Python 科学计算基础

第十一章 数值计算

2025 年 9 月 5 日

目录

一元插值

给定一些离散数据点，插值方法求解一个通过这些点的连续函数，求解结果称为插值函数 (interpolant)。插值的应用领域包括绘制平滑曲线和曲面、估算函数值、求解非线性方程和数值积分等。scipy.interpolate 模块提供了多个实现插值方法的类。本节介绍如何使用多项式函数和样条函数进行一元插值。

给定二维平面上的互不相同的 n 个点 $\{(x_i, y_i)\}_{i=1}^{n}$，拉格朗日插值公式可以计算以下的多项式插值函数：

$$f(x) = \sum_{i=1}^{n} y_i L_i(x), \quad L_i(x) = \prod_{j=1, j\neq i}^{n} \left(\frac{x - x_j}{x_i - x_j}\right), \; i = 1, 2, ..., n$$

多项式函数

也可定义一个次数为 $n-1$ 的多项式函数 $p(x)=a_0+a_1x+\ldots+a_{n-1}x^{n-1}$ 作为插值函数，然后求解线性方程组 $\{p(x_i)=y_i\}_{i=1}^{n}$ 得到所有系数 $\{a_i\}_{i=0}^{n-1}$。

程序 11.1 使用 numpy.polynomial 模块的 Polynomial 类的 fit 函数求解多项式函数的系数。当 n 较大时，插值得到的高次多项式函数在两个数据点之间可能出现大幅度波动现象。

样条函数

为了避免高次多项式函数在插值时的病态性质，通常使用分段多项式函数作为插值函数，即把给定的数据点的 x 坐标值的取值范围划分为若干个子区间，在每个子区间上使用低次多项式函数进行插值。

样条函数是一种特殊类型的分段多项式函数。一个 k 次分段多项式函数称为样条函数，如果它在所有相邻子区间的连接点 (称为节点) 处具有连续的 $k-1$ 阶导数。在实际应用中通常选择 $k=3$，三次样条函数在所有节点处具有连续的二阶导数。

样条函数

scipy.interpolate 模块的 interp1d 类实现了一元样条函数插值。该类实现了__call__方法，可以作为函数调用。前两个参数分别是存储了数据点的 x 坐标值和 y 坐标值的数组，关键字实参 kind 可指定样条函数的次数。

程序 11.1 的运行结果显示使用三次样条插值得到的函数图像较为平滑，没有出现大幅度波动。

二元插值

给定一些分布在三维空间的数据点，二元插值生成一个通过这些点的曲面。本节假设这些数据点在 x-y 平面上的投影分布在一个网格上。网格上 x 坐标值的集合为 $\{x_1, x_2, ..., x_m\}$，y 坐标值的集合为 $\{y_1, y_2, ..., y_n\}$，则数据点为 $\{(x_i, y_j, z_k)\}$，$1 \le i \le m$，$1 \le j \le n$，$k = (i-1) \times n + j - 1$。

二元插值

如果网格在 x 轴方向和 y 轴方向都是等间距的，可使用 RectBivariateSpline 函数进行二元样条函数插值。它的前两个参数是分别存储了网格上 x 坐标值和 y 坐标值的数组，第三个参数是网格上所有点的 z 坐标值，关键字实参 kx 和 ky 分别指定样条函数在 x 轴方向和 y 轴方向的次数。

如果给定的数据点的网格不满足以上条件，可使用 interp2d 函数进行二元插值。它的前三个参数和 RectBivariateSpline 函数相同。指定关键字实参 kind 为'cubic' 表示使用三次样条函数。

二元插值

程序 11.2 的运行结果的左上子图绘制了函数 $z = (x^7 - y^6 + x^5 - y^4 + x^3 - y^2 + x - 1)e^{-x^2-y^2}$ 在区间 $[-3, 3] \times [-2.5, 2.5]$ 上的等高线图。中上子图和右上子图分别绘制了生成的在 20×20 等间距网格和在 40×40 等间距网格上的二元样条插值函数的等高线图。左下子图、中下子图和右下子图分别绘制了生成的在 20×20 随机网格、40×40 随机网格和 80×80 随机网格上的二元样条插值函数的等高线图。

