数值计算实验报告

崔晨*(201605130115)
山东大学计算机科学与技术学院菁英班

2018 年 12 月

摘要

本文是数值计算课程的实验报告，涉及非线性方程的求解、线性方程组的数值解、插值与多项式逼近、曲线拟合、数值微积分、微分方程求解等算法，对于大部分算法都给出了详细的推导，运行过程示例以及针对某一特定问题的解。

Contents

*邮箱：chentsuei@gmail.com，指导教师：刘保东老师

1 综述

数值计算指有效使用数字计算机求数学问题近似解的方法与过程, 主要研究如何利用计算机更好的解决各种数学问题, 包括连续系统离散化和离散形方程的求解, 并考虑误差、收敛性和稳定性等问题.

从数学类型来分, 数值运算的研究领域包括数值逼近、数值微分和数值积分、数值代数、最优化方法、常微分方程数值解法、积分方程数值解法、偏微分方程数值解法、计算几何、计算概率统计等. 随着计算机的广泛应用和发展, 许多计算领域的问题, 如计算物理、计算力学、计算化学、计算经济学等都可归结为数值计算问题.

本学期实验对非线性方程的求解、线性方程组的数值解、插值与多项式逼近、曲线拟合、数值微积分、微分方程求解等算法进行总结, 并用 Python 语言编程实现，每个算法都给出了详尽的说明和精美的图表。

2 第 1 章

2.1 题目 1

根据以下方法构造算法和 MATLAB 程序，以便精确计算所有情况下的二次方程的根，包括 $|b| \approx \sqrt{b^2 - 4ac}$ 的情况。

分析

设 $a \neq 0, b^2 - 4ac > 0$，且有方程 $ax^2 + bx + c = 0$，则通过如下二次根公式可解出方程的根：

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \qquad x_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \tag{1}$$

通过将分子有理化，可以等价变换成下列公式

$$x_1 = \frac{-2c}{b + \sqrt{b^2 - 4ac}} \qquad x_2 = \frac{-2c}{b - \sqrt{b^2 - 4ac}} \tag{2}$$

当 $|b| \approx \sqrt{b^2 - 4ac}$，必须小心处理，以避免其值过小而引起巨量消失（catastrophic cancellation）而带来精度损失。

- 当 $b > 0$ 的时候应使用公式 (2) 计算 x_1，应使用公式 (1) 计算 x_2。
- 当 $b < 0$ 的时候应使用公式 (1) 计算 x_1，应使用公式 (2) 计算 x_2。

实验结果

方程 $x2 + 2.001x + 1 = 0$: $x0 = -0.968873270798, x1 = -1.032126729202$.
方程 $x2 - 1000.001x + 1 = 0$: $x0 = 1000.000000000000, x1 = 0.001000000000$.
方程 $x2 - 1000.0001x + 1 = 0$: $x0 = 999.999099999100, x1 = 0.001000000900$.
方程 $x2 - 1000.00001x + 1 = 0$: $x0 = 999.999009999010, x1 = 0.001000000990$.
方程 $x2 - 1000.000001x + 1 = 0$: $x0 = 999.999000999001, x1 = 0.001000000999$.

代码

```
def solve_quad(a, b, c):
    delta = b * b - 4 * a * c
    if delta < 0:
        return None
    elif delta == 0:
        return [-b / (2 * a)]
    elif b > 0:
        return [-2 * c / (b + np.sqrt(delta)),
                (-b - np.sqrt(delta)) / (2 * a)]
    elif b < 0:
        return [(-b + np.sqrt(delta)) / (2 * a),
                -2 * c / (b - np.sqrt(delta))]

def disp_solve_quad(a, b, c):
    res = solve_quad(a, b, c)
    x = sp.Symbol('x')
    f = a * x * x + b * x + c
    if res is None:
        display(Math('方程 %s = 0 无解.' % sp.latex(f)))
    elif len(res) == 1:
        display(Math('方程 %s = 0 有一解: x = %.12f.'
                     % (sp.latex(f), res[0])))
```

```
    else:
        display(
            Math('方程 %s = 0 有两解: x_0 = %.12f, x_1 = %.12f.'
                 % (sp.latex(f), res[0], res[1])))

disp_solve_quad(1, -1000.001, 1)
disp_solve_quad(1, -1000.0001, 1)
disp_solve_quad(1, -1000.00001, 1)
disp_solve_quad(1, -1000.000001, 1)
```

2.2 题目 2

对下列 3 个差分方程计算出前十个数值近似值。在每种情况下引入一个小的初始误差。如果没有初始误差，则每个差分方程将生成序列 $\{1/2^n\}_{n=1}^{\infty}$，构造误差表和误差图。

1. $r_0 = 0.994, r_n = \frac{1}{2}r_{n-1}, n = 1, 2, \cdots$
2. $p_0 = 1, p_1 = 0.497, p_n = \frac{3}{2}p_{n-1} - \frac{1}{2}p_{n-2}, n = 2, 3, \cdots$
3. $q_0 = 1, q_1 = 0.497, q_n = \frac{5}{2}q_{n-1} - q_{n-2}, n = 2, 3, \cdots$

基础知识：误差

1. 误差的来源由于计算机中二进制数精度有限, 存在截断误差 (10 进制与 2 进制互相转化, 2 进制计算);
2. 误差的类型

   截断误差: 通常指的是, 用一个基本表达式替换一个相当复杂的算术表达式时, 所引入的误差. 这个术语从用截断泰勒级数替换一个复杂表达式的技术衍生而来.

   舍入误差: 计算机表示的示数受限于尾数的固定精度, 因此有时并不能确切地表示真实值, 这一类型的误差称为舍入误差.
3. 误差度量方法: 设 $\hat{p}$ 是 p 的近似值,

   相对误差

   $$R_p = \frac{|p - \hat{p}|}{p}, p \neq 0$$

   绝对误差

   $$E_p = |p - \hat{p}|$$

当 $|p|$ 远离 1 时 (大于或小于), 相对误差 R_p 比误差 E_p 能更好地表示近似值的精确程度.

基础知识：误差的收敛阶

序列的收敛阶设 $\lim\limits_{n\to\infty} x_n = x$, 有序列 $\{r_n\}_{n=1}^{\infty}$, 且 $\lim\limits_{n\to\infty} r_n = 0$. 如果存在常量 $K > 0$, 满足 $\frac{|x_n - x|}{|r_n|} \leq K$, n 足够大, 则称 $\{x_n\}_{n=1}^{\infty}$ 以收敛阶 $O(r_n)$ 收敛于 x. 可以将其表示为 $x_n = x + O(r_n)$, 或表示为 $x_n \to x$, 收敛阶为 $O(r_n)$.

求解：生成序列

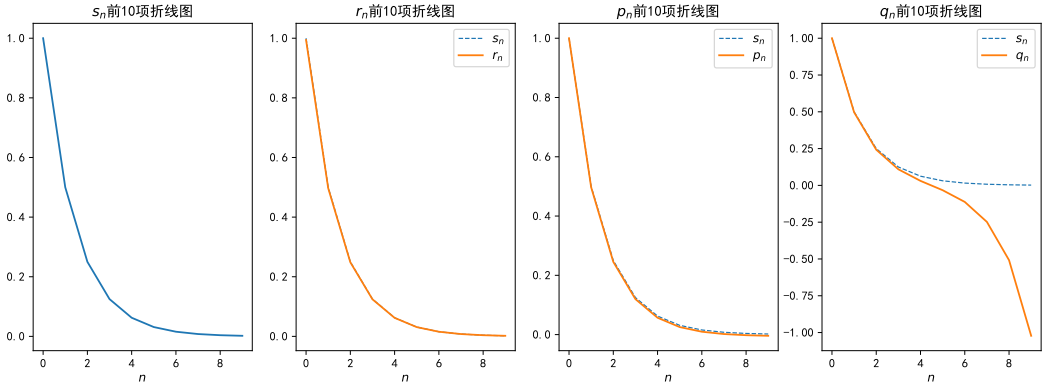
设 $\{s_n\} = \left\{1/2^n\right\}_{n=1}^{\infty}$ 为标准序列，生成得到前 10 项序列值：

Table 1: 生成序列表

n	s_n	r_n	p_n	q_n
1	1.000000000000	0.994000000000	1.000000000000	1.000000000000
2	0.500000000000	0.497000000000	0.497000000000	0.497000000000
3	0.250000000000	0.248500000000	0.245500000000	0.242500000000
4	0.125000000000	0.124250000000	0.119750000000	0.109250000000
5	0.062500000000	0.062125000000	0.056875000000	0.030625000000
6	0.031250000000	0.031062500000	0.025437500000	-0.032687500000
7	0.015625000000	0.015531250000	0.009718750000	-0.112343750000
8	0.007812500000	0.007765625000	0.001859375000	-0.248171875000
9	0.003906250000	0.003882812500	-0.002070312500	-0.508085937500
10	0.001953125000	0.001941406250	-0.004035156250	-1.022042968750

使用折线图将四个序列进行可视化，

Figure 1: 生成序列的折线图



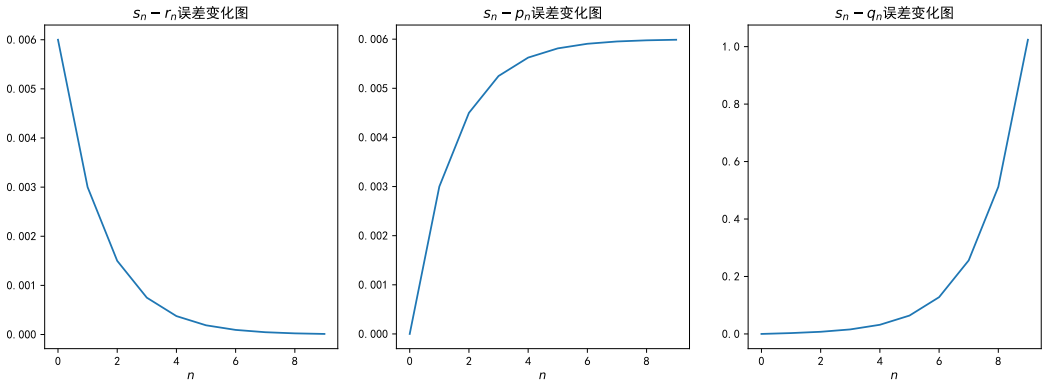
数列 $\{s_n\}$ 与 $\{r_n\}$,$\{p_n\}$,$\{q_n\}$ 之差的前 10 项：

Table 2: 生成序列误差表

n	$s_n - r_n$	$s_n - p_n$	$s_n - q_n$
1	0.006000000000	0.000000000000	0.000000000000
2	0.003000000000	0.003000000000	0.003000000000
3	0.001500000000	0.004500000000	0.007500000000
4	0.000750000000	0.005250000000	0.015750000000
5	0.000375000000	0.005625000000	0.031875000000
6	0.000187500000	0.005812500000	0.063937500000
7	0.000093750000	0.005906250000	0.127968750000
8	0.000046875000	0.005953125000	0.255984375000
9	0.000023437500	0.005976562500	0.511992187500
10	0.000011718750	0.005988281250	1.023996093750

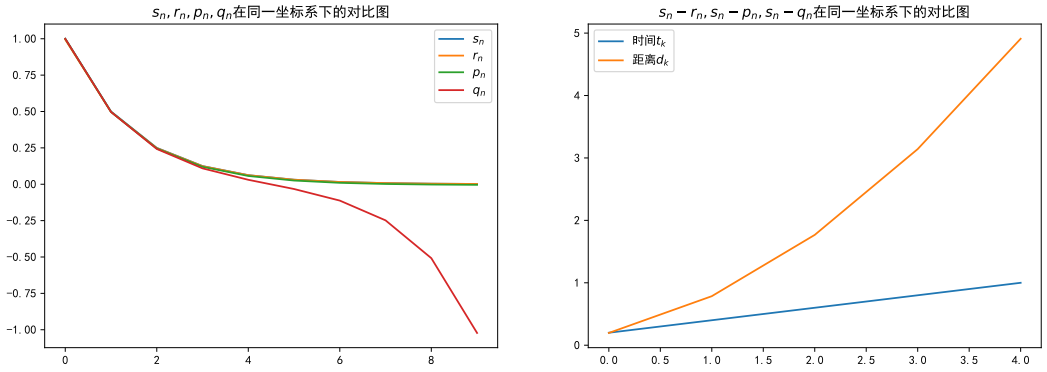
绘制每个序列的误差变化折线图，

Figure 2: 生成序列的误差变化折线图



将其放在同一坐标系下，

Figure 3: 同一坐标系对比图



分析 由图表可以发现 p_n 和 r_n 都可以较好地生成需要的序列，而 q_n 却随着 n 的增加离需要构造的序列真实值越来越远。

3 第 2 章

3.1 题目 1

利用牛顿法求解方程

$$\frac{1}{2} + \frac{1}{4}x^2 - x\sin x - \frac{1}{2}\cos 2x = 0$$

分别取 $x_0 = \frac{\pi}{2}, 5\pi, 10\pi$, 使精度不超过 10^{-5}, 比较初值对计算结果的影响。

牛顿法

推导：假设 $f \in C^2[a,b]$, 并且 x^* 是 $f(x)=0$ 的一个解.

令 $\overline{x} \in [a,b]$ 是对 x_* 的一个近似, 使得 $f'(\overline{x}) \neq 0$ 且 $|\overline{x}-x^*|$ 比较小. 考虑 $f(x)$ 在 $\overline{x}$ 处展开的一阶泰勒多项式 $f(x)=f(\overline{x})+(x-\overline{x})f'(\overline{x})+\frac{(x-\overline{x})^2}{2}f''(\xi(x))$, 其中 $\xi(x)$ 在 x 和 $\overline{x}$ 之间. 因为 $f(x^*)=0$, 令 $x=x^*$, 此时有 $0=f(x^*)=f(\overline{x})+(x^*-\overline{x})f'(\overline{x})+\frac{(x^*-\overline{x})^2}{2}f''(\xi(x))$. 忽略余项, 得到 $0=f(x^*)\approx f(\overline{x})+(x^*-\overline{x})f'(\overline{x})$

求得

$$x^* \approx \overline{x}-\frac{f(\overline{x})}{f'(\overline{x})}$$

因此定义迭代序列为: $x_n=x_{n-1}-\frac{f(x_{n-1})}{f'(x_{n-1})}, \forall n \geq 1.$

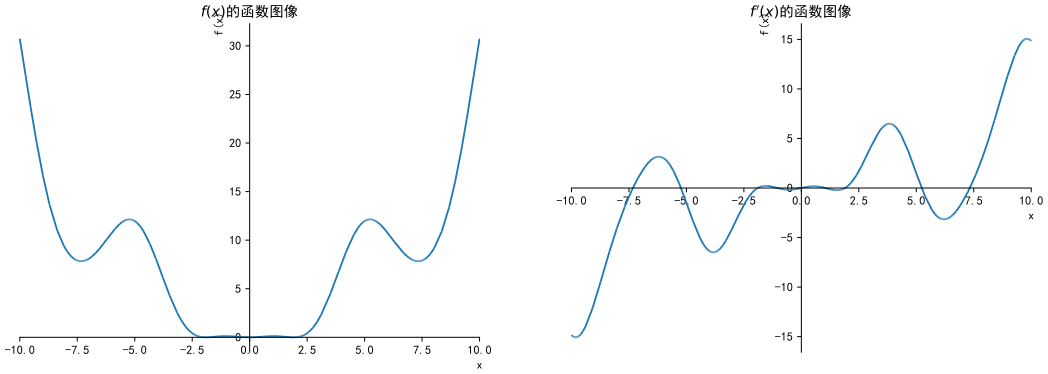
求解

使用 Python 的 `sympy` 包求得 $f(x)=\frac{x^2}{4}-x\sin(x)-\frac{1}{2}\cos(2x)+\frac{1}{2}$ 的导函数为

$$f'(x)=-x\cos(x)+\frac{x}{2}-\sin(x)+\sin(2x)$$

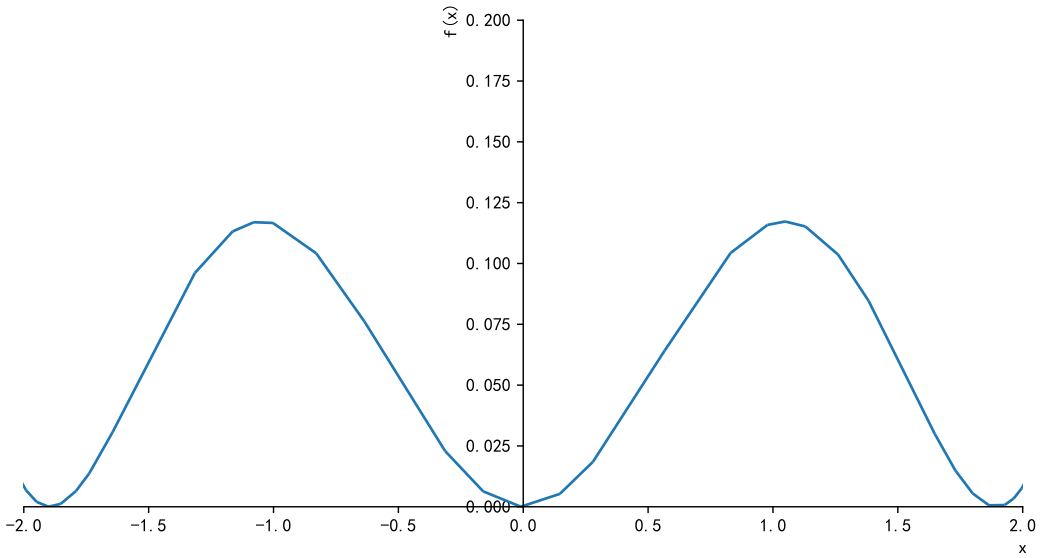
然后我们做出 $f(x)=\frac{x^2}{4}-x\sin(x)-\frac{1}{2}\cos(2x)+\frac{1}{2}$ 及其导函数的图像进行观察。

