Homework 0 实验报告

PB20000296 郑滕飞

1、功能实现

实验要求：计算级数并给出保证误差在 1e-6 内时需要计算的项数。
代码结构：
c0.m 主程序脚本[调用 c0_count]
c0_count.m 根据给定的 x 计算 1e-6 精度的级数[调用 c0_estimateK]
c0_estimateK.m 根据给定 x 估计需要的最小项数 k

2、算法简述

注意到

$$\frac{1}{k(k+x)}=\frac{1}{x}\left(\frac{1}{k}-\frac{1}{k+x}\right)$$

裂项相消可知当 x 为正整数的时候级数有精确解

$$\varphi(x)=\frac{1}{x}\sum_{i=1}^{x}\frac{1}{i}$$

于是当 x 不为整数时，可以通过取和 x 最接近的整数估算来得到最小项数 k(注意到 $\varphi(0)=\frac{\pi^2}{6}$)，也即

$$k(x)=\min\left\{k\ \middle|\ \varphi(\bar{x})-\sum_{i=1}^{k}\frac{1}{i(i+\bar{x})}<10^{-6}\right\}$$

其中$\bar{x}$为如上所说的和 x 最接近的整数。

有了 k 的估计后，直接对前 k 项求和即可。

3、结果展示

将运行结果整理成表格如下：

x	result	k	x	result	k
0	1.6449	1000000	110	0.048019	999945
0.1	1.5346	1000000	120	0.04474	999940
0.2	1.4409	1000000	130	0.041911	999935
0.3	1.3601	1000000	140	0.039445	999930
0.4	1.2896	1000000	150	0.037274	999925
0.5	1.2274	1000000	160	0.035346	999920
0.6	1.1721	1000000	170	0.033622	999915
0.7	1.1225	1000000	180	0.032071	999910
0.8	1.0778	1000000	190	0.030667	999905
0.9	1.0371	1000000	200	0.029389	999900
1	1	1000000	210	0.028221	999895
10	0.2929	999995	220	0.02715	999890

20	0.17989	999990	230	0.026162	999885
30	0.13317	999985	240	0.025249	999880
40	0.10696	999980	250	0.024402	999875
50	0.089983	999975	260	0.023614	999870
60	0.077997	999970	270	0.022879	999865
70	0.06904	999965	280	0.022191	999860
80	0.062067	999960	290	0.021547	999855
90	0.056472	999955	300	0.020941	999850
100	0.051873	999950			

注意到，0 到 1 之间近似为 0 与 1 得到的 k 是一样的，因此这个 k 是精确的结果，而后续计算的整数值也是精确结果。

4、讨论

结果中值得注意的是，在 x 每次增加 10 时，k 似乎是以 5 的等差数列下降的。继续计算更大的数据会发现，这个 5 的等差数列并不是完全精确，但在 10 到 3000 之内也只出现了一次偏差(出现一次下降了 4 而不是 5)。

这是由于，我们实质计算的是满足

$$\sum_{i=k}^{\infty} \frac{1}{i(i+x)} < 10^{-6}$$

的最小 i，而估算可以得到在 1000000 附近的接近性。

附录-代码

```
for i = 0:10
    c0_count(i / 10.0);
end
for i = 1:30
    c0_count(i*10);
end
```

```
function res = c0_count(x)
    res = 0;
    k = 0;
    maxk = c0_estimateK(x);
    while k < maxk
        k = k + 1;
        res = res + 1/(k*(k+x));
    end
    disp([num2str(x), '  ', num2str(res), '  ', num2str(maxk - 1)]);
end
```

```
function k = c0_estimateK(x)
    a = int32(x);
    if a == 0
```

```
        sum = pi * pi / 6;
    else
        sum = 0;
        for i = 1:a
            sum = sum + 1 / double(i);
        end
        sum = sum / double(a);
    end
    k = 1;
    while sum > 1e-6
        sum = sum - 1.0 / (k*(k+double(a)));
        k = k + 1;
    end
end
```

Homework 1 实验报告

PB20000296 郑滕飞

1、功能实现

实验要求：计算某函数的均匀点与切比雪夫点的 Lagrange 插值并比较。
代码结构：
c1.m 主程序脚本[调用 c1_lagrange]
c1_lagrange.m 实现插值所需的各种函数
-- lagrange_uni 计算均匀点的 Lagrange 插值与误差
-- lagrange_ch 计算切比雪夫点的 Lagrange 插值与误差
-- count_res 给定插值点，计算所给输入在插值多项式的输出
-- count_err 计算所给插值多项式输出与真实结果的最大误差

2、算法简述

插值多项式的算法实现与书上相同，当给定点 X(0)到 X(N)时，插值的结果为

$$f(x)=\sum_{i=0}^{N} y_i \prod_{j\neq i} \frac{x-x_j}{x_i-x_j}$$

由于所给函数性质，此处由$y_i=\frac{1}{1+x_i^2}$得到最终结果。

其余的代码都是计算误差与画图的简单细节，如用 if_draw 参数确定只算误差还是进行绘制，画图时用蓝线绘制真实的函数，而红线代表拟合的多项式结果，由此可以直观对比出差异。

