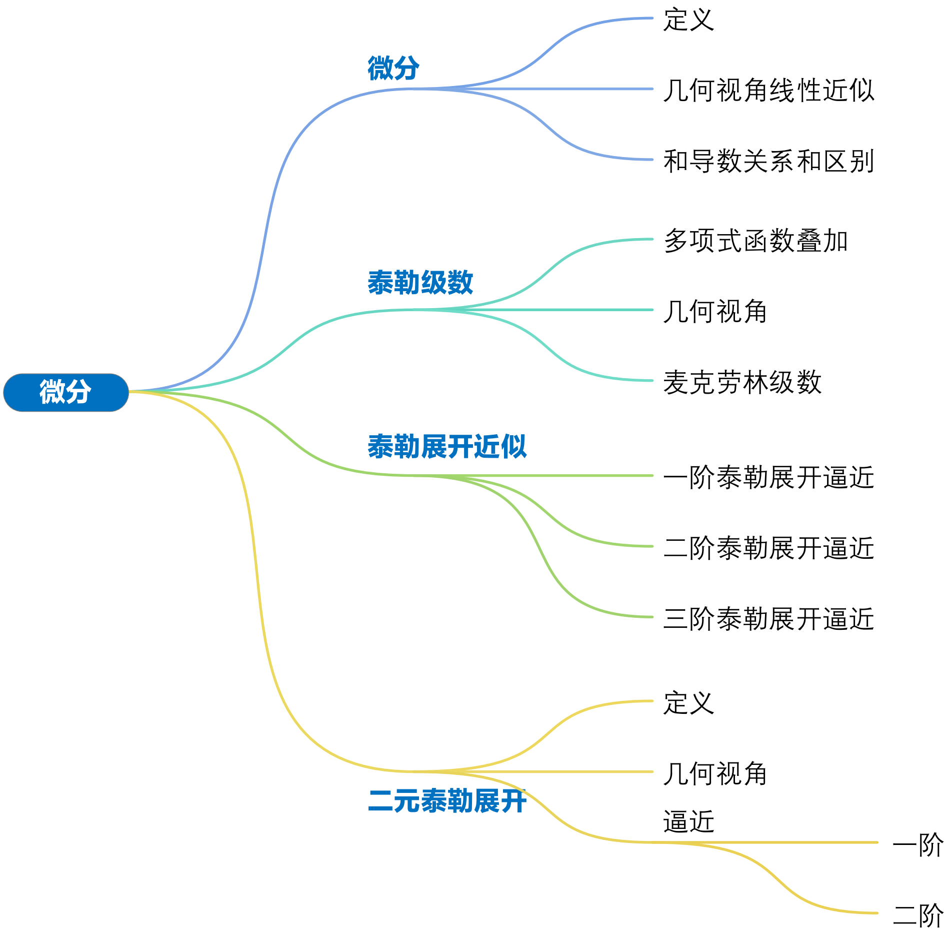
17 Differential 微分

微分是线性近似

> 我看的比别人更远，那是因为我站在一众巨人们的臂膀之上。
>
> ***If I have seen further than others, it is by standing upon the shoulders of giants.***
>
> —— 艾萨克·牛顿 (Isaac Newton) | 英国数学家、物理学家 | 1643 ~ 1727

- `numpy.meshgrid()` 获得网格数据
- `sympy.abc import x` 定义符号变量 x
- `sympy.abc import x` 定义符号变量 x
- `sympy.diff()` 求解符号导数和偏导解析式
- `sympy.evalf()` 将符号解析式中未知量替换为具体数值
- `sympy.lambdify()` 将符号表达式转化为函数
- `sympy.series()` 求解泰勒展开级数符号式
- `sympy.symbols()` 定义符号变量


微分
微分
定义
几何视角线性近似
和导数关系和区别
泰勒级数
多项式函数叠加
几何视角
麦克劳林级数
泰勒展开近似
一阶泰勒展开逼近
二阶泰勒展开逼近
三阶泰勒展开逼近
二元泰勒展开
定义
几何视角
逼近
一阶
二阶