Figure 4: $f(x)=\frac{x^2}{4}-x\sin(x)-\frac{1}{2}\cos(2x)+\frac{1}{2}$ 及其导函数图像



观察图像可知，函数零点在 $(-2,2)$ 这段区间上，因此我们将其放大进行观察：

Figure 5: $f(x)$ 在 $(-2,2)$ 区间上的放大图

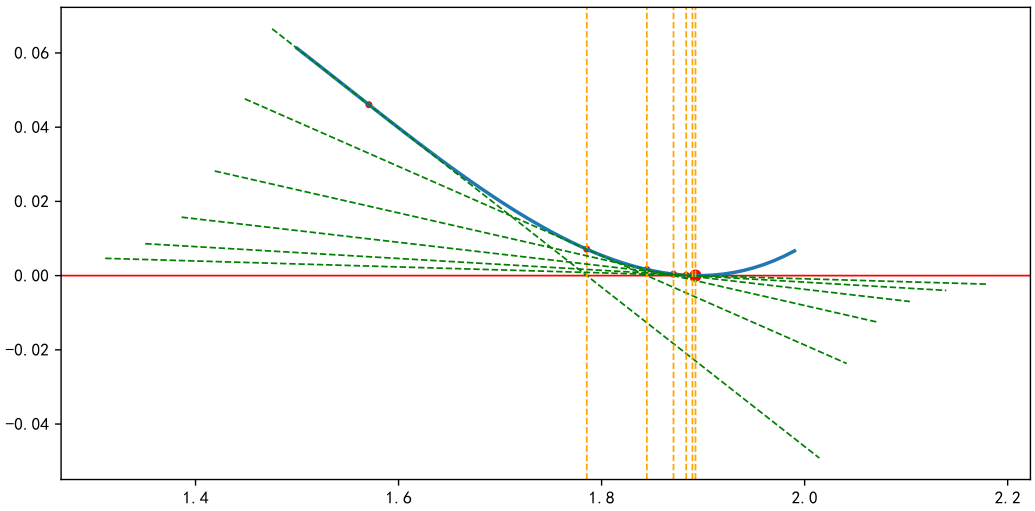


然后使用牛顿法求解方程，分别取 $x_0 = \frac{\pi}{2}, 5\pi, 10\pi$。

$x_0 = \frac{\pi}{2}$ **时** 在 6 次迭代后找到零点

$$x = 1.8924896245342444$$

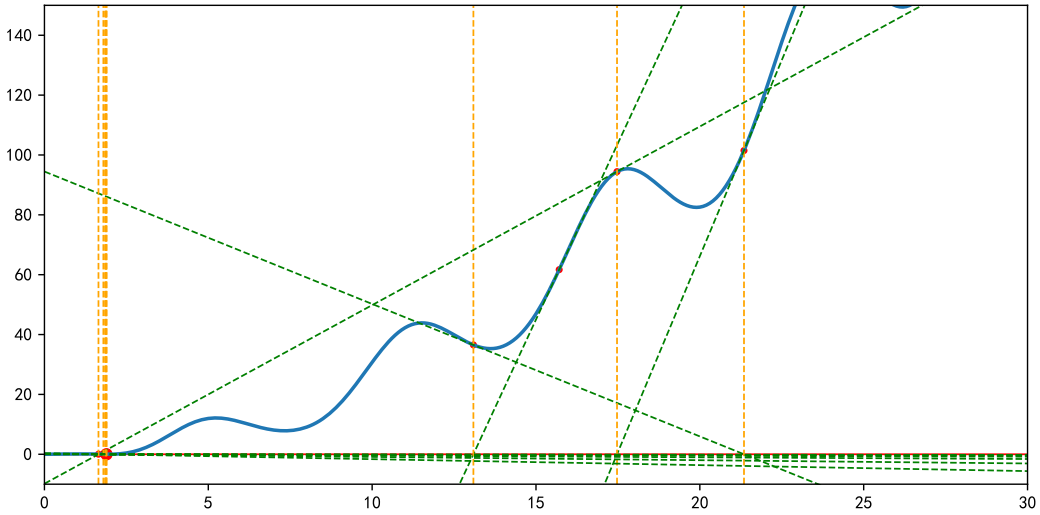
Figure 6: $x_0 = \pi/2$ 时牛顿法的过程图



$x_0 = 5\pi$ **时** 在 10 次迭代后找到零点

$$x = 1.8927898018266247$$

Figure 7: $x_0 = 5\pi$ 时牛顿法的过程图



$x_0 = 10\pi$ **时** 在 10313 次迭代后找到零点

$$x = 1.898094316843809$$

由于当 $x_0 = 10\pi$ 时时迭代次数过多，计算量过大，所以难以将计算过程进行可视化，仅计算其结果，不进行可视化。

代码

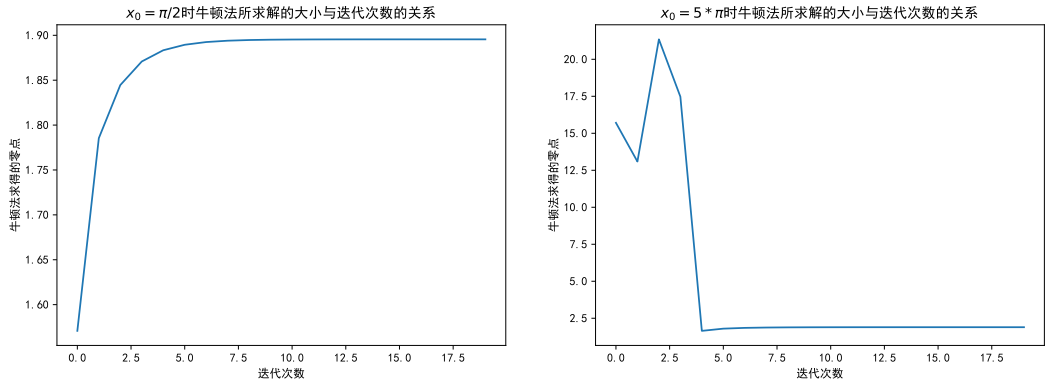
```
def newton(f, df, x0, eps, max_iter, ax=None, px=None):
    x = x0
    if ax is not None:
        ax.axhline(y=0, color='red', linestyle='-', linewidth=1)
        ax.plot(px, f(px), linewidth=2)

    for n in range(max_iter):
        fx = f(x)
```

```
        if ax is not None:
            ax.scatter(x, fx, c='red', marker='.')
        if abs(fx) < eps:
            print('在%d次迭代后找到解.' % n)
            if ax is not None:
                ax.scatter(x, fx, c='red')
            return x
        dfx = df(x)
        if ax is not None:
            abline(ax, x, fx, dfx)
        if dfx == 0:
            print('导数值为 0，无法找到解.')
            return None
        x = x - fx / dfx
        if ax is not None:
            ax.axvline(x=x, linestyle='--', c='orange',
                       linewidth=1)
    print('超过最大迭代次数，无法找到解.')
    return None

px = np.arange(1.5, 2.0, 0.01)
display(Math('找到零点 x = %s.' %
  newton(f_eval, df_eval, np.pi / 2, 1e-5, 100, ax, px)))
px = np.arange(0, 30, 0.1)
plt.xlim(0, 30)
plt.ylim(-10, 150)
display(Math('找到零点 x = %s.' %
  newton(f_eval, df_eval, 5 * np.pi, 1e-5, 100, ax, px)))
display(Math('找到零点 x = %s.' %
  newton(f_eval, df_eval, 10 * np.pi, 1e-5, 20000)))
```

解与迭代次数的关系探究

下面对解和迭代次数的关系进行探究（以初值 $x_0 = \frac{\pi}{2}$ 为例）：

Figure 8: 解与迭代次数的关系图



随着迭代次数的增加，解的大小的绝对量变化越来越小；初值的选择对于迭代的次数与解的质量有很大关系。

3.2 题目 2

已知

$$f(x) = 5x - e^x$$

在 $(0,1)$ 之间有一个实根，试分别用二分法、牛顿法、割线法、错位法设计相应的计算格式，并编程求解。

二分法

如果 $f \in C(a,b)$，且存在数 $r \in [a,b]$，满足 $f(r) = 0$。如果 $f(a)f(b) < 0$ 则在区间 $[a,b]$ 内有奇数个零点，若只有一个零点，可以递归用下述方法找到零点：

每次取中点 z, 若 $f(z) = 0$, 则 z 就是零点。如果 $f(z)f(a) < 0$ 则区间变换到 $[a,z]$，否则区间变为 $[z,b]$。

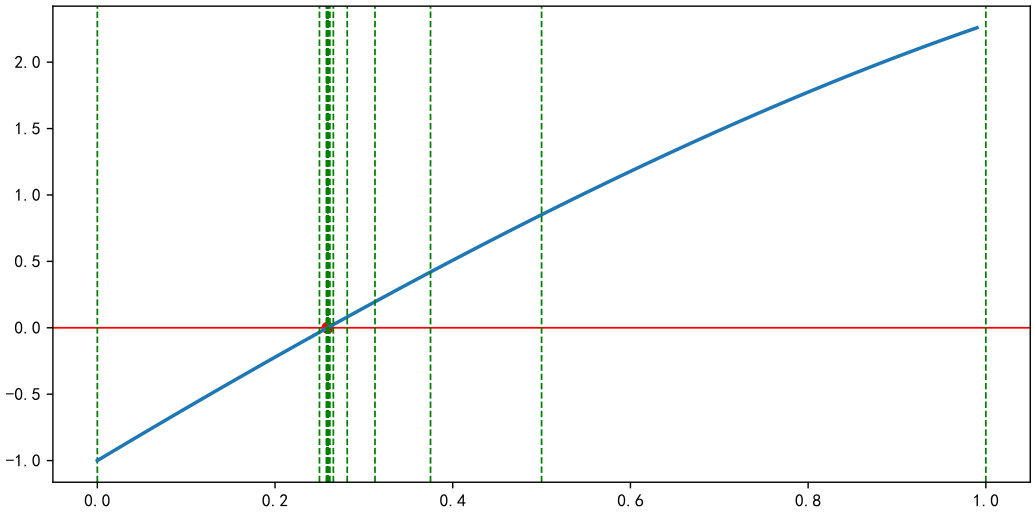
从上述递归中容易知道，每次都将零点存在的区间缩小一倍，所以如果在区间 $[a,b]$ 上，如果迭代 n 次，可以得到精度为：$(b-a)/2^{n+1}$

因此，当 $n \to \infty$ 的时候，上式右边趋于 0，所以得到 $r \to c_n$，所以只要 n 足够大，最后一定会收敛到目标点。

求解

使用二分法进行求解得到根 0.25917110181907377

Figure 9: 二分法求解过程图



二分法代码

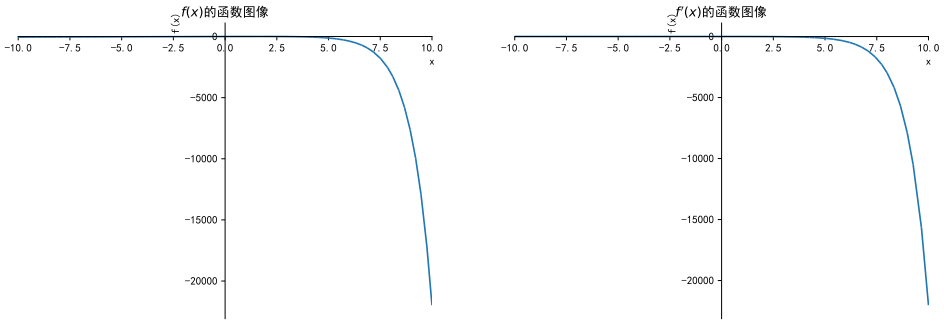
```
def bisection(f, a, b, max_iter, ax=None, px=None):
    if ax is not None:
        ax.axhline(y=0, color='red', linestyle='-', linewidth=1)
        ax.axvline(x=a, linestyle='--', c='green', linewidth=1)
        ax.axvline(x=b, linestyle='--', c='green', linewidth=1)
        ax.plot(px, f(px), linewidth=2)
    if f(a) * f(b) >= 0:
        print(" 二分法失败.")
        return None
    for _ in range(max_iter):
        c = (a + b) / 2
        if ax is not None:
            ax.axvline(x=c, linestyle='--', c='green', linewidth=1)

        fc = f(c)
        if f(a) * fc < 0:
            b = c
```

```
        elif f(b) * fc < 0:
            a = c
        elif fc == 0:
            print(" 找到准确的解.")
            return c
        else:
            print(" 二分法失败.")
            return None
    if ax is not None:
        ax.scatter((a + b) / 2, 0, c='red')
    return (a + b) / 2
```

牛顿法

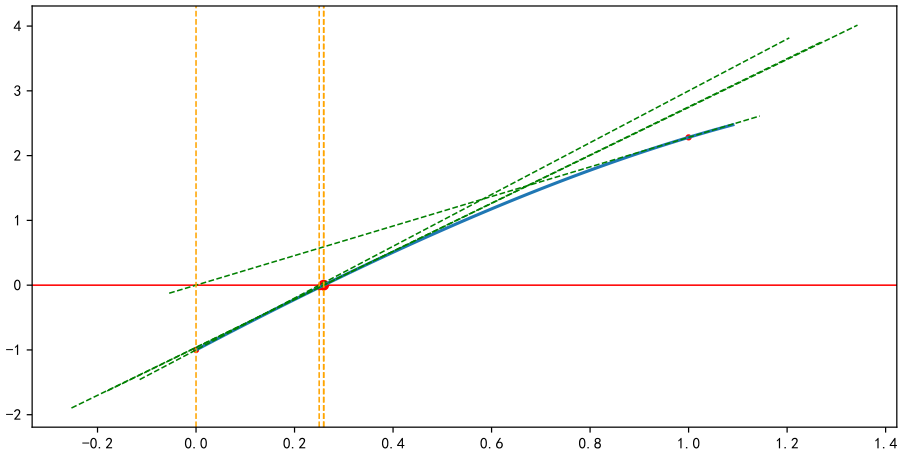
首先使用 Python 的 `sympy` 包对函数 $f(x) = 5x - e^x$ 进行求导，得 $f(x) = 5x - e^x$ 的导函数为 $f'(x) = -e^x + 5$，然后对其图像进行观察。

Figure 10: $f(x)$ 与 $f'(x)$ 的函数图像



运行牛顿法在 4 次迭代后找到解 0.2591711017819102。

Figure 11: 牛顿法过程图



牛顿法代码 牛顿法代码请见题目 1。

```
px = np.arange(0, 1.1, 0.01)
newton(f_eval, df_eval, 1, 1e-5, 100, ax, px)
```

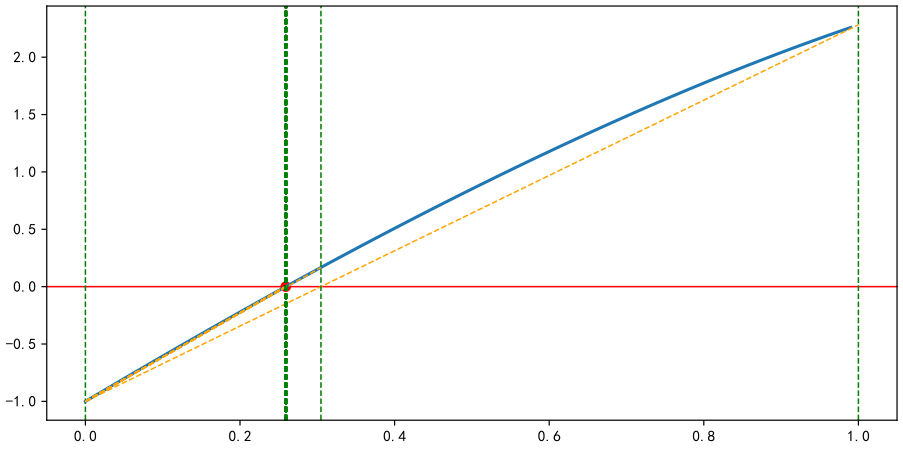
割线法

推导: 用割线近似代替牛顿法中的切线.

得到公式 $x_{k+1} = x_k - f(x_k)\frac{x_k - x_{k-1}}{f(x_k) - f(x_{k-1})}$。

运行割线法找到解 0.25917110181907377。

Figure 12: 割线法过程图



割线法代码

```
def secant(f, a, b, max_iter, ax=None, px=None):
    if ax is not None:
        ax.axhline(y=0, color='red', linestyle='-', linewidth=1)
        ax.axvline(x=a, linestyle='--', c='green', linewidth=1)
        ax.axvline(x=b, linestyle='--', c='green', linewidth=1)
        ax.plot(px, f(px), linewidth=2)
    if f(a) * f(b) >= 0:
        print(" 割线法失败.")
        return None
    for _ in range(max_iter):
        fa, fb = f(a), f(b)
        c = a - fa * (b - a) / (fb - fa)
        if ax is not None:
            ax.plot([a, b], [fa, fb], c='orange', linestyle='--',
                    linewidth=1)
            ax.axvline(x=c, linestyle='--', c='green', linewidth=1)
        fc = f(c)
        if fa * fc < 0:
            b = c
        elif fb * fc < 0:
            a = c
        elif fc == 0:
            print(" 找到准确的解.")
            return c
        else:
            print(" 割线法失败.")
            return None
    if ax is not None:
        ax.scatter(a - f(a) * (b - a) / (f(b) - f(a)), 0, c='red')
    return a - f(a) * (b - a) / (f(b) - f(a))
```

错位法

推导：要在区间 $[p_0, p_1]$ 上寻找 $f(x) = 0$ 的解, 其中 $f(p_0) f(p_1) < 0$.

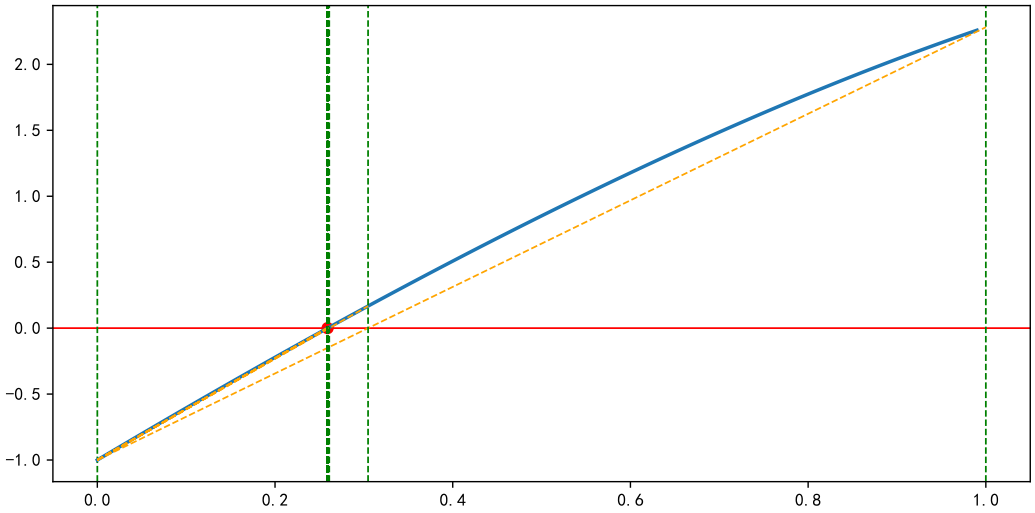
使用与切线法相同的方式, 找到近似点 p_2.

为确定使用哪一条线计算 p_3, 计算 $f(p_2) f(p_1)$ 和 $f(p_2) f(p_0)$.

如果 $f(p_2) f(p_1) < 0$, 说明 p_1, p_2 中间包含了一个根, 那么取 $(p_1, f(p_1)), (p_2, f(p_2))$ 线段的斜率, 作为切线法中的斜率.

