

However, we may improve the quadrature just by adding the information we gain by using the new function value $f(a+b)/2$. We only add to T_1 the value of the area of the triangle D_1 sitting on the top of the trapez T_1:

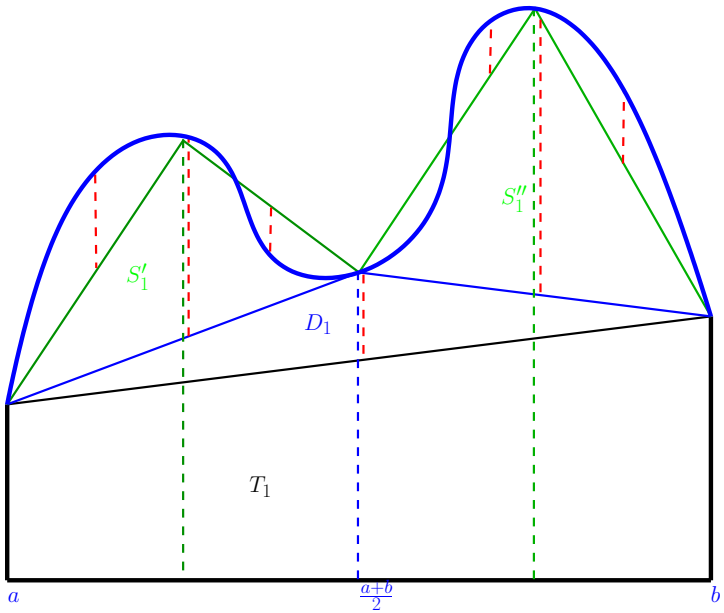
$$T_2(f,a,b) = T_1(f,a,b) + D_1(f,a,b),$$

with

$$D_1(f,a,b) = \left(f(\frac{a+b}{2}) - \frac{f(a)+f(b)}{2}\right)\frac{b-a}{2} = g_1(f,a,b)\frac{b-a}{2}.$$

We call here $g_1(f,a,b) = f(\frac{a+b}{2}) - \frac{f(a)+f(b)}{2}$ the hierarchical surplus of the function f on the interval $[a,b]$. The error is then the sum of the area of the triangle D_1 and the two smaller parts S_1' and S_1'':

$$S_1(f,a,b) = D_1(f,a,b) + S_1(f,a,\frac{a+b}{2}) + S_1(f,\frac{a+b}{2},b).$$



We expect to have smaller errors S_1' and S_2'' and hence smaller hierarchical surpluses when we repeat the procedure.

Let us focus on the interval $[0,1]$, which we divide in 2^ℓ sub-intervals $\frac{k}{2^\ell}, \frac{k+1}{2^\ell}$ for $k = 0,1,\dots,2^\ell-1$. We call ℓ level and consider Q_ℓ^1 a simple quadrature formula on each interval. Typical examples are the tapezoidal rule and the midpoint rule. The Clenshaw-Curtis rule and Gaussian rules are of the same flavour, but they spread the quadrature points non-uniformly.

The procedure described previously may be formulated as a simple telescopic sum of the details at each level:

$$\begin{aligned} Q_\ell^1 f &= Q_0^1 f + (Q_1^1 f - Q_0^1 f) + (Q_2^1 f - Q_1^1 f) + \dots + (Q_\ell^1 f - Q_{\ell-1}^1 f) \\ &= Q_0^1 f + \Delta_1^1 f + \Delta_2^1 f + \dots + \Delta_\ell^1 f. \end{aligned}$$

As long as we remain in 1-dimension, there is no gain in this reformulation. Things change fundamentally when going to d dimensions.

The tensor-product quadrature for the levels $\ell = (\ell_1,\dots,\ell_d)$ is

$$\begin{aligned} Q_\ell^d &= Q_{\ell_1}^1 \otimes \dots \otimes Q_{\ell_d}^1 \\ &= \sum_{j_1=1}^{N_1} \dots \sum_{j_d=1}^{N_d} w_{j_1}^1 \dots w_{j_d}^d f(c_{j_1}^1,\dots,c_{j_d}^d) \\ &= \sum_{j=1}^{d} \sum_{1\le k_j\le \ell_j} \left(\Delta_{k_1}^1 \otimes \dots \otimes \Delta_{k_d}^1\right) f. \end{aligned}$$

In the case of an isotropic grid $\ell = (\ell,\dots,\ell)$ we denote

$$Q_\ell^d = \sum_{j=1}^{d} \sum_{1\le k_j\le \ell} \left(\Delta_{k_1}^1 \otimes \dots \otimes \Delta_{k_d}^1\right) f = \sum_{|\mathbf{k}|_\infty \le \ell} \left(\Delta_{k_1}^1 \otimes \dots \otimes \Delta_{k_d}^1\right) f.$$

Idea: In the case that f is a smooth function, many of the details $\left(\Delta_{k_1}^1 \otimes \dots \otimes \Delta_{k_d}^1\right) f$ are so small that they may be neglected.