17.1 几何角度看微分：线性近似

微分 (differential) 是函数的局部变化的一种线性描述。如图 1 所示，微分可以近似地描述当函数自变量取值出现足够小 Δx 变化时，函数值如何变化。

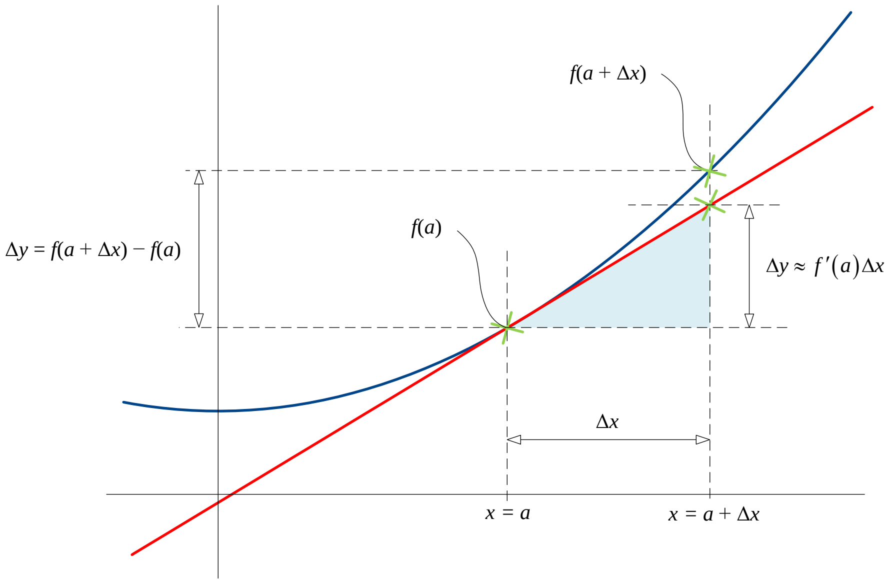


图 1. 对一元函数来说，微分是线性近似

假设函数 $f(x)$ 在某个区间内有定义。给定该区间内一点 a，当 a 变动到 $a + \Delta x$ (也在该区间内) 时，函数实际增量为 Δy：

$$\Delta y = f(a+\Delta x) - f(a) \tag{1}$$

而增量 Δy 可以近似为：

$$\Delta y = f(a+\Delta x) - f(a) \approx f'(a)\Delta x \tag{2}$$

其中，$f'(a)$ 为函数在 $x = a$ 处一阶导数；本书前文讲过，函数 $f(x)$ 在某一点处一阶导数值是函数在该点处切线的斜率值。

整理上式，$f(a + \Delta x)$ 的近似写成：

$$f(a+\Delta x) \approx f'(a)\Delta x + f(a) \tag{3}$$

令

$$x = a + \Delta x \tag{4}$$

(3) 可以写成：

$$f(x) \approx f'(a)(x-a)+f(a) \tag{5}$$

上式就是一次函数的点斜式；一次函数通过 $(a, f(a))$ 这点，斜率为 $f'(a)$。

如图 1 所示，从几何角度，微分用切线这条斜线代替曲线。实践中，复杂的非线性函数可以通过局部线性化来简化。

图 2 和图 3 分别所示为高斯函数和其一阶导数函数在若干点处的切线。

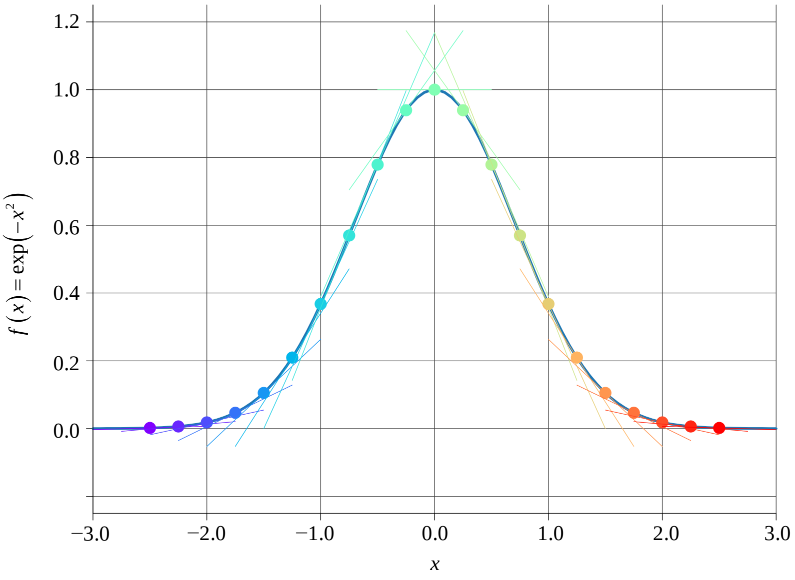


图 2. 高斯函数不同点处切线

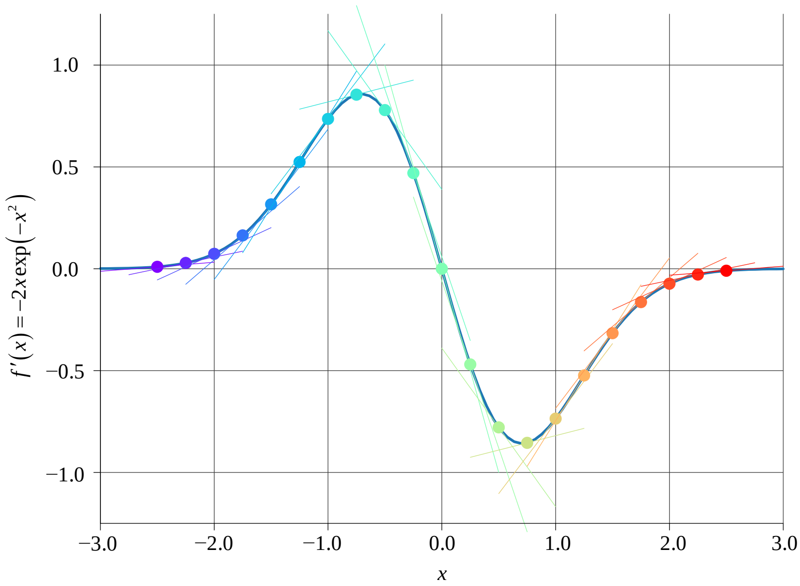


图 3. 高斯函数一阶导数不同点处切线

并不是所有函数都能得到导数的解析解，很多函数需要用数值方法近似求得导数。这种近似的思路和微分如出一辙。

图 4 给出三种一次导数的数值估算的方法：向前差分 (forward difference)、向后差分 (backward difference)、中心差分 (central difference)。

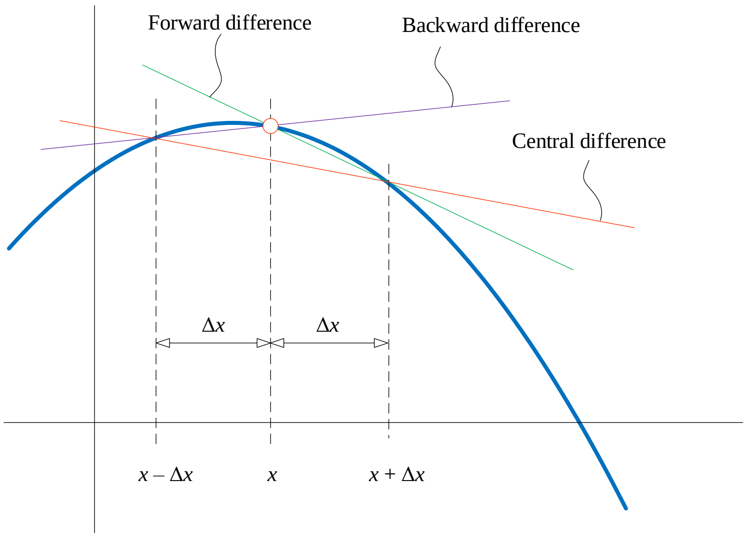


图 4. 三种一次导数的数值估计方法

一阶导数向前差分的具体公式为：

$$f'(x) \approx \frac{f(x+\Delta x)-f(x)}{\Delta x} \tag{6}$$

一阶导数向后差分的公式如下：

$$f'(x) \approx \frac{f(x)-f(x-\Delta x)}{\Delta x} \tag{7}$$

一阶导数的中心差分第一种形式如下：

$$f'(x) \approx \frac{f(x+\Delta x)-f(x-\Delta x)}{2\Delta x} \tag{8}$$

多元函数的偏微分也可以通过类似数值方法进行估算。本系列丛书后续会介绍这几种数值估算导数的方法。

17.2 泰勒级数：多项式函数近似

英国数学家布鲁克·泰勒 (Sir Brook Taylor) 在 1715 年发表了泰勒级数 (Taylor's theorem)。泰勒级数是一种强大的函数近似工具。

布鲁克·泰勒 (Brook Taylor)
英国数学家 | 1685年 ~ 1731年
以泰勒公式和泰勒级数闻名

当展开点 (expansion point) 为 $x = a$ 时，一元函数 $f(x)$ 泰勒展开 (Taylor expansion) 形式为：

$$
\begin{aligned}
f(x) &= \sum_{n=0}^{\infty} \frac{f^{(n)}(a)}{n!}(x-a)^n \\
&= \underbrace{f(a)}_{\text{Constant}} + \underbrace{\frac{f'(a)}{1!}(x-a)}_{\text{Linear}} + \underbrace{\frac{f''(a)}{2!}(x-a)^2}_{\text{Quadratic}} + \underbrace{\frac{f'''(a)}{3!}(x-a)^3}_{\text{Cubic}} + \cdots
\end{aligned}
\tag{9}
$$

其中，a 为展开点 (expansion point)。式中的阶乘是多项式求导产生的。展开点为 0 的泰勒级数又叫做麦克劳林级数 (Maclaurin series)。

如图 5 所示，泰勒展开相当于一系列多项式函数叠加，用来近似某个复杂函数。注意，图中常数函数图像对应的高度 $f(a)$ 提供了 $x = a$ 处 $f(x)$ 的函数值；而剩余其他多项式函数在展开点 $x = a$ 处函数值均为 0。

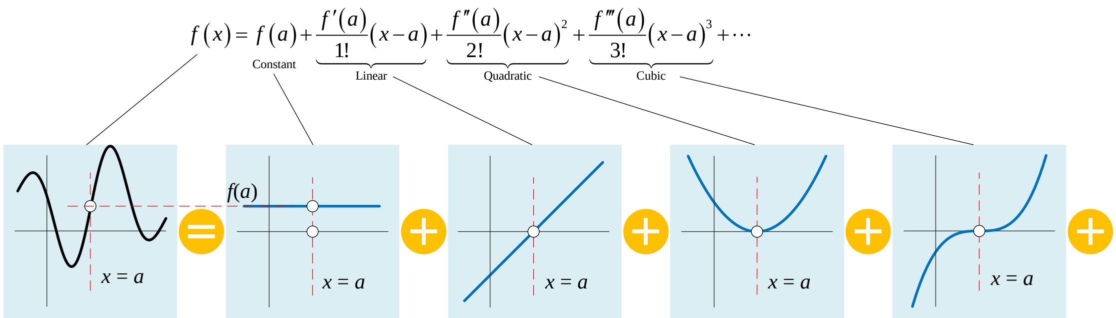


图 5. 一元函数泰勒展开原理

实际应用中，在应用泰勒公式近似计算时需要截断，也就是只取有限项。

上一节介绍微分时，(5) 实际上就是泰勒公式取前两项，即用“常数函数 + 一次函数”叠加近似原函数 $f(x)$：

$$
f(x) \approx \underbrace{f(a)}_{\text{Constant}} + \underbrace{\frac{f'(a)}{1!}(x-a)}_{\text{Linear}} = f(a) + f'(a)(x-a)
\tag{10}
$$

