Lecture 6: Numerical Integration

Numerical Integration

Numerical integration (数值积分) simply calculates the area under a given curve. The basic ideas for performing such an operation come from the definition of integration

$$\int_a^b f(x)dx = \lim_{h\to 0}\sum_{j=0}^{N} f(x_j)\,h$$

where $b-a = Nh$. Thus the area under the curve, from the calculus standpoint (微积分的角度), is thought of as a limiting process (极限法) of summing up an ever-increasing number of rectangles (矩形). This process is known as **numerical quadrature (数值积分法)**. Specifically, any sum can be represented as follows

$$Q[f] = \sum_{j=0}^{N} w_k f(x_k) = w_0 f(x_0) + w_1 f(x_1) + \cdots + w_N f(x_N)$$

where $a = x_0 < x_1 < x_2 < \cdots < x_N = b$.

Figure 4.3 gives the standard representation of the integration (积分) process and the division of the integration interval into a finite set of integration intervals.

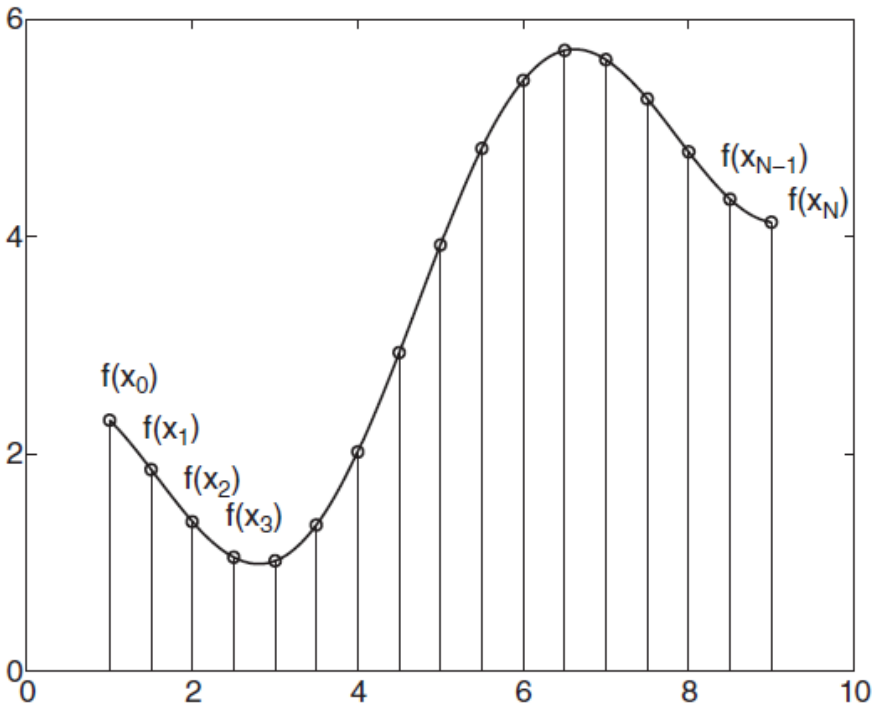


Figure 4.3: Graphical representation of the integration process. The integration interval is broken up into a finite set of points. A quadrature rule then determines how to sum up the area of a finite number of rectangles.

Thus the integral is evaluated as

$$\int_a^b f(x)dx = Q[f] + E[f]$$

where the term *E[f]* is the error in approximating the integral (积分) by the quadrature (求积法) sum.

Typically, the error *E[f]* is due to truncation error (截断误差). To integrate, use will be made of polynomial fits (多项式拟合) to the y-values $f(x_j)$. Thus we assume the function *f(x)* **can be approximated by a polynomial**

$$P_n(x) = a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0$$

where the truncation error in this case is proportional (成正比) to the $(n+1)_{th}$ **derivative** $E[f] = Af^{(n+1)}(c)$ **and** *A* **is a constant**. **This process of polynomial fitting the data gives the Newton-Cotes Formulas (牛顿-柯特斯公式)**.

Newton-Cotes Formulas(牛顿-柯特斯公式)

The following integration approximations result from using a polynomial fit through the data to be differentiated. It is assumed that

$$x_k = x_0 + hk \quad f_k = f(x_k)$$

This gives the following integration algorithms:

Trapezoid Rule $\int_{x_0}^{x_1} f(x)dx = \frac{h}{2}(f_0 + f_1) - \frac{h^3}{12}f''(c)$

Simpson's Rule $\int_{x_0}^{x_2} f(x)dx = \frac{h}{3}(f_0 + 4f_1 + f_2) - \frac{h^5}{90}f''''(c)$

Simpson's 3/8 Rule $\int_{x_0}^{x_3} f(x)dx = \frac{3h}{8}(f_0 + 3f_1 + 3f_2 + f_3) - \frac{3h^5}{80}f''''(c)$

Boole's Rule $\int_{x_0}^{x_4} f(x)dx = \frac{2h}{45}(7f_0 + 32f_1 + 12f_2 + 32f_3 + 7f_4) - \frac{8h^7}{945}f^{(6)}(c)$

https://baike.baidu.com/item/%E6%A2%AF%E5%BD%A2%E6%B3%95%E5%88%99

https://zhuanlan.zhihu.com/p/159058519

- These algorithms have varying degrees of accuracy (精度不同). Specifically, they are $O(h^2)$, $O(h^4)$, $O(h^4)$ and $O(h^6)$ accurate schemes respectively. **The accuracy condition is determined from the truncation terms of the polynomial fit**.
- Note that the **Trapezoid rule** uses a sum of simple trapezoids to approximate the integral.
- **Simpson's rule** fits a quadratic curve (二次曲线) through **three points** and calculates the area under the quadratic curve.
- **Simpson's 3/8 rule** uses **four points** and **a cubic polynomial (三次多项式)** to evaluate the area,
- while **Boole's rule** uses **five points** and a **quintic polynomial (五次多项式)** fit to generate an evaluation of the integral.