The classical sparse grid quadrature (Smolyak) rule is defined by

$$S_\ell^d f := \sum_{|\mathbf{k}|_1 \le \ell+d-1} \left(\Delta_{k_1}^1 \otimes \dots \otimes \Delta_{k_d}^1\right) f.$$

One can prove the combinations formula:

$$S_\ell^d f = \sum_{\ell \le |\mathbf{k}|_1 \le \ell+d-1} (-1)^{\ell+d-|\mathbf{k}|_1-1} \binom{d-1}{|\mathbf{k}|_1-\ell} \left(Q_{k_1}^1 \otimes \dots \otimes Q_{k_d}^1\right) f,$$

Num. Meth. Phys.
Gradinaru D-MATH
7.4 p. 410

which is used in the practical parallel implementations.

The sparse grid is then the grid formed by the reunion of the anisotropic full grids used in the combinations formula. Its cardinality is $N = O(2^\ell \ell^{d-1})$ which is much less than $O(2^{d\ell})$ of the full grid. The error of the classical sparse grid quadrature is $O\left(N^{-r}\log^{(d-1)(r+1)}(N)\right)$ for f of a certain smoothness r.

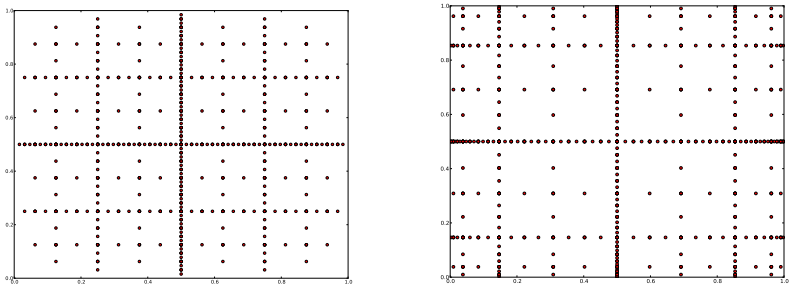


Figure 7.1: Sparse Grids based on midpoint rule and on open Clenshaw-Curtis rule

The Clenshaw-Curtis rule on a full grid in $d = 3$ dimensions at level $n = 6$ needs 274625 function evaluations and executes in about 6.6 seconds on my laptop to run. The same quadrature rule on the sparse grid of the same level $n = 6$ needs only 3120 function evaluations and executes in about 0.1 seconds. The error in the full grid case is $1.5 \cdot 10^{-5}$ and in the sparse grid case $1.5 \cdot 10^{-6}$ for the function $f(\mathbf{x}) = (1 + 1/d)^d (x_1 \cdot \ldots \cdot x_d)^{1/2}$.

The same example in $d = 4$ dimensions requires in the full grid case about 17.8 millions function evaluations with error $3.3 \cdot 10^{-5}$ and runs in 426 seconds, while the sparse gird algorithms needs 9065 function evaluations and runs in 0.3 seconds with error $8 \cdot 10^{-6}$.

7.5 Monte-Carlo Quadrature

Monte-Carlo integration: instead of step-functions as quadrature,

$$I = \int_0^1 f(t)dt \approx h \sum_{i=1}^{N} f(t_i)$$

where $t_i = (i - \frac{1}{2})h$, $h = \frac{1}{N}$, the $\{f(t_i)\}$ may be reordered in any way. In particular, we can order them randomly:

$$I = \int_0^1 f(t)dt \approx \frac{1}{N} \sum_{i=1}^{N} f(t_i)$$

where $t_i \in (0, 1)$ are uniformly distributed and sampled from a **random number generator**. A little more generally,

$$I = \int_a^b f(t)dt = |b - a|\langle f\rangle \approx |b - a|\frac{1}{N}\sum_{i=1}^{N} f(t_i)$$

where $t_i = a + (b - a) \cdot RNG$ (`rand` e.g.).

Each Monte-Carlo Method needs:

- a domain for the "experiment": here $[0, 1]^d$
- generated random numbers : here t_i
- a deterministic computation: here $|b - a|\frac{1}{N}\sum_{i=1}^{N} f(t_i)$
- a representation of the result: here $P(I \in [I_N - \sigma_N, I_N + \sigma_N]]) = 0.683$

Random variables and statistics are not subject of this lecture.