在 (10) “常数函数 + 一次函数” 基础上，再增加“二次函数”成分，我们便得到二次近似：

$$f(x) \approx \underbrace{f(a)}_{\text{Constant}} + \underbrace{\frac{f'(a)}{1!}(x-a)}_{\text{Linear}} + \underbrace{\frac{f''(a)}{2!}(x-a)^2}_{\text{Quadratic}} \tag{11}$$

图 6 和图 7 分别所示为高斯函数和其一阶导数函数的二次近似。泰勒公式把复杂函数转换为多项式叠加；相较其他函数，多项式函数更容易计算微分、积分。本章后续将会介绍利用泰勒展开近似。

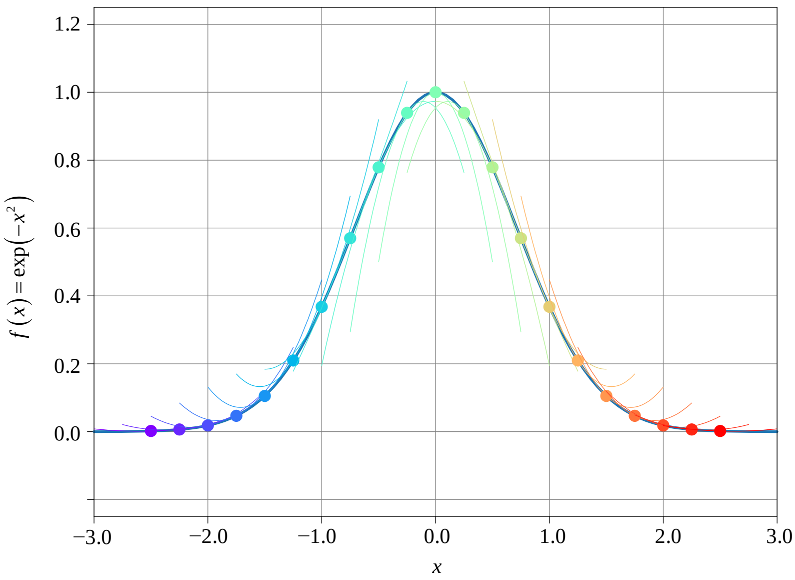


图 6. 高斯函数不同点处二次近似

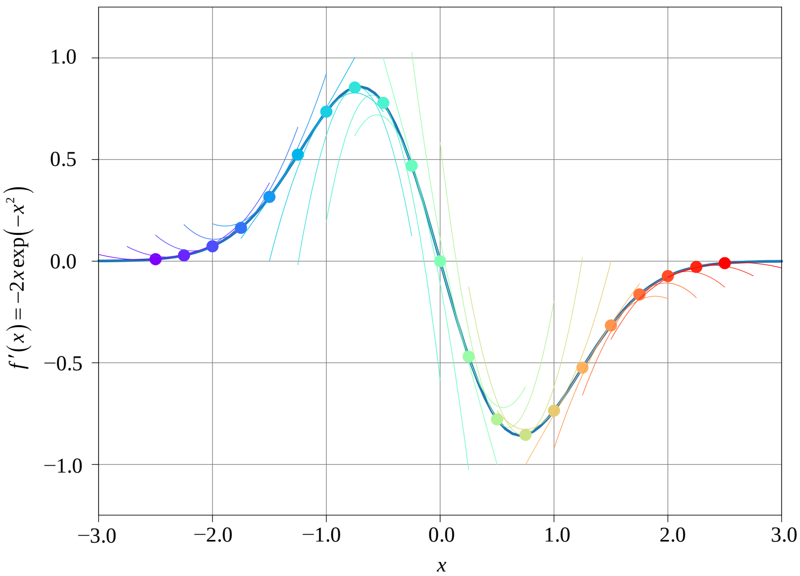


图 7. 高斯函数一阶导数不同点处二次近似

Bk3_Ch17_01.py 绘制图 6 和图 7。

```
# Bk3_Ch17_01

from sympy import lambdify, diff, evalf, sin, exp
from sympy.abc import x
import numpy as np
from matplotlib import pyplot as plt
from matplotlib import cm

f_x = exp(-x**2)

x_array   = np.linspace(-3,3,100)
x_0_array = np.linspace(-2.5,2.5,21)
# f_x.evalf(subs = {x: 0})

f_x_fcn = lambdify(x,f_x)
f_x_array = f_x_fcn(x_array)

#%% plot quadratic approx for original fcn

plt.close('all')

colors = plt.cm.rainbow(np.linspace(0,1,len(x_0_array)))

f_x_1_diff = diff(f_x,x)
f_x_1_diff_fcn = lambdify(x,f_x_1_diff)

f_x_2_diff = diff(f_x,x,2)
f_x_2_diff_fcn = lambdify(x,f_x_2_diff)

fig, ax = plt.subplots()

ax.plot(x_array, f_x_array, linewidth = 1.5)
ax.set_xlabel("$\it{x}$")
ax.set_ylabel("$\it{f}(\it{x})$")

for i in np.arange(len(x_0_array)):

    color = colors[i,:]

    x_0 = x_0_array[i]

    y_0 = f_x.evalf(subs = {x: x_0})
    x_t_array = np.linspace(x_0-0.5, x_0+0.5, 50)

    b = f_x_1_diff.evalf(subs = {x: x_0})

    a = f_x_2_diff.evalf(subs = {x: x_0})

    second_order_f = a*(x - x_0)**2 + b*(x - x_0) + y_0

    second_order_f_fcn = lambdify(x,second_order_f)
    second_order_f_array = second_order_f_fcn(x_t_array)

    ax.plot(x_t_array, second_order_f_array, linewidth = 0.25, color = color)
    ax.plot(x_0,y_0,marker = '.', color = color,
            markersize = 12)

ax.grid(linestyle='--', linewidth=0.25, color=[0.5,0.5,0.5])
ax.set_xlim((x_array.min(),x_array.max()))

ax.set_xlim(-3,3)
ax.set_ylim(-0.25,1.25)

#%% plot quadratic approx for first-order derivative
```

```python
f_x_1_diff_new = diff(f_x,x)
print(f_x_1_diff)
f_x_1_diff_fcn_new = lambdify(x,f_x_1_diff_new)
f_x_1_diff_array_new = f_x_1_diff_fcn_new(x_array)

colors = plt.cm.rainbow(np.linspace(0,1,len(x_0_array)))

f_x_1_diff = diff(f_x,x,2)
f_x_1_diff_fcn = lambdify(x,f_x_1_diff)

f_x_2_diff = diff(f_x,x,3)
f_x_2_diff_fcn = lambdify(x,f_x_2_diff)


fig, ax = plt.subplots()

ax.plot(x_array, f_x_1_diff_array_new, linewidth = 1.5)
ax.set_xlabel("$\it{x}$")
ax.set_ylabel("$\it{f}(\it{x})$")

for i in np.arange(len(x_0_array)):

    color = colors[i,:]

    x_0 = x_0_array[i]

    y_0 = f_x_1_diff_new.evalf(subs = {x: x_0})
    x_t_array = np.linspace(x_0-0.5, x_0+0.5, 50)

    b = f_x_1_diff.evalf(subs = {x: x_0})

    a = f_x_2_diff.evalf(subs = {x: x_0})

    second_order_f = a*(x - x_0)**2 + b*(x - x_0) + y_0

    second_order_f_fcn = lambdify(x,second_order_f)
    second_order_f_array = second_order_f_fcn(x_t_array)

    ax.plot(x_t_array, second_order_f_array, linewidth = 0.25, color = color)
    ax.plot(x_0,y_0,marker = '.', color = color,
            markersize = 12)

ax.grid(linestyle='--', linewidth=0.25, color=[0.5,0.5,0.5])
ax.set_xlim((x_array.min(),x_array.max()))

ax.set_xlim(-3,3)
ax.set_ylim(-1.25,1.25)
```