Figure 4.4 represents graphically the different quadrature rules(求积规则) and its approximation (dotted line) to the actual function.

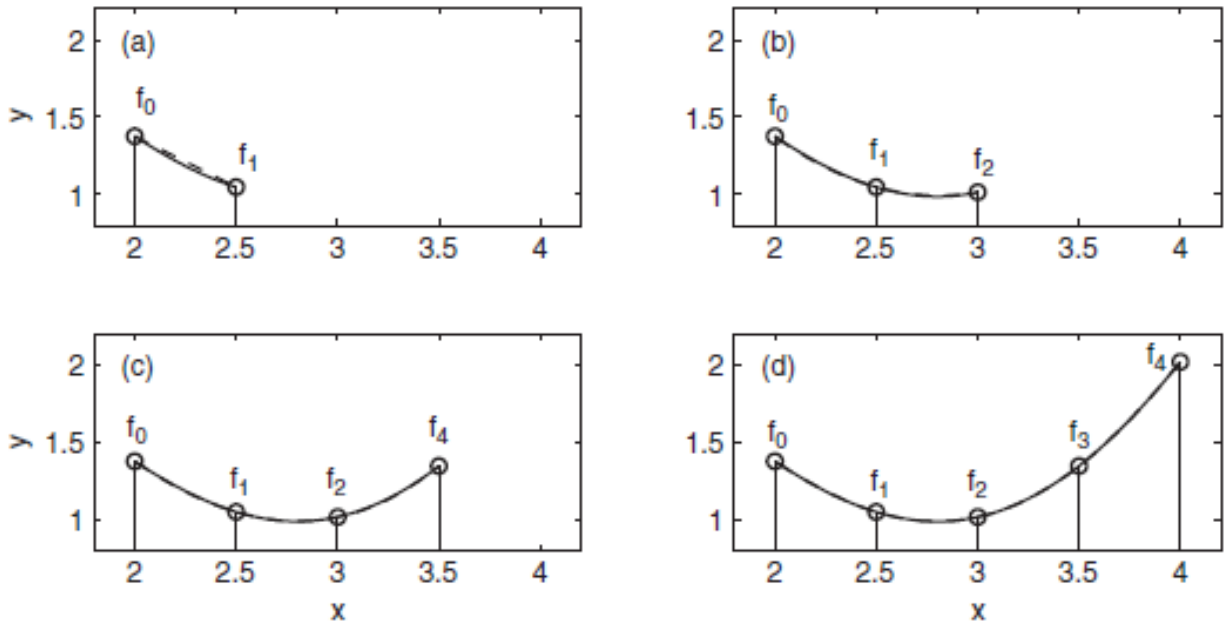


Figure 4.4: Graphical representation of the integration process for the quadrature rules in (4.2.6). The quadrature rules represented are (a) trapezoid rule, (b) Simpson's rule, (c) Simpson's 3/8 rule and (d) Boole's rule. In addition to the specific function considered, the dotted line represents the polynomial fit to the actual function for the four different integration rules. Note that the dotted line and function line are essentially identical for (c) and (d).

The derivation of these integration rules follows from simple polynomial fits through a specified number of data points.

To derive the Simpson's rule, consider a second degree polynomial (拉格朗日多项式，二次多项式) through the three points (x_0, f_0), (x_1, f_1), and (x_2, f_2):

$$p_2(x) = f_0 \frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} + f_1 \frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)} + f_2 \frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)}.$$

This **quadratic fit is derived by using Lagrange coefficients (乘子)**. The truncation error could also be included, but we neglect it for the present purposes. By plugging in (4.2.7) into the integral

$$\int_{x_0}^{x_2} f(x)dx \approx \int_{x_0}^{x_2} p_2(x)dx = \frac{h}{3}\left(f_0 + 4f_1 + f_2\right).$$

拉格朗日（插值）多项式

如对实践中的某个物理量进行观测，在若干个不同的地方得到相应的观测值，拉格朗日插值法可以找到一个多项式，其恰好在各个观测的点取到观测到的值。这样的多项式称为拉格朗日（插值）多项式。

概念

一般地，若已知$y=f(x)$在互不相同 n+1 个点$x_0,x_1,\cdots,x_n$处的函数值$y_0,y_1,\cdots,y_n$（即该函数过$(x_0,y_0),(x_1,y_1),\cdots,(x_n,y_n)$这n+1个点），则可以考虑构造一个过这n+1个点的、次数不超过n的多项式$y=P_n(x)$，使其满足：

$$P_n(x_k)=y_k,k=0,1,\cdots,n \quad (*)$$

要估计任一点ξ，ξ≠x_i,i=0,1,2,...,n,则可以用P_n（ξ）的值作为准确值f(ξ)的近似值，此方法叫做"插值法"。

称式（*）为插值条件（准则），含x_i(i=0,1,...,n)的最小区间[a,b]，其中a=min{x_0,x_1,...,x_n}，b=max{x_0,x_1,...,x_n}。