运行错位法找到解 0.25917110181907377。

Figure 13: 错位法过程图



错位法代码

```
def secant(f, a, b, max_iter, ax=None, px=None):
    if ax is not None:
        ax.axhline(y=0, color='red', linestyle='-', linewidth=1)
        ax.axvline(x=a, linestyle='--', c='green', linewidth=1)
        ax.axvline(x=b, linestyle='--', c='green', linewidth=1)
        ax.plot(px, f(px), linewidth=2)
    if f(a) * f(b) >= 0:
        print(" 割线法失败.")
        return None
```

```
    for _ in range(max_iter):
        fa, fb = f(a), f(b)
        c = a - fa * (b - a) / (fb - fa)
        if ax is not None:
            ax.plot([a, b], [fa, fb], c='orange', linestyle='--',
                    linewidth=1)
            ax.axvline(x=c, linestyle='--', c='green', linewidth=1)
        fc = f(c)
        if fa * fc < 0:
            b = c
        elif fb * fc < 0:
            a = c
        elif fc == 0:
            print(" 找到准确的解.")
            return c
        else:
            print(" 割线法失败.")
            return None
    if ax is not None:
        ax.scatter(a - f(a) * (b - a) / (f(b) - f(a)), 0, c='red')
    return a - f(a) * (b - a) / (f(b) - f(a))

fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(10, 5))
px = np.arange(0, 1, 0.01)
regula(f_eval, 0, 1, 100, ax, px)
```

4 第三章

4.1 题目 1

求解线性方程组

$$4x - y + z = 7 \tag{1}$$

$$4x - 8y + z = -21 \tag{2}$$

$$-2x + y + 5z = 15 \tag{3}$$

1. 使用 LU 分解求此方程组
2. 分别用 Jacobi，Gauss-Seidel 方法求解此方程组

4.1.1 题目 1.1

使用 LU 分解求此方程组。

LU 分解

给定一个无行交换的高斯消去法可求解一般线性方程组 $AX = B$，则矩阵 A 可分解为一个下三角矩阵 L 和一个上三角矩阵 U 的乘积:

$$A = LU$$

而且 L 的对角线元素为 1，U 的对角线元素非零。得到 LU 矩阵之后，原方程变为：

$$LUX = B$$

这时可以先把 UX 当成新的未知数，即求解 $LX_{new} = B$ 得到 $X_{new}w$ 之后，再求解 $UX = X_{new}$ 最终得到 X.

分解矩阵

利用已知求未知。最开始已知 L 的对角线全是 1，所以用 L 的第一行分别与 U 的第一列、第二列…第 n 列相乘，得到 U 的第一行。然后用 L 的第二行与 U 的第一列相乘求得 L 中第二行的所有元素，然后用 L 的第二行分别与 U 的所有列相乘求得 U 的第二行，然后一直重复这样的步骤，就可以得到分解的 LU 矩阵。

求解矩阵 X_{new}

用 L 的第一行乘以 X_{new} 的列，求得 X_{new} 的第一行，然后用 L 的第二行乘以 X_{new} 的列，求得 X_{new} 的第二行。重复这样的步骤就可以得到 X_{new} 的所有值。

求解矩阵 X

类似 X_{new} 的求法，但是是从 U 矩阵的最后一行向上分别和 X 的列相乘。LU 分解的代码和分解之后求解方程组的代码:

设 $\mathrm{A} = \left(a_{i,j}\right)_{n\times n}$, 分解为 $\mathrm{A} = \mathrm{LU}$ 的形式, 其中 L 为对角元为 1 的下三角矩阵, U 为上三角矩阵. 即

$$\begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix} = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ l_{21} & 1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1} & l_{n2} & \cdots & 1 \end{pmatrix} \begin{pmatrix} u_{11} & u_{12} & \cdots & u_{1n} \\ 0 & u_{22} & \cdots & u_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & u_{nn} \end{pmatrix}$$

按照矩阵乘法规则, 比较系数, 可得

$$\begin{cases} u_{1j} = a_{1j} & (j = 1, 2, \cdots, n) \\ l_{i1} = \dfrac{a_{i1}}{u_{11}} & (i = 2, 3, \cdots, n) \\ u_{ij} = a_{ij} - \displaystyle\sum_{k=1}^{i-1} l_{ik}u_{kj} & (i = 2, \cdots, n, j = i, \cdots, n) \\ l_{ij} = \left(a_{ij} - \displaystyle\sum_{k=1}^{j-1} l_{ik}u_{kj}\right) & (j = 1, 2, \cdots, n, i = j+1, \cdots, n) \end{cases}$$

分解之后的矩阵:

$$\mathrm{L} = \begin{pmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ -0.5 & -0.0714 & 1 \end{pmatrix} \mathrm{U} = \begin{pmatrix} 4 & -1 & 1 \\ 0 & -7 & 0 \\ 0 & 0 & 5.5 \end{pmatrix}$$

方程的解:

1. $x_1 = 2$;
2. $x_2 = 4$;
3. $x_3 = 3$;

LU 分解代码

```
A = np.array([[4, -1, 1], [4, -8, 1], [-2, 1, 5]])
B = np.array([7, -21, 15]).reshape(-1, 1)
```

```
def pivot_matrix(A):
    m = A.shape[0]
    I = np.identity(m)
    for j in range(m):
        row = max(range(j, m), key=lambda i: abs(A[i, j]))
        if j != row:
            I[[j, row]] = I[[row, j]]
    return I

def lu_decomposition(A):
    n = A.shape[0]
    L = np.zeros((n, n), dtype=float)
    U = np.zeros((n, n), dtype=float)
    A = np.dot(pivot_matrix(A), A)
    for i in range(n):
        for k in range(i, n):
            s = sum(L[i][j] * U[j][k] for j in range(i))
            U[i][k] = A[i][k] - s
        for k in range(i, n):
            if i == k:
                L[i][i] = 1
            else:
                s = sum(L[k][j] * U[j][i] for j in range(i))
                L[k][i] = (A[k][i] - s) / U[i][i]
    return A, L, U

AB = np.column_stack([A, B])
AB, L, U = lu_decomposition(AB)

UX = backsubL(L, B).reshape(-1, 1)
X = backsubU(U, UX).reshape(-1, 1)
```

4.1.2 题目 1.2

分别用 Jacobi，Gauss-Seidel 方法求解此方程组。

推导 假设方程组

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n = b_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n = b_2 \\ \quad\quad \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n = b_1 \end{cases}$$

的系数矩阵 A 非奇异, 不妨设 $a_{ii} \neq 0\,(i = 1, 2, \cdots, n)$. 将方程组变形为:

$$\begin{cases} x_1 = \dfrac{1}{a_{11}}\left(-a_{12}x_2 - a_{13}x_3 - \cdots - a_{1n}x_n + b_1\right) \\ x_2 = \dfrac{1}{a_{22}}\left(-a_{21}x_1 - a_{23}x_3 - \cdots - a_{2n}x_n + b_2\right) \\ \quad\quad \vdots \\ x_n = \dfrac{1}{a_{nn}}\left(-a_{n1}x_1 - a_{n2}x_2 - \cdots - a_{n,n-1}x_{n-1} + b_n\right) \end{cases}$$

建立迭代公式:

$$\begin{cases} x_1^{(k+1)} = \dfrac{1}{a_{11}}\left(-a_{12}x_2^{(k)} - a_{13}x_3^{(k)} - \cdots - a_{1n}x_n^{(k)} + b_1\right) \\ x_2^{(k+1)} = \dfrac{1}{a_{22}}\left(-a_{21}x_1^{(k)} - a_{23}x_3^{(k)} - \cdots - a_{2n}x_n^{(k)} + b_2\right) \\ \quad\quad \vdots \\ x_n^{(k+1)} = \dfrac{1}{a_{nn}}\left(-a_{n1}x_1^{(k)} - a_{n2}x_2^{(k)} - \cdots - a_{n,n-1}x_{n-1}^{(k)} + b_n\right) \end{cases}$$

选定初始向量 $\mathrm{x}^{(0)}$ 后, 反复迭代可以得到向量序列 $\left\{\mathrm{x}^{(k)}\right\}$ 迭代公式为:

$$\begin{cases} \mathrm{x}^{(0)} = \left(x_1^{(0)}, x_2^{(0),\cdots,x_n^{(0)}}\right)^T \\ x_i^{(k+1)} = \dfrac{1}{a_{ii}}\left(b_i - \displaystyle\sum_{j=1, j\neq i}^{n} a_{ij}x_j^{(k)}\right) \end{cases}$$

Jacobi 迭代法也可以写为向量递推的形式

$$\mathrm{x}^{(k)} = \mathrm{T}\mathrm{x}^{(k)} + \mathrm{c}$$

设 D 是对角元与 A 相同的对角阵; –L 是严格下三角矩阵, 下三角元素与 A 相同; –U 是严格上三角矩阵, 上三角元素与 A 相同. 因此 $\mathrm{A} = \mathrm{D} - \mathrm{L} - \mathrm{U}$.

对于方程 $\mathrm{Ax} = \mathrm{b}$, 或 $(\mathrm{D} - \mathrm{L} - \mathrm{U})\,\mathrm{x} = \mathrm{b}$,

可以变形为

$$\mathrm{Dx} = (\mathrm{L} + \mathrm{U})\,\mathrm{x} + \mathrm{b}$$

也就是

$$\mathrm{x} = \mathrm{D}^{-1}\,(\mathrm{L} + \mathrm{U})\,\mathrm{x} + \mathrm{D}^{-1}\mathrm{b}$$

写成递推式的形式:

$$\mathrm{x}^{(k)} = \mathrm{D}^{-1}\,(\mathrm{L} + \mathrm{U})\,\mathrm{x}^{(k-1)} + \mathrm{D}^{-1}\mathrm{b}$$

分析 由方程组可得:$x = \frac{7+y+z}{4}$,$y = \frac{21+4*x+z}{8}$,$z = \frac{15+2*x-y}{5}$ 这样就提出了下列 Jacobi 迭代过程：

$$\begin{cases} x_{k+1} = \dfrac{7 + 1 * y_k + z_k}{4} \\ y_{k+1} = \dfrac{21 + 4 * x_k + z_k}{8} \\ z_{k+1} = \dfrac{15 + 2 * x_k - y_k}{5} \end{cases}$$

$\mathrm{x} = (2, 4, 3)$, 与求解线性方程组结果一致。

Jacobi 迭代的解随迭代次数的变化 下面对 Jacobi 迭代的解随迭代次数的变化进行探究，运行 Jacobi 迭代程序，得到下表。

Table 3: Jacobi 迭代的解随迭代次数的变化表

迭代次数	x_0	x_1	x_2
0	0.000000000000	0.000000000000	0.000000000000
1	1.750000000000	2.625000000000	3.000000000000
2	1.656250000000	3.875000000000	3.175000000000
3	1.925000000000	3.850000000000	2.887500000000
4	1.990625000000	3.948437500000	3.000000000000
5	1.987109375000	3.995312500000	3.006562500000
6	1.997187500000	3.994375000000	2.995781250000
7	1.999648437500	3.998066406250	3.000000000000
8	1.999516601563	3.999824218750	3.000246093750
9	1.999894531250	3.999789062500	2.999841796875
10	1.999986816406	3.999927490234	3.000000000000
11	1.999981872559	3.999993408203	3.000009228516
12	1.999996044922	3.999992089844	2.999994067383
13	1.999999505615	3.999997280884	3.000000000000
14	1.999999320221	3.999999752808	3.000000346069
15	1.999999851685	3.999999703369	2.999999777527
16	1.999999981461	3.999999898033	3.000000000000
17	1.999999974508	3.999999990730	3.000000012978
18	1.999999994438	3.999999988876	2.999999991657
19	1.999999999305	3.999999996176	3.000000000000
20	1.999999999044	3.999999999652	3.000000000487
21	1.999999999791	3.999999999583	2.999999999687
22	1.999999999974	3.999999999857	3.000000000000
23	1.999999999964	3.999999999987	3.000000000018
24	1.999999999992	3.999999999984	2.999999999988
25	1.999999999999	3.999999999995	3.000000000000
26	1.999999999999	4.000000000000	3.000000000001
27	2.000000000000	3.999999999999	3.000000000000
28	2.000000000000	4.000000000000	3.000000000000
29	2.000000000000	4.000000000000	3.000000000000

Figure 14: Jacobi 迭代的解随迭代次数的变化

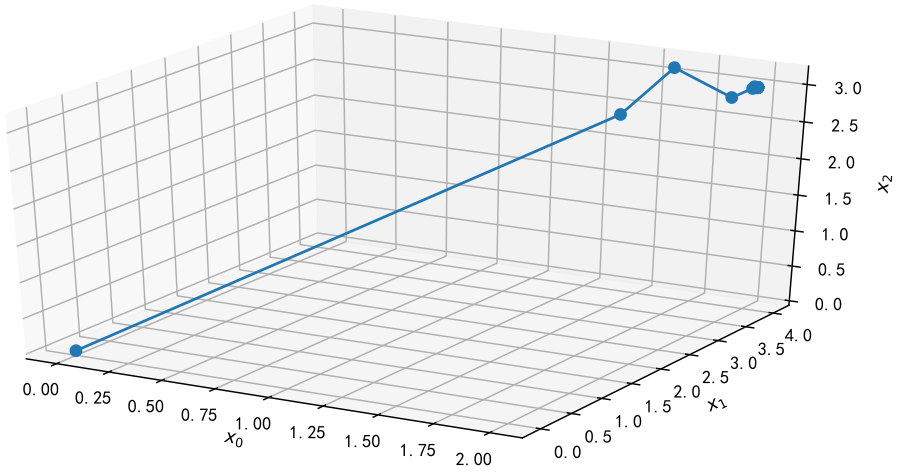
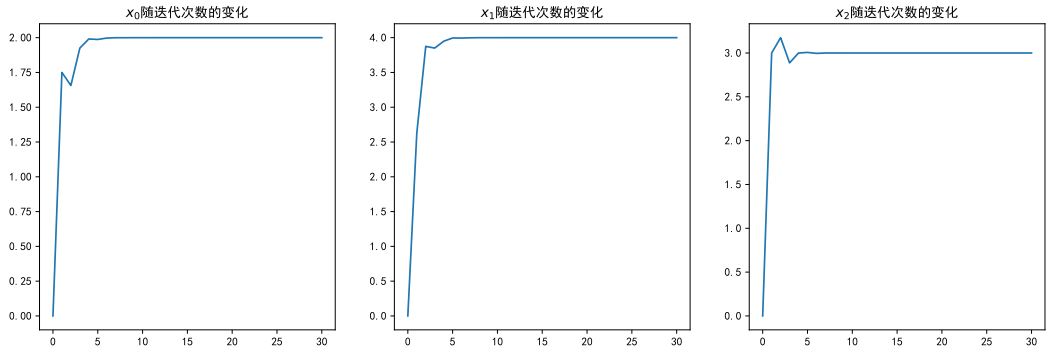


Figure 15: Jacobi 迭代的每个解随迭代次数的变化



Jacobi 迭代代码

```
def jacobi(A, B, max_iter, x=None, ax=None):
    B = B.reshape(-1)
    if x is None:
        x = np.zeros_like(B)
    show = ax is not None and x.size == 3
    showx = [[] for _ in range(3)]
```

```
    D = np.diag(A)
    R = A - np.diagflat(D)
    if show:
        for i in range(len(showx)):
            showx[i].append(x[i])
    for _ in range(max_iter):
        x = (B - np.dot(R, x)) / D
        if show:
            for i in range(len(showx)):
                showx[i].append(x[i])
    if show:
        ax.plot(showx[0], showx[1], showx[2], marker='o')
        ax.set_xlabel('$x_0$')
        ax.set_ylabel('$x_1$')
        ax.set_zlabel('$x_2$')
        return x, np.array(showx)
    return x
```

Gauss-Seidel **方法**

如果把 Jacobi 迭代公式改成以下形式

$$\begin{cases} x_1^{(k+1)} = \dfrac{1}{a_{11}}\left(-a_{12}x_2^{(k)} - a_{13}x_3^{(k)} - \cdots - a_{1n}x_n^{(k)} + b_1\right) \\ x_2^{(k+1)} = \dfrac{1}{a_{22}}\left(-a_{21}x_1^{(k+1)} - a_{23}x_3^{(k)} - \cdots - a_{2n}x_n^{(k)} + b_2\right) \\ \qquad\qquad \vdots \\ x_n^{(k+1)} = \dfrac{1}{a_{nn}}\left(-a_{n1}x_1^{(k+1)} - a_{n2}x_2^{(k+1)} - \cdots - a_{n,n-1}x_{n-1}^{(k+1)} + b_n\right) \end{cases}$$

选取初始向量 $\mathrm{x}^{(0)}$, 用迭代公式:

$$\begin{cases} \mathrm{x}^{(0)} = \left(x_1^{(0)}, x_2^{(0),\cdots,x_n^{(0)}}\right)^T \\ x_i^{(k+1)} = \dfrac{1}{a_{ii}}\left(b_i - \displaystyle\sum_{j=1}^{i-1} a_{ij}x_j^{(k+1)} - \sum_{j=i+1}^{n} a_{ij}x_j^{(k)}\right) \end{cases}$$

Gauss-Seidel 迭代法也可以写为向量递推的形式:

$$\mathrm{x}^{(k)} = \mathrm{T}_g \mathrm{x}^{(k)} + \mathrm{c}_g$$

或

$$(\mathrm{D} - \mathrm{L})\,\mathrm{x}^{(k)} = \mathrm{U}\mathrm{x}^{(k-1)} + \mathrm{b}$$

即

$$\mathrm{x}^{(k)} = (\mathrm{D} - \mathrm{L})^{-1}\,\mathrm{U}\mathrm{x}^{k-1} + (\mathrm{D} - \mathrm{L})^{-1}\,\mathrm{b}$$

由方程组可得:$x = \frac{7+y+z}{4}$,$y = \frac{21+4*x+z}{8}$,$z = \frac{15+2*x-y}{5}$ 这样就提出了下列 Gauss-Seidel 迭代过程：

$$\begin{cases} x_{k+1} = \dfrac{7 + 1 * y_k + z_k}{4} \\ y_{k+1} = \dfrac{21 + 4 * x_{k+1} + z_k}{8} \\ z_{k+1} = \dfrac{15 + 2 * x_{k+1} - y_{k+1}}{5} \end{cases}$$

Jacobi 方法没有利用刚计算出来的值，而 Gauss-Seidel 方法利用了刚计算出来的值，Gauss-Seildel 方法对于相同的精度应该需要的迭代次数应该比 Jacobi 方法的要少，即收敛得更快。实际上因为有 Stein-Rosenberg 定理有: 如果对每个 $i\ j, a_{ij}\ 0$, 且对每个 $i = 1, 2, ..., n\ a_{ii} > 0$, 那么一下结论有且只有一个成立:

$$\begin{aligned} 0 \le p(T_g) < p(T_g) < 1 \\ 1 < p(T_j) < p(T_g) \\ p(T_j) = p(T_g) = 0 \\ p(T_j) = p(T_g) = 1 \end{aligned}$$