可见在网格的点数相同时，随机网格上的插值效果不如等间距网格。对于两种网格，增加网格的点数都可以改善插值的效果。

数值积分

如果一个函数 $f(x)$ 不存在解析形式的积分原函数或函数的定义未知，只能通过数值方法求解其在区间 $[a, b]$ 上的定积分，即用区间内选取的 $n+1$ 个点 x_i $(i = 0, 1, ..., n)$(称为积分节点)上的函数值的加权和近似计算：

$$\int_a^b f(x)\,dx \approx \sum_{i=0}^{n} w_i f(x_i) \tag{1}$$

其中 w_i 是函数值 $f(x_i)$ 的权值，称为积分系数。不同的数值计算公式的区别体现在积分节点和积分系数上。

插值求积公式

插值求积公式 (interpolatory quadrature) 的原理是使用基于区间内一些节点进行插值得到的多项式函数作为原函数的近似。如果一个插值求积公式对于次数不超过 k 的多项式函数是精确的，但对次数为 $k+1$ 的多项式函数不精确，则称公式的精度为 k。插值求积公式分为两类：

- 牛顿-柯特斯型求积公式 (Newton-Cotes quadrature)：节点之间必须是等距的；
- 高斯型求积公式 (Gaussian quadrature)：节点选取为正交多项式 (例如勒让德 Legendre 多项式、切比雪夫 Chebyshev 多项式等) 的零点。

插值求积公式的精度

梯形公式和辛普森公式属于牛顿-柯特斯型公式。梯形公式的节点有 2 个，精度为 1。辛普森公式的节点有 3 个，精度为 3。

当节点个数为 $n+1$ 时，高斯-勒让德公式的精度等于 $2n+1$。2 个节点的高斯-勒让德公式的精度为 3。

通常将积分区间划分为多个子区间，在每个子区间分别使用插值求积公式，由此得到的求积公式称为复合求积公式。实验 5 的第二题列出了复合梯形公式、复合辛普森公式和复合高斯-勒让德公式。

插值求积公式

程序 11.3 根据精度的定义，即

$$\int_a^b x^j \, dx = \sum_{i=0}^{n} w_i x_i^j \quad \text{for} \quad j = 0, \ldots, k \quad ,$$

得出方程组，然后利用 sympy 解方程组，可以求得梯形公式和辛普森公式的积分系数，以及 2 个节点的高斯-勒让德公式的积分节点和积分系数。

scipy 库的数值积分函数

scipy 库的 integrate 模块提供了多个数值积分函数。

如果被积函数已给定，quad 函数使用高斯型公式计算其定积分，积分区间的上界和下界可以是正负无穷大。

- 返回值为一个元组，由积分值和估计的绝对误差组成。
- 如果被积函数包含多个变量，则 quad 只对第一个变量计算积分，可通过 args 关键字实参提供其他变量的值。
- 如果要积分的变量不是函数的第一个变量 (例如 bessel 函数 jv(3, x))，可利用 lambda 函数将其交换到第一个变量。
- 如果函数在求积公式的某个取样点处发散，可用 points 关键字实参指定需要规避的取样点。

scipy 库的数值积分函数

dblquad(f,a,b,c,d) 函数计算二重积分 $\int_a^b \int_{c(x)}^{d(x)} f(x,y)\,dx\,dy$。其中两个常数 a 和 b 是对 x 积分的区间的下界和上界，两个以 x 为自变量的 Python 函数 c 和 d 是对 y 积分的区间的的下界和上界。

类似地，tplquad 函数计算三重积分
$\int_a^b \int_{c(x)}^{d(x)} \int_{g(x,y)}^{h(x,y)} f(x,y,z)\,dx\,dy\,dz$。

nquad 函数计算 n 重积分，它的第一个参数是被积函数，第二个参数是一个列表，由对每个变量积分的区间的下界和上界组成的元组组成。

scipy 库的数值积分函数

如果被积函数未知，只给定了函数的一些数据点 (例如通过实验和观测得到)，可使用牛顿-柯特斯型求积公式。

simps 函数和 trapz 函数分别实现了辛普森公式和梯形公式。它们的参数为两个数组，分别存储了数据点的 y 坐标值和 x 坐标值。

如果数据点的个数是 $2^k + 1$(k 是一个正整数) 且数据点之间是等距的，可使用 romb 函数。romb 函数实现了 Romberg 公式，它的第一个参数是存储了数据点的 y 坐标值的数组，第二个参数是相邻数据点之间的间距。 程序 11.4