定理

满足插值条件的、次数不超过n的多项式是存在而且是唯一的。

在平面上有$(x_0,y_0),(x_1,y_1),\cdots,(x_{n-1},y_{n-1})$共n个点，现作一条函数$f(x)$使其图像经过这n个点。

作法：设集合D_n是关于点(x,y)的角标的集合，$D_n=\{0,1,\ldots,n-1\}$，作n个多项式$p_j(x),j\in D_n$。对于任意$k\in D_n$，都有$p_k(x),B_k=\{i|i\neq k,i\in D_n\}$

使得

$$p_k(x)=\prod_{i\in B_k}\frac{x-x_i}{x_k-x_i}$$

$p_k(x)$是n-1次多项式，且满足$\forall m\in B_k,p_k(x_m)=0$并且$p_k(x_k)=1$。

最后可得$L_n(x)=\sum_{j=0}^{n-1}y_jp_j(x)$。

形如上式的插值多项式$L_n(x)$称为拉格朗日（Lagrange）插值多项式。

例如：当n=4时，上面的公式可简化为：

$$f(x)=\frac{(x-x_1)(x-x_2)(x-x_3)}{(x_0-x_1)(x_0-x_2)(x_0-x_3)}y_0+\frac{(x-x_0)(x-x_2)(x-x_3)}{(x_1-x_0)(x_1-x_2)(x_1-x_3)}y_1+\frac{(x-x_0)(x-x_1)(x-x_3)}{(x_2-x_0)(x_2-x_1)(x_2-x_3)}y_2+\frac{(x-x_0)(x-x_1)(x-x_2)}{(x_3-x_0)(x_3-x_1)(x_3-x_2)}y_3$$

定理 7.1 的证明 从基于 $x_0, x_1, \cdots, x_M$ 的 $f(x)$ 的拉格朗日逼近多项式 $P_M(x)$ 开始。用它来逼近

$$f(x) \approx P_M(x) = \sum_{k=0}^{M} f_k L_{M,k}(x) \tag{12}$$

其中 $f_k = f(x_k), k = 0, 1, \cdots, M$。积分的一种逼近方法是用多项式 $P_M(x)$ 来代替被积函数 $f(x)$，这是得到牛顿－科特斯公式的一般方法：

$$\begin{aligned}\int_{x_0}^{x_M} f(x)\,dx &\approx \int_{x_0}^{x_M} P_M(x)\,dx \\ &= \int_{x_0}^{x_M} \left(\sum_{k=0}^{M} f_k L_{M,k}(x)\right) dx = \sum_{k=0}^{M} \left(\int_{x_0}^{x_M} f_k L_{M,k}(x)\,dx\right) \\ &= \sum_{k=0}^{M} \left(\int_{x_0}^{x_M} L_{M,k}(x)\,dx\right) f_k = \sum_{k=0}^{M} w_k f_k\end{aligned} \tag{13}$$

式(13)中系数 w_k 的计算细节是烦琐的。以下给出辛普森公式的一个例证，其中 $M = 2$，这种情况下用到了逼近多项式

$$P_2(x) = f_0\frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)} + f_1\frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)} + f_2\frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)} \quad (14)$$

由于 f_0, f_1 和 f_2 对于积分运算而言是常数，因此由式(13)得到

$$\begin{aligned}\int_{x_0}^{x_2} f(x)\,dx \approx f_0\int_{x_0}^{x_2}\frac{(x-x_1)(x-x_2)}{(x_0-x_1)(x_0-x_2)}dx + f_1\int_{x_0}^{x_2}\frac{(x-x_0)(x-x_2)}{(x_1-x_0)(x_1-x_2)}dx \\ + f_2\int_{x_0}^{x_2}\frac{(x-x_0)(x-x_1)}{(x_2-x_0)(x_2-x_1)}dx\end{aligned} \quad (15)$$

引入变量代换，用 $dx = h\,dt$ 替换 $x = x_0 + ht$。新的积分限从 $t=0$ 到 $t=2$。等距节点 $x_k = x_0 + kh$ 变为 $x_k - x_j = (k-j)h$ 和 $x - x_k = h(t-k)$，这可使式(15)简化为

$$\begin{aligned}\int_{x_0}^{x_2} f(x)\,dx &\approx f_0\int_0^2\frac{h(t-1)h(t-2)}{(-h)(-2h)}h\,dt + f_1\int_0^2\frac{h(t-0)h(t-2)}{(h)(-h)}h\,dt \\ &\quad + f_2\int_0^2\frac{h(t-0)h(t-1)}{(2h)(h)}h\,dt \\ &= f_0\frac{h}{2}\int_0^2(t^2-3t+2)\,dt - f_1h\int_0^2(t^2-2t)\,dt + f_2\frac{h}{2}\int_0^2(t^2-t)\,dt \\ &= f_0\frac{h}{2}\left(\frac{t^3}{3}-\frac{3t^2}{2}+2t\right)\Big|_{t=0}^{t=2} - f_1h\left(\frac{t^3}{3}-t^2\right)\Big|_{t=0}^{t=2} \\ &\quad + f_2\frac{h}{2}\left(\frac{t^3}{3}-\frac{t^2}{2}\right)\Big|_{t=0}^{t=2} \\ &= f_0\frac{h}{2}\left(\frac{2}{3}\right) - f_1h\left(\frac{-4}{3}\right) + f_2\frac{h}{2}\left(\frac{2}{3}\right) \\ &= \frac{h}{3}(f_0+4f_1+f_2)\end{aligned} \quad (16)$$

证明完毕。7.2 节将给出推论 7.1 的一个例证。 ●

5.1.3Newton-Cotes公式

将 $[a,b]$ 等分为 n 个区间，使用 $n+1$ 个点.