17.3 多项式近似和误差

再次强调，泰勒展开的核心是用一系列多项式函数叠加，来逼近某个函数。实际应用中，泰勒级数常用来近似计算复杂非线性函数，并估计误差。

给定原函数 $f(x)$为自然指数函数：

$$f(x)=\exp(x)=\mathrm{e}^{x} \tag{12}$$

在 $x = 0$ 处，该函数的泰勒级数展开为：

$$e^x = \sum_{n=0}^{\infty} \frac{x^n}{n!} = \frac{x^0}{0!} + \frac{x^1}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \cdots = 1 + x + \frac{x^2}{2} + \frac{x^3}{6} + \frac{x^4}{24} + \frac{x^5}{120} + \cdots \quad (13)$$

如前文所述，在具体应用场合，泰勒公式需要截断，只取有限项进行近似运算；一个函数的有限项的泰勒级数叫做泰勒展开式。

常数函数

在 $x = 0$ 点处，$f(x)$ 函数值为：

$$f(0) = \exp(0) = 1 \quad (14)$$

图 8 所示为用常数函数来近似原函数：

$$f_0(x) = \underbrace{1}_{\text{Constant}} \quad (15)$$

图 9 比较原函数和常数函数，并给出误差随 x 变化。常数函数为平行横轴的直线，它的估计能力显然明显不足。

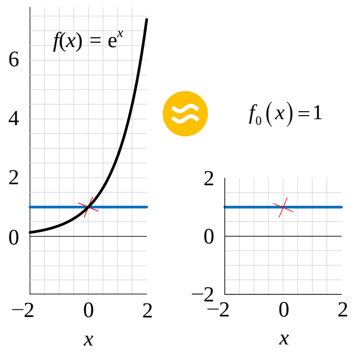


图 8. 常数函数近似

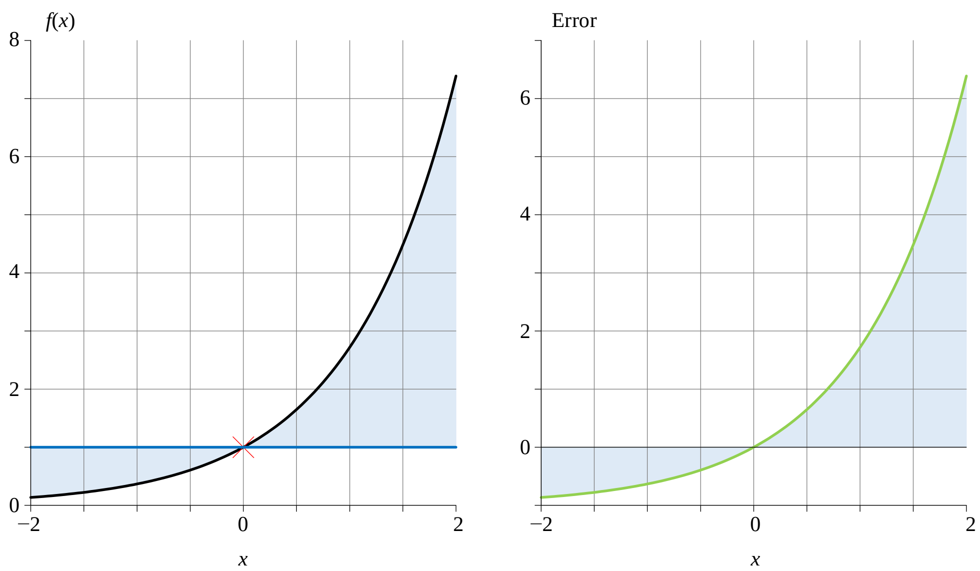


图 9. 常数函数近似及误差

一次函数

原函数 $f(x)$ 一阶导数为：

$$f'(x) = \exp(x) \tag{16}$$

$x = 0$ 处一阶导数为切线斜率：

$$f'(0) = \exp(0) = 1 \tag{17}$$

用一次函数来近似原函数：

$$f_1(x) = \underbrace{1}_{\text{Constant}} + \underbrace{x}_{\text{Linear}} \tag{18}$$

图 10 所示为"常数函数 + 一次函数"近似的原理。叠加常数函数和一次函数，常被称作一阶泰勒展开 (first-order Taylor polynomial/expansion/approximation 或 first-degree Taylor polynomial)。一阶泰勒展开是最常用的逼近手段。

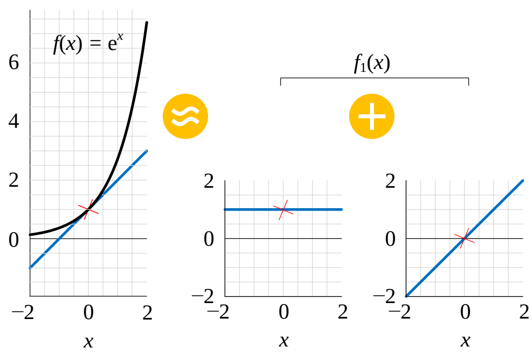


图 10. "常数函数 + 一次函数"近似

原函数和泰勒多项式的差被称作为泰勒公式的余项，即误差：

$$R(x) = f(x) - f_1(x) = f(x) - \left(\underbrace{1}_{\text{Constant}} + \underbrace{x}_{\text{Linear}}\right) \tag{19}$$

图 11 所示为一阶泰勒展开近似和误差；离展开点 $x = a$ 越远，误差越大。

也就是说，非线性函数在 $x = a$ 附近可以用这个一次函数近似；当 x 远离 a，这个近似就越不准确。

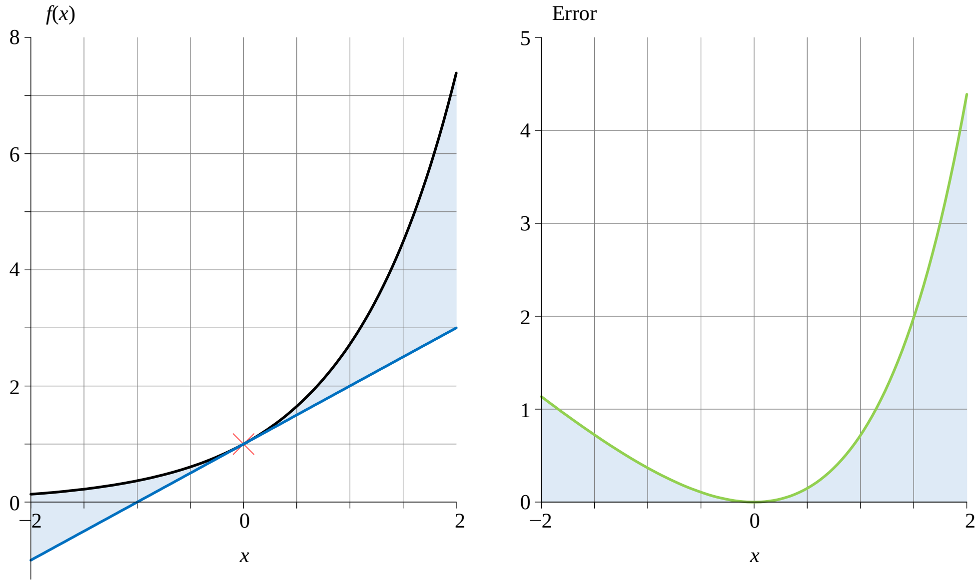


图 11. 一阶泰勒展开近似及误差

二次函数

用二次多项式函数近似原函数：

$$f_2(x) = \underbrace{1}_{\text{Constant}} + \underbrace{x}_{\text{Linear}} + \underbrace{\frac{x^2}{2}}_{\text{Quadratic}} \tag{20}$$