Essential: good RNG: fullfill statistical tests and is deterministic (reproductible).

Uniform RNG: Mersene-Twister (actual in python and Matlab), better is Marsaglia (C-MWC), best is WELL (2006); SPRNG (Masagni) is especially suited to large-scale parallel Monte Carlo applications. Normal RNG: Box-Muller improved by Marsaglia, better is Ziggurat-method of Marsaglia (1998)

Note: methods based on the inversion of the distribution function resides offen on badly-conditioned zero solvers and hence have to be avoided.

What is really important is the statistical error, in d-dimensions,

$$\text{error} = \frac{k_d}{\sqrt{N}}.$$

The constant $k_d = \sqrt{\text{variance}}$.

Note the different meaning of this error: it is now of a probabilistic nature: 68.3% of the time, the estimate is within one standard deviation of the correct answer.

The method is very general. For example, $A \subset R^d$,

$$\int_A f(\mathbf{x})dx_1dx_2\cdots d\mathbf{x}_d \approx |A|\langle f\rangle$$

where $|A|$ is the volume of region A.

The error is always $\propto N^{-1/2}$ independetly on the dimension d !

But k_d can be reduced significantly. Two such methods: antithetic variates & control variates. An example,

$$I_0(x) = \frac{1}{\pi}\int_0^{\pi} e^{-x\cos t}dt\ .$$

$I_0(x)$ is a modified Bessel function of order zero.

Code 7.5.1: Plain Monte-Carlo

```
"""
 Computes integral
 I0(1) = (1/pi) int(z=0..pi)  exp(-cos(z)) dz by raw MC.
 Abramowitz and Stegun give I0(1) = 1.266066
"""

import numpy as np
import time

t1 = time.time()

M = 100 # number of times we run our MC inetgration
asval = 1.266065878
ex = np.zeros(M)
print 'A and S tables:  I0(1) = ',asval
print 'sample       variance        MC I0 val'
print '------------------------------------------------------------'
k = 5 # how many experiments
N = 10**np.arange(1,k+1)
v = []; e = []
for n in N:
    for m in xrange(M):
        x = np.random.rand(n) # sample
        x = np.exp(np.cos(-np.pi*x))
        ex[m] = sum(x)/n # quadrature
    ev = sum(ex)/M
    vex = np.dot(ex,ex)/M
    vex -= ev**2
    v += [vex]; e += [ev]
    print n, vex, ev

t2 = time.time()
t = t2-t1

print "Serial calculation completed, time = %s s" % t
```

➢ General Principle of Monte Carlo: If, at any point of a Monte Carlo calculation, we can replace an estimate by an exact value, we shall replace an estimate by an exact value, we shall reduce the sampling error in the final result.

➢ Mark Kac: "You use Monte Carlo until you understand the problem."

Antithetic variates: usually only 1-D. Estimator for

$$I = I_a + I_b$$

where

$$I_a \approx \theta_a = \frac{1}{N}\sum_{i=1}^{N} f^{[a]}(x_i) \quad \text{and} \quad I_b \approx \theta_b = \frac{1}{N}\sum_{i=1}^{N} f^{[b]}(x_i)$$

so the variance is

$$\begin{aligned}\mathsf{Var}_{ab} &= \langle(\theta_a+\theta_b-I)^2\rangle \\ &= \langle(\theta_a-I_a)^2\rangle + \langle(\theta_b-I_b)^2\rangle + 2\langle(\theta_a-I_a)(\theta_b-I_b)\rangle \\ &= \mathsf{Var}_a + \mathsf{Var}_b + 2\mathsf{Cov}_{ab}\end{aligned}$$

If $\mathsf{Cov}_{ab} = \langle(\theta_a-I_a)(\theta_b-I_b)\rangle < 0$ (negatively correlated), Var_{ab} is reduced.