其分点为

$$x_i = a + ih, \quad i = 0, 1, 2, \cdots, n, \quad h = \frac{b-a}{n}$$

插值多项式为

$$P_n(x) = \sum_{i=0}^{n} \frac{\omega(x)}{(x - x_i)\,\omega'(x_i)} f(x_i)$$

用插值多项式代替被积函数得到

$$\begin{aligned}\int_a^b f(x)\mathrm{d}x \approx \int_a^b P_n(x)\mathrm{d}x &= \int_a^b \left(\sum_{i=0}^{n} \frac{\omega(x)}{(x - x_i)\,\omega'(x_i)} f(x_i) \right) \mathrm{d}x \\ &= \sum_{i=0}^{n} \left(\int_a^b \frac{\omega(x)}{(x - x_i)\,\omega'(x_i)} \mathrm{d}x \right) f(x_i) = \sum_{i=0}^{n} A_i f(x_i)\end{aligned}$$

这里的 $A_i = \int_a^b \frac{\omega(x)}{(x-x_i)\omega'(x_i)}\mathrm{d}x$ ，它是拉格朗日插值的基函数在区间 $[a,b]$ 上的积分，作变量替换 $x = a + th$ ，这个积分等于

$$A_i = (b-a)c_i^{(n)} = \frac{(-1)^{n-i}}{n \cdot (i!) \cdot (n-i)!} \int_0^n \frac{t(t-1)\cdots(t-n)}{t-i}\mathrm{d}t$$

这里 $c_i^{(n)}$ 不依赖函数 $f(x)$ 和区间 $[a,b]$ ，可以**事先计算出来**。称这些数为Newton-Cotes系数.

5.2.2梯形公式误差估计

定理 梯形公式的误差为

$$R_T(f) = \int_a^b f(x)dx - \frac{b-a}{2}[f(a)+f(b)]$$
$$= -\frac{(b-a)^3}{12}f''(\eta), \quad a \le \eta \le b$$

证明 首先利用插值多项式误差估计式

$$R_T(f) = \int_a^b [f(x) - p_1(x)]\,dx$$
$$= \int_a^b \frac{1}{2!}f''(\xi)(x-a)(x-b)dx$$

$f''(\xi)$ 是关于 x 的函数，它是连续的，并且 $(x-a)(x-b)$ 非正，利用积分中值定理得到

$$R_T(f) = \frac{1}{2!}f''(\eta)\int_a^b (x-a)(x-b)dx$$
$$= -\frac{(b-a)^3}{12}f''(\eta)$$

- The integral calculation is easily performed since it only involves integrating powers of x^2 or less. Evaluating at the limits (在极限处求值) then causes many terms to cancel and drop out.
- Thus the Simpson's rule is recovered. The trapezoid rule, Simpson's 3/8 rule, and Boole's rule are all derived in a similar fashion.
- To make connection with the quadrature rule (求积规则) (4.2.2), $Q = w_0f_0+w_1f_1+w_2f_2$, Simpson's rule gives $w_0 = h/3$, $w_1 = 4h/3$, and $w_2 = h/3$ as weighting factors.

Composite Rules (可分解法则)

The integration methods (4.2.6) give values for the integrals over only a small part of the integration domain. Trapezoid rule, for instance, only gives a value for $x \in [x_0, x_1]$. However, our fundamental aim is to evaluate the integral over the entire domain $x \in [a, b]$. Assuming once again that our interval is divided as $a = x_0 < x_1 < x_2 < \cdots < x_N = b$, then the trapezoid rule applied over the interval gives the total integral

$$\int_a^b f(x)dx \approx Q[f] = \sum_{j=1}^{N-1} \frac{h}{2}(f_j + f_{j+1})$$

Writing out this sum gives

$$\sum_{j=1}^{N-1} \frac{h}{2}(f_j + f_{j+1}) = \frac{h}{2}(f_0 + f_1) + \frac{h}{2}(f_1 + f_2) + \cdots + \frac{h}{2}(f_{N-1} + f_N)$$

$$= \frac{h}{2}(f_0 + 2f_1 + 2f_2 + \cdots + 2f_{N-1} + f_N)$$

$$= \frac{h}{2}\left(f_0 + f_N + 2\sum_{j=1}^{N-1} f_j\right)$$

The final expression no longer double counts the values of the points between f_0 and f_N. Instead, the final sum only counts the intermediate values once, thus making the algorithm about twice as fast as the previous sum expression. These are computational savings which should always be exploited if possible.

Recursive Improvement of Accuracy(精度递归改进)

Given an integration procedure and a value of *h*, a function or data set can be integrated to a prescribed (规定的) accuracy.

However, it may be desirable to improve the accuracy without having to disregard previous approximations to the integral.

To see how this might work, consider the trapezoidal rule for **a step size of *2h***.