上式也叫二阶泰勒展开 (second-order Taylor polynomial/expansion/approximation)。图 12 所示，二阶泰勒展开叠加了三个成分，“常数函数 + 一次函数 + 二次函数”。

图 13 给出的是二阶泰勒展开近似及误差；相较图 11，图 13 中误差明显变小。

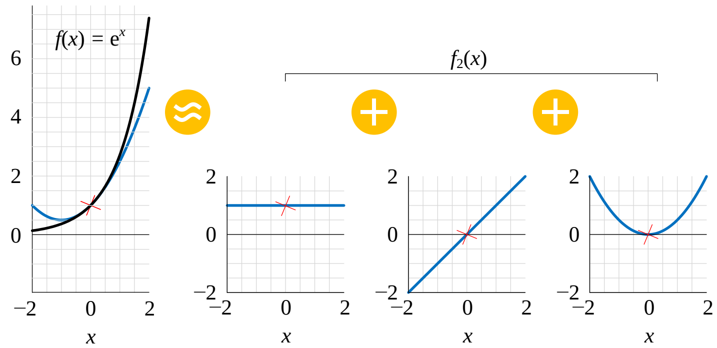


图 12. “常数函数 + 一次函数 + 二次函数”近似原函数

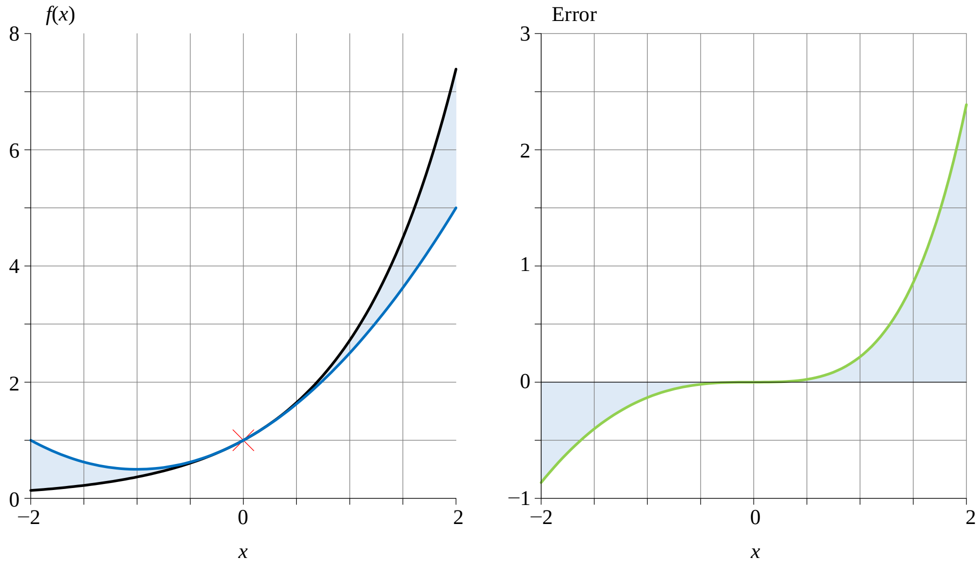


图 13. 二阶泰勒展开近似及误差

三次函数

用三次多项式函数近似原函数:

$$f_3(x) = \underbrace{1}_{\text{Constant}} + \underbrace{x}_{\text{Linear}} + \underbrace{\frac{x^2}{2}}_{\text{Quadratic}} + \underbrace{\frac{x^3}{6}}_{\text{Cubic}} \tag{21}$$

图 14 所示为“常数函数 + 一次函数 + 二次函数 + 三次函数”叠加近似原函数。比较图 13 和图 15，增加三次项后，逼近效果提高，误差进一步减小。

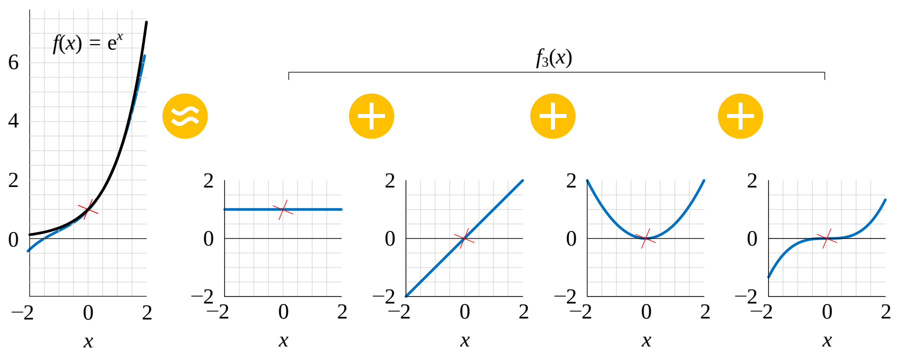


图 14. “常数函数 + 一次函数 + 二次函数 + 三次函数”近似原函数

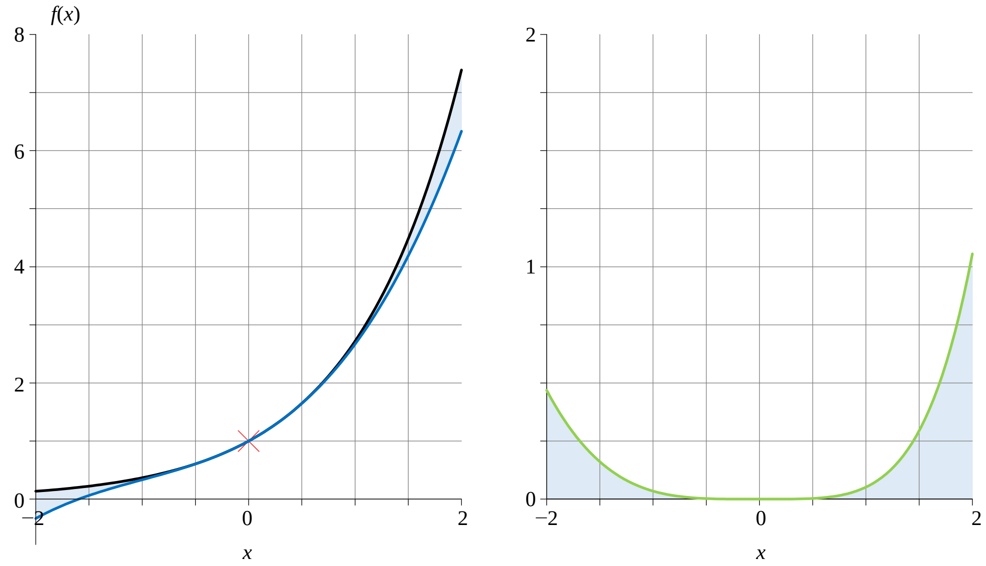


图 15. 三函数近似原函数及误差

四次函数

图 16 所示为用四次多项式函数近似原函数：

$$f(x)=\exp(x)\approx f_4(x)=\underbrace{1}_{\text{Constant}}+\underbrace{x}_{\text{Linear}}+\underbrace{\frac{x^2}{2}}_{\text{Quadratic}}+\underbrace{\frac{x^3}{6}}_{\text{Cubic}}+\underbrace{\frac{x^4}{24}}_{\text{Quartic}} \tag{22}$$