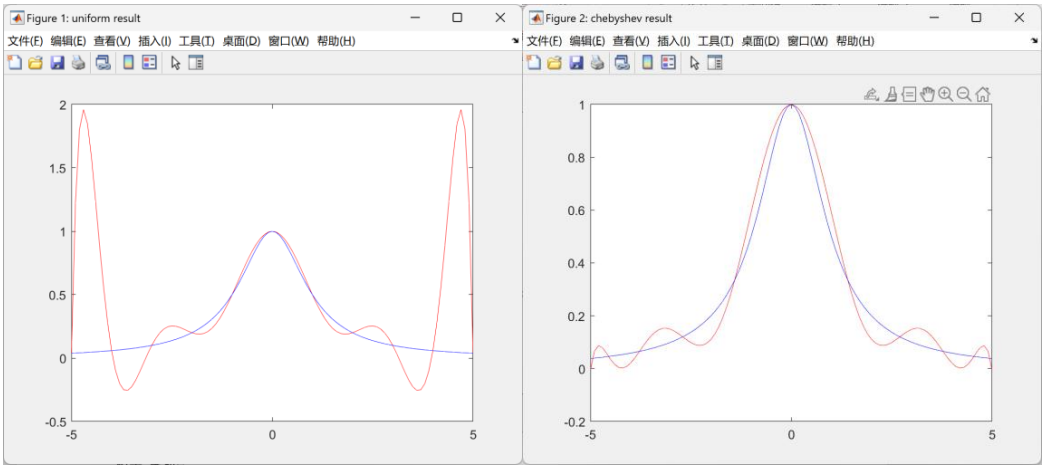
3、结果展示

控制台输出如下：

```
N=5
Max Error of grid (1): 0.43269
Max Error of grid (2): 0.55591
N=10
Max Error of grid (1): 1.9156
Max Error of grid (2): 0.10893
N=20
Max Error of grid (1): 58.2781
Max Error of grid (2): 0.015325
N=40
Max Error of grid (1): 78689.0375
Max Error of grid (2): 0.00027386
```

可以看出，插值点增多时切比雪夫点的误差能逐渐减小，而均匀选取反而会导致误差越来越大(龙格现象)。

N = 10 时绘制如下：



左图为均匀选取的结果，右图为切比雪夫点的结果，可以发现均匀选取在边界处的误差显著非常大，而切比雪夫点不存在此问题。

4、讨论

最核心的观察已经在结果展示部分给出，也即龙格函数会使得均匀取点时的插值多项式误差趋于无穷。而对切比雪夫点，在误差估计时会具有更良好的性态。

附录-代码

```
fun = c1_lagrange;
disp('N=5');
disp(['Max Error of grid (1): ', num2str(fun.lagrange_uni(5, 0))]);
disp(['Max Error of grid (2): ', num2str(fun.lagrange_ch(5, 0))]);
disp('N=10');
disp(['Max Error of grid (1): ', num2str(fun.lagrange_uni(10, 0))]);
disp(['Max Error of grid (2): ', num2str(fun.lagrange_ch(10, 0))]);
disp('N=20');
disp(['Max Error of grid (1): ', num2str(fun.lagrange_uni(20, 0))]);
disp(['Max Error of grid (2): ', num2str(fun.lagrange_ch(20, 0))]);
disp('N=40');
disp(['Max Error of grid (1): ', num2str(fun.lagrange_uni(40, 0))]);
disp(['Max Error of grid (2): ', num2str(fun.lagrange_ch(40, 0))]);
fun.lagrange_uni(10, 1);
fun.lagrange_ch(10, 1);
```

```
function Funcollect = c1_lagrange
    Funcollect.lagrange_uni = @lagrange_uni;
    Funcollect.lagrange_ch = @lagrange_ch;
    Funcollect.count_res = @count_res;
    Funcollect.count_err = @count_err;
end
```

```
function err = lagrange_uni(N, if_draw)
    X = zeros(1, N + 1);
    for i = 0:N
        X(i + 1) = 5 - 10 * i / double(N);
    end
    Xin = zeros(1, 101);
    for i = 0:100
        Xin(i + 1) = double(i) / 10 - 5;
    end
    Yout = count_res(X, Xin);
    if if_draw ~= 0
        figure('name', 'uniform result');
        plot(Xin, Yout, 'r');
        hold on;
        Yin = zeros(1, 101);
        for i = 1:101
            Yin(i) = 1 / (1 + Xin(i) * Xin(i));
        end
        plot(Xin, Yin, 'b');
    end
    err = count_err(Xin, Yout);
end

function err = lagrange_ch(N, if_draw)
    X = zeros(1, N + 1);
    for i = 0:N
        X(i + 1) = -5 * cos((2 * i + 1) * pi / (2 * N  + 2));
    end
    Xin = zeros(1, 101);
    for i = 0:100
        Xin(i + 1) = double(i) / 10 - 5;
    end
    Yout = count_res(X, Xin);
    if if_draw ~= 0
        figure('name', 'chebyshev result');
        plot(Xin, Yout, 'r');
        hold on;
        Yin = zeros(1, 101);
        for i = 1:101
            Yin(i) = 1 / (1 + Xin(i) * Xin(i));
        end
        plot(Xin, Yin, 'b');
    end
    err = count_err(Xin, Yout);
```

```
end

function res = count_res(X, Xin)
    l = length(X);
    Y = zeros(1, l);
    for i = 1:l
        Y(i) = 1 / (1 + X(i) * X(i));
    end
    lout = length(Xin);
    res = zeros(1, lout);
    for k = 1:lout
        for i = 1:l
            sum = 1;
            for j = 1:l
                if j ~= i
                    sum = sum * (Xin(k) - X(j)) / (X(i) - X(j));
                end
            end
            res(k) = res(k) + sum * Y(i);
        end
    end
end

function err = count_err(Xin, Yout)
    err = 0;
    for i = 1:length(Xin)
        err_now = 1 / (1 + Xin(i) * Xin(i)) - Yout(i);
        if abs(err_now) > err
            err = abs(err_now);
        end
    end
end
```