Thus, the even data points are the only ones of interest and we have the basic one-step integral

$$\int_{x_0}^{x_2} f(x)dx = \frac{2h}{2}\left(f_0 + f_2\right) = h\left(f_0 + f_2\right)$$

The composite rule associated with this is then

$$\int_a^b f(x)dx \approx Q[f] = \sum_{j=0}^{N/2-1} h\left(f_{2j} + f_{2j+2}\right)$$

Writing out this sum gives

$$\sum_{j=0}^{N/2-1} h\left(f_{2j}+f_{2j+2}\right) = h\left(f_0+f_2\right)+h\left(f_2+f_4\right)+\cdots+h\left(f_{N-2}+f_N\right)$$

$$= h\left(f_0+2f_2+2f_4+\cdots+2f_{N-2}+f_N\right)$$

$$= h\left(f_0+f_N+2\sum_{j=1}^{N/2-1} f_{2j}\right)$$

Comparing the middle expressions in (4.2.10) and (4.2.13) gives a great deal of insight into how recursive schemes for improving accuracy work.

Specifically, we note that the more accurate scheme with step size *h* contains all the terms in the integral approximation (积分近似值) using step size *2h*. Quantifying this gives

$$Q_h = \frac{1}{2}Q_{2h} + h\left(f_1 + f_3 + \cdots + f_{N-1}\right)$$

where Q_h and Q_{2h} are the quadrature approximations to the integral with step size *h* and *2h* respectively.

This then allows us to half the value of *h* and improve accuracy without jettisoning (抛弃) the work required to approximate the solution with the accuracy given by a step size of *2h*. This recursive procedure can be continued so as to give higher accuracy results. Further, this type of procedure holds for Simpson's rule as well as any of the integration schemes developed here.

This recursive routine is used in MATLAB in order to generate results to a prescribed (预先设定的) accuracy.

Implementation of Differentiation and Integration

This lecture focuses on the **implementation of differentiation and integration methods**. Since they form the backbone of calculus (微积分的骨干框架), accurate methods to approximate these calculations are essential.

To begin, we consider a specific function to be differentiated (可微分的), namely a hyperbolic secant (双曲正割). Part of the reason for considering this function is that the exact value of its derivative (导数) is known.

This allows us to make a comparison of our approximate methods with the actual solution. Thus, we consider

$$u = \operatorname{sech}(x)$$

whose derivative is

$$\frac{du}{dx} = -\operatorname{sech}(x)\tanh(x)$$
$$\frac{d^2u}{dx^2} = \operatorname{sech}(x) - \operatorname{sech}^3(x).$$

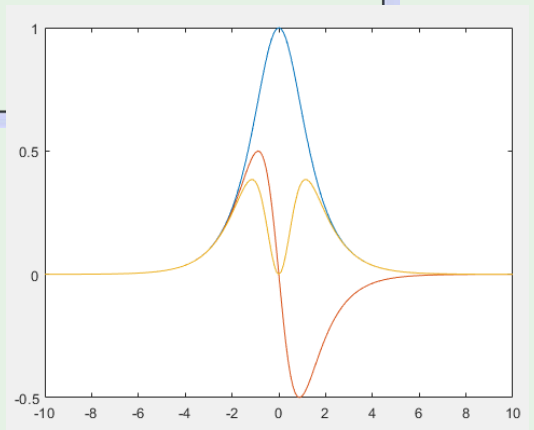
Differentiation

To begin the calculations, we define a spatial interval. For this example we take the interval $x \in [-10, 10]$. In MATLAB, the spatial discritization, *dx*, must also be defined. This gives

```
dx=0.1; %spatial discritization
x=-10:dx:10; %spatial domain
```

Once the spatial domain has been defined, the function to be differentiated must be evaluated

```
u=sech(x);
ux_exact=-sech(x).*tanh(x);
uxx_exact=sech(x)-sech(x).^3;
figure(1), plot(x,u,x,ux_exact,x,uxx_exact)
```



MATLAB figure 1 produces the function and its derivative.

To calculate the derivative numerically, we use the center-, forward-, and backward-difference formulas derived for differentiation. Specifically, we will make use of the following four first-derivative approximations:

$$\text{center - difference } O\left(h^2\right): \frac{y_{n+1}-y_{n-1}}{2h}$$

$$\text{center - difference } O\left(h^4\right): \frac{-y_{n+2}+8y_{n+1}-8y_{n-1}+y_{n-2}}{12h}$$

$$\text{forward - difference } O\left(h^2\right): \frac{-3y_n+4y_{n+1}-y_{n+2}}{2h}$$

$$\text{backward - difference } O\left(h^2\right): \frac{3y_n-4y_{n-1}+y_{n-2}}{2h}.$$

Here we have used $h=\Delta x$ and $y_n=y(x_n)$.

Second-order accurate derivative(二阶精确差分)

To calculate the second order accurate derivative, we use the first, third and fourth equations of (4.3.3). In the interior of the domain (在区间内), we use the center-difference scheme. However, at the left and right boundaries, there are no left and right neighboring points respectively to calculate the derivative with. Thus we require the use of forward- and backward-difference schemes respectively. This gives the basic algorithm

```
1 n=length (x)
2 % 2nd order accurate
3 ux(1)=(-3*u(1)+4*u(2)-u(3))/(2*dx);
4 for j=2:n-1
5    ux(j)=(u(j+1)-u(j-1))/(2*dx);
6 end
7 ux(n)=(3*u(n)-4*u(n-1)+u(n-2))/(2*dx);
```

The values of *ux(1)* and *ux(n)* are evaluated with the forward- and backward difference schemes.