Gauss-Seidel 方法的解随迭代次数的变化　下面是 Gauss-Seidel 方法的解随迭代次数的变化表与变化图：

Table 4: Gauss-Seidel 方法的解随迭代次数的变化表

迭代次数	x_0	x_1	x_2
0	0.000000000000	0.000000000000	0.000000000000
1	1.750000000000	3.500000000000	3.000000000000
2	1.875000000000	3.937500000000	2.962500000000
3	1.993750000000	3.992187500000	2.999062500000
4	1.998281250000	3.999023437500	2.999507812500
5	1.999878906250	3.999877929688	2.999975976563
6	1.999975488281	3.999984741211	2.999993247070
7	1.999997873535	3.999998092651	2.999999530884
8	1.999999640442	3.999999761581	2.999999903860
9	1.999999964430	3.999999970198	2.999999991733
10	1.999999994616	3.999999996275	2.999999998592
11	1.999999999421	3.999999999534	2.999999999861
12	1.999999999918	3.999999999942	2.999999999979
13	1.999999999991	3.999999999993	2.999999999998
14	1.999999999999	3.999999999999	3.000000000000
15	2.000000000000	4.000000000000	3.000000000000
16	2.000000000000	4.000000000000	3.000000000000
17	2.000000000000	4.000000000000	3.000000000000
18	2.000000000000	4.000000000000	3.000000000000
19	2.000000000000	4.000000000000	3.000000000000
20	2.000000000000	4.000000000000	3.000000000000
21	2.000000000000	4.000000000000	3.000000000000
22	2.000000000000	4.000000000000	3.000000000000
23	2.000000000000	4.000000000000	3.000000000000
24	2.000000000000	4.000000000000	3.000000000000
25	2.000000000000	4.000000000000	3.000000000000
26	2.000000000000	4.000000000000	3.000000000000
27	2.000000000000	4.000000000000	3.000000000000
28	2.000000000000	4.000000000000	3.000000000000
29	2.000000000000	4.000000000000	3.000000000000

Figure 16: Gauss-Seidel 方法的解随迭代次数的变化

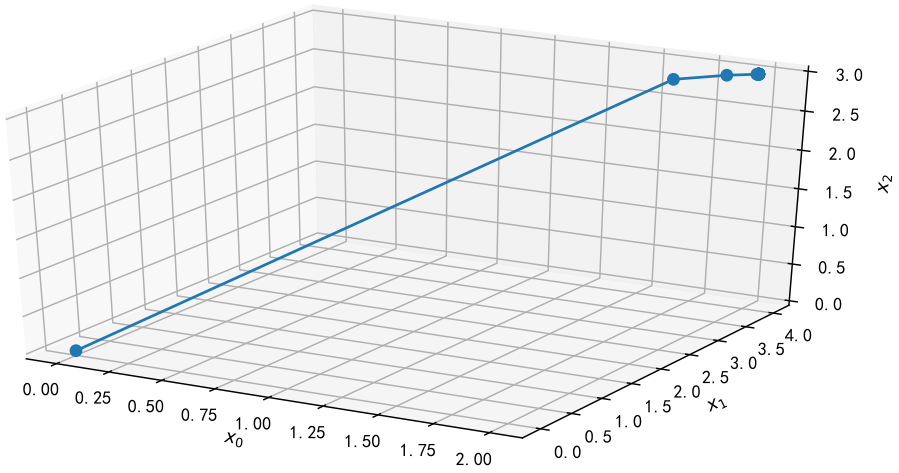
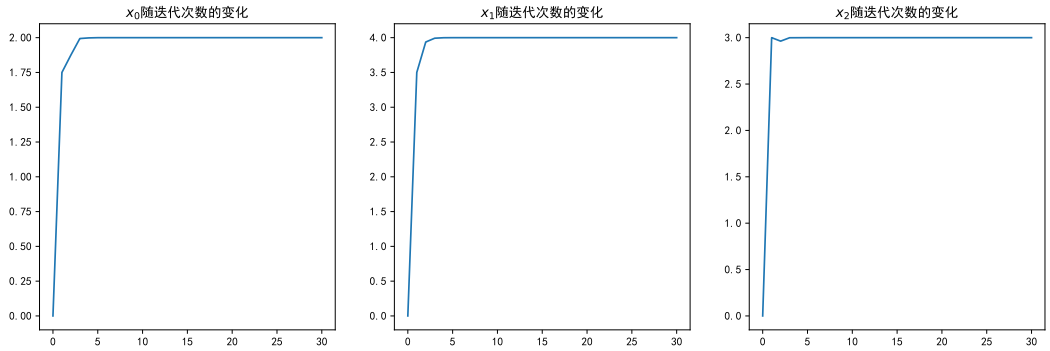


Figure 17: Gauss-Seidel 方法的每个解随迭代次数的变化



Gauss-Seidel 方法代码

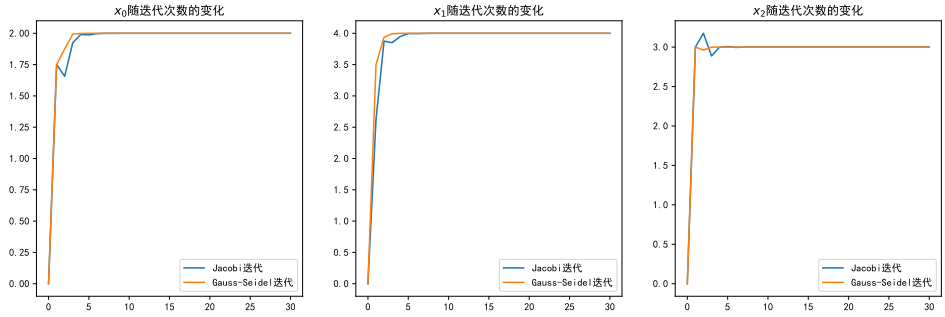
```
def gauss_seidel(A, B, max_iter, x=None, ax=None):
    B = B.reshape(-1)
    if x is None:
        x = np.zeros_like(B)
    show = ax is not None and x.size == 3
    showx = [[] for _ in range(3)]
```

```
    if show:
        for i in range(len(showx)):
            showx[i].append(x[i])
    for _ in range(max_iter):
        x_new = np.zeros_like(x)
        for i in range(A.shape[0]):
            s1 = np.dot(A[i, :i], x_new[:i])
            s2 = np.dot(A[i, i + 1:], x[i + 1:])
            x_new[i] = (B[i] - s1 - s2) / A[i, i]
        x = x_new
        if show:
            for i in range(len(showx)):
                showx[i].append(x[i])
    if show:
        ax.plot(showx[0], showx[1], showx[2], marker='o')
        ax.set_xlabel('$x_0$')
        ax.set_ylabel('$x_1$')
        ax.set_zlabel('$x_2$')
        return x, np.array(showx)
    return x
```

Jacobi 迭代与 Gauss-Seidel 方法对比

下图是 Jacobi 迭代与 Gauss-Seidel 方法在每个解上的对比图：

Figure 18: Jacobi 迭代与 Gauss-Seidel 方法对比图



4.2 题目 2

设有如下三角线性方程组，而且系数矩阵具有严格对角优势：

$$\begin{array}{ccccccccccc}
d_1x_1 & + & c_1x_2 & & & & & & & = & b_1 \\
a_1x_1 & + & d_2x_2 & + & c_2x_3 & & & & & = & b_2 \\
 & & a_2x_2 & + & d_3x_3 & + & c_3x_4 & & & = & b_3 \\
 & & \cdot & & & & & & & \cdot & \\
 & & & \cdot & & & & & & \cdot & \\
 & & & & \cdot & & & & & \cdot & \\
 & & & & & & a_{N-1}x_{N-1} & + & d_Nx_N & = & b_N
\end{array}$$

设计一个算法来求解上述方程组. 算法必须有效地利用系数矩阵的稀疏性。

分析

对第一行 $d_x x_1 + c_1 x_2 = b$，将其转变为 $x_1 + \frac{c_1}{d_1} x_2 = \frac{b_1}{d_1}$

对将第一行乘 a_1 与第二行相减，把第二行转变为

$$(d_2 - \frac{a_1 c_1}{d_1}) x_2 + c_2 x_3 = \frac{c_2}{d_2} - \frac{a_2 b_1}{d_1}$$

然后用第二行转化第三行，第三行转化第四行，以此类推。最终矩阵的形式为

$$\begin{pmatrix}
1 & r_1 & & & \\
 & 1 & \ddots & & \\
 & & 1 & \ddots & \\
 & & & \ddots & r_1 \\
 & & & & 1
\end{pmatrix}
\begin{pmatrix}
x_1 \\ x_2 \\ \vdots \\ x_{n-1} \\ x_n
\end{pmatrix} =
\begin{pmatrix}
\rho_1 \\ \rho_2 \\ \vdots \\ \rho_{n-1} \\ \rho_n
\end{pmatrix}$$

三对角矩阵是容易求解的。无论是多大的矩阵，利用托马斯算法我们可以仅仅使用两次迭代就求出了精确的解。如果使用迭代算法 Guass-Seidel 算法往往需要数十次的迭代。所以利用矩阵的稀疏性可以大大提高求解的效率。

4.3 题目 3

利用 Gauss-Seidel 迭代法求解下列带状方程。

$$AX = B$$

其中，

$$A = \begin{bmatrix} 12. & -2. & 1. & \dots & 0. & 0. & 0. \\ -2. & 12. & -2. & \dots & 0. & 0. & 0. \\ 1. & -2. & 12. & \dots & 0. & 0. & 0. \\ \dots & & & & & & \\ 0. & 0. & 0. & \dots & 12. & -2. & 1. \\ 0. & 0. & 0. & \dots & -2. & 12. & -2. \\ 0. & 0. & 0. & \dots & 1. & -2. & 12. \end{bmatrix} \quad B = \begin{bmatrix} 5. \\ 5. \\ 5. \\ \dots \\ 5. \\ 5. \\ 5. \end{bmatrix}$$

使用 Gauss-Seidel 方法解得

$$X = \begin{bmatrix} 0.46379552 & 0.53728461 & 0.50902292 & 0.49822163 & 0.49894186 & 0.49998535 \\ 0.50008872 & 0.50001532 & 0.49999479 & 0.49999786 & 0.50000011 & 0.5000002 \\ 0.50000002 & 0.49999999 & 0.5 & 0.5 & 0.5 & 0.5 \\ 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 \\ 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 \\ 0.5 & 0.5 & 0.5 & 0.5 & 0.5 & 0.5 \\ 0.49999999 & 0.50000002 & 0.5000002 & 0.50000011 & 0.49999786 & 0.49999479 \\ 0.50001532 & 0.50008872 & 0.49998535 & 0.49894186 & 0.49822163 & 0.50902292 \\ 0.53728461 & 0.46379552 & & & & \end{bmatrix}$$

Gauss-Seidel 方法求解代码

```
A = 12 * np.identity(50)
for i in range(50):
    if i + 1 < 50: A[i][i + 1] = -2
    if i + 2 < 50: A[i][i + 2] = 1
    if i - 1 >= 0: A[i][i - 1] = -2
    if i - 2 >= 0: A[i][i - 2] = 1
B = 5 * np.ones(50).reshape(-1, 1)
x = gauss_seidel(A, B, 30)
```

5 第四章

5.1 题目 1

在区间 $[-5,5]$ 上，生成 11 个等距插值节点 $x_i, i = 0,1,2,3,\cdots,10$。在相应的插值节点上计算函数

$$y(x) = \frac{1}{1+x^2}$$

的函数值作为观测值 $y(x_i), i = 1,2,3,\cdots,10$。

5.1.1 题目 1.1

利用这 11 个数据点，生成一个 10 次拉格朗日插值多项式 $P_{10}(x)$, 并作出插值函数与原函数的结果对比图。

拉格朗日插值多项式

对于给定的 $n+1$ 个点, n 次拉格朗日插值多项式以此通过这 $n+1$ 个点.

拉格朗日多项式形式如下:

$$P(x) = \sum_{k=0}^{n} L_{n,k}(x) f(x)$$

.

其中 $L_{n,k}(x)$ 为基函数, 满足这样的性质:

$$L_{n,k}(x_i) = \begin{cases} 0, & i \neq k \\ 1, & i = k \end{cases}$$

取

$$L_{n,k}(x) = \prod_{i=0, i\neq k}^{n} \frac{x - x_i}{x_k - x_i}$$

.

拉格朗日余项

$f(x) = P(x) + R(x)$, 其中 $P(x)$ 为 n 次多项式,

$$R(x) = \frac{f^{(n+1)}(\xi(x))}{(n+1)!} \prod_{i=0}^{n} (x - x_i)$$

为余项. 用于误差估计。

生成 11 个数据点

$$
\begin{aligned}
0 &: (-5, 0.038461538461538464) \\
1 &: (-4, 0.058823529411764705) \\
2 &: (-3, 0.1) \\
3 &: (-2, 0.2) \\
4 &: (-1, 0.5) \\
5 &: (0, 1.0) \\
6 &: (1, 0.5) \\
7 &: (2, 0.2) \\
8 &: (3, 0.1) \\
9 &: (4, 0.058823529411764705) \\
10 &: (5, 0.038461538461538464)
\end{aligned}
$$

生成代码

```
x = np.arange(-5, 6, 1)
y = 1 / (1 + x * x)
for i, (xi, yi) in enumerate(zip(x, y)):
display(Math("%s: %s" % (i, str((xi, yi)))))
```

插值函数与原函数的结果对比

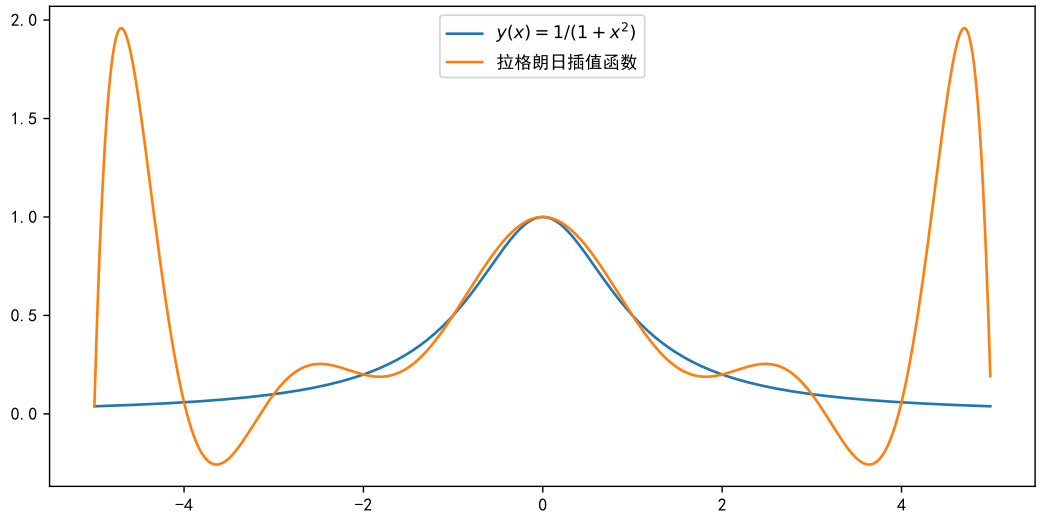
原函数为

$$f(x) = \frac{1}{1+x^2}$$

拉格朗日插值函数为

$$
\begin{aligned}
f(x) = &-2.262e-05x^{10} - 1.864e-19x^9 + 0.001267x^8 - 1.108e-17x^7 - 0.02441x^6 \\
&+1e-16x^5 + 0.1974x^4 - 7.284e-17x^3 - 0.6742x^2 - 1.448e-16x + 1
\end{aligned}
$$

Figure 19: 插值函数与原函数的结果对比图



拉格朗日插值代码

```
def lagrange(X, Y):
    w = len(x)
    n = w - 1
    L = np.zeros((w, w), dtype=float)
    for k in range(n + 1):
        V = 1
        for j in range(n + 1):
            if k != j:
                V = np.convolve(V, np.poly(X[j])) / (X[k] - X[j])
        L[k] = V
    C = np.dot(Y, L)
    return C

L = lagrange(x, y)
f = np.poly1d(L)
xs = np.arange(-5, 5, 0.01)
ys_true = 1 / (1 + xs * xs)
```

```
ys_pred = f(xs)
plt.figure(figsize=(10, 5))
plt.plot(xs, ys_true, label='$y(x)=1 / ({1+x^2)}$')
plt.plot(xs, ys_pred, label='拉格朗日插值函数')
plt.legend(loc='upper center')
plt.savefig('fig18.pdf', bbox_inches='tight')
```

5.1.2 题目 1.2

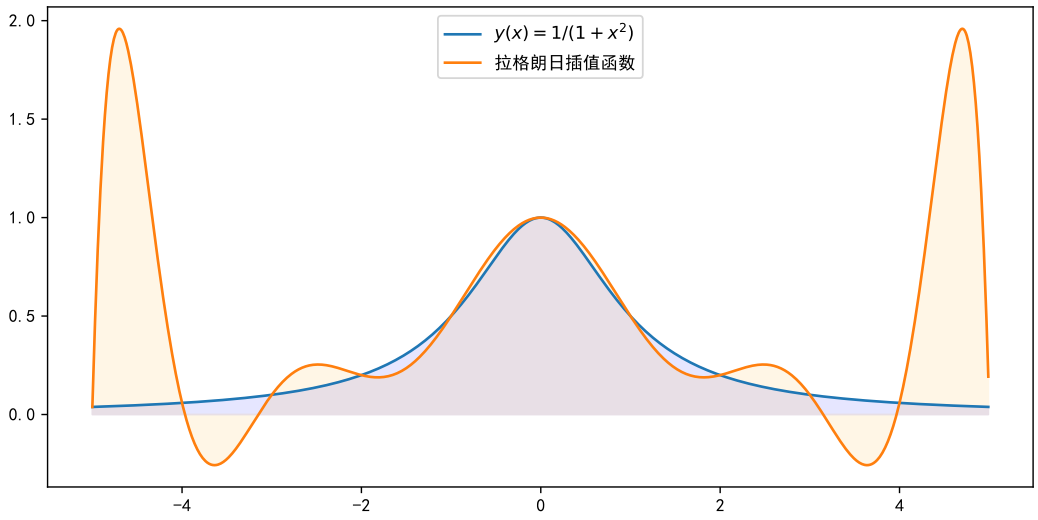
利用此多项式近似计算

$$\int_{-5}^{5} \frac{1}{1+x^2} dx \approx \int_{-5}^{5} P_{10}(x) dx$$

与解析解比较，分析误差产生的原因。

近似解与解析解可视化比较 下图是近似解与解析解的可视化比较：

Figure 20: 插值函数与原函数的结果对比图



解析解结果

$$\int_{-5}^{5} 1/(1+x^2) = [atan(x)]_{-5}^{5} = 2.746801533890032$$

解析解代码

```
x_sym = sp.Symbol('x')
f_sym = 1 / (1 + x_sym * x_sym)
F_sym = sp.integrate(f_sym, x_sym)
F_eval = sp.lambdify(x_sym, F_sym)
display(Math("%s = %s = %s" % ("\int_{-5}^{5}{1 / ({1+x^2})}",
             "[%s]_{-5}^{5}" % F_sym, F_eval(5) - F_eval(-5))))
```

近似解结果

积分结果为 4.673300555654807

近似解代码

```
_L = [l for l in L]
for i, l in enumerate(_L):
    _L[i] = l / (11 - i)
_L.append(0)
_f = np.poly1d(_L)
print("%s \n的积分是\n %s" % (f, _f))
display(Math(" 积分结果为%s" % (_f(5) - _f(-5))))
```