Homework 2 实验报告

PB20000296 郑滕飞

1、功能实现

实验要求：计算某函数的均匀点与切比雪夫点的 Newton 插值并比较。
代码结构：
c2.m 主程序脚本[调用 c2_newton]
c2_newton.m 实现插值所需的各种函数
-- newton_uni 计算均匀点的 Newton 插值与误差
-- newton_ch 计算切比雪夫点的 Newton 插值与误差
-- count_res 给定插值点，计算所给输入在插值多项式的输出
-- count_err 计算所给插值多项式输出与真实结果的最大误差

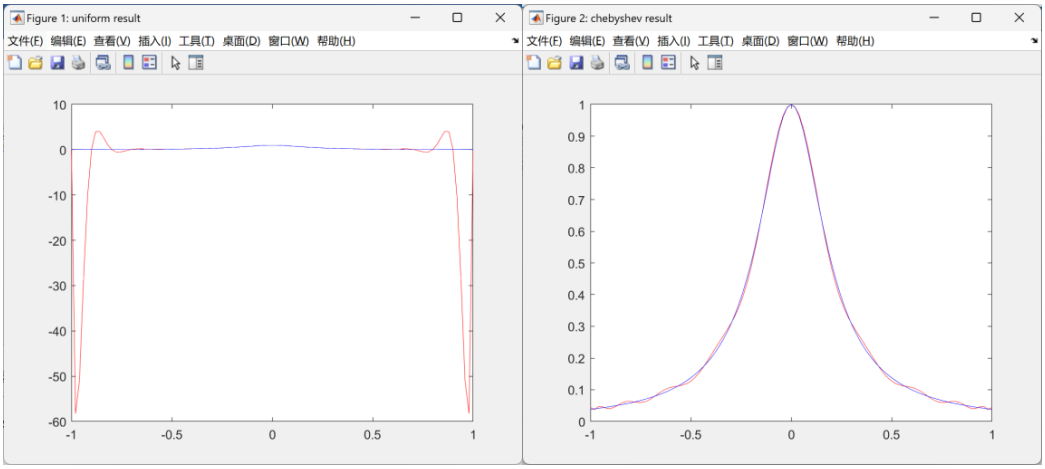
2、算法简述

除了改变数字外，区别主要在 count_res 的构造方式。牛顿插值的系数计算方法已在教材上详细给出，其他操作与 Lagrange 插值相同：

```
c[0] = y[0]
for k = 1 to n
    d = x[k] – x[k-1]
    u = c[k-1]
    for i = k-2 downto 0
        u = u(x[k] - x[i]) + c[i]
        d = d(x[k] – x[i])
    c[k] = (y[k] – u) / d
```

3、结果展示

N=20 时绘图如下(左图为均匀选取的结果，右图为切比雪夫点的结果)：

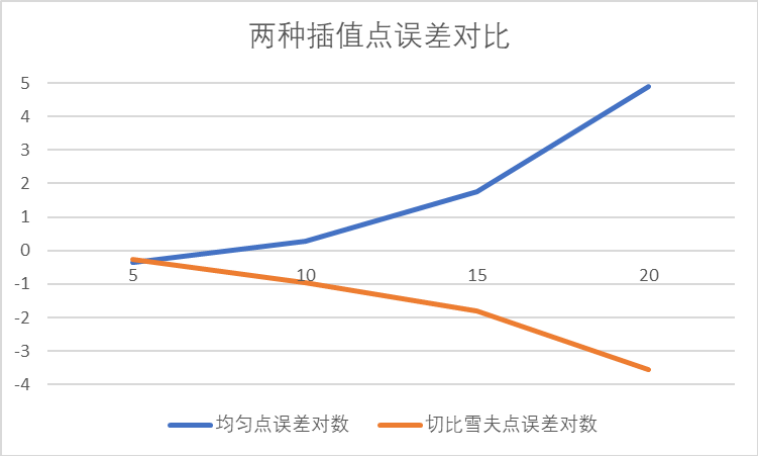


控制台输出如下：

```
N=5
```

```
Max Error of grid (1): 0.43269
Max Error of grid (2): 0.55591
N=10
Max Error of grid (1): 1.9156
Max Error of grid (2): 0.10893
N=20
Max Error of grid (1): 58.2781
Max Error of grid (2): 0.015325
N=40
Max Error of grid (1): 78689.0462
Max Error of grid (2): 0.00027387
```