Fourth-order accurate derivative (四阶精确微分)

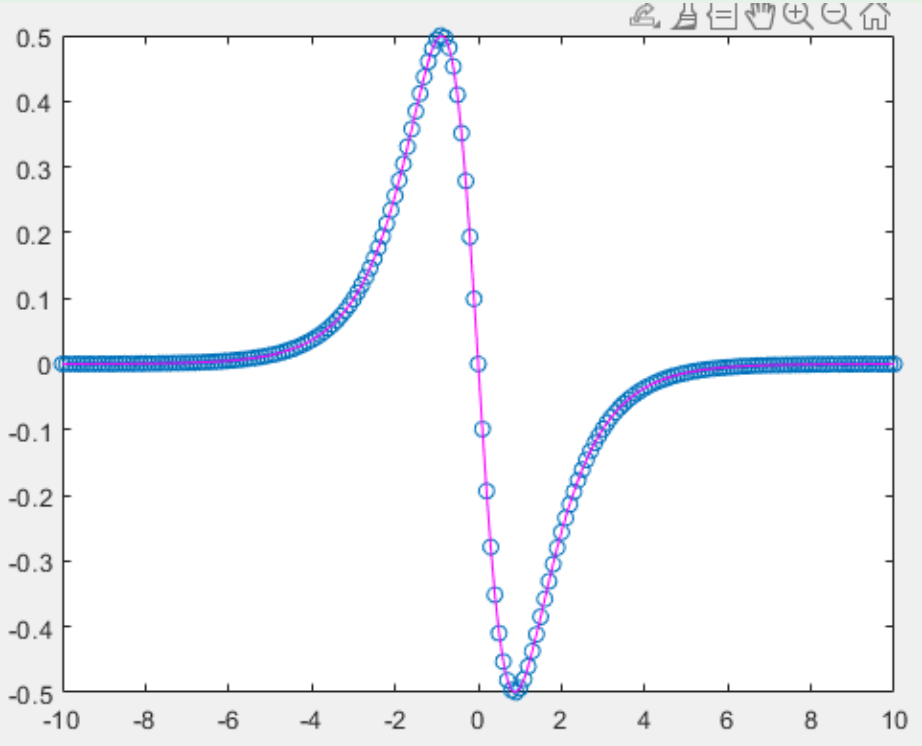
A higher degree of accuracy can be achieved by using a fourth-order scheme. A fourth-order center-difference scheme such as the second equation of (4.3.3) relied on two neighboring points. Thus the first two and last two points of the computational domain must be handled separately.

In what follows, we use a second-order scheme at the boundary points, and a fourth-order scheme for the interior. This gives the algorithm

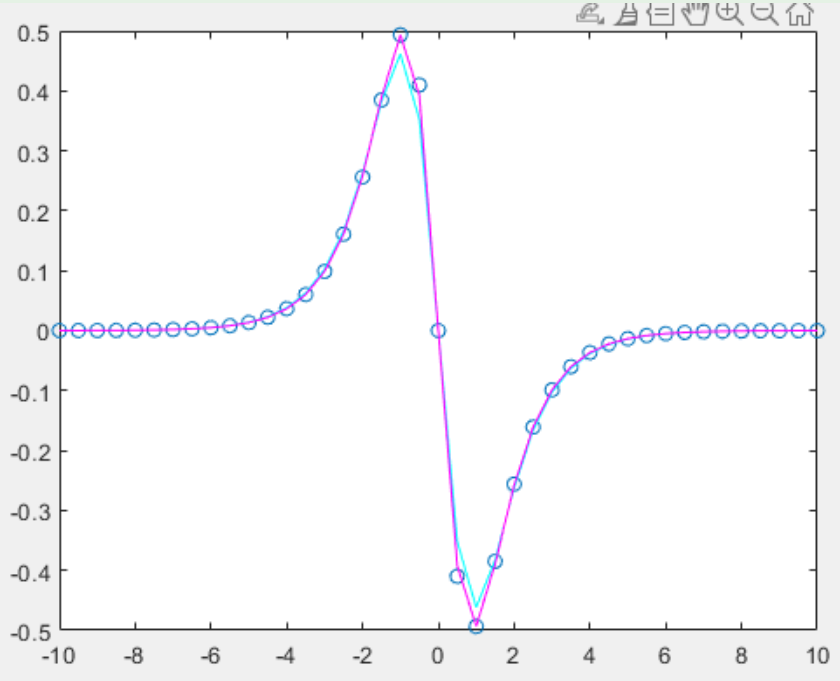
```
% 4th order accurate
ux2(1)=(-3*u(1)+4*u(2)-u(3))/(2*dx);
ux2(2)=(-3*u(2)+4*u(3)-u(4))/(2*dx);
for  j=3:n-2
   ux2(j)=(-u(j+2)+8*u(j+1)-8*u(j-1)+u(j-2))/(12*dx);
end
ux2(n-1)=(3*u(n-1)-4*u(n-2)+u(n-3))/(2*dx);
ux2(n)=(3*u(n)-4*u(n-1)+u(n-2))/(2*dx);
```