线性方程组求解

线性方程组的一般形式是 $\mathbf{Ax} = \mathbf{b}$，其中 $\mathbf{A}$ 是一个 m 行 n 列的矩阵。

- SymPy 库的 linsolve 函数可以求解规模较小的线性方程组，并得到精确解。$\mathbf{A}$ 的元素可以使用符号变量。
- 当 $m = n$ 并且 $\mathbf{A}$ 非奇异时，可使用 scipy.linalg 模块中定义的 solve 函数求解。
- numpy.linalg 模块的 lstsq 函数通过最小化欧式范数 $\|\mathbf{Ax} - \mathbf{b}\|_2^2$ 求最小二乘解。$\mathbf{A}$ 的线性无关行的个数可以大于或小于或等于它的线性无关列的个数。程序 11.5 演示了最小二乘解。

非线性方程的数值求解

SymPy 库的 solveset 函数可求解非线性方程。大多数非线性方程无法获得解析解，只能使用数值方法求近似解。数值方法求解非线性方程的基本方法有三种：二分法、割线法和牛顿法。这三种方法的共同点是通过多次迭代求解，区别在于收敛的速度和可靠性。后两种方法的基本思想都是使用线性函数作为非线性函数的近似。

以下用 $f(x) = 0$ 表示要求解的方程，α 表示未知解。定义第 k 次迭代时的解为 x_k，它和未知解的绝对误差为 $\delta_k = |x_k - \alpha|$。如果对于一个迭代算法可以证明 $\lim_{k \to +\infty} \frac{\delta_{k+1}}{\delta_k{}^q}$ 是一个常数，则它的收敛阶为 q。

二分法

二分法依据的定理是：若函数 $f(x)$ 在 $[a,b]$ 上连续，且 $f(a)f(b)<0$，则在 (a,b) 上至少存在一个解。

二分法的原理是通过每次迭代把包含解的区间的宽度缩小为原来的一半。令 $a_1=a$ 和 $b_1=b$。设第 k 次迭代时的区间是 $[a_k,b_k]$，定义 x_k 为区间的中点：$x_k=(a_k+b_k)/2$。

- 若 $f(x_k)=0$，则已找到解。
- 若 $f(a_k)f(x_k)<0$，设定 $a_{k+1}=a_k$ 和 $b_{k+1}=x_k$，然后进行下一次迭代。
- 若 $f(a_k)f(x_k)>0$，设定 $a_{n+1}=x_k$ 和 $b_{k+1}=b_k$，然后进行下一次迭代。

二分法

x_k 和未知解的绝对误差
$\delta_k = |\alpha - x_k| \leq (b_k - a_k)/2 = (b-a)/2^k$。为了使得 δ_k 不超过一个预先设定的值 ϵ，即 $(b-a)/2^n \leq \epsilon$，需要迭代的次数为 $\lceil \log \frac{b-a}{\epsilon} / \log 2 \rceil$。

二分法的优点是可靠性：从满足条件的区间开始迭代可以保证获得解。二分法的缺点是收敛阶为 1，收敛速度比另外两种方法慢。

割线法

割线法从两个给定的起始点 x_0 和 x_1 开始迭代。其原理是在第 k 次迭代时，用通过当前解 x_k 和第 $k-1$ 次的解 x_{k-1} 的数据点 $(x_k, f(x_k))$ 和 $(x_{k-1}, f(x_{k-1}))$ 的割线 $y = f(x_k) + \frac{f(x_k)-f(x_{k-1})}{x_k-x_{k-1}}(x - x_k)$ 作为函数 $f(x)$ 的近似，把割线和 x 轴的交点的横坐标作为 x_{k+1} 进行下一次迭代。求解线性方程 $0 = f(x_k) + \frac{f(x_k)-f(x_{k-1})}{x_k-x_{k-1}}(x - x_k)$ 可得 $x_{k+1} = x_k - \frac{x_k-x_{k-1}}{f(x_k)-f(x_{k-1})}f(x_k)$。

当两个起始点与未知解足够接近时，割线法可以保证收敛 (称为局部收敛)，收敛阶 (至少) 是 1.61803。

牛顿法

牛顿法从一个给定的起始点 x_0 开始迭代。其原理是在第 k 次迭代时，用通过当前解 x_k 的数据点 $(x_k, f(x_k))$ 的切线 $y = f(x_k) + f'(x_k)(x - x_k)$ 作为函数 $f(x)$ 的近似，把切线和 x 轴的交点的横坐标作为 x_{k+1} 进行下一次迭代。求解线性方程 $0 = f(x_k) + f'(x_k)(x_{k+1} - x_k)$ 可得 $x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$。