作图如下(由于量级差别较大，这里对误差取以 10 为底的对数)：



4、讨论

与 Lagrange 插值时的讨论相同，即切比雪夫点具有更好的误差性态。此外，这里的牛顿插值法计算系数已经使用了类似动态规划的操作，因此复杂度在$O(n^2)$。

附录-代码

```
fun = c2_newton;
disp('N=5');
disp(['Max Error of grid (1): ', num2str(fun.newton_uni(5, 0))]);
disp(['Max Error of grid (2): ', num2str(fun.newton_ch(5, 0))]);
disp('N=10');
disp(['Max Error of grid (1): ', num2str(fun.newton_uni(10, 0))]);
disp(['Max Error of grid (2): ', num2str(fun.newton_ch(10, 0))]);
disp('N=20');
disp(['Max Error of grid (1): ', num2str(fun.newton_uni(20, 0))]);
disp(['Max Error of grid (2): ', num2str(fun.newton_ch(20, 0))]);
disp('N=40');
disp(['Max Error of grid (1): ', num2str(fun.newton_uni(40, 0))]);
```

```
disp(['Max Error of grid (2): ', num2str(fun.newton_ch(40, 0))]);
fun.newton_uni(20, 1);
fun.newton_ch(20, 1);
```

```
function Funcollect = c2_newton
    Funcollect.newton_uni = @newton_uni;
    Funcollect.newton_ch = @newton_ch;
    Funcollect.count_res = @count_res;
    Funcollect.count_err = @count_err;
end

function err = newton_uni(N, if_draw)
    X = zeros(1, N + 1);
    for i = 0:N
        X(i + 1) = 1 - 2 * i / double(N);
    end
    Xin = -1:0.02:1;
    Yout = count_res(X, Xin);
    if if_draw ~= 0
        figure('name', 'uniform result');
        plot(Xin, Yout, 'r');
        hold on;
        Yin = zeros(1, 101);
        for i = 1:101
            Yin(i) = 1 / (1 + 25 * Xin(i) * Xin(i));
        end
        plot(Xin, Yin, 'b');
    end
    err = count_err(Xin, Yout);
end

function err = newton_ch(N, if_draw)
    X = zeros(1, N + 1);
    for i = 0:N
        X(i + 1) = -cos((2 * i + 1) * pi / (2 * N  + 2));
    end
    Xin = -1:0.02:1;
    Yout = count_res(X, Xin);
    if if_draw ~= 0
        figure('name', 'chebyshev result');
        plot(Xin, Yout, 'r');
        hold on;
        Yin = zeros(1, 101);
        for i = 1:101
            Yin(i) = 1 / (1 + 25 * Xin(i) * Xin(i));
```

```
        end
        plot(Xin, Yin, 'b');
    end
    err = count_err(Xin, Yout);
end

function res = count_res(X, Xin)
    l = length(X);
    Y = zeros(1, l);
    for i = 1:l
        Y(i) = 1 / (1 + 25 * X(i) * X(i));
    end

    c = zeros(1, l);
    c(1) = Y(1);
    for k = 2:l
        d = X(k) - X(k-1);
        u = c(k - 1);
        for i = k-2:-1:1
            u = u * (X(k) - X(i)) + c(i);
            d = d * (X(k) - X(i));
        end
        c(k) = (Y(k) - u) / d;
    end

    lout = length(Xin);
    res = zeros(1, lout);
    for t = 1:lout
        res(t) = c(l);
        for i = l-1:-1:1
            res(t) = res(t) * (Xin(t) - X(i)) + c(i);
        end
    end
end

function err = count_err(Xin, Yout)
    err = 0;
    for i = 1:length(Xin)
        err_now = 1 / (1 + 25 * Xin(i) * Xin(i)) - Yout(i);
        if abs(err_now) > err
            err = abs(err_now);
        end
    end
end
```

Homework 3 实验报告

PB20000296 郑滕飞

1、功能实现

实验要求：计算某函数的一次样条与给定端点导数的三次样条并比较。
代码结构：
c3.m 主程序脚本[调用 c3_linear、c3_cubic]
c2_linear.m 计算一次样条与其误差
c3_cubic.m 计算符合要求的三次样条与其误差

2、算法简述

对于一次线性插值，由于我们需要估计的是每个插值区间中点的值，事实上就是以左右两端点的平均进行估计，也即

$$f\left(\frac{2i+1}{2n}\right)=\frac{\exp\left(\frac{i}{n}\right)+\exp\left(\frac{i+1}{n}\right)}{2}$$

对于三次样条，与书上的自然样条不同，我们的边界条件是$S_0'(0)$与$S_{n-1}'(1)$，且由等距选取所有的 h 都为共同的，因此由书可构造方程组为：

$$\begin{pmatrix} h/3 & h/6 & & & & \\ h & 4h & h & & & \\ & h & 4h & h & & \\ & & & \cdots & & \\ & & & h & 4h & h \\ & & & & h/6 & h/3 \end{pmatrix}\begin{pmatrix} z_0 \\ z_1 \\ z_2 \\ \vdots \\ z_{n-1} \\ z_n \end{pmatrix}=\begin{pmatrix} (y_1-y_0)/h-1 \\ 6(y_2-2y_1+y_0)/h \\ 6(y_3-2y_2+y_1)/h \\ \vdots \\ 6(y_n-2y_{n-1}+y_{n-2})/h \\ (y_{n-1}-y_n)/h+\mathrm{e} \end{pmatrix}$$