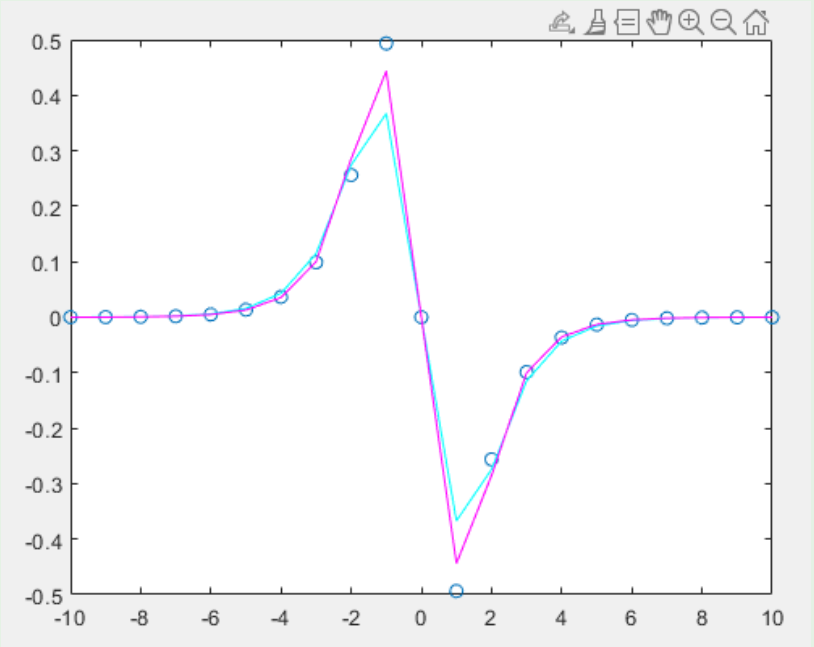
For the *dx = 0.1* considered here, the second-order and fourth-order schemes result in errors of 10^{-2} and 10^{-4} respectively. To view the accuracy, the results are plotted together with the analytic solution for the derivative

```
1 figure(2),plot(x,ux_exact,'o',x,ux,'c',x,ux2,'m')
```



To the naked eye (肉眼), these results all look to be exactly on the exact solution. However, by zooming in on a particular point, it quickly becomes clear that the errors for the two schemes are indeed 10^{-2} and 10^{-4} respectively. The failure of accuracy can also be observed by taking large values of dx. For instance, $dx = 0.5$ and $dx = 1$ illustrate how the **derivatives fail to be properly determined with large *dx* values**.





As a final note, if you are given a set of data to differentiate. It is always recommended that you first run a spline through the data with **an appropriately small *dx*** and then differentiate the spline. This will help to give smooth differentiated data. Otherwise, the data will tend to be highly inaccurate and choppy (波浪起伏的).

Integration

There are a large number of integration routines (例子) built into (内置) MATLAB. So unlike the differentiation routines presented here, we will simply make use of the built in MATLAB functions. The most straight-forward (直接的) integration routine is the trapezoidal rule. Given a set of data, the *trapz* command can be used to implement this integration rule. For the *x* and *y* data defined previously for the hyperbolic secant (双曲正割), the command structure gives

```
int_sech=trapz(x,y.2);
```

where we have integrated $sech_2(x)$. The value of this integral is exactly two.

The *trapz* command gives a value that is within 10^{-7} of the true value. To generate the cumulative values, the *cumtrapz* command is used.

```
int_sech2=cumtrapz(x,y.2);
figure(3),plot(x,int_sech2)
```

MATLAB figure 3 gives the value of the integral as a function of *x*.