牛顿法也是局部收敛的。当 $f''(\alpha) \neq 0$ 时，牛顿法的收敛阶是 2。如果起始点和未知解相差较大，可能导致收敛速度很慢或不收敛，例如 $f(x) = x^{20} - 1$ 和 $x_0 = 1/2$。

Brent 方法

Brent 方法是一种综合了二分法、割线法以及逆二次插值的复杂方法。它具有二分法的可靠性和接近割线法的收敛速度。如果函数的导数未知，则 Brent 方法是首选方法。

scipy 库的求解非线性方程的函数

scipy 库的 optimize 模块定义了多个求解非线性方程的函数，它们的第一个参数都是 $f(x)$。

- bisect 函数实现了二分法，它的第二个和第三个参数分别是区间的下界和上界。
- newton 函数实现了牛顿法和割线法，它的第二个参数是起始点。如果提供了 $f(x)$ 的导数作为 fprime 关键字实参，则使用牛顿法，否则使用割线法。
- brentq 函数实现了 Brent 方法，它的第二个和第三个参数分别是区间的下界和上界。

程序 11.6 演示了如何使用这些函数求解方程 $e^x - 3x^3 - 6x^2 = 0$。

牛顿法和割线法的求解过程示例

程序 11.7 的运行结果显示了牛顿法和割线法的前几次迭代过程。

非线性方程组求解

SymPy 库的 nonlinsolve 函数可求解一些非线性方程组。大多数非线性方程组只能使用数值方法求近似解。含有 n 个变量的非线性方程组的一般形式为

$$\begin{aligned} f_1(x_1, x_2, ..., x_n) &= 0 \\ f_2(x_1, x_2, ..., x_n) &= 0 \\ ... &= 0 \\ f_n(x_1, x_2, ..., x_n) &= 0 \end{aligned}$$

定义 $\mathbf{f} = (f_1, f_2, ..., f_n)$ 为一个以 n 维向量 $\mathbf{x} = (x_1, x_2, ..., x_n)$ 为自变量的函数，它的函数值也是 n 维向量。

非线性方程组求解

以上方程组表示为 $\mathbf{f}(\mathbf{x}) = \mathbf{0}$。对于一元函数使用的牛顿法可推广到 n 维向量函数。

在第 k 次迭代时，使用线性函数 $\mathbf{f}(\mathbf{x_k}) + \mathbf{J_f}(\mathbf{x_k})(\mathbf{x} - \mathbf{x_k})$ 作为函数 $\mathbf{f}(\mathbf{x})$ 的近似。其中 $\mathbf{J_f}(\mathbf{x_k})$ 是函数 $\mathbf{f}(\mathbf{x})$ 在 $\mathbf{x_k}$ 处的 Jacobian 矩阵，$[\mathbf{J_f}(\mathbf{x})]_{ij} = \frac{\partial f_i(\mathbf{x})}{\partial x_j}$。求解线性方程 $\mathbf{0} = \mathbf{f}(\mathbf{x_k}) + \mathbf{J_f}(\mathbf{x_k})(\mathbf{x_{k+1}} - \mathbf{x_k})$ 可得 $\mathbf{x_{k+1}} = \mathbf{x_k} - \mathbf{J_f}(\mathbf{x_k})^{-1}\mathbf{f}(\mathbf{x_k})$。为了避免求矩阵的逆，可以定义 $\Delta_k = \mathbf{x_{k+1}} - \mathbf{x_k}$，则每次迭代只需求解线性方程组 $\mathbf{J_f}(\mathbf{x_k})\Delta_k = -\mathbf{f}(\mathbf{x_k})$，然后设定 $\mathbf{x_{k+1}} = \mathbf{x_k} + \Delta_k$。

scipy 库的 root 函数

scipy 库的 optimize 模块的 root 函数可求解非线性方程组，它的第一个参数是要求解的方程组。

关键字实参 method 可指定多种求解方法，例如 ‘hybr’ (MINPACK 实现的 Powell 方法)、‘lm’ (Levenberg-Marquardt 算法) 和 ‘broyden1’ (近似计算 Jacobian 矩阵的 Broyden 方法) 等。对于前两种方法，可以通过关键字实参 jac 说明是否提供 Jacobian 矩阵。第三种方法无需提供 Jacobian 矩阵。

scipy 库的 root 函数

程序 11.8 演示了用以上方法求解非线性方程组：

$$\begin{aligned} 3x - \cos(yz) - 0.5 &= 0 \\ 4x^2 - 625y^2 + 2y - 1 &= 0 \\ e^{-xy} + 20z - 9\pi &= 0 \end{aligned}$$