求解出z_i后，仍与书上相同方法计算插值，即

$$f\left(\frac{2i+1}{2n}\right)=\frac{h}{2}\left(\frac{h}{2}\left(\frac{h}{2}\frac{z_{i+1}-z_i}{6h}+\frac{z_i}{2}\right)-\frac{z_{i+1}h}{6}-\frac{z_ih}{3}+\frac{y_{i+1}-y_i}{3}\right)+y_i$$

上方$h=\frac{1}{n}, y_i=\exp\left(\frac{i}{n}\right)$。

3、结果展示

输出结果列表如下：

线性样条			三次样条		
节点数	误差	阶数	结点数	误差	阶数
5	1.2e-2	/	5	1.1e-5	/
10	3.2e-3	1.93	10	7.0e-7	3.97
20	8.3e-4	1.96	20	4.4e-8	3.99
40	2.1e-4	1.98	40	2.8e-9	3.99

4、讨论

从结果上来说，三次样条无论是误差大小还是收敛阶数都远好于一次样条，而其付出的一方面是更多信息的估算(两端点导数值的额外提供)：当两端点导数值不准确时，收敛的阶数就会显著下降，甚至不如一次样条(如假设两端点导数为 0 只有一阶收敛，而自然样条的性态也并不好)；另一方面则是更大的计算开销(一次样条计算所有时间复杂度是$O(n)$，而三次样条则为$O(n^2)$，如果只需要单个结果，一次样条复杂度只有$O(1)$，三次样条不变)，这是由于一次样条只需要局部数据，而更高次则引入了整体信息。

附录-代码

```
disp('linear error:');
now = c3_linear(5);
disp(['N=5:  ', num2str(now)]);
past = now;
now = c3_linear(10);
disp(['N=10: ', num2str(now), ' ord=', num2str(log(past/now) / log(2))]);
past = now;
now = c3_linear(20);
disp(['N=20: ', num2str(now), ' ord=', num2str(log(past/now) / log(2))]);
past = now;
now = c3_linear(40);
disp(['N=40: ', num2str(now), ' ord=', num2str(log(past/now) / log(2))]);
disp('cubic error:');
now = c3_cubic(5);
disp(['N=5:  ', num2str(now)]);
past = now;
now = c3_cubic(10);
disp(['N=10: ', num2str(now), ' ord=', num2str(log(past/now) / log(2))]);
past = now;
now = c3_cubic(20);
disp(['N=20: ', num2str(now), ' ord=', num2str(log(past/now) / log(2))]);
past = now;
now = c3_cubic(40);
disp(['N=40: ', num2str(now), ' ord=', num2str(log(past/now) / log(2))]);
```

```
function err = c3_linear(N)
    step = 1 / double(2*N);
    err = 0;
    for i = step:2*step:1
        now = exp(i) - (exp(i-step) + exp(i+step)) / 2;
        if abs(now) > err
            err = abs(now);
        end
    end
```

```
end
```

```
function err = c3_cubic(N)
    h = 1 / double(N);
    y = exp(0:h:1);
    A = zeros(N + 1);
    b = zeros(N + 1, 1);
    A(1, 1) = h / 3;
    A(1, 2) = h / 6;
    b(1) = (y(2) - y(1)) / h - 1;
    for i = 2:N
        A(i, i-1) = h;
        A(i, i+1) = h;
        A(i, i) = 4 * h;
        b(i) = 6 / h * (y(i+1) - 2 * y(i) + y(i-1));
    end
    A(N+1, N) = h / 6;
    A(N+1, N+1) = h / 3;
    b(N+1) = (y(N) - y(N+1)) / h + exp(1);
    z = sparse(A) \ b;
    err = 0;
    for i = 1:N
        t = (2 * i - 1) * h / 2;
        res = exp(t);
        sim = (z(i+1) - z(i)) / (6 * h);
        sim = sim*h/2 + z(i) / 2;
        sim = sim*h/2 - h * z(i+1)/6 - h * z(i)/3 + (y(i+1) - y(i)) / h;
        sim = sim*h/2 + y(i);
        now = abs(res - sim);
        if now > err
            err = now;
        end
    end
end
```

Homework 4 实验报告

PB20000296 郑滕飞

1、功能实现

实验要求：实现利用 Richardson 外推计算导数。
代码结构：
c4.m 主程序脚本[调用 c4_R]
c4_R.m 计算 Richardson 外推得到的逼近矩阵