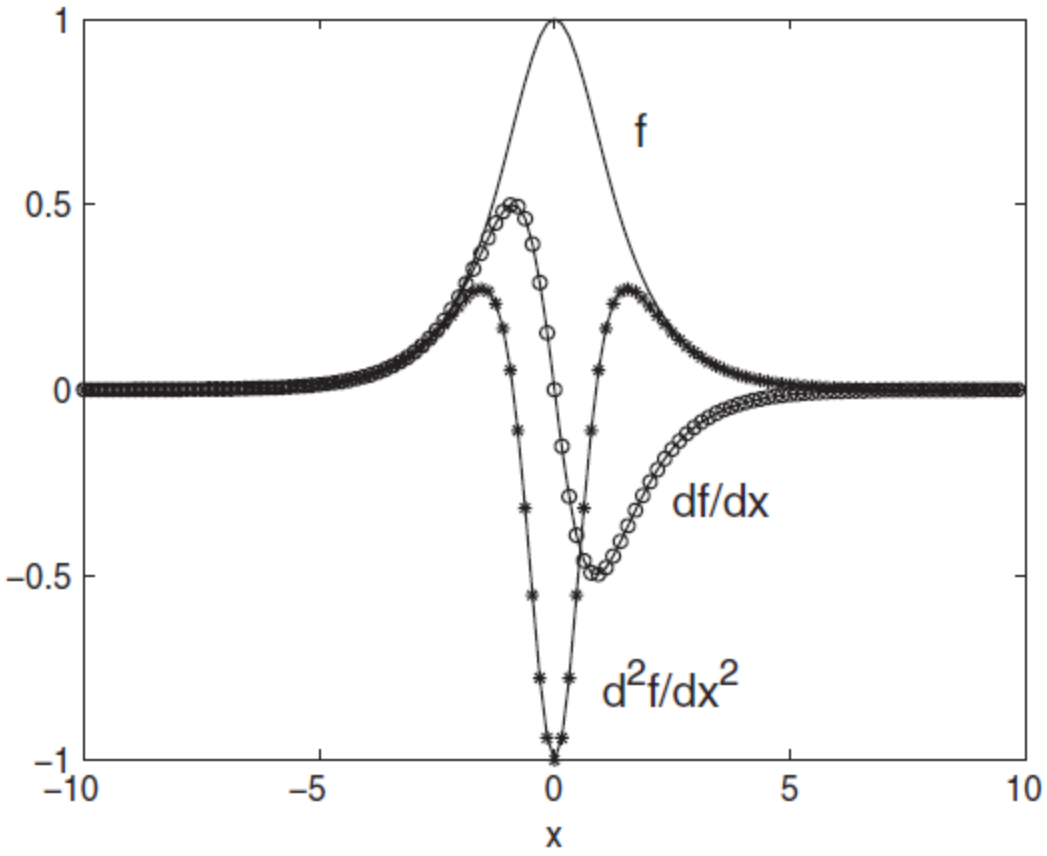


Figure 4.5: Calculation of the first and second derivatives of $f(x) = \mathrm{sech}(x)$ using the finite difference method. The approximate solutions for the first and second derivatives are compared with the exact analytic solutions (circles and stars, respectively).

Alternatively, a function can be specified for integration **over a given domain (指定区间内)**. The *quad* command is implemented by specifying a function and the range of integration. This will give a value of the integral using a recursive Simpson's rule that is accurate to within 10^{-6}. The command structure for this evaluation is as follows

```
1 int_quad=quad(inline('sech(x).2'),-10,10)
```

Here the inline command allows us to circumvent (回避) a traditional function call. The *quad* command can, however, be used with a function call. This command executes the integration of $\mathrm{sech}^2(x)$ over $x \in [-10,.10]$

Double and triple integrals (二重积分和三重积分) over two-dimensional rectangles and three-dimensional boxes can be performed with the *dbequad* and *triplequad* commands. Note that no version of the *quad* command exists that produces cumulative integration values. However, this can be easily handled in a for loop.

Double integrals can also be performed with the trapz command using a trapz imbedded within another trapz. As an example of a two-dimensional integration, we consider the dblquad command for integration of the following two-dimensional function

```
1 f(x,y)=cos(x)sech(x)sech(y)+1.5
```

on the box $x \in [-5, 5]$ and $y \in [-5, 5]$.

The *trapz* MATLAB code for solving this problem is the following

```
1 Area=dblquad(inline('cos (x).*sech(x).*sech(y)+.5')
      ,-5,5,-5,5)
```

Figure 4.7 demonstrates the trapezoidal rule that can be used to handle the two-dimensional integration procedure. A similar procedure is carried out in three dimensions.

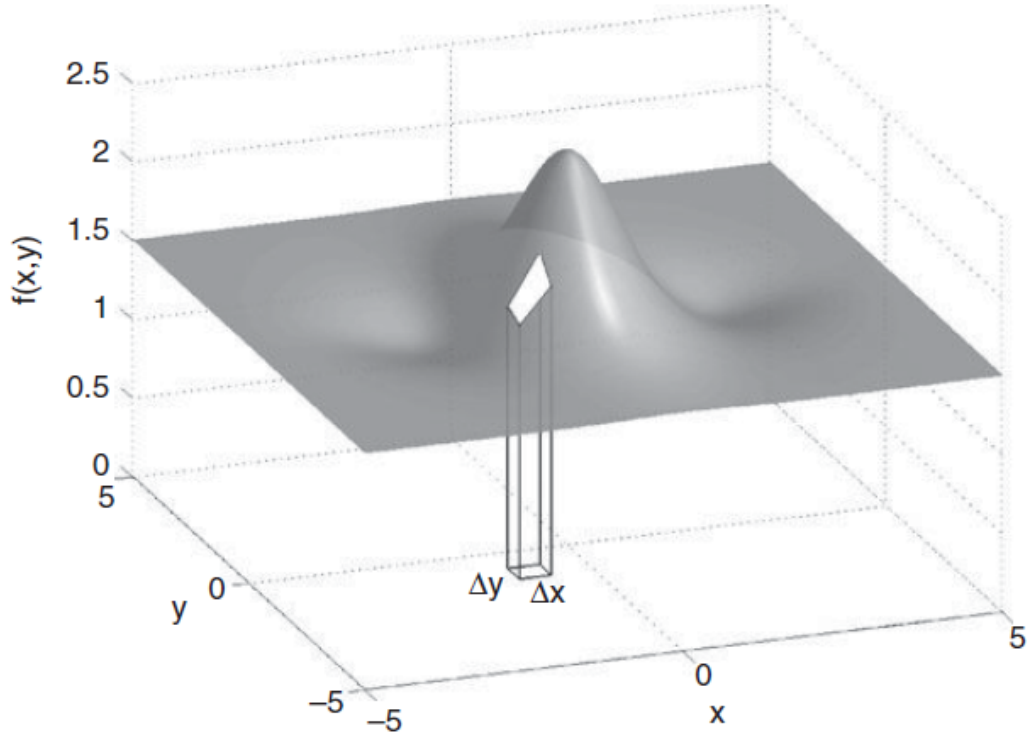


Figure 4.7: Two-dimensional trapezoidal rule integration procedure. As with one dimension, the domain is broken down into a grid of width and height Δx and Δy. The area of each parallelepiped is then easily calculated and summed up.