第 5 行至第 7 行定义了一个列向量 f_mat，它的每个分量是组成方程组的一个方程，方程中的数学函数 (例如 cos) 由 SymPy 库定义。

scipy 库的 root 函数

第 8 行使用 SymPy 计算了 Jacobian 矩阵

$$\begin{bmatrix} 3 & z\sin(yz) & y\sin(yz) \\ 8x & 2-1250y & 0 \\ -ye^{-xy} & -xe^{-xy} & 20 \end{bmatrix}$$

。

第 10 行至第 13 行定义了一个函数 f，它返回一个列表，其中的每个元素是组成方程组的一个方程。方程中需要求解的各变量 (x,y,z) 用一个数组 x 的各元素 (x[0],x[1],x[2]) 分别表示。第 14 行至第 20 行定义了一个函数 f2，它返回一个元组。其中的第一个元素是 f(x)，第二个元素是存储了 Jacobian 矩阵的二维数组。关键字实参 jac 设定为 True 时 root 函数的第一个实参是 f2，否则是 f。

常微分方程求解

常微分方程指包含未知函数和其对于单个自变量的一阶或高阶导数的方程。常微分方程的阶数由方程中出现的最高阶导数确定。

求解常微分方程的方法是对其进行解析积分或数值积分，一个 n 阶常微分方程在积分时会产生 n 个积分常数，需要指定 n 个边界条件才能确定所有积分常数。如果这些边界条件都位于自变量的同一个点上，称为初值问题。如果这些边界条件位于自变量的多个点上，称为边值问题。SymPy 库的 dsolve 函数可求解一些常微分方程。很多常微分方程只能使用数值方法求解。

常微分方程的初值问题

常微分方程的初值问题的标准形式是求解一个满足以下一阶常微分方程和初值条件的向量值函数 $\mathbf{y}(x) \in C^1[a,b]$：

$$\frac{d\mathbf{y}}{dx} = \mathbf{f}(x,\mathbf{y}), \quad a \le x \le b; \quad \mathbf{y}(a) = \mathbf{y}_0.$$

出现了高阶导数的常微分方程可以通过引入新的变量转换为标准形式。例如牛顿第二运动定律

$$F(s) = m\frac{d^2s}{dt^2}$$

可以转换为标准形式：

$$\frac{d\mathbf{y}}{dt} = \mathbf{f}(t,\mathbf{y}), \quad \mathbf{y} = [s,v]^T, \quad v = \frac{ds}{dt}$$

常微分方程的初值问题

关于初值问题的解的基本定理是：若 $\mathbf{f}(x, \mathbf{y})$ 对于 $x \in [a, b]$ 连续，并且对于某种范数 $|\cdot|$ 满足 Lipschitz 条件：

$$|\mathbf{f}(x, \mathbf{y}) - \mathbf{f}(x, \mathbf{y}^*)| \leq L|\mathbf{y} - \mathbf{y}^*|, \quad \forall x \in [a, b], \quad \forall \mathbf{y} \in R^d, \quad \forall \mathbf{y}^* \in R^d$$

则以上初值问题对于所有 $\mathbf{y}_0 \in R^d$ 存在唯一解 $\mathbf{y}(x)$。

初值问题的数值求解方法是在区间 $[a, b]$ 上的一些离散的点 x_n 上计算 $\mathbf{y}(x_n)$ 的近似值 $\mathbf{u}_n$。这些点 x_n 满足条件：$a = x_0 < x_1 < \ldots x_N < x_{N+1} = b$，通常设定相邻点之间具有相同的间距 h，即 $x_n = x_0 + nh$。求解方法可分为两类：单步方法和多步方法。

单步方法

单步方法根据 x_n、$\mathbf{u}_n$ 和步长 $h = x_{n+1} - x_n$ 计算 $\mathbf{u}_{n+1}$，一般形式为：

$$\mathbf{u}_{n+1} = \mathbf{u}_n + h\Phi(x_n, \mathbf{u}_n; h)$$

定义局部离散化误差 $\tau(x, \mathbf{u}; h) = \Phi(x, \mathbf{u}; h) - \frac{\mathbf{y}(x+h)-\mathbf{y}(x)}{h}$。