Our example: break the integral into two pieces $0 < x < \pi/2$ and $x + \pi/2$. The new integrand is $e^{\sin(x)} + e^{-\cos(x)}$, for $0 < x < \pi/2$, and strictly **monotone**.

$$\begin{aligned}I_0(1) &\approx I_+ + I_- \\ &= \frac{1}{4N}\sum_{i=1}^{N} e^{\sin \pi u_i/2} + e^{\sin \pi(1-u_i)/2} \\ &\quad + e^{-\cos \pi u_i/2} + e^{-\cos \pi(1-u_i)/2}.\end{aligned}$$

Code 7.5.2: Antitetic Variates Monte-Carlo

```
"""
 Computes integral
 I0(1) = (1/pi) int(z=0..pi)  exp(-cos(z)) dz
 by  antithetic variates.
 We split the range into 0 < x < pi/2 and pi/2<x<pi.
 The resulting integrand is
 exp(sin(x)) + exp(-cos(x)), which is monotone
 increasing in 0 < x < pi/2, so antithetic variates
 can be used.
 Abramowitz and Stegun give I0(1) = 1.266066
"""

import numpy as np

M = 100 # number of times we run our MC inetgration
asval = 1.266065878
ex = np.zeros(M)
print 'A and S tables:  I0(1) = ',asval
print 'sample       variance        MC I0 val'
print '------------------------------------------------------------'
k = 5 # how many experiments
N = 10**np.arange(1,k+1)
v = []; e = []
```

```
pi2 = 0.5*np.pi
for n in N:
    for m in xrange(M):
        up = pi2*np.random.rand(n)  # sample
        dn = pi2-up
        x = np.exp(np.sin(up))  + np.exp(np.sin(dn))
        x += np.exp(np.cos(-up))  + np.exp(-np.cos(dn))
        ex[m] = 0.25*sum(x)/n  # quadrature
    ev = sum(ex)/M
    vex = np.dot(ex,ex)/M
    vex -=  ev**2
    v += [vex]; e += [ev]
    print n, vex, ev
```


Control variates Integral

$$I = \int_0^1 f(t)dt$$

can be re-written

$$I = \int_0^1 (f(t) - \phi(t))dt + \int_0^1 \phi(t)dt$$

$$\approx \frac{1}{N}\sum_{i=1}^{N}[f(t_i) - \phi(t_i)] + I_\phi$$

Pick ϕ:

- $\phi(u) \approx g(u)$ is *nearby*, and
- $I_\phi = \int \phi(u)du$ is known.

Variance is reduced if

$$\text{var}(f - \phi) \ll \text{var}(f)$$

To see how it works, our problem

$$\begin{aligned} f(t) &= e^{-cos(\pi t)} \\ &= 1 - cos(\pi t) + \frac{1}{2}(cos(\pi t))^2 + \ldots \\ \phi(t) &= \text{first three terms} \end{aligned}$$

Code 7.5.3: Control Variates Monte-Carlo

```
"""
 Computes integral
 I0(1) = (1/pi) int(z=0..pi)  exp(-cos(z)) dz
 by  control variate. The splitting is
 < exp(-cos) >  =  < exp(-cos) - phi > + int(0..pi) phi dz
 where phi = 1 + cos - (1/2)*cos*cos is the control.
 The exact integral of the control
 int(z=0..pi) (1 + cos - (1/2)*cos*cos) dz = 1.25.
 Abramowitz and Stegun give I0(1) = 1.266066
"""

import numpy as np

M = 100  # number of times we run our MC inetgration
asval = 1.266065878
ex = np.zeros(M)
print 'A and S tables:  I0(1) = ',asval
print 'sample       variance        MC I0 val'
print '------       --------        ---------'
k = 5  # how many experiments
N = 10**np.arange(1,k+1)
v = []; e = []
for n in N:
    for m in xrange(M):
        x = np.pi*np.random.rand(n)  # sample
        ctv = np.exp(-np.cos(x)) - 1. + np.cos(x) -
          0.5*np.cos(x)*np.cos(x)
        ex[m] = 1.25 + sum(ctv)/n  # quadrature
    ev = sum(ex)/M
    vex = np.dot(ex,ex)/M
    vex -=  ev**2
    v += [vex]; e += [ev]
    print n, vex, ev
```


Importance Sampling

Idea: Concentrate the distribution of the sample points in the parts of the interval that are of most importance instead of spreading them out evenly.

Importance Sampling:

$$\theta = \int_0^1 f(x)\,dx = \int_0^1 \frac{f(x)}{g(x)}g(x)\,dx = \int_0^1 \frac{f(x)}{g(x)}dG(x),$$

where g and G satisfy

$$G(x) = \int_0^x g(y)\,dy, \quad G(1) = \int_0^1 g(y)\,dy = 1,$$

and $G(x)$ is a distribution function. Variance

$$\sigma^2_{f/g} = \int_0^1 (f(x)/g(x) - \theta)^2 dG(x)$$

How to select a good sampling function?

How about $g = cf$? g must be simple enough for us to know its integral theoretically.