5.1.3 题目 1.3

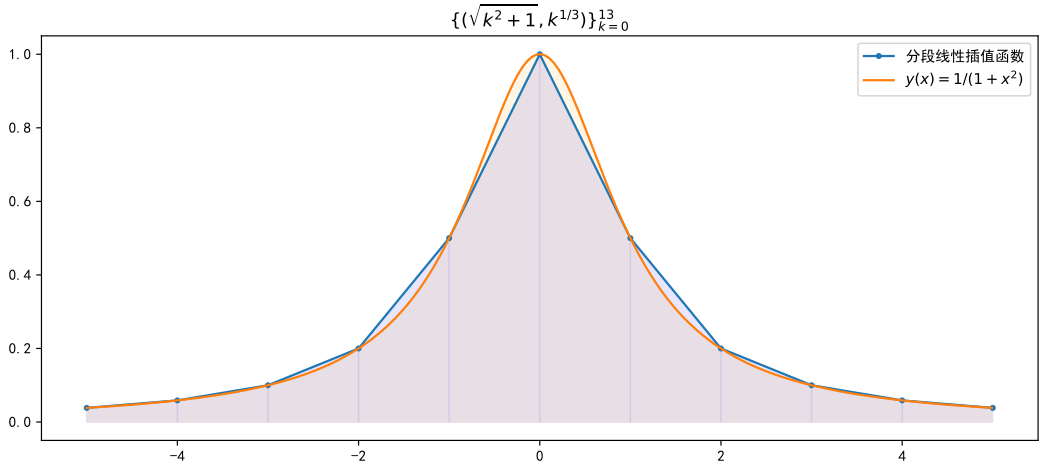
利用 $\{(x_i, y(x_i))\}_{i=0}^{10}$, 构造分片线性插值多项式 $P(x)$，并利用此分片插值多项式近似计算积分

$$\int_{-5}^{5} \frac{1}{1+x^2} dx \approx \int_{-5}^{5} P(x) dx$$

与解析解比较，分析误差产生的原因。

近似解与解析解可视化比较 分段线性插值函数与原函数的结果对比如图所示，

Figure 21: 分段线性插值函数与原函数的结果对比图



近似解结果

积分结果为 2.756108597285068

近似解代码

```
plt.figure(figsize=(12, 5))
plt.plot(x, y, marker='.', label='分段线性插值函数')
plt.title('${\{(\sqrt{k^2+1},k^{1/3})\}}_{k=0}^{13}$')
res = 0
for ki in range(1, len(x)):
    h = x[ki] - x[ki - 1]
    res += h / 2 * (y[ki - 1] + y[ki])
    plt.fill_between([x[ki], x[ki - 1]], [y[ki], y[ki - 1]],
                     color = "blue", alpha = 0.1)
plt.plot(xs, ys_true, label='$y(x)=1 / ({1+x^2)}$')
plt.fill_between(xs, ys_true, color = "orange", alpha = 0.1,
                 linewidth=1)
plt.legend(loc='upper right')
```

5.1.4 题目 1.4

若希望提高积分的计算精度，试提出你自己的建议，并进行实验测试验证。

分片线性插值增加分段数，可以有效的减少误差。Lagrange 插值如果分段过多会产生剧烈的龙格震荡现象，误差迅速增高。

拉格朗日插值与分段线性插值在不同分段数的对比 拉格朗日插值与分段线性插值在不同分段数的对比如下表所示，

Table 5: 拉格朗日插值与分段线性插值在不同分段数的对比表

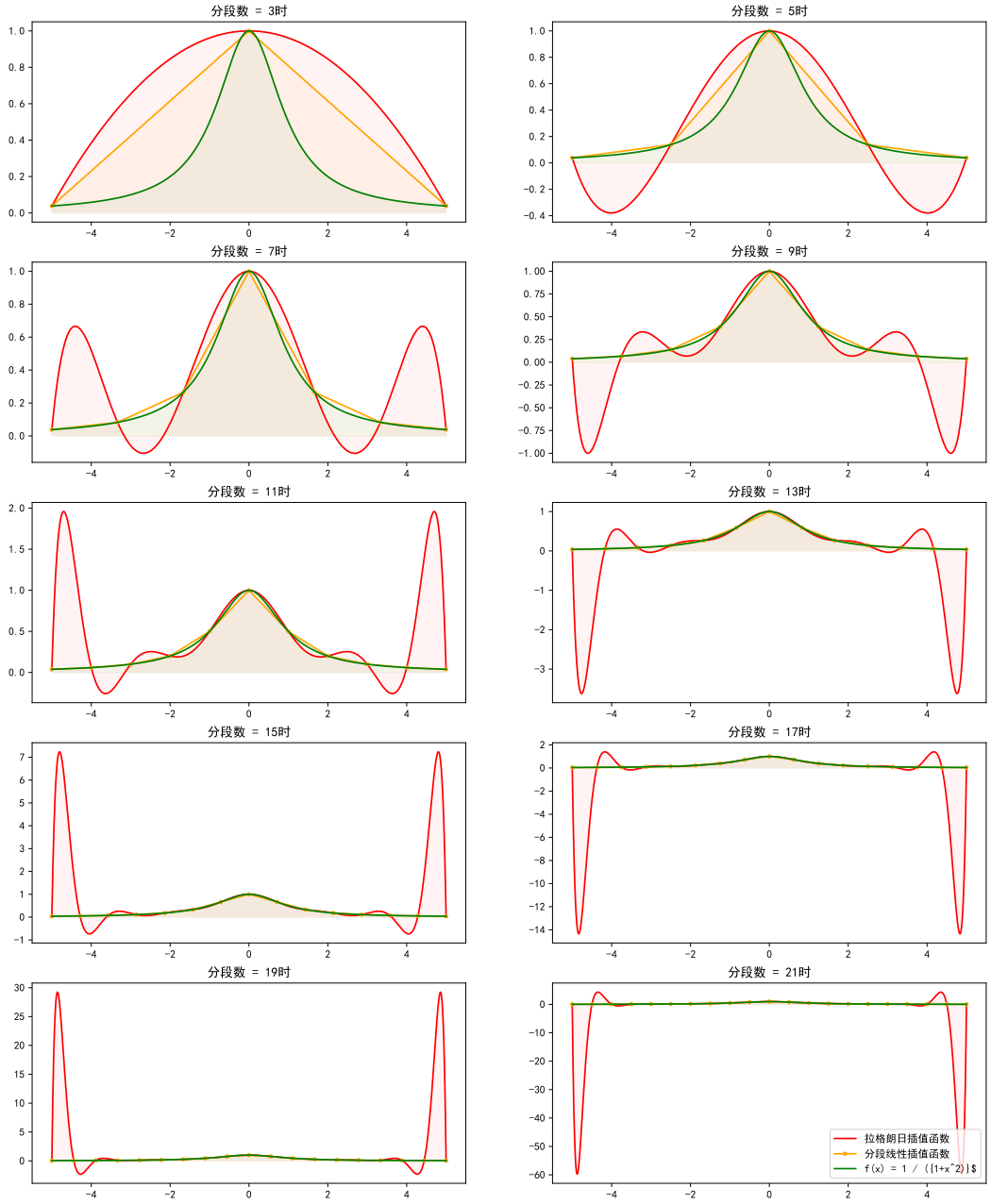
分段数	拉格朗日插值函数结果	分段线性插值函数结果
2	6.794871794872	5.192307692308
3	2.081447963801	1.892911010558
4	2.374005305040	3.285809018568
5	2.307692307692	2.476923076923
6	3.870448673471	2.888351529744
7	2.898994409748	2.665500985478
8	1.500488907128	2.784489369116
9	2.398617897843	2.721866472615
10	4.673300555655	2.756108597285
11	3.244772940278	2.738522227070
12	-0.312936515721	2.748438937965
13	1.919797216870	2.743567654383
14	7.899544641088	2.746498723781
15	4.155558992513	2.745202176481
16	-6.241437311025	2.746111618044
17	0.260509426355	2.745806196517
18	18.876621314144	2.746119227086
19	7.246026098451	2.746078836991
20	-26.849551804843	2.746208162460

打印对比表代码 使用下面代码打印对比表的值：

```
data = []
for n in range(2, 21, 1):
    x = np.linspace(-5., 5., n + 1)
    y = 1 / (1 + x * x)

    L = lagrange(x, y)
    f = np.poly1d(L)

    _L = [l for l in L]
    for i, l in enumerate(_L):
        _L[i] = l / (n + 1 - i)
    _L.append(0)
    _f = np.poly1d(_L)
    res1 = _f(5) - _f(-5)

    res2 = 0
    for ki in range(1, len(x)):
        h = x[ki] - x[ki - 1]
        res2 += h / 2 * (y[ki - 1] + y[ki])

    data.append((n, res1, res2))
df = pd.DataFrame(data, columns=['分段数', '拉格朗日插值函数结果',
                                 '分段线性插值函数结果'])
```

将拉格朗日插值与分段线性插值在不同分段数进行可视化，

Figure 22: 拉格朗日插值与分段线性插值的结果随分段数变化对比图



由上面的图表可知，分片线性插值增加分段数，可以有效的减少误差，Lagrange插值如果分段过多会产生剧烈的龙格震荡现象，误差迅速增高。

可视化代码 使用下面代码对拉格朗日插值与分段线性插值进行可视化：

```
data = []
for n in range(2, 21, 1):
x = np.linspace(-5., 5., n + 1)
y = 1 / (1 + x * x)

L = lagrange(x, y)
f = np.poly1d(L)

_L = [l for l in L]
for i, l in enumerate(_L):
_L[i] = l / (n + 1 - i)
_L.append(0)
_f = np.poly1d(_L)
res1 = _f(5) - _f(-5)

res2 = 0
for ki in range(1, len(x)):
h = x[ki] - x[ki - 1]
res2 += h / 2 * (y[ki - 1] + y[ki])

data.append((n, res1, res2))
df = pd.DataFrame(data, columns=['分段数', '拉格朗日插值函数结果',
'分段线性插值函数结果'])
```

6 第五章

6.1 题目 1

根据下列数据，使用幂曲线拟合，求解重力常量。

Table 6: a

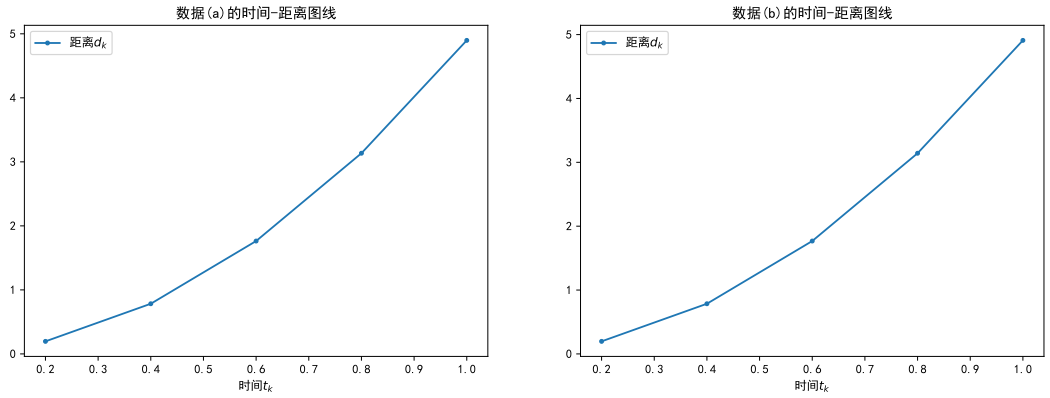
t_k	d_k
0.200	0.1960
0.400	0.7835
0.600	1.7630
0.800	3.1345
1.000	4.8975

Table 7: b

x_k	d_k
0.2	0.1965
0.4	0.7855
0.6	1.7675
0.8	3.1420
1.0	4.9095

时间-距离图线 根据原数据作出时间-距离图线：

Figure 23: 时间-距离图线



曲线拟合 设 $\{(x_k, y_k)\}_{k=1}^{N}$ 有 N 个点，其中横坐标是确定的，最小二乘幂函数拟合曲线 $y = Ax^M$ 的系数 A 为

$$A = \frac{\sum_{k=1}^{N} x_k^M y_k}{\sum_{k=1}^{N} x_k^{2M}}$$

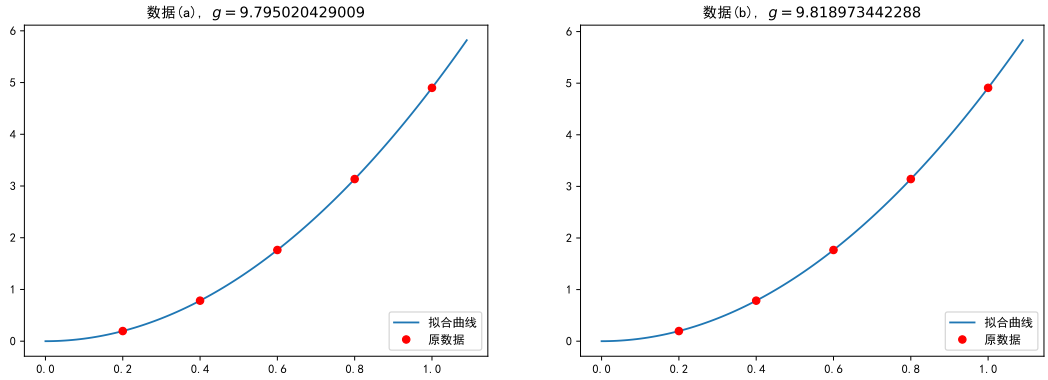
代入数据解得 $g_1 = 9.795020429009, g_2 = 9.818973442288$。

曲线拟合代码

```
M = 2
# data (a)
x1 = df1['时间 $t_k$'].values
y1 = df1['距离 $d_k$'].values
g1 = sum(xi**M * yi for xi, yi in zip(x1, y1 * 2)) /
       sum(xi**(2*M) for xi in x1)
display(Math('g_1 = %.12f' % g1))

# data (b)
M = 2
x2 = df2['时间 $t_k$'].values
y2 = df2['距离 $d_k$'].values
g2 = sum(xi**M * yi for xi, yi in zip(x2, y2 * 2)) /
       sum(xi**(2*M) for xi in x2)
display(Math('g_2 = %.12f' % g2))
```

拟合曲线与原始数据对比可视化 下面是拟合曲线与原始数据的对比图，

Figure 24: 拟合曲线与原始数据对比图



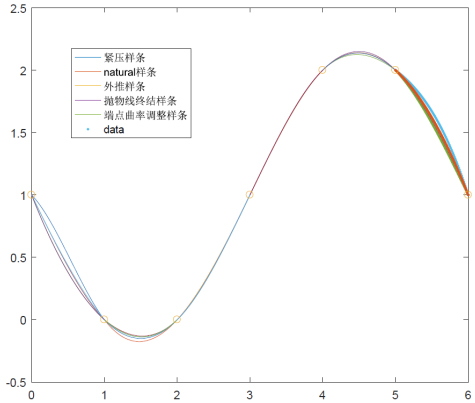
拟合曲线可视化代码

```
px = np.arange(0, 1.1, 0.01)
py1 = 0.5 * g1 * px**2
py2 = 0.5 * g2 * px**2
fig = plt.figure(figsize=(15, 5))
plt.subplot(1, 2, 1)
plt.title('数据 (a), $g = %.12f$' % g1)
plt.plot(px, py1, label='拟合曲线')
plt.plot(x1, y1, 'ro', label='原数据')
plt.legend(loc='lower right')
plt.subplot(1, 2, 2)
plt.title('数据 (b), $g = %.12f$' % g2)
plt.plot(px, py2, label='拟合曲线')
plt.plot(x2, y2, 'ro', label='原数据')
plt.legend(loc='lower right')
```

6.2 题目 2

编写程序，根据点 $(0,1),(1,0),(2,0),(3,1),(4,2),(5,2),(6,1)$。求 5 种不同的三次样条插值，其中 $S'(0)=-0.6, S'(6)=-1.8, S''(0)=1, S''(6)=-1$。在同一坐标系下，画出这 5 个三次样条插值和这些数据点。

Figure 25: 样条插值结果



7 第六章

7.1 题目 1

使用极限的微分求解下列函数在 x 处的导数近似值，精度为小数点后 13 位。（有必要改写程序中的 $max1$ 的值和 h 的初始值）

$$f(x) = \tan\left(\cos\left(\frac{\sqrt{5}+\sin x}{1+x^2}\right)\right); \qquad x = \frac{1+\sqrt{5}}{3}$$

极限的微分求解 在研究求解函数 $f(x)$ 的导数的近似值的过程，$f(x)$ 的导数可以表示为:

$$f^{'}(x) = \lim_{h\to 0} \frac{f(x+h)-f(x)}{h}$$

非常理想情况下就是直接选取一个序列 h_k，并且序列极限为 0，然后计算序列的极限: $D_k = \frac{f(x+h_k)-f(x)}{h_k}, k = 1, 2, ..., n, ...$

但是在计算机求解的过程中，序列 h_k 在趋近于 0 的时候，$f(x+h)-f(x)$ 同样也会趋近于 0，而且 $f(x+h)-f(x)$ 的差值会和之前求抛物线的根一样的情况，会损失精度，所以并不是迭代的步骤越多越好，需要及时终止。

求解

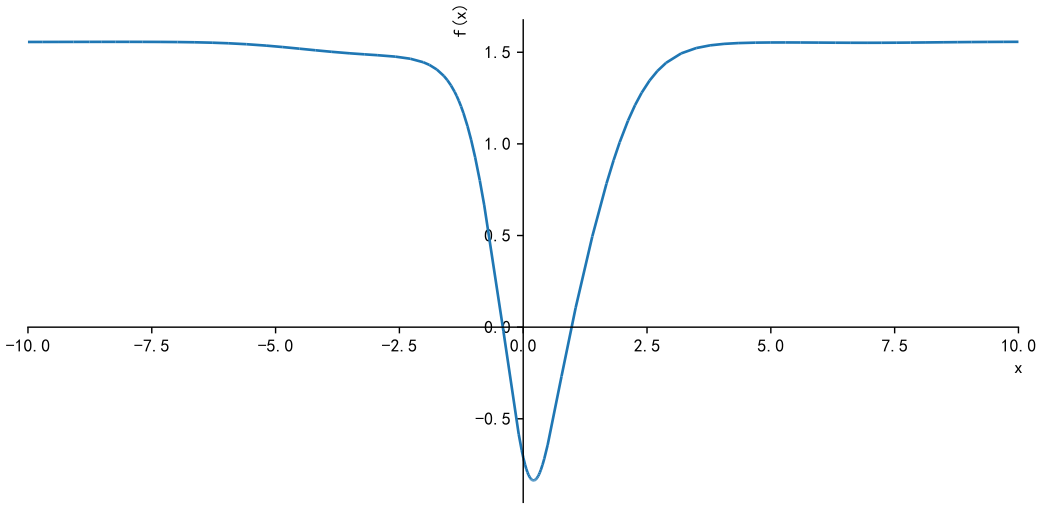
将 TOLERANCE 设置为 10^{-10}，最大迭代次数为 100 次，计算得到的导数值为 1.2285974239。

极限的微分求解代码

```
def diff_lim(f, x, tol, max_iter):
    h = 1
    D0 = (f(x + h) - f(x - h)) / (2 * h)
    for _ in range(max_iter):
        h /= 10.0
        D1 = (f(x + h) - f(x - h)) / (2 * h)
        if abs(D0 - D1) < tol:
            break
        D0 = D1
    return D0
```