2、算法简述

Richardson 外推实际上是通过动态规划的思想构造导数的逼近矩阵，出于 MATLAB 起始下标 1，需要构造(M+1, M+1)方阵，并迭代：

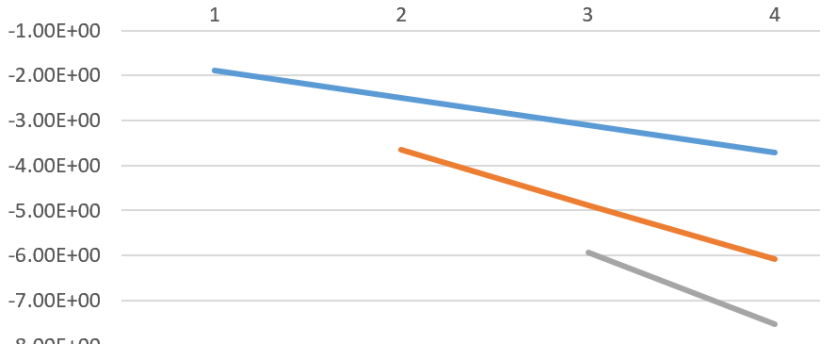
$$D_{n1}=\frac{f(x+2^{1-n})-f(x-2^{1-n})}{2^{2-n}},n=1,\dots,M+1$$

$$D_{nk}=\frac{4^{k-1}}{4^{k-1}-1}D_{n,k-1}-\frac{1}{4^{k-1}-1}D_{n-1,k-1},1<k\le n\le M+1$$

最终输出整个方阵，事实上一般以$D_{M+1,M+1}$作为最终近似。

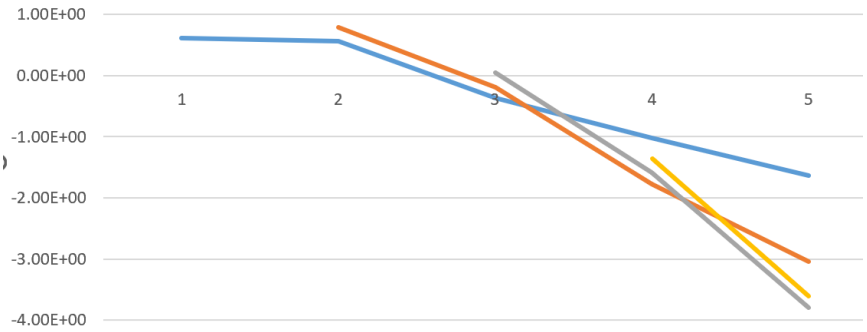
3、结果展示

ln(x)在 3 处的精确结果为 1/3，D(i,j)与其误差取 log 后作图：



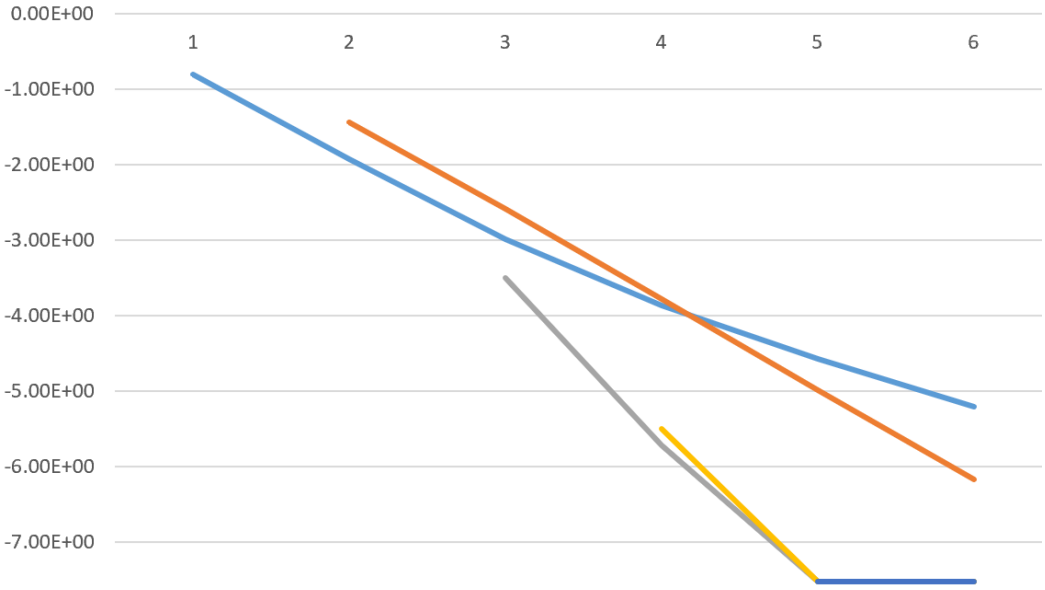
其中蓝、橙、灰线为 D(n,1)、D(n,2)、D(n,3)。

tan(x)在 arcsin(0.8)处精确结果为 25/9，D(i,j)与其误差取 log 后作图：



这里的黄线为 D(n,4)。

sin(x^2+x/3)在 0 处精确结果为 1/3，D(i,j)与其误差取 log 后作图：



这里的深蓝线为 D(n,5)。由于输出是 7 位有效数字，深蓝线、黄线、灰线的重合部分代表已经到有效数字下的最优结果。

4、讨论

六位有效数字下观察，Richardson 外推在很小次数的迭代下就得到了精确的结果，与理论的收敛速度一致。此外，其$O(M^2)$复杂度的也易于编写、计算，是导数的良好近似算法。