若对于一种单步方法，$\lim_{h\to 0} \tau(x, \mathbf{u}; h) = \mathbf{0}$，则称其为一致的。

若 $\tau(x, \mathbf{u}; h) = O(h^p)$ 当 $h \to 0$，则称这种方法的阶为 p。

欧拉方法

欧拉方法是一种简单的单步方法，仅使用一个点。欧拉方法的阶是 1。

显式欧拉方法的公式是：$\mathbf{u}_{n+1} = \mathbf{u}_n + h\mathbf{f}(x_n, \mathbf{u}_n)$。

隐式欧拉方法的公式是：$\mathbf{u}_{n+1} = \mathbf{u}_n + h\mathbf{f}(x_n, \mathbf{u}_{n+1})$。隐式方法需要通过迭代求解非线性方程，计算量大于显式方法，但具有更好的稳定性。

Runge-Kutta 方法

Runge-Kutta 方法使用多个点以提高计算结果的精确度。显式 Runge-Kutta 方法的定义是：

$$\Phi(x_n, \mathbf{u}_n, h) = \sum_{s=1}^{r} \alpha_s \mathbf{k}_s,$$

$$\mathbf{k}_1(x_n, \mathbf{u}_n) = \mathbf{f}(x_n, \mathbf{u}_n), \quad \mathbf{k}_s(x_n, \mathbf{u}_n, h) = \mathbf{f}(x_n + \mu_s h, \mathbf{u}_n + h \sum_{j=1}^{r} \lambda_{sj} \mathbf{k}_j), \quad s = 2, 3, ..., r.$$

Runge-Kutta 方法

显式 4 阶 Runge-Kutta 方法的公式是

$$\Phi(x_n, \mathbf{u}_n, h) = \frac{1}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4),$$

$$\mathbf{k}_1(x_n, \mathbf{u}_n) = \mathbf{f}(x_n, \mathbf{u}_n),\ \mathbf{k}_2(x_n, \mathbf{u}_n, h) = \mathbf{f}(x_n + \frac{1}{2}h, \mathbf{u}_n + \frac{1}{2}h\mathbf{k}_1),$$

$$\mathbf{k}_3(x_n, \mathbf{u}_n, h) = \mathbf{f}(x_n + \frac{1}{2}h, \mathbf{u}_n + \frac{1}{2}h\mathbf{k}_2),\ \mathbf{k}_4(x_n, \mathbf{u}_n, h) = \mathbf{f}(x_n + h, \mathbf{u}_n + h\mathbf{k}_3)$$

Runge-Kutta 方法

隐式 Runge-Kutta 方法的定义是：

$$\Phi(x_n, \mathbf{u}_n, h) = \sum_{s=1}^{r} \alpha_s \mathbf{k}_s(x_n, \mathbf{u}_n, h),$$

$$\mathbf{k}_s(x_n, \mathbf{u}_n, h) = \mathbf{f}(x_n + \mu_s h, \mathbf{u}_n + h \sum_{j=1}^{r} \lambda_{sj} \mathbf{k}_j), \quad s = 1, 2, ..., r.$$

半隐式 Runge-Kutta 方法的定义和隐式方法基本相同，唯一的区别是第二个公式中的求和项的下标范围改为从 $j = 1$ 到 s。隐式方法和半隐式方法都需要求解非线性方程组以确定 $\mathbf{k}_s$，计算量大于显式方法，但具有更好的稳定性。

Runge-Kutta 方法

单步方法的步长 h 如果太大，则局部离散化误差过大，导致计算结果不准确。步长 h 如果太小，则需要计算的步数太多，导致计算代价太大，并且舍入误差也会随着步数的增加不断积累。

为了最优化步长，Runge-Kutta-Fehlberg 方法在每一步使用两种阶数 (4 阶和 5 阶) 的 Runge-Kutta 方法，根据当前步的估计误差动态调整下一步的步长。