与解析解比较验证 首先画出 $f(x)$ 的图像，

Figure 26: $f(x) = \tan\left(\cos\left(\frac{\sqrt{5}+\sin x}{1+x^2}\right)\right)$ 的图像



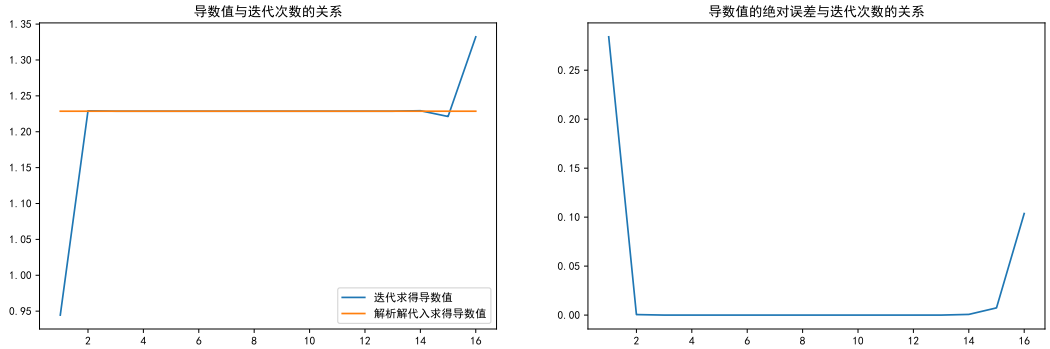
使用 Python 的 `sympy` 包求解出 $f'(x)$ 的解析解：

$$f'(x) = -\left(-\frac{2x}{(x^2+1)^2}\left(\sin(x)+\sqrt{5}\right)+\frac{\cos(x)}{x^2+1}\right)\left(\tan^2\left(\cos\left(\frac{\sin(x)+\sqrt{5}}{x^2+1}\right)\right)+1\right)\sin\left(\frac{\sin(x)+\sqrt{5}}{x^2+1}\right)$$

将 $x = \frac{1+\sqrt{5}}{3}$ 代入得

$$f'\left(\frac{1+\sqrt{5}}{3}\right) = 1.228597423376$$

Figure 27: 极限的微分求解结果与解析解比较验证



比较验证及其可视化代码

```
plt.figure(figsize=(16, 5))

def _diff_lim(f, x, max_iter):
    h = 1
    D0 = (f(x + h) - f(x - h)) / (2 * h)
    data = [(1, D0)]
    for i in range(max_iter):
        h /= 10.0
        D0 = (f(x + h) - f(x - h)) / (2 * h)
        data.append((i + 2, D0))
    return data
data = _diff_lim(f_eval, (1 + np.sqrt(5)) / 3.0, 15)
px, py = zip(*data)
px = np.array(px)
py = np.array(py)
plt.subplot(1, 2, 1)
plt.title('导数值与迭代次数的关系')
plt.plot(px, py, label='迭代求得导数值')
plt.plot(px, np.ones_like(px) * (df_eval((1 + np.sqrt(5)) / 3.0)),
         label= '解析解代入求得导数值')
plt.legend(loc='lower right')
plt.subplot(1, 2, 2)
plt.title('导数值的绝对误差与迭代次数的关系')
plt.plot(px, abs(df_eval((1 + np.sqrt(5)) / 3.0) - py))
```

8 第七章

8.1 题目 1

使用组合辛普森公式求定积分，精确到小数点后 11 位

$$l = \int_a^b \sqrt{1 + (f'(x)^2)} dx$$

1. $f(x) = x^3, 0 \le x \le 1$
2. $f(x) = \sin x, 0 \le x \le \pi/4$
3. $f(x) = e^{-x}, 0 \le x \le 1$

组合梯形公式

梯形法简介 ：把被积区间划分成若干小区间, 对于每一小区间, 用函数两个端点与 x 轴围成的梯形面积, 近似代替函数图像与 x 轴围成的面积. 将所有小梯形面积累加, 即为结果.(或者说, 每一个小区间都用线性拉格朗日插值多项式来代替).

积分公式可以表述为:

$$f(x)\,dx = \frac{h}{2}\left[f(a) + 2\sum_j^{n-1} f\left(x_j\right) + f(b)\right] - \frac{b-a}{12}h^2 f''(\mu)$$

辛普森公式

推导

设 $x_1 = x_0 + h, x_2 = x_0 + 2h\,(h > 0)$ $\forall x \in [x_0, x_2]$,$\exists \xi(x) \in [x_0, x_2]$, 使得

$$f(x) = f(x_1) + f'(x_1)(x - x_1) + \frac{f''(x_1)}{2}(x - x_1)^2 +$$

$$\frac{f'''(x_1)}{6}(x - x_1)^3 + \frac{f^{(4)}(\xi(x))}{24}(x - x_1)^4$$

. 积分, 得 $\int_{x_0}^{x_2} f(x)\,dx =$

$$\left[f(x_1)(x - x_1) + \frac{f'(x_1)}{2}(x - x_1)^2 + \frac{f''(x_1)}{6}(x - x_1)^3 + \frac{f'''(x_1)}{24}(x - x_1)^4\right]_{x_0}^{x_2}$$

$$+\frac{1}{24}\int_{x_0}^{x_2} f^{(4)}\left(\xi\left(x\right)\right)\left(x-x_1\right)^4 dx$$

由于 $\forall x \in [x_0, x_2], (x-x_1)^4 \geq 0$, 因此

$$\frac{1}{24}\int_{x_0}^{x_2} f^{(4)}\left(\xi\left(x\right)\right)\left(x-x_1\right)^4 dx = \frac{f^{(4)}\left(\xi_1\right)}{24}\int_{x_0}^{x_2}\left(x-x_1\right)^4 dx = \frac{f^{(4)}\left(\xi_1\right)}{120}\left(x-x_1\right)^5 |_{x_0}^{x_2}$$

其中 $\xi_1 \in (x_0, x_2)$.

利用 $h = x_2 - x_1 = x_1 - x_0$, 上述积分表述为

$$\int_{x_0}^{x_2} f\left(x\right) dx = 2hf\left(x_1\right) + \frac{h^3}{3}f^{''}\left(x_1\right) + \frac{f^{(4)(\xi_1)}}{60}h^5$$

.

对于 $f^{''}\left(x_1\right)$, 用如下方式处理:

$$f\left(x_0+h\right) = f\left(x_0\right) + f^{'}\left(x_0\right)h + \frac{1}{2}f^{''}\left(x_0\right)h^2 + \frac{1}{6}f^{(3)}\left(x_0\right)h^3 + \frac{1}{24}f^{(4)}\left(\xi_1\right)h^4$$

.

$$f\left(x_0+h\right) = f\left(x_0\right) - f^{'}\left(x_0\right)h + \frac{1}{2}f^{''}\left(x_0\right)h^2 - \frac{1}{6}f^{(3)}\left(x_0\right)h^3 + \frac{1}{24}f^{(4)}\left(\xi_1\right)h^4$$

. 两式相加,

$$f\left(x_0+h\right) + f\left(x_0-h\right) = 2f\left(x_0\right) + f^{''}\left(x_0\right)h^2 + \frac{h^4}{24}\left[f^{(4)}\left(\xi_1\right) + f^{(4)}\left(\xi_{-1}\right)\right]$$

取 $f^{(4)}\left(\xi\right) = \frac{1}{2}\left[f^{(4)}\left(\xi_1\right) + f^{(4)}\left(\xi_{-1}\right)\right]$, 把 $f^{''}\left(x_1\right)$ 的值代入, 可以求得

$$\int_{x_0}^{x_2} f\left(x\right) dx = \frac{h}{3}\left[f\left(x_0\right) + 4f\left(x_1\right) + f\left(x_2\right)\right] - \frac{h^5}{12}\left[\frac{1}{3}f^{(4)}\left(\xi_2\right) - \frac{1}{5}f^{(4)}\left(\xi_1\right)\right]$$

余项可以写成 $\frac{h^5}{90}f^{(4)}\left(\xi\right)$ 的形式.

对于复合辛普森公式,

$$\int_a^b f\left(x\right) dx = \sum_{j=1}^{n/2}\int_{x_{2j-2}}^{x_{2j}} f\left(x\right) dx =$$

$$\frac{h}{3}\left[f\left(x_0\right) + 2\sum_{j=1}^{n/2-1} f\left(x_{2j}\right) + 4\sum_{j=1}^{n/2} f\left(x_{2j-1} + f\left(x_n\right)\right)\right] + E\left(f\right)$$

其中 $E\left(f\right) = -\frac{h^5}{90}\sum_{j=1}^{n/2} f^{(4)}\left(\xi_j\right) = -\frac{b-a}{180}h^4 f^{(4)}\left(\mu\right)$

求解

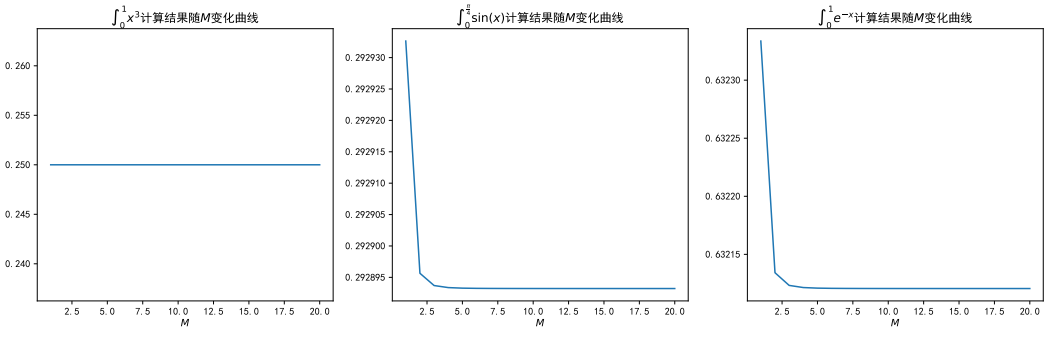
$$\int_0^1 x^3 = 0.250000$$
$$\int_0^{\frac{\pi}{4}} \sin(x) = 0.292893$$
$$\int_0^1 e^{-x} = 0.632121$$

辛普森公式代码

```
int1 = simpson(sp.lambdify(x, f1), a1, b1, 10)
display(Math('\int_{%s}^{%s}{%s} = %f' %
             (sp.latex(a1), sp.latex(b1), sp.latex(f1), int1)))
int2 = simpson(sp.lambdify(x, f2), a2, b2, 10)
display(Math('\int_{%s}^{%s}{%s} = %f' %
             (sp.latex(a2),sp.latex(sp.pi/4),sp.latex(f2),int2)))
int3 = simpson(sp.lambdify(x, f3), a3, b3, 10)
display(Math('\int_{%s}^{%s}{%s} = %f' %
             (sp.latex(a3), sp.latex(b3), sp.latex(f3), int3)))
```

计算结果随 M 的变化曲线　下面是辛普森公式计算的结果随 M 的变化曲线

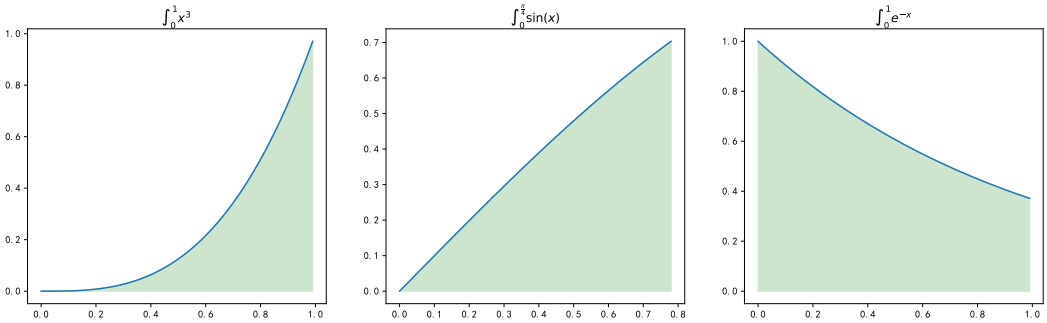
Figure 28: 计算结果随 M 的变化曲线



解析解验证

$$\int_0^1 x^3 = [\frac{x^4}{4}]_0^1 = 0.250000$$
$$\int_0^{\frac{\pi}{4}} \sin(x) = [-\cos(x)]_0^{\frac{\pi}{4}} = 0.292893$$
$$\int_0^1 e^{-x} = [-e^{-x}]_0^1 = 0.632121$$

Figure 29: 解析解积分图示

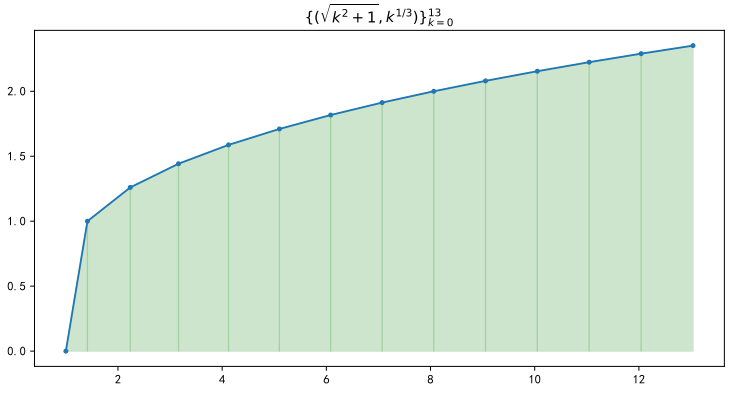


8.2 题目 2

修改组合梯形公式，使之可以求只有若干个点函数值的已知函数的积分。将程序修改为求区间 $[a, b]$ 上过 M 个给定点的函数 $f(x)$ 的积分逼近。注意节点不需要等距，利用该程序求过点 $\{(\sqrt{k^2+1}, k^{1/3})\}_{k=0}^{13}$ 的函数的积分逼近。

求解 解得积分近似为 21.84106920647963.

Figure 30: 过点 $\{(\sqrt{k^2+1}, k^{1/3})\}_{k=0}^{13}$ 的函数的积分逼近



代码

```
k = np.arange(14)
x = np.sqrt(k * k + 1)
y = k ** (1 / 3)
plt.figure(figsize=(10, 5))
plt.plot(x, y, marker='.')
plt.title('${\{(\sqrt{k^2+1},k^{1/3})\}}_{k=0}^{13}$')
res = 0
for ki in range(1, 14):
    h = x[ki] - x[ki - 1]
    res += h / 2 * (y[ki - 1] + y[ki])
    plt.fill_between([x[ki], x[ki - 1]], [y[ki], y[ki - 1]],
                     color = "green", alpha = 0.2)
print(res)
```

9 第九章

9.1 题目 1

流行病的数学模型描述如下：设有 L 个成员的构成的群落，其中有 P 个感染个体，Q 为未感染个体。令 $y(t)$ 表示时刻 t 感染个体的数量。对于温和的疾病，如普通感冒，每个个体保持存活，流行病从感染者传播到未感染者。由于两组间有 PQ 种可能的接触，$y(t)$ 的变化率正比于 PQ。故该题目可以描述为初值题目：

$$y' = ky(L - y) \qquad y(0) = y_0$$

9.1.1 题目 1.1

用 $L = 25000, k = 0.00003, h = 0.2$ 和初值条件 $y(0) = 250$, 编写程序求解 $[0, 60]$ 上的欧拉近似解。

欧拉显式格式 ：

公式推导 根据泰勒展开, 有

$$y\left(t_{j+1}\right)+hf\left(t_j,y\left(t_j\right)\right)+\frac{h^2}{2}y''\left(\xi_j\right)$$

忽略余项, 可以得到欧拉显式格式:

$$\begin{cases} w_0=\alpha \\ w_{j+1}=w_j+hf\left(t_j,w_j\right) \end{cases}$$

误差分析 假设 f 在 $D=\left\{(t,y)\middle|a\le t\le b,-\infty<y<\infty\right\}$ 连续, 满足 Lipschitz 条件 (Lipschitz 常数为 L), 且满足 $\left|y''(t)\right|\le M,\forall t\in[a,b]$. 则 $\left|y\left(t_j\right)-w_j\right|\le\frac{hM}{2L}\left[e^{L\left(t_j-a\right)}-1\right]$. 在某些情况下, M,L 较难确定. 但可以发现, 随着向后递推, 欧拉显式格式的误差逐渐增大。

欧拉法误差分析 对于

$$\left|y\left(x_j\right)-w_j\right|\le\frac{hM}{2L}\left[e^{L\left(x_j-a\right)}-1\right]$$

, 需要确定 M,L 的值.

由于

$$M=\max\left|y''(x)\right|,x\in[a,b]$$

,

而

$$y(x)=\sqrt{2x+1}$$

,

$$y''(x)=\frac{-1}{(2x+1)^{3/2}}$$

有

$$M=\max\left|y''(x)\right|=\left|f''(-1)\right|=1$$

$$\frac{\partial f}{\partial y}=1+\frac{2x}{y^2}=1+\frac{2x}{1+2x}$$

,

$$L=\max\left|\frac{\partial f}{\partial y}\right|=\left|\frac{\partial f}{\partial y}\Big|_{x=1}\right|\approx1.6667$$

.