附录-代码

```
disp('Task 1:');
f = @(x)log(x);
M = 3;
D = c4_R(f, 3, M);
for i = 1:M+1
    for j = 1:i
        fprintf('%ld\t', D(i,j));
    end
    fprintf('\n');
end
fprintf('\n');

disp('Task 2:');
f = @(x)tan(x);
M = 4;
D = c4_R(f, asin(0.8), M);
```

```
for i = 1:M+1
    for j = 1:i
        fprintf('%ld\t', D(i,j));
    end
    fprintf('\n');
end
fprintf('\n');

disp('Task 3:');
f = @(x)sin(x^2 + x/3);
M = 5;
D = c4_R(f, 0, M);
for i = 1:M+1
    for j = 1:i
        fprintf('%ld\t', D(i,j));
    end
    fprintf('\n');
end
```

```
function D = c4_R(f, x, M)
    D = zeros(M + 1);
    h = 1;
    for i = 1:M+1
        D(i, 1) = phi(f, h, x);
        h = h / 2;
    end
    for k = 2 : M+1
        for n = k : M+1
            r = 1 / (4^(k - 1) - 1);
            D(n, k) = (r + 1) * D(n,k-1) - r * D(n-1,k-1);
        end
    end
end

function y = phi(f, h, x)
    y = (f(x + h) - f(x - h)) / (2 * h);
end
```

Homework 5 实验报告

PB20000296 郑滕飞

1、功能实现

实验要求：实现不同的数值积分估计方式。
代码结构：
c5.m 主程序脚本[调用 c5_simp、c5_trap、c5_Lagrange]
c5_simp.m 计算复化 Simpson 方法数值积分结果
c5_trap.m 计算复化梯形方法数值积分的结果
c5_lagrange.m 计算拉格朗日插值估算数值积分的结果

2、算法简述

a. 两种方式估算$\int_0^4 \sin x \,\mathrm{d}\, x$

复化 Simpson 方法与复化梯形方法在本题中的估计公式为($h = \frac{4}{N}$):

$$\int_0^4 \sin x \,\mathrm{d}\, x \approx \frac{h}{3}\sum_{i=1}^{N/2}(\sin(2i-2)h + 4\sin(2i-1)h + \sin 2ih)$$

$$\int_0^4 \sin x \,\mathrm{d}\, x \approx \frac{h}{2}\left(2\sum_{i=1}^{N-1}\sin ih + \sin 4\right)$$

b. 等距与(近似)切比雪夫点的拉格朗日插值估算$\int_{-1}^{1}\frac{1}{1+25x^2}\mathrm{d}\, x$

拉格朗日插值本题中的估计公式则为:

$$\int_{-1}^{1}\frac{1}{1+25x^2}\mathrm{d}\, x \approx \sum_{\mathrm{i}=0}^{\mathrm{N}} y_i \int_{-1}^{1} l_i(x)\mathrm{d}\, x$$

其中x_i为第 i 个节点，且

$$y_i = \frac{1}{1+25x_i^2}, l_i(x) = \prod_{j\neq i}\frac{x-x_j}{x_i-x_j}$$

在 MATLAB 中，我们通过符号计算来实现$l_i(x)$的积分，以达到精确的效果。

3、结果展示

控制台完整输出如下:

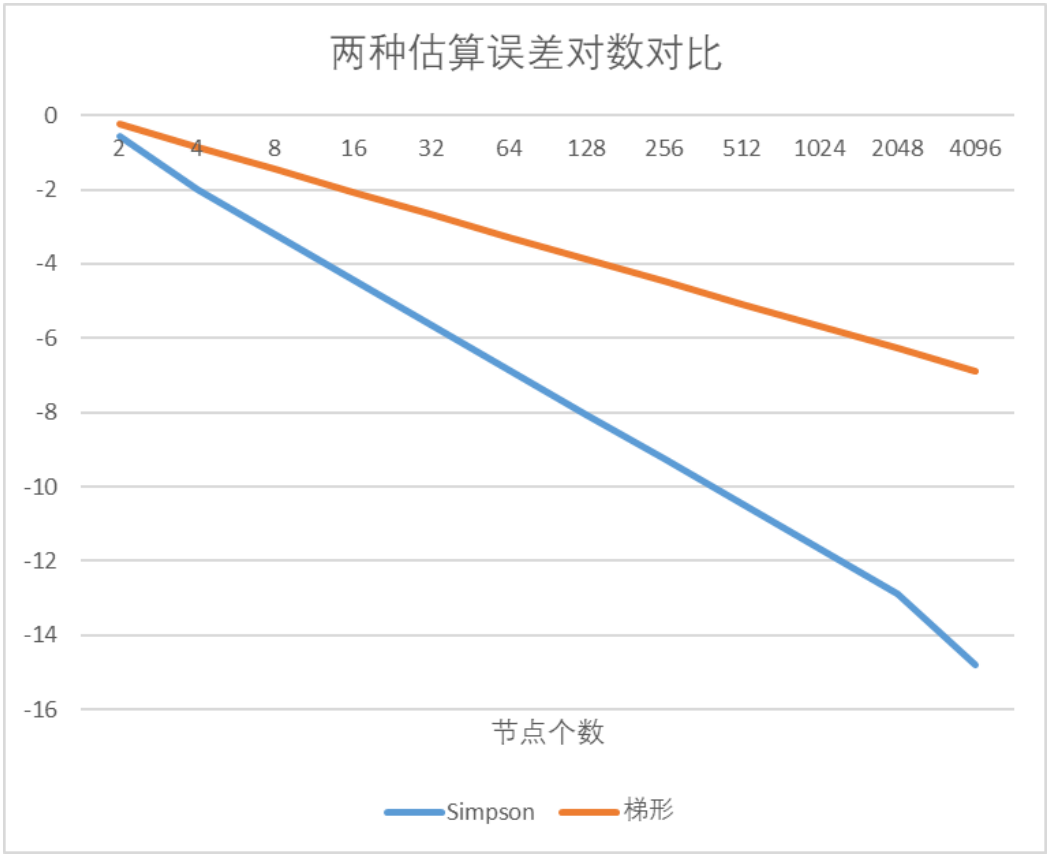
```
Task1:
N       err1              ord1          err2              ord2
2       2.666145e-01                    5.918513e-01
4       1.040849e-02      4.678923      1.401564e-01      2.078197
8       5.917309e-04      4.136676      3.459531e-02      2.018390
```