多步方法

线性多步方法的一般形式是：

$$\mathbf{u}_{n+k}+\alpha_{k-1}\mathbf{u}_{n+k-1}+...+\alpha_0\mathbf{u}_n = h[\beta_k\mathbf{f}_{n+k}+\beta_{k-1}\mathbf{f}_{n+k-1}+...+\beta_0\mathbf{f}_n], \quad n=0,1,2,...,N-k,$$

$$\mathbf{f}_r = \mathbf{f}(x_r,\mathbf{u}_r), \quad x_r = a+rh, \quad r=0,1,2,...,N$$

其中 $\alpha_i(0\le i\le k-1)$ 和 $\beta_i(0\le i\le k)$ 是给定的系数。

若 α_0 和 β_0 不同时为 0，则以上公式称为 k 步方法。若 β_k 为 0，即需要计算的 $\mathbf{u}_{n+k}$ 只出现在公式的左边，则以上公式称为显式的，否则称为隐式的。在获得相同的计算精确度的前提下，隐式方法可以比显式方法使用更大的步长 h，但需要通过迭代求解非线性方程。

初值问题的数值求解方法

scipy 库的 integrate 模块定义的 solve_ivp 函数使用数值方法求解初值问题，

$$\frac{d\mathbf{y}}{dt} = \mathbf{f}(t, \mathbf{y}), \quad t_0 \le x \le t_f; \quad \mathbf{y}(t_0) = \mathbf{y}_0.$$

它的第一个参数 fun 是方程的右边表达式，第二个参数 t_span 是一个包含 t_0 和 t_f 的元组，第三个参数 y0 是 $\mathbf{y}_0$，第四个参数 method 指定求解方法 (表)，第五个参数 t_eval 指定一个取值范围在 t_span 内的数组表示返回的求解结果的自变量值。

solve_ivp 函数的参数 method

方法名称	方法含义
'RK45'	显式 Runge-Kutta 方法，误差控制基于 4 阶方法，步长基于 5 阶方法。
'RK23'	显式 Runge-Kutta 方法，误差控制基于 2 阶方法，步长基于 3 阶方法。
'DOP853'	显式 8 阶 Runge-Kutta 方法，精确度高于前两种方法。
'Radau'	隐式 Radau IIA 5 阶 Runge-Kutta 方法。
'BDF'	隐式多步可变阶方法。

表: method 参数可指定的求解方法：前三种仅适用于非刚性微分方程，后两种适用于刚性微分方程。

Holling-Tanner 模型

Holling-Tanner 模型是一种描述捕食者 (predator) 和食饵 (prey) 的数量变化情况的模型：

$$\begin{aligned} \frac{dx}{dt} &= x(1-\frac{x}{7}) - \frac{6xy}{7+7x} \\ \frac{dy}{dt} &= 0.2y(1-\frac{Ny}{x}) \end{aligned}$$

方程中 $x(t) \neq 0$ 和 $y(t)$ 分别表示捕食者和食饵在时刻 t 的数量。$x(1-\frac{x}{7})$ 表示在无捕食者时食饵数量的增长速率，$\frac{6xy}{7+7x}$ 表示捕食者对食饵的捕食效果。$0.2y(1-\frac{Ny}{x})$ 表示当 x 个食饵可以最多支撑 x/N 个捕食者时捕食者数量的增长速率。

Holling-Tanner 模型

程序 11.9 输出的图的右子图显示：当 $N = 0.5$ 时，不论 x 和 y 的初值如何，捕食者和食饵的数量变化情况都会呈现周期性的增长和减少。

左子图显示了相平面上 $x(t)$ 和 $y(t)$ 的轨迹最终形成了一个稳定极限环。

洛伦兹方程

洛伦兹方程 (Lorenz equation) 是描述对流流体运动的一个简化模型：

$$\begin{aligned} \frac{dx}{dt} &= \sigma(y - x) \\ \frac{dy}{dt} &= rx - y - xz \\ \frac{dz}{dt} &= xy - bz \end{aligned}$$

方程中 x 表示对流翻转的速率 (风速)，y 表示水平方向的温度变化，z 表示竖直方向的温度变化，σ 表示与流体粘度相关的 Prandtl 数， r 表示与温差相关的 Rayleigh 数，b 是一个尺度因子。对于某些参数值，方程的解显示了混沌现象。

洛伦兹方程

程序 11.10 输出的图的两个子图展示了由两组参数值生成的三维相空间的两个轨迹，其特点是轨迹以不可预测的方式环绕两个吸引子，环绕的具体形式对初值条件敏感，但环绕的总体形式和初值条件无关。