因此

$$|y(1) - w_n| \leq \frac{hM}{2L}\left[e^L - 1\right]$$

。

求解 使用欧拉法求得 $[0, 60]$ 上的欧拉近似解，前十个解如下表所示

Table 8: 前十个欧拉近似解

t	y
0.000000	250.000000
0.200669	287.125000
0.401338	329.699105
0.602007	378.501762
0.802676	434.417445
1.003344	498.447751
1.204013	571.724212
1.404682	655.521633
1.605351	751.271626
1.806020	860.575916

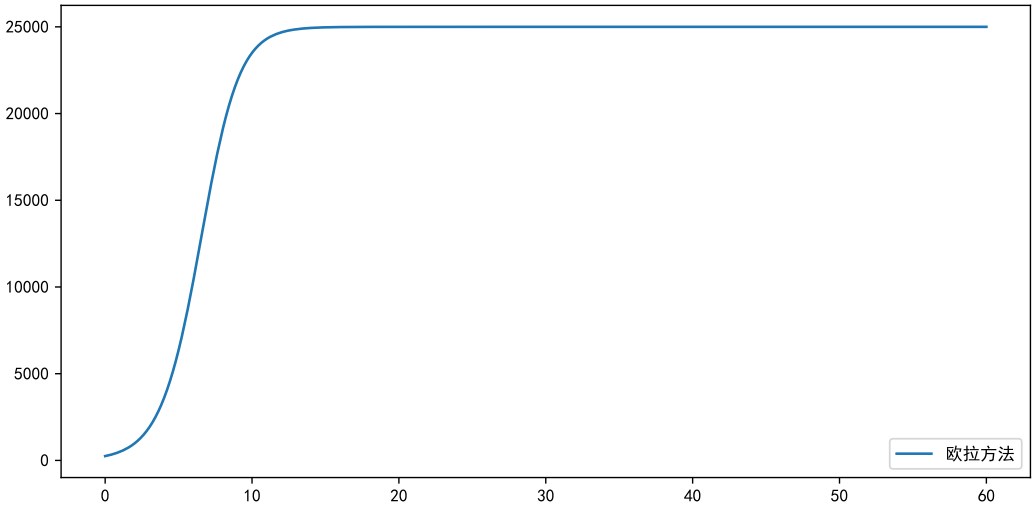
欧拉法代码

```
h = 0.2
M = int(60 / h)
def f(y):
    return 0.00003 * y * (25000 - y)
t = np.linspace(0, 60, M)
y = np.zeros(M)
y[0] = 250
for k in range(M - 1):
    y[k + 1] = y[k] + h * f(y[k])
```

9.1.2 题目 1.2

画出题目 1.1 中的近似解。

Figure 31: 近似解可视化



可视化代码

```
plt.figure(figsize=(10, 5))
plt.plot(t, y, label='欧拉方法')
plt.legend(loc='lower right')
plt.show()
```

9.1.3 题目 1.3

通过求题目 1.1 中欧拉方法的纵坐标平均值来估计平均感染个体数目。

编程求得平均感染数目为 22292.56481490718。

代码

```
np.mean(y)
```

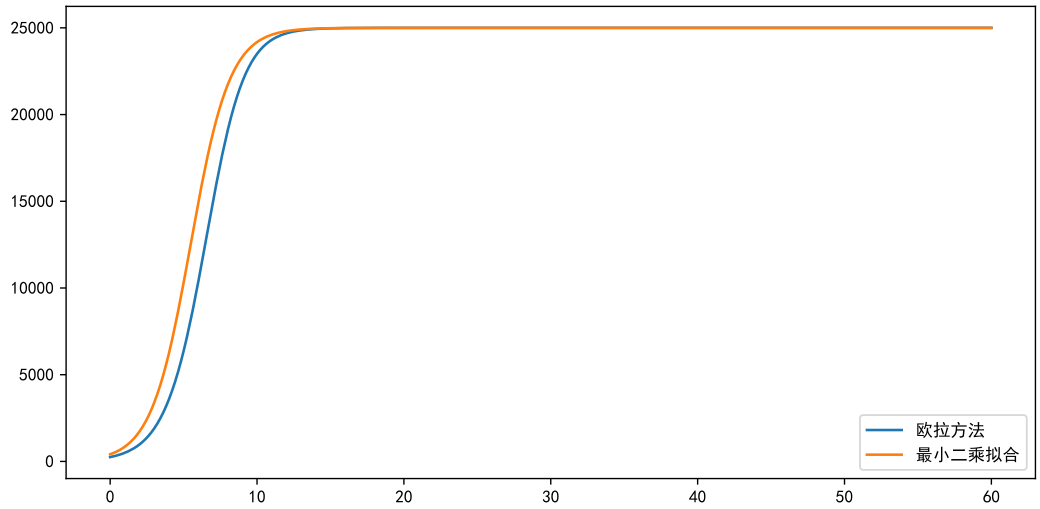
9.1.4 题目 1.4

通过使用曲线拟合题目 1.1 中的数据，并使用积分中值定理，估计平均感染个体的数目。

Logistic 模型

$$f(x) = \frac{L}{1 + e^{ax+b}}$$

Figure 32: 欧拉方法与最小二乘拟合对比



使用积分中值定理估计平均感染个体的数目为 22709.106862921362。

代码

```
integrate.quad(lambda x: 25000.0 / (1 + np.exp(p(x))),
               0, 60)[0] / 60
```

9.2 题目 2

考虑一阶积分-常微分方程 (intergro-ordinary differential equation)

$$y' = 1.3y - 0.25y^2 - 0.0001y \int_0^t y(\tau)d\tau$$

常微分方程组解法

给定一个常微分方程组:

$$\begin{cases} \dfrac{dy_1}{dx} = f_1\left(x, y_1, y_2, \cdots, y_n\right) \\ \dfrac{dy_2}{dx} = f_2\left(x, y_1, y_2, \cdots, y_n\right) \\ \quad \vdots \\ \dfrac{dy_n}{dx} = f_n\left(x, y_1, y_2, \cdots, y_n\right) \end{cases}$$

初始状态 $y_0^{(0)}, y_2^{(0)}, \cdots, y_n^{(0)}$ 已知. 令

$$\begin{cases} K_i^{(1)} = h * f_i\left(x^{(k)}, y_1^{(k)}, \cdots, y_n^{(k)}\right) \\ K_i^{(2)} = h * f_i\left(x^{(k)} + \dfrac{h}{2}, y_1^{(k)} + \dfrac{1}{2}K_1^{(1)}, \cdots, y_n^{(k)} + \dfrac{1}{2}K_n^{(1)}\right) \\ K_i^{(3)} = h * f_i\left(x^{(k)} + \dfrac{h}{2}, y_1^{(k)} + \dfrac{1}{2}K_1^{(2)}, \cdots, y_n^{(k)} + \dfrac{1}{2}K_n^{(2)}\right) \\ K_i^{(4)} = h * f_i\left(x^{(k)} + h, y_1^{(k)} + K_1^{(3)}, \cdots, y_n^{(k)} + K_n^{(3)}\right) \end{cases}$$

高阶微分方程数值解

给定一个 n 阶微分方程, 且知道这个方程在某初始点处函数值和 1 到 $n-1$ 阶导数值. 求解方程:

$$y^{(n)} = f\left(x, y, y', \cdots, y^{(n-1)}\right)$$

令 $y = y_1, y' = y_2, y'' = y_3, \cdots, y^{(n-1)} = y_n$ 可以转化为下面的形式:

$$\begin{cases} \quad y = y_1 \\ \dfrac{dy_1}{dx} = y_2 \\ \dfrac{dy_2}{dx} = y_3 \\ \quad \vdots \\ \dfrac{dy_n}{dx} = f\left(x, y, y_1, \cdots, y_n\right) \end{cases}$$

9.2.1 题目 2.1

在区间 $[0, 20]$ 上，用欧拉方法和 $h = 0.2, y(0) = 250$ 以及梯形公式求方程的近似解。

分析 欧拉方法的一般步长可以修改为

$$y_(k+1) = y_k + h(1.3y_k - 0.25y_k^2 - 0.0001y_k \int_0^{t_k} y(\tau)d\tau)$$

如果梯形公式用于逼近积分，则该表达式为

$$y_(k+1) = y_k + h(1.3y_k - 0.25y_k^2 - 0.0001y_k T_k(h))$$

其中 $T_0(h) = 0$ 且 $T_k(h) = T_{k-1}(h) + \frac{h}{2}(y_{k-1} + y_k)$ 其中 $k = 0, 1, ..., 99$

求解 使用欧拉方法和梯形公式进行求解。

Table 9: 前 9 个近似解

t	y
0.000000	2.500000e+02
0.202020	-2.810000e+03
0.404040	-3.983600e+05
0.606061	-7.935358e+09
0.808081	-3.148621e+18
1.010101	-4.957105e+35
1.212121	-1.228694e+70
1.414141	-7.548744e+138
1.616162	-2.849290e+276

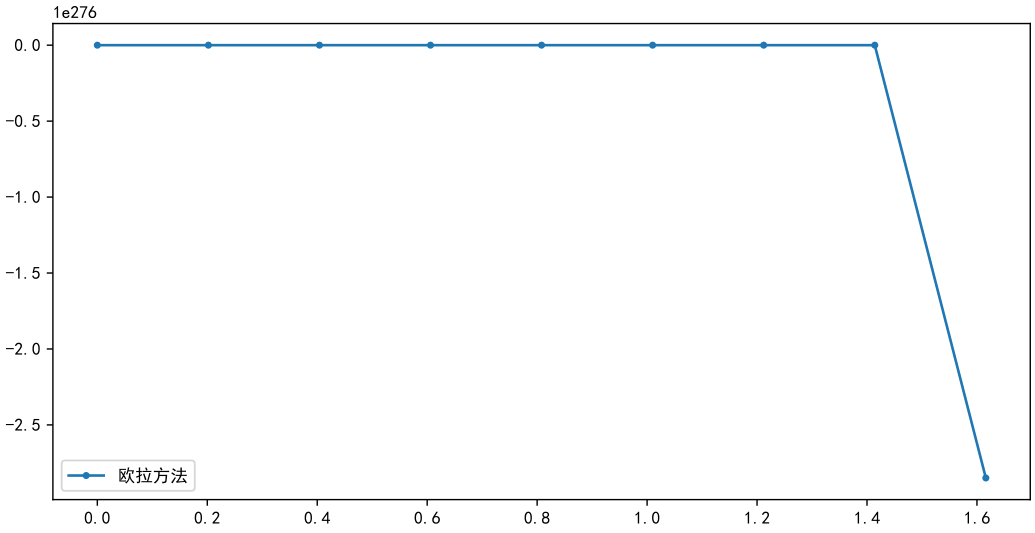
代码

```
h = 0.2
M = int(20 / h)
x = np.linspace(0, 20, M)
y = np.zeros(M)
t = np.zeros(M)
```

```
y[0] = 250
t[0] = 0
for k in range(M - 1):
    if k > 0:
        t[k] = t[k - 1] + h * (y[k - 1] + y[k]) / 2
    y[k + 1] = y[k] + h * (1.3 * y[k]
                - 0.25 * y[k] * y[k] - 0.0001 * y[k] * t[k])
display(pd.DataFrame(list(zip(x, y))[:10], columns=['$x$', '$y$']))
display(Math(r'\cdots'))
```

迭代过程可视化

Figure 33: 迭代过程可视化



9.2.2 题目 2.2

用初值 $y(0) = 200$ 和 $y(0) = 300$ 重复计算题目 2.1 的值。

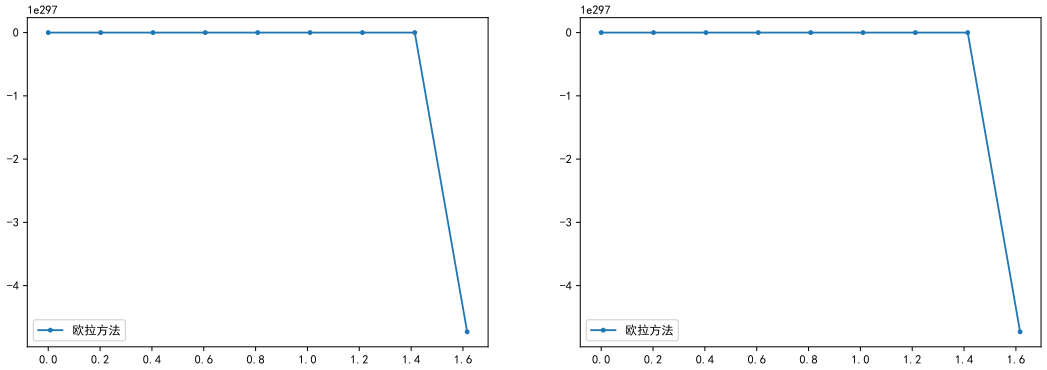
求解 使用欧拉方法和梯形公式进行求解。

Table 10: $y(0) = 200$ 和 $y(0) = 300$ 时前 9 个近似解

t	y	t	y
0.000000	2.000000e+02	0.000000	3.000000e+02
0.202020	-1.748000e+03	0.202020	-4.122000e+03
0.404040	-1.549831e+05	0.404040	-8.547694e+05
0.606061	-1.201232e+09	0.606061	-3.653409e+10
0.808081	-7.215084e+16	0.808081	-6.673966e+19
1.010101	-2.602976e+32	1.010101	-2.227180e+38
1.212121	-3.387877e+63	1.212121	-2.480265e+75
1.414141	-5.739085e+125	1.414141	-3.075980e+149
1.616162	-1.646921e+250	1.616162	-4.731015e+297

迭代过程可视化

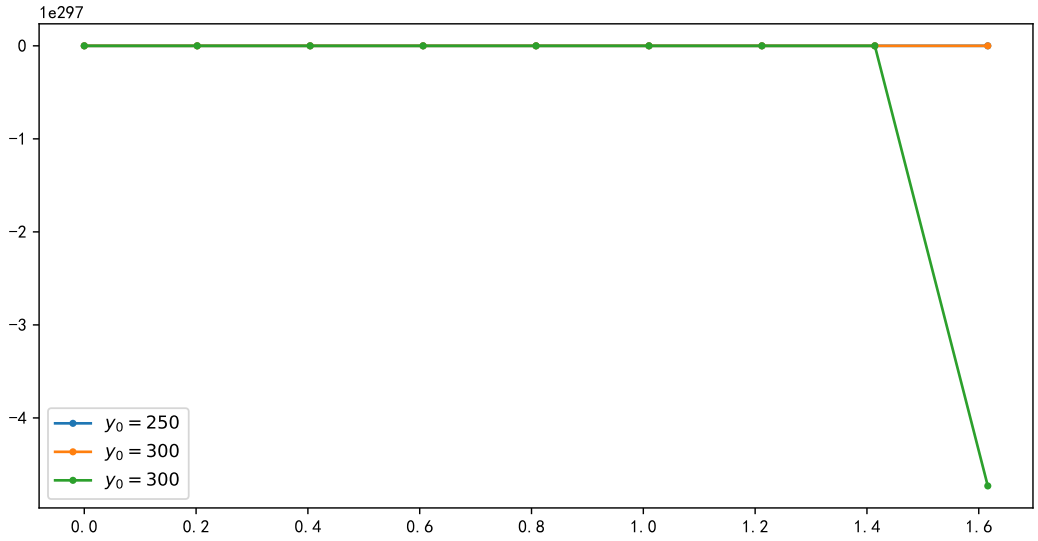
Figure 34: 迭代过程可视化



9.2.3 题目 2.3

在同一坐标系下画出题目 2.1 和题目 2.3 的近似解，如下图所示。

Figure 35: 不同初值在同一坐标系下的对比图



9.3 题目 3

使用休恩方法求解微分方程

$$y' = 3y + 3t \qquad y(0) = 1 \qquad y(t) = \frac{4}{3}e^{3t} - t - \frac{1}{3}$$

9.3.1 题目 3.1

令 $h = 0.1$, 使程序执行 20 步，然后令 $h = 0.05$，使程序执行 40 步。

```
def heun(f, a, b, ya, M):
    h = (b - a) / M
    T = np.linspace(a, b, M)
    Y = np.zeros(M)
    Y[0] = ya
    for j in range(M - 1):
        k1 = f(T[j], Y[j])
        k2 = f(T[j + 1], Y[j] + h * k1)
        Y[j + 1] = Y[j] + (h / 2) * (k1 + k2)
    return T, Y
```

9.3.2 题目 3.2

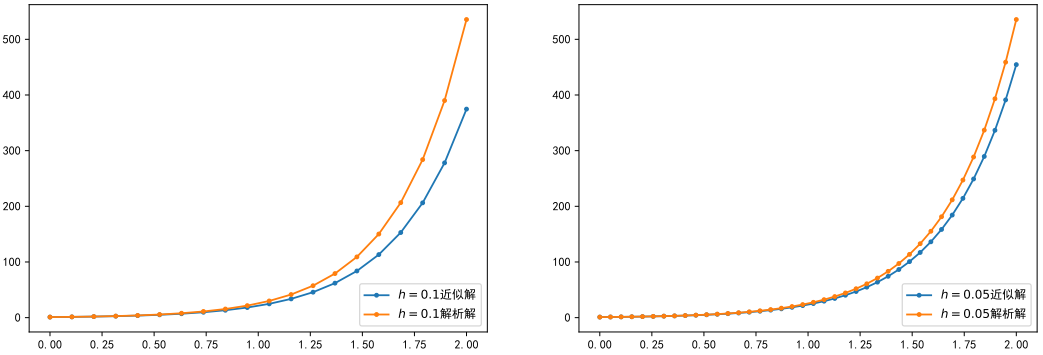
比较 (a) 中两个近似解与精确解 $y(2)$。

Table 11: $h = 0.1$ 和 $h = 0.0.5$ 时近似解与解析解对比表

t	$\hat{y}$	y	t	$\hat{y}$	y
0.000000	1.000000	1.000000	0.000000	1.000000	1.000000
0.105263	1.360789	1.389859	0.051282	1.165096	1.170467
0.210526	1.882367	1.963577	0.102564	1.365083	1.377812
0.315789	2.620205	2.789429	0.153846	1.605588	1.628171
0.421053	3.648912	3.961043	0.205128	1.893142	1.928696
0.526316	5.068840	5.606815	0.256410	2.235335	2.287730
0.631579	7.014958	7.902818	0.307692	2.640975	2.715005
0.736842	9.668802	11.090512	0.358974	3.120293	3.221870
0.842105	13.274539	15.501017	0.410256	3.685172	3.821560

$h = 0.1$ 和 $h = 0.0.5$ 时近似解与解析解对比图

Figure 36: $h = 0.1$ 和 $h = 0.0.5$ 时近似解与解析解对比图



9.3.3 题目 3.3

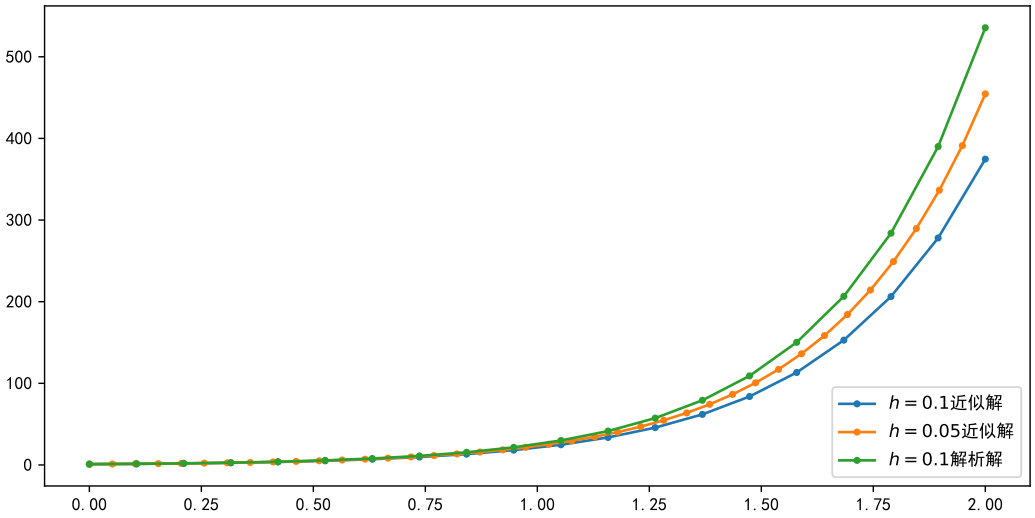
当 h 减半时，(a) 中的最终全局误差是否和预期相符？

使用欧拉方法也有误差 $O(h)$，和外面的 $\frac{2}{h}$ 合在一起就是 $O(h^2)$，所以总的误差就是 $O(h^2)$。所以如果步长变为原来的 $\frac{1}{2}$ 误差应该变为原来的 $\frac{1}{4}$