一般来说，泰勒多项式展开的项数越多，也就是说多项式幂次越高，逼近效果越好；但是，实际应用中，线性逼近和二次逼近用的最为广泛。

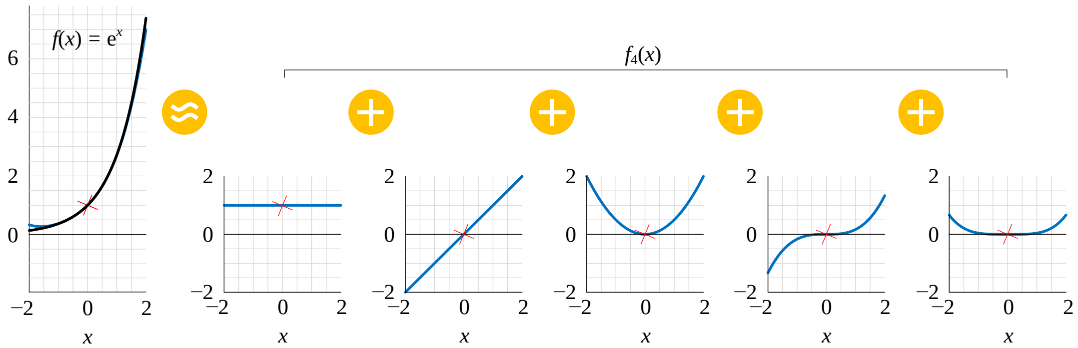


图 16.“ 常数函数 + 一次函数 + 二次函数 + 三次函数 + 四次函数”近似原函数

Bk3_Ch17_02.py 绘制本节图像；请大家改变展开点位置，比如 $x = 1$、$x = -1$，并观察比较近似及误差。

```
# Bk3_Ch17_02

from sympy import latex, lambdify, diff, sin, log, exp, series
from sympy.abc import x
import numpy as np
from matplotlib import pyplot as plt
from matplotlib import cm

f_x = exp(x)
x_array = np.linspace(-2,2,100)
x_0 = 0 # expansion point

# f_x = sin(x)
# x_array = np.linspace(-10,10,100)
# x_0 = np.pi/6 # expansion

# f_x = log(x + 1) # ln(y + 1) = r
# x_array = np.linspace(-0.8,2,100)

y_0 = f_x.evalf(subs = {x: x_0})

f_x_fcn = lambdify(x,f_x)
f_x_array = f_x_fcn(x_array)

# Visualization

plt.close('all')

fig, ax = plt.subplots()

ax.plot(x_array, f_x_array, 'k', linewidth = 1.5)
ax.plot(x_0, y_0, 'xr', markersize = 12)
ax.set_xlabel("$\it{x}$")
ax.set_ylabel("$\it{f}(\it{x})$")

highest_order = 5
order_array = np.arange(0,highest_order + 1)

colors = plt.cm.rainbow(np.linspace(0,1,len(order_array)))

i = 0

for order in order_array:

    f_series = f_x.series(x,x_0,order + 1).removeO()
    # order + 1 = number of terms

    f_series_fcn = lambdify(x,f_series)
    f_series_array = f_series_fcn(x_array)

    ax.plot(x_array, x_array*0 + f_series_array, linewidth = 0.5,
            color = colors[i,:],
            label = 'Order = %0.0f'%order)

    i += 1

ax.grid(linestyle='--', linewidth=0.25, color=[0.5,0.5,0.5])
ax.set_xlim(x_array.min(),x_array.max())
ax.spines['right'].set_visible(False)
ax.spines['top'].set_visible(False)

# ax.set_ylim(x_array.min(),x_array.max())
```

```
# ax.set_aspect('equal', 'box')
plt.legend()

#%% Error

fig = plt.figure(figsize=plt.figaspect(0.5))
ax = fig.add_subplot(1, 2, 1)

ax.plot(x_array, f_x_array, 'k', linewidth = 1.5)
ax.plot(x_0, y_0, 'xr', markersize = 12)
ax.set_xlabel("$\it{x}$")
ax.set_ylabel("$\it{f}(\it{x})$")

highest_order = 2

f_series = f_x.series(x,x_0,highest_order + 1).removeO()
# order + 1 = number of terms

f_series_fcn = lambdify(x,f_series)
f_series_array = f_series_fcn(x_array)
f_series_array = x_array*0 + f_series_array

ax.plot(x_array, f_series_array, linewidth = 1.5,
        color = 'b')

ax.fill_between(x_array,
                f_x_array,
                x_array*0 + f_series_array,
                color = '#DEEAF6')

ax.grid(linestyle='--', linewidth=0.25, color=[0.5,0.5,0.5])
ax.set_xlim(x_array.min(),x_array.max())

ax.set_ylim(np.floor(f_x_array.min()),
            np.ceil(f_x_array.max()))
# ax.set_aspect('equal', 'box')
# plt.legend()
ax.spines['right'].set_visible(False)
ax.spines['top'].set_visible(False)

ax = fig.add_subplot(1, 2, 2)

error = f_x_array - f_series_array
ax.plot(x_array, error, 'r', linewidth = 1.5)
ax.fill_between(x_array,
                error,
                color = '#DEEAF6')
plt.axhline(y=0, color='k', linestyle='--', linewidth = 0.25)
ax.set_xlabel("$\it{x}$")
ax.set_ylabel("Error")

ax.grid(linestyle='--', linewidth=0.25, color=[0.5,0.5,0.5])
ax.set_xlim(x_array.min(),x_array.max())
ax.set_ylim(np.floor(error.min()),np.ceil(error.max()))
ax.spines['right'].set_visible(False)
ax.spines['top'].set_visible(False)
```

17.4 二元泰勒展开：用多项式曲面近似

上一节介绍的一元泰勒展开也可以扩展到多元函数。本节以二元函数为例介绍多元函数泰勒展开。

给定二元函数 $f(x_1, x_2)$，它的泰勒展开可以写成：

$$f(x_1,x_2) = \underbrace{f(a,b)}_{\text{Constant}} + \underbrace{f_{x1}(a,b)(x_1-a) + f_{x2}(a,b)(x_2-b)}_{\text{Plane}} + \underbrace{\frac{1}{2!}\left[f_{x1x1}(a,b)(x_1-a)^2 + 2f_{x1x2}(a,b)(x_1-a)(x_2-b) + f_{x2x2}(a,b)(x_2-b)^2\right]}_{\text{Quadratic}} + \cdots \tag{23}$$

注意，式中假定两个混合偏导相同，即 $f_{x1x2}(a, b) = f_{x2x1}(a, b)$。

将 (23) 写成矩阵运算形式：

$$f(x_1,x_2) = f(a,b) + \begin{bmatrix} f_{x1}(a,b) \\ f_{x2}(a,b) \end{bmatrix}^{\mathrm{T}} \begin{bmatrix} x_1-a \\ x_2-b \end{bmatrix} + \frac{1}{2!}\begin{bmatrix} x_1-a \\ x_2-b \end{bmatrix}^{\mathrm{T}} \begin{bmatrix} f_{x1x1}(a,b) & f_{x2x1}(a,b) \\ f_{x1x2}(a,b) & f_{x2x2}(a,b) \end{bmatrix} \begin{bmatrix} x_1-a \\ x_2-b \end{bmatrix} + \dots \tag{24}$$

如图 17 所示，从几何角度，二元函数泰勒展开相当于，水平面、斜面、二次曲面、三次曲面等多项式曲面叠加。

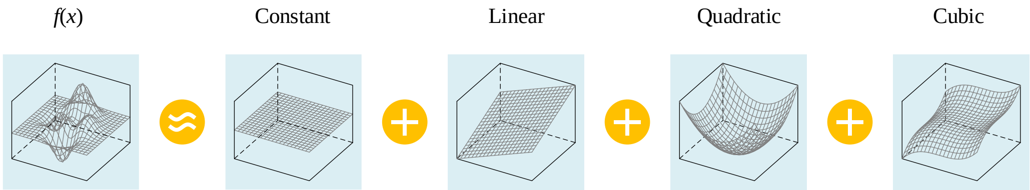


图 17. 二元函数泰勒展开原理

举例

举个例子，给定如下二元高斯函数：

$$y = f(x_1,x_2) = \exp\left(-\left(x_1^2 + x_2^2\right)\right) \tag{25}$$

二元函数 $f(x_1, x_2)$ 一阶偏导：

$$\begin{aligned} f_{x1}(x_1,x_2) &= \frac{\partial f}{\partial x_1}(x_1,x_2) = -2x_1\exp\left(-\left(x_1^2+x_2^2\right)\right) \\ f_{x2}(x_1,x_2) &= \frac{\partial f}{\partial x_2}(x_1,x_2) = -2x_2\exp\left(-\left(x_1^2+x_2^2\right)\right) \end{aligned} \tag{26}$$