Example 7.5.4. $\int_0^1 f(x)dx$ with $f(x) = 1/\sqrt{x(1-x)}$ has singularities at $x = 0, 1$. General trick: isolate them!

$$g(x) = \frac{1}{4\sqrt{x}} + \frac{1}{4\sqrt{1-x}} \quad \Rightarrow \quad \int_0^1 h(x)dG(x)$$

with

$$h(x) = \frac{4}{\sqrt{x} + \sqrt{1-x}}$$

and $dG(x)$ will be sampled as

```
u = rand(N)
v = rand(N)
x = u*u
w = where(v>0.5)
x[w] = 1 - x[w]
```

Code 7.5.5: Importance Sampling Monte-Carlo

```
from scipy import where, sqrt, arange, array
from numpy.random import rand
from time import time

from scipy.integrate import quad
f = lambda x: 1/sqrt(x*(1-x))
print 'quad(f,0,1) = ', quad(f,0,1)

func = lambda x: 4./(sqrt(x)+sqrt(1-x))

def exotic(N):
    u = rand(N)
    v = rand(N)
    x = u*u
    w = where(v>0.5)
    x[w] = 1 - x[w]
    return x

def ismcquad():
    k = 5 # how many experiments
    N = 10**arange(1,k+1)
    ex = []
    for n in N:
        x = exotic(n) # sample
        x = func(x)
        ex += [x.sum()/n] # quadrature
    return ex

def mcquad():
    k = 5 # how many experiments
    N = 10**arange(1,k+1)
    ex = []
    for n in N:
        x = rand(n) # sample
        x = f(x)
        ex += [x.sum()/n] # quadrature
    return ex

M = 100 # number of times we run our MC inetgration

t1 = time()
results = []
for m in xrange(M):
    results += [ismcquad()]

t2 = time()
t = t2-t1
```



```

print "ISMC Serial calculation completed, time = %s s" % t

ex = array(results)
ev = ex.sum(axis=0)/M
vex = (ex**2).sum(axis=0)/M
vex = vex - ev**2
print ev[-1]
print vex

t1 = time()
results = []
for m in xrange(M):
    results += [mcquad()]

t2 = time()
t = t2-t1

print "MC Serial calculation completed, time = %s s" % t

ex = array(results)
ev = ex.sum(axis=0)/M
vex = (ex**2).sum(axis=0)/M
vex = vex - ev**2
```

```
print ev[-1]
print vex
```

. Compare with the analytical computed value.

7.6 Essential Skills Learned in Chapter 7

You should know:

- several (compozite) polynomial quadrature formulas with their convergence order
- what is special about the trapezoidal rule
- Gaussian quadrature rules
- how to compute a high-dimensional integral
- particularities of Monte-Carlo integration
- how to reduce the variance in Monte-Carlo integration

x

Part III

Integration of Ordinary Differential Equations

8 Single Step Methods

8.1 Initial value problems (IVP) for ODEs

Some grasp of the meaning and theory of ordinary differential equations (ODEs) is indispensable for understanding the construction and properties of numerical methods. Relevant information can be found in [52, Sect. 5.6, 5.7, 6.5].

Example 8.1.1 (Growth with limited resources). [1, Sect. 1.1]

$y : [0,T] \mapsto \mathbb{R}$: bacterial population density as a function of time

Model: autonomous logistic differential equations

$$\dot{y} = f(y) := (\alpha - \beta y)\, y \tag{8.1.1}$$

✎ Notation (Newton): dot $\dot{}$ $\hat{=}$ (total) derivative with respect to time t

- $y \hat{=}$ population density, $[y] = \frac{1}{\mathrm{m}^2}$
- growth rate $\alpha - \beta y$ with growth coefficients $\alpha, \beta > 0$, $[\alpha] = \frac{1}{\mathrm{s}}$, $[\beta] = \frac{\mathrm{m}^2}{\mathrm{s}}$: decreases due to more fierce competition as population density increases.

Note: we can only compute a solution of (8.1.3), when provided with an initial value $y(0)$.

The logisitc differential equation arises in autocatalytic reactions (as in haloform reaction, tin pest, binding of oxygen by hemoglobin or the spontaneous degradation of aspirin into salicylic acid and acetic acid, causing very old aspirin in sealed containers to smell mildly of vinegar):

$$A + B \longrightarrow 2B \quad \text{with rate } r = k c_A c_B \tag{8.1.2}$$

As $\dot{c}_A = -r$ and $\dot{c}_B = -r + 2r = r$ we have that $c_A + c_B = c_A(0) + c_B(0) = D$ is constant and we get two decoupled equations

$$\dot{c}_A = -k(D - c_A)c_A \tag{8.1.3}$$