9.3.4 题目 3.4

将两个近似解和精确解画在同一坐标系中。

Figure 37: $h = 0.1$ 和 $h = 0.0.5$ 时近似解与解析解对比图



9.4 题目 4

使用 $N = 4$ 的龙格-库塔方法求解微分方程

$$y' = 3y + 3t \qquad y(0) = 1 \qquad y(t) = \frac{4}{3}e^{3t} - t - \frac{1}{3}$$

9.4.1 题目 4.1

令 $h = 0.1$, 使程序执行 20 步，然后令 $h = 0.05$，使程序执行 40 步。

```
def rk4(f, a, b, ya, M):
    h = (b - a) / M
    T = np.linspace(a, b, M)
    Y = np.zeros(M)
    Y[0] = ya
    for j in range(M - 1):
        k1 = h * f(T[j], Y[j])
        k2 = h * f(T[j] + h / 2, Y[j] + k1 / 2)
```

```
        k3 = h * f(T[j] + h / 2, Y[j] + k2 / 2)
        k4 = h * f(T[j] + h, Y[j] + k3)
        Y[j + 1] = Y[j] + (k1 + 2 * k2 + 2 * k3 + k4) / 6
    return T, Y
```

9.4.2 题目 4.2

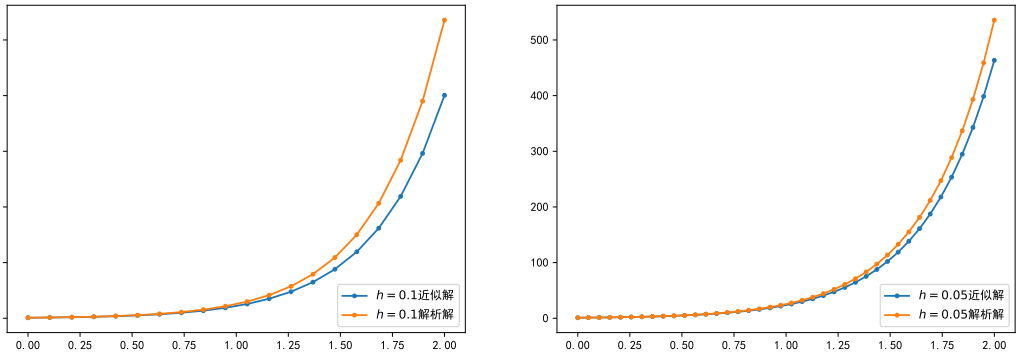
比较 (a) 中两个近似解与精确解 $y(2)$。

Table 12: $h = 0.1$ 和 $h = 0.0.5$ 时近似解与解析解对比表

t	$\hat{y}$	y	t	$\hat{y}$	y
0.000000	1.000000	1.000000	0.000000	1.000000	1.000000
0.105263	1.360789	1.389859	0.051282	1.165096	1.170467
0.210526	1.882367	1.963577	0.102564	1.365083	1.377812
0.315789	2.620205	2.789429	0.153846	1.605588	1.628171
0.421053	3.648912	3.961043	0.205128	1.893142	1.928696
0.526316	5.068840	5.606815	0.256410	2.235335	2.287730
0.631579	7.014958	7.902818	0.307692	2.640975	2.715005
0.736842	9.668802	11.090512	0.358974	3.120293	3.221870
0.842105	13.274539	15.501017	0.410256	3.685172	3.821560

$h = 0.1$ 和 $h = 0.0.5$ 时近似解与解析解对比图

Figure 38: $h = 0.1$ 和 $h = 0.0.5$ 时近似解与解析解对比图



9.4.3 题目 4.3

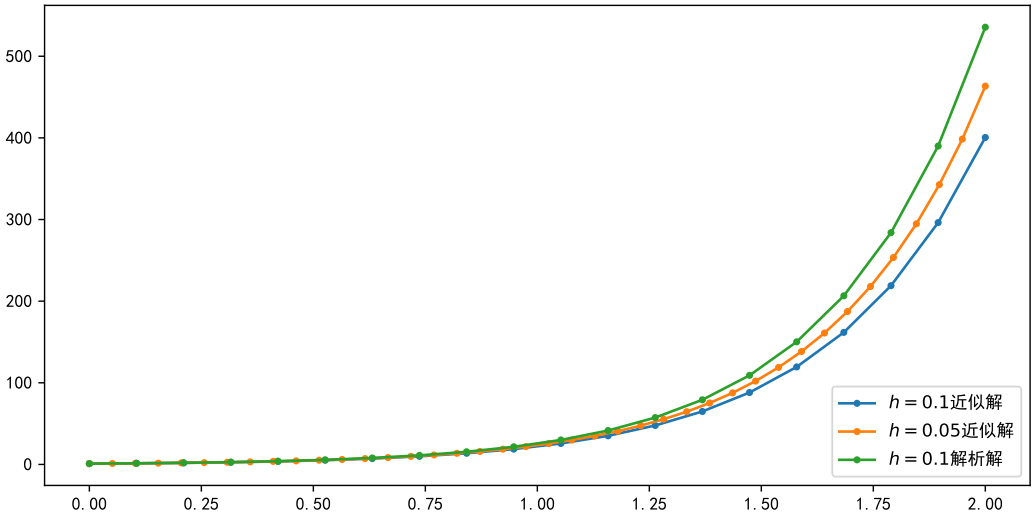
当 h 减半时，(a) 中的最终全局误差是否和预期相符？

是的。

9.4.4 题目 4.4

将两个近似解和精确解画在同一坐标系中。

Figure 39: $h = 0.1$ 和 $h = 0.0.5$ 时近似解与解析解对比图



10 第十一章

已知矩阵

$$\mathrm{A} = \begin{bmatrix} 4 & -1 & 1 \\ -1 & 3 & -2 \\ 1 & -2 & 3 \end{bmatrix}$$

是一个对称矩阵，且其特征值为 $\lambda_1 = 6, \lambda_2 = 3, \lambda_3 = 1$. 分别利用幂法、对称幂法、反幂法求其最大特征值和特征向量。注意：可取初始向量 $x^{(0)} = (111)^T$.

幂法 假定 $n \times n$ 的矩阵 A 有 n 个特征值 $|\lambda_1| > |\lambda_2| \geq \cdots \geq |\lambda_n|$, 对应线性无关的特征向量 $\left\{\mathrm{v}^{(1)}, \mathrm{v}^{(2)}, \cdots, \mathrm{v}^{(n)}\right\}$,

$\forall \mathrm{x} \in \mathbb{R}^n, \exists \beta_1, \beta_2, \cdots, \beta_n, st\ \mathrm{x} = \beta_1 \mathrm{v}^{(1)} + \beta_2 \mathrm{v}^{(2)} + \cdots + \beta_n \mathrm{v}^{(n)} = \sum_{j=1}^{n} \beta_j \mathrm{v}^{(j)}$

即

$$\begin{aligned}
\mathrm{Ax} &= \sum_{j=1}^{n} \beta_j \mathrm{A}\mathrm{v}^{(j)} = \sum_{j=1}^{n} \beta_j \lambda_j \mathrm{v}^{(j)} \\
\mathrm{A}^2\mathrm{x} &= \sum_{j=1}^{n} \beta_j \mathrm{A}\mathrm{v}^{(j)} = \sum_{j=1}^{n} \beta_j \lambda_j^2 \mathrm{v}^{(j)} \\
\mathrm{A}^2\mathrm{x} &= \sum_{j=1}^{n} \beta_j \mathrm{A}\mathrm{v}^{(j)} = \sum_{j=1}^{n} \beta_j \lambda_j^2 \mathrm{v}^{(j)} \\
&\vdots \\
\mathrm{A}^k\mathrm{x} &= \sum_{j=1}^{n} \beta_j \mathrm{A}\mathrm{v}^{(j)} = \sum_{j=1}^{n} \beta_j \lambda_j^k \mathrm{v}^{(j)} \\
&= \lambda_1^k \left(\beta_1 \mathrm{v}^{(1)} + \sum_{j=2}^{n} \beta_j \left(\frac{\lambda_j}{\lambda_1} \right)^k \mathrm{v}^{(j)} \right)
\end{aligned}$$

由于 $\forall j \in \{2, \cdots, n\}, |\lambda_1| > \left|\lambda_j\right|$, 因此 $\lim\limits_{k \to \infty} \left(\frac{\lambda_j}{\lambda_1} \right)^k = 0.$ 即

$$\lim_{k \to \infty} \mathrm{A}^k \mathrm{x} = \lim_{k \to \infty} \lambda_1^k \beta_1 \mathrm{v}^{(1)}$$

用 $x_i^{(k)}$ 表示 $\mathrm{x}^{(k)}$ 的第 i 个分量. 由于

$$\frac{x_i^{(k+1)}}{x_i^{(k)}} \approx \frac{\lambda_1^{k+1} \beta_1 \mathrm{v}_i^{(1)}}{\lambda_1^k \beta_1 \mathrm{v}_i^{(k)}} = \lambda_1$$

需要注意, 当 $|\lambda_1| > 1$ 时, $\mathrm{x}^{(k)}$ 的各分量趋于无穷, 当 $|\lambda_1| < 1$ 时, $\mathrm{x}^{(k)}$ 的各分量趋于 0. 为了克服这一缺点, 需要将迭代向量规范化. 采用如下方式进行迭代:

$$\begin{cases}
\mathrm{x}^{(0)} = \mathrm{y}^{(0)} \neq 0 \\
\mathrm{x}^{(k)} = \mathrm{A}\mathrm{y}^{(k-1)} \\
\mathrm{y}^{(k)} = \dfrac{\mathrm{x}^{(k)}}{\left\| \mathrm{x}^{(k)} \right\|_\infty}
\end{cases}$$

$$\lim_{k \to \infty} \mathrm{y}^{(k)} = \frac{v^{(1)}}{\left\| \mathrm{v}^{(1)} \right\|}, \lim_{k \to \infty} ||\mathrm{x}||_\infty = \lambda_1$$

.

易知最终的 x 为所求特征向量.

结果 解得最大特征值为 6.00000，特征向量为 $[1,-1,1]$，迭代次数为 22 次。

代码

```
def pow2(A, X, step, eps):
    l = 0
    cnt = 0
    err = 1
    while err > eps and cnt < step:
        cnt += 1
        Y = A * X
        C[cnt] = max(abs(Y))
        dc = abs(l - C[cnt])
        Y = (1 / C[cnt]) * Y
        dv = norm(X - Y)
        err = max(dc, dv)
        X = Y
        l = C[cnt]
    V = X
    return C, l, V
```

对称幂法 假设 A 是对称矩阵, A 有 n 个实数特征向量 $|\lambda_1| > |\lambda_2| > \cdots > |\lambda_n|$, 对应 n 个标准正交特征向量 $\left\{\mathrm{v}^{(1)}, \mathrm{v}^{(2)}, \cdots, \mathrm{v}^{(n)}\right\}$.

$\forall \mathrm{x}_0 \in \mathbb{R}^n, \mathrm{x}_0 = \beta_1 \mathrm{v}^{(1)} + \beta_2 \mathrm{v}^{(2)} + \cdots + \mathrm{v}^{(n)}$.

对于 $\mathrm{x}_k = \mathrm{A}^k \mathrm{x}_0, \mathrm{x}_k = \beta_1 \lambda_1^k \mathrm{v}^{(1)} + \beta_2 \lambda_2^k \mathrm{v}^{(2)} + \cdots + \beta_n \lambda_n^k \mathrm{v}^{(n)}$. 因为 n 个特征向量标准正交, 因此

$$\mathrm{x}_k^T \mathrm{x}_k = \sum_{j=1}^{n} \beta_j^2 \lambda_j^{2k} = \beta_1^2 \lambda_1^{2k} \left[1 + \sum_{j=2}^{n} \left(\frac{\beta_j}{\beta_1}\right)^2 \left(\frac{\lambda_j}{\lambda_n}\right)^{2k}\right]$$

且

$$\mathrm{x}_k^T \mathrm{A} \mathrm{x}_k = \sum_{j=1}^{n} \beta_j^2 \lambda_j^{2k+1} = \beta_1^2 \lambda_1^{2k+1} \left[1 + \sum_{j=2}^{n} \left(\frac{\beta_j}{\beta_1}\right)^2 \left(\frac{\lambda_j}{\lambda_1}\right)^{2k+1}\right]$$

于是

$$\lim_{k\to\infty} \frac{\mathrm{x}_k^T \mathrm{A} \mathrm{x}_k}{\mathrm{x}_k^T \mathrm{x}_k} = \lambda_1, \lim_{k\to\infty} \frac{\mathrm{x}_k}{||\mathrm{x}_k|_2|} = \frac{\mathrm{v}^{(1)}}{\left\|\mathrm{v}^{(1)}\right\|_2}$$

.

结果 解得最大特征值为 6.00000，特征向量为 $[0.5774, -0.5773, 0.5773]$，迭代次数为 18 次。

代码

```
def sym_pow(A, X, step, eps):
    cnt = 0
    err = 1
    X = X / norm(X, 2)
    while err > eps and cnt < step:
        cnt += 1
        Y = A * X
        C[cnt] = X * Y
        dc = norm(Y, 2)
        err = norm(X - Y / norm(Y, 2), 2)
        X = Y / norm(Y, 2)
    V = X
    l = u
    return C, l, V
```

反幂法 设 A 为 $n \times n$ 阶非奇异矩阵, λ 和 v 为对应的特征向量, 即 $\mathrm{Au} = \lambda \mathrm{v}$.

由于 $\mathrm{A}^{-1}\mathrm{v} = \frac{1}{\lambda}\mathrm{v}$. 如果 A 的特征值的顺序为 $|\lambda_1| \geq |\lambda_2| \geq \cdots \geq |\lambda_n|$,

则 A^{-1} 的特征值 $\left|\frac{1}{\lambda_n}\right| \geq \left|\frac{1}{\lambda_{n-1}}\right| \geq \cdots \geq \left|\frac{1}{\lambda_1}\right|$.

因此, 若对矩阵 A^{-1} 用幂法, 即可计算出 A^{-1} 的最大特征值 $\frac{1}{\lambda_n}$, 从而求得 A 按模最小的特征值 λ_n.

因为 A^{-1} 的计算比较麻烦, 因此在实际运算时, 以求解方程组 $\mathrm{Ax}^{(k+1)} = \mathrm{x}^{(k)}$ 替代.

在本题中, 需要求解最大特征值. 根据圆盘定理, 矩阵 A 的特征值有上界 Ma. 设 λ 为 A 的特征值, $k \in \mathbb{R}$. 则 $(\mathrm{A} - k\mathrm{I})\,\mathrm{x} = (\lambda - k)\,\mathrm{x}$. 此时令 $k = Ma + 2$, 则 $(\mathrm{A} - (Ma+2)\,\mathrm{I})$ 的特征值均小于 0. 设用反幂法求出 $(\mathrm{A} - (Ma+2)\,\mathrm{I})^{-1}$ 的绝对值最小特征值为 $|\lambda^*|$, 则 A 的最大特征值为 $Ma + 2 - \frac{1}{|\lambda^*|}$.

注: 圆盘定理: 设 A 是一个 $n \times n$ 的矩阵, $\mathbb{R}_i$ 表示以 a_{ii} 为圆心, $\sum_{j=1,j\neq i}^{n} |a_{ij}|$ 的圆. 令

$$\mathbb{R}_i = \left\{ z \in \mathbb{C} \middle| |z - a_{ii}| \leq \sum_{j=1,j\neq i}^{n} |a_{ij}| \right\}$$

A 的特征值包含在 $\bigcup_{i=1}^{n} \mathbb{R}_i$ 中.

结果

1. 当 $\alpha = 5.5$ 时，特征值为 6.0000，特征向量为 $[-1, -1, -1]$，迭代次数为 10 次。
2. 当 $\alpha = 2.5$ 时，特征值为 3.0000，特征向量为 $[1, 0.5, -0.5]$，迭代次数为 13 次。
3. 当 $\alpha = 0.6$ 时，特征值为 1.0000，特征向量为 $[0, 1, 1]$，迭代次数为 9 次。

代码

```
def inv_pow(A, X, alpha, step, eps):
    n = A.size()
    A -= alpha * np.identity(n)
    l = 0
    cnt = 0
    err = 1
    while err > eps and cnt < step:
        cnt += 1
        Y = A / X
        C[cnt] = max(Y)
        dc = abs(l - C[cnt])
        Y = (1 / C[cnt]) * Y
        dv = norm(X - Y)
        err = max(dc, dv)
        X = Y
        l = C[cnt]
    l = alpha + 1 / C[cnt]
    V = X
    return C, l, V
```

11 总结

计算机科学根本上是一门抽象的科学，算法是其灵魂，数学是其基础，要求很高的实践性，而《数值计算》这门课几乎涵盖了计算机科学的这几大部分，既有适用于计算机求解的实用数值算法，又有严谨的数学推导与证明，还为许多高级算法、应用程序提供了基础设施的支持，有着很强的实践性，在许多任务中，这些基本的数值方法往往是效率的关键。

《数值计算》这门课在我的整个本科阶段的学习中也起到了承前启后的作用，既承接了之前的《高等数学》、《线性代数》等数学基础课程，又为后面更加高级的《算法分析》、《图形学》和《机器学习》等课程奠定了基础。在这门课中，我进一步体会到了数学与算法之美，能够更加注重自己抽象思维和计算思维的培养，更加注重算法功底的牢固，更加注重数学知识的深入与拓宽，更加注重实践，通过这几次实验，能够在实践中深化和检验理论。

感谢刘保东老师的辛勤付出!

参考文献

[1] Mathews, John H., and Kurtis D. Fink. Numerical methods using MATLAB. Vol. 4. Upper Saddle River, NJ: Pearson Prentice Hall, 2004.

[2] Stefan Behnel, Robert Bradshaw, Craig Citro, Lisandro Dalcin, Dag Sverre Seljebotn and Kurt Smith. Cython: The Best of Both Worlds, Computing in Science and Engineering, 13, 31-39 (2011), DOI:10.1109/MCSE.2010.118

[3] Fernando Pérez and Brian E. Granger. IPython: A System for Interactive Scientific Computing, Computing in Science & Engineering, 9, 21-29 (2007), DOI:10.1109/MCSE.2007.53

[4] Travis E, Oliphant. A guide to NumPy, USA: Trelgol Publishing, (2006).

[5] John D. Hunter. Matplotlib: A 2D Graphics Environment, Computing in Science & Engineering, 9, 90-95 (2007), DOI:10.1109/MCSE.2007.55

[6] Jones E, Oliphant E, Peterson P, et al. SciPy: Open Source Scientific Tools for Python, 2001-, http://www.scipy.org/ [Online; accessed 2019-01-01].

[7] Wes McKinney. Data Structures for Statistical Computing in Python, Proceedings of the 9th Python in Science Conference, 51-56 (2010)

[8] Meurer A, Smith CP, Paprocki M, Čertík O, Kirpichev SB, Rocklin M, Kumar A, Ivanov S, Moore JK, Singh S, Rathnayake T, Vig S, Granger BE, Muller RP, Bonazzi F, Gupta H, Vats S, Johansson F, Pedregosa F, Curry MJ, Terrel AR, Roučka Š, Saboo A, Fernando I, Kulal S, Cimrman R, Scopatz A. SymPy: symbolic computing in Python, PeerJ Computer Science 3:e103 (2017)