图 18 所示为两个一阶偏导函数的平面等高线图；图中 × 为展开点位置，水平面位置坐标 (−0.1, −0.2)。

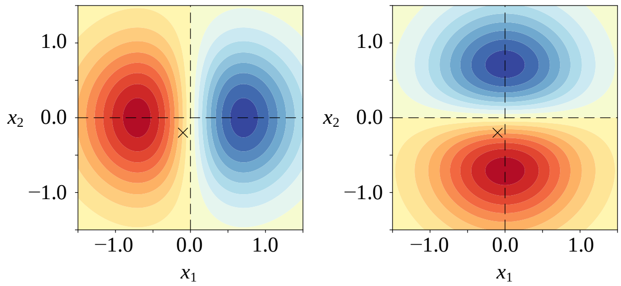


图 18. 一阶偏导数曲面等高线

(25) 中二元函数 $f(x_1, x_2)$ 二阶偏导：

$$
\begin{aligned}
f_{x1x1}(x_1,x_2) &= \frac{\partial^2 f}{\partial x_1^2}(x_1,x_2) = \left(-2+4x_1^2\right)\exp\left(-\left(x_1^2+x_2^2\right)\right) \\
f_{x2x2}(x_1,x_2) &= \frac{\partial^2 f}{\partial x_2^2}(x_1,x_2) = \left(-2+4x_2^2\right)\exp\left(-\left(x_1^2+x_2^2\right)\right) \\
f_{x1x2}(x_1,x_2) &= \frac{\partial^2 f}{\partial x_2 \partial x_1}(x_1,x_2) = 4x_1x_2\exp\left(-\left(x_1^2+x_2^2\right)\right) \\
f_{x2x1}(x_1,x_2) &= \frac{\partial^2 f}{\partial x_1 \partial x_2}(x_1,x_2) = 4x_1x_2\exp\left(-\left(x_1^2+x_2^2\right)\right)
\end{aligned}
\tag{27}
$$

显然，两个混合偏导相同，即，

$$
f_{x1x2}(x_1,x_2) = f_{x2x1}(x_1,x_2) \tag{28}
$$

图 19 所示为两个二阶偏导函数的平面等高线图。

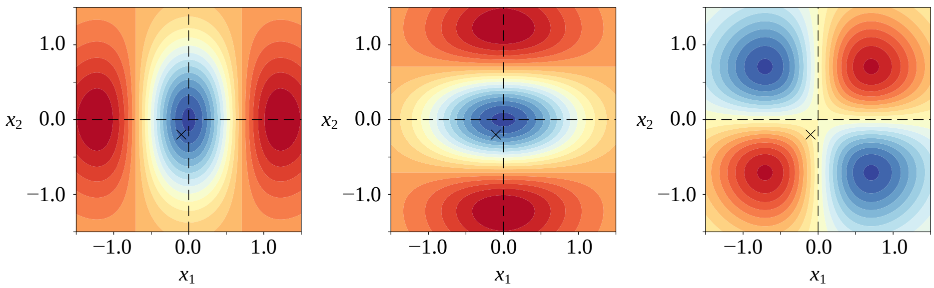


图 19. 二阶偏导数曲面等高线

展开点 × (−0.1, −0.2) 处函数值、一阶、二阶偏导数具体值为：

$$
f(-0.1,-0.2) = 0.951, \quad \begin{cases} f_{x1}(-0.1,-0.2) = 0.190 \\ f_{x2}(-0.1,-0.2) = 0.380 \end{cases}, \quad \begin{cases} f_{x1x1}(-0.1,-0.2) = -1.864 \\ f_{x2x2}(-0.1,-0.2) = -1.750 \\ f_{x1x2}(-0.1,-0.2) = f_{x2x1}(-0.1,-0.2) = 0.076 \end{cases} \tag{29}
$$

常数函数

类似前文，我们本节也采用逐步分析。首先用二元常函数数来估计 $f(x_1, x_2)$：

$$f\left(x_1,x_2\right) \approx \underbrace{f\left(a,b\right)}_{\text{Constant}} \tag{30}$$

这相当于用一个平行于 x_1x_2 平面的水平面来近似 $f(x_1, x_2)$。

图 20 所示为用常数函数估计二元高斯函数，常数函数对应解析式为：

$$f\left(x_1,x_2\right) \approx f\left(-0.1,-0.2\right) = 0.951 \tag{31}$$

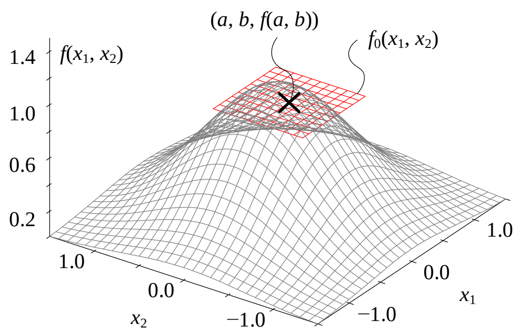


图 20. 用常数函数估计二元高斯函数

一次函数

用一次泰勒展开估计 $f(x_1, x_2)$：

$$f\left(x_1,x_2\right) \approx \underbrace{f\left(a,b\right)}_{\text{Constant}} + \underbrace{f_{x1}\left(a,b\right)\left(x_1-a\right) + f_{x2}\left(a,b\right)\left(x_2-b\right)}_{\text{Plane}} \tag{32}$$

相当于"水平面 + 斜面"叠加来近似 $f(x_1, x_2)$。

图 21 所示为一阶泰勒展开估计原函数，二元一次函数对应解析式为：

$$\begin{aligned} f\left(x_1,x_2\right) &\approx f\left(-0.1,-0.2\right) + f_{x1}\left(-0.1,-0.2\right)\left(x_1-\left(-0.1\right)\right) + f_{x2}\left(-0.1,-0.2\right)\left(x_2-\left(-0.2\right)\right) \\ &= 0.951 + 0.190\left(x_1+0.1\right) + 0.380\left(x_2+0.2\right) \end{aligned} \tag{33}$$

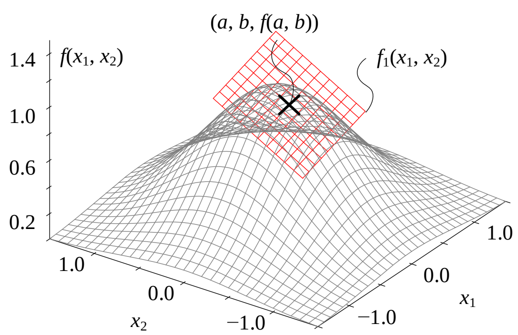


图 21. 用二元一次函数估计二元高斯函数

二次函数

用二次泰勒展开估计 $f(x_1, x_2)$:

$$\begin{aligned} f(x_1,x_2) \approx & \underbrace{f(a,b)}_{\text{Constant}} + \underbrace{f_{x1}(a,b)(x_1-a) + f_{x2}(a,b)(x_2-b)}_{\text{Plane}} \\ & + \underbrace{\frac{1}{2!}\left[f_{x1x1}(a,b)(x_1-a)^2 + 2 f_{x1x2}(a,b)(x_1-a)(x_2-b) + f_{x2x2}(a,b)(x_2-b)^2 \right]}_{\text{Quadratic}} \end{aligned} \tag{34}$$