边值问题的数值求解方法

求解边值问题的基本方法有两种：打靶法 (shooting method) 和有限差分法 (finite difference method)。

- 打靶法将边值问题转换为多个初值问题进行求解。
- 有限差分法用差商近似计算导数，将边值问题转换为线性方程组进行求解。

打靶法

使用打靶法求解以下二阶常微分方程的边值问题。

$$y'' = f(x, y, y'), \quad y(a) = \alpha, \quad y(b) = \beta$$

设 $y'(a) = s$，则以条件 $y'(a) = s$ 和 $y(a) = \alpha$ 可以求解一个初值问题，用 $F(x, s)$ 表示它的解。原问题转换为一个代数方程，即求解满足方程 $F(b, s) = \beta$ 的 s，可以使用二分法和牛顿法等方法迭代求解，在求解过程中对每个 s 的迭代值都需要求解一个对应的初值问题。

得到代数方程的解 $s = t$ 以后再以条件 $y'(a) = t$ 和 $y(a) = \alpha$ 求解初值问题，即可得到边值问题的解。

有限差分法

使用有限差分法求解以下二阶常微分方程的边值问题。

$$-y'' + q(x)y = g(x), \quad y(a) = \alpha, \quad y(b) = \beta$$

在区间 $[a, b]$ 上选取 $n+2$ 个等间距的点：
$a = x_0 < x_1 < ...x_n < x_{n+1} = b$，相邻两个点之间的间距 $h = (b-a)/(n+1)$。在点 $x_j (j = 1, ..., n)$ 处使用中心差分公式近似计算二阶导数

$$y''(x_j) \approx \frac{y(x_{j+1}) - 2y(x_j) + y(x_{j-1})}{h^2}$$

可得到如下的线性方程组：

有限差分法

$$\frac{1}{h^2}\begin{pmatrix} 2+g(x_1)h^2 & -1 & & 0 \\ -1 & 2+g(x_2)h^2 & \ddots & \\ & \ddots & \ddots & -1 \\ 0 & & -1 & 2+g(x_n)h^2 \end{pmatrix}\begin{pmatrix} u_1 \\ u_2 \\ \vdots \\ u_n \end{pmatrix} = \begin{pmatrix} g(x_1)+\frac{\alpha}{h^2} \\ g(x_2) \\ \vdots \\ g(x_{n-1}) \\ g(x_n)+\frac{\beta}{h^2} \end{pmatrix}$$

求解该线性方程组得到的向量 $(u_1, u_2, ..., u_n)^T$ 即为 $y(x_1), y(x_2), ..., y(x_n)$ 的近似值。

边值问题求解示例

以下边值问题：

$$-y'' + 400y = 800x - 400\cos^2 \pi x - 2\pi^2 \cos 2\pi x, \quad y(0) = 0, \quad y(1) = 2$$

的解析解是

$$y(x) = 2x + \frac{e^{-20}}{1 + e^{-20}} e^{20x} + \frac{1}{1 + e^{-20}} e^{-20x} - \cos^2 \pi x$$

程序 11.11 分别使用打靶法和有限差分法求解。输出的图的左子图中用红色圆点标示了打靶法的求解结果，右子图中用红色圆点标示了有限差分法的求解结果。两个子图中的绿色曲线绘制了解析解的函数图像。

实验 11： SciPy 库简介

本实验的目的是掌握使用 SciPy 库进行插值、代数方程求解和常微分方程求解的方法。

在 Blackboard 系统提交一个文本文件 (txt 后缀)，文件中记录每道题的源程序和运行结果。

1. 插值

使用一元三次样条函数对函数 $y = (5\sin(\frac{3x}{8}) + 3)\cos(\frac{-x^2}{9})$ 上横坐标由数组 np.linspace(0, 10, num=21, endpoint=True) 指定的 21 个数据点进行插值。绘制这些数据点和插值函数的曲线，并标注图例。

2. 求解非线性方程组

使用 optimize 模块的 root 函数求解以下非线性方程组，选择 Levenberg-Marquardt 算法并设置关键字实参 jac 为 True。

$$y\sin(x) = 4 \tag{2}$$

$$xy - x = 5 \tag{3}$$

答案：
[1.6581955 4.015326]

3. 求解常微分方程的初值问题

使用 solve_ivp 函数求解实验 10 第 3 题，t_eval 参数是 np.linspace(0, 10, 1000)。绘制数值解的数据点和解析解的曲线，并标注图例。