相当于"水平面 + 斜面 + 二次曲面"叠加来近似 $f(x_1, x_2)$。

图 22 所示为二阶泰勒展开估计原函数，二元二次函数对应解析式为:

$$\begin{aligned} f_2(x_1,x_2) = & 0.951 + 0.190(x_1+0.1) + 0.380(x_2+0.2) \\ & + \frac{1}{2}\left[-1.864(x_1+0.1)^2 + 0.152(x_1+0.1)(x_2+0.2) - 1.750(x_2+0.2)^2 \right] \end{aligned} \tag{35}$$

请大家用本节代码自行展开整理上式。

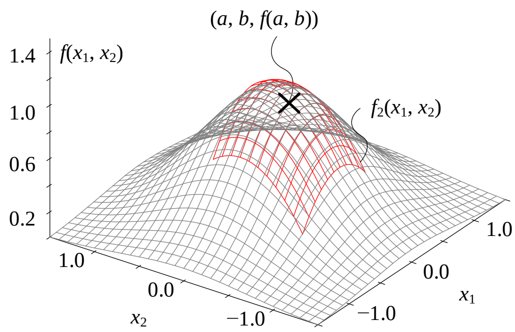


图 22. 用二次函数估计二元高斯函数

Bk3_Ch17_03.py 绘制图 20、图 21、图 22 三幅图。

```
# Bk3_Ch17_03

import numpy as np
from sympy import lambdify, diff, exp, latex, simplify
from sympy.abc import x, y
import numpy as np
from matplotlib import pyplot as plt
from matplotlib import cm


num = 301; # number of mesh grids
x_array = np.linspace(-1.5,1.5,num)
```

```python
y_array = np.linspace(-1.5,1.5,num)

# global mesh
xx,yy = np.meshgrid(x_array,y_array)

num_stride = 10

plt.close('all')

f_xy = exp(-x**2 - y**2)
f_xy_fcn = lambdify([x,y],f_xy)
f_xy_zz = f_xy_fcn(xx,yy)

# expansion point
x_a = -0.1
y_b = -0.2

# local mesh
x_a_array = np.linspace(x_a - 0.5,x_a + 0.5,101)
y_b_array = np.linspace(y_b - 0.5,y_b + 0.5,101)

xx_local,yy_local = np.meshgrid(x_a_array,y_b_array)

f_xy_zz_local = f_xy_fcn(xx_local,yy_local)


# expansion point
f_ab = f_xy_fcn(x_a,y_b)


#%% constant approximation

f_ab = f_xy_fcn(x_a,y_b)

fig, ax = plt.subplots(subplot_kw={'projection': '3d'})

ax.plot_wireframe(xx,yy, f_xy_zz,
                  color = [0.5,0.5,0.5],
                  rstride=num_stride, cstride=num_stride,
                  linewidth = 0.25)

approx_zero_order = f_ab + xx_local*0

ax.plot_wireframe(xx_local,yy_local, approx_zero_order,
                  color = [1,0,0],
                  rstride=num_stride, cstride=num_stride,
                  linewidth = 0.25)

ax.plot(x_a,y_b,f_ab, marker = 'x', color = 'r',
        markersize = 12)

ax.set_proj_type('ortho')

ax.set_xlabel('$x_1$')
ax.set_ylabel('$x_2$')
ax.set_zlabel('$f(x_1,x_2)$')

ax.set_xlim(xx.min(), xx.max())
ax.set_ylim(yy.min(), yy.max())
ax.set_zlim(f_xy_zz.min(), 1.5)

ax.view_init(azim=-145, elev=30)
# ax.view_init(azim=-90, elev=0)

plt.tight_layout()
ax.grid(False)
plt.show()

#%% first order approximation

df_dx = f_xy.diff(x)
```

```
df_dx_fcn = lambdify([x,y],df_dx)
df_dx_a_b = df_dx_fcn(x_a,y_b)

df_dy = f_xy.diff(y)
df_dy_fcn = lambdify([x,y],df_dy)
df_dy_a_b = df_dy_fcn(x_a,y_b)

approx_first_order = approx_zero_order + df_dx_a_b*(xx_local - x_a) +
df_dy_a_b*(yy_local - y_b)

fig, ax = plt.subplots(subplot_kw={'projection': '3d'})

ax.plot_wireframe(xx,yy, f_xy_zz,
                  color = [0.5,0.5,0.5],
                  rstride=num_stride, cstride=num_stride,
                  linewidth = 0.25)

ax.plot_wireframe(xx_local, yy_local, approx_first_order,
                  color = [1,0,0],
                  rstride=num_stride, cstride=num_stride,
                  linewidth = 0.25)

ax.plot(x_a,y_b,f_ab, marker = 'x', color = 'r',
        markersize = 12)

ax.set_proj_type('ortho')

ax.set_xlabel('$x_1$')
ax.set_ylabel('$x_2$')
ax.set_zlabel('$f(x_1,x_2)$')

ax.set_xlim(xx.min(), xx.max())
ax.set_ylim(yy.min(), yy.max())
ax.set_zlim(f_xy_zz.min(), 1.5)

ax.view_init(azim=-145, elev=30)
# ax.view_init(azim=-90, elev=0)

plt.tight_layout()
ax.grid(False)
plt.show()

#%% second order approximation

d2f_dxdx = f_xy.diff(x,2)
d2f_dxdx_fcn = lambdify([x,y],d2f_dxdx)
d2f_dxdx_a_b = d2f_dxdx_fcn(x_a,y_b)

d2f_dxdy = f_xy.diff(x,y)
d2f_dxdy_fcn = lambdify([x,y],d2f_dxdy)
d2f_dxdy_a_b = d2f_dxdy_fcn(x_a,y_b)

d2f_dydy = f_xy.diff(y,2)
d2f_dydy_fcn = lambdify([x,y],d2f_dydy)
d2f_dydy_a_b = d2f_dydy_fcn(x_a,y_b)

approx_second_order = approx_first_order + (d2f_dxdx_a_b*(xx_local - x_a)**2
                                            + 2*d2f_dxdy_a_b*(xx_local - x_a)*(yy_local
- y_b)
                                            + d2f_dydy_a_b*(yy_local - y_b)**2)/2

fig, ax = plt.subplots(subplot_kw={'projection': '3d'})

ax.plot_wireframe(xx,yy, f_xy_zz,
                  color = [0.5,0.5,0.5],
                  rstride=num_stride, cstride=num_stride,
                  linewidth = 0.25)

ax.plot_wireframe(xx_local,yy_local, approx_second_order,
                  color = [1,0,0],
                  rstride=num_stride, cstride=num_stride,
```

```
                  linewidth = 0.25)

ax.plot(x_a,y_b,f_ab, marker = 'x', color = 'r',
        markersize = 12)

ax.set_proj_type('ortho')

ax.set_xlabel('$x_1$')
ax.set_ylabel('$x_2$')
ax.set_zlabel('$f(x_1,x_2)$')

ax.set_xlim(xx.min(), xx.max())
ax.set_ylim(yy.min(), yy.max())
ax.set_zlim(f_xy_zz.min(), 1.5)

ax.view_init(azim=-145, elev=30)
# ax.view_init(azim=-90, elev=0)

plt.tight_layout()
ax.grid(False)
plt.show()
```

本章有两个关键点——微分、泰勒展开。

再次强调，导数是函数的变化率，而微分是函数线性近似。从几何视角来看，微分用切线近似非线性函数。

泰勒展开是一系列多项式函数叠加，用来近似某个复杂函数；最为常用的是一阶泰勒展开和二阶泰勒展开。