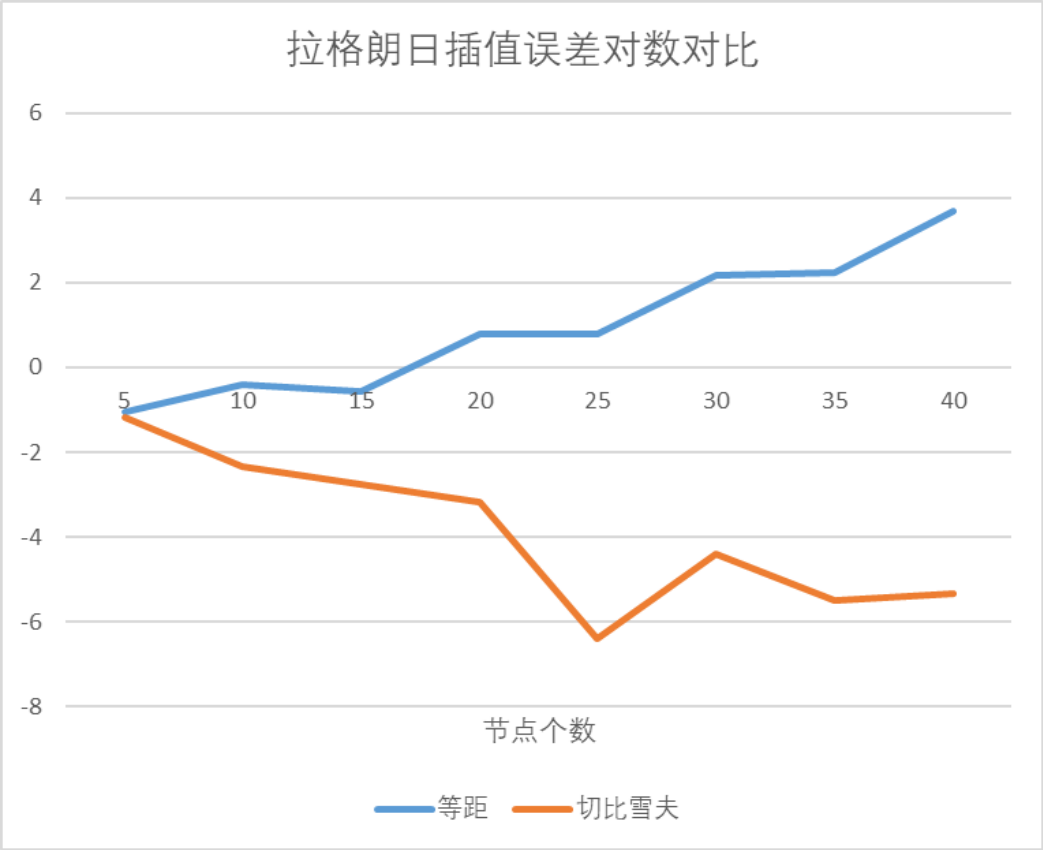
16	3.615514e-05	4.032669	8.621712e-03	2.004530
32	2.247077e-06	4.008079	2.153743e-03	2.001128
64	1.402463e-07	4.002014	5.383305e-04	2.000282
128	8.762338e-09	4.000503	1.345761e-04	2.000070
256	5.475955e-10	4.000133	3.364360e-05	2.000018
512	3.422485e-11	3.999995	8.410875e-06	2.000004
1024	2.140066e-12	3.999317	2.102717e-06	2.000001
2048	1.321165e-13	4.017772	5.256792e-07	2.000000
4096	1.554312e-15	6.409391	1.314198e-07	2.000000

Task2:

N	res1	err1	res2	err2
5	0.461538	8.782185e-02	0.481140	6.821986e-02
10	0.934660	3.852998e-01	0.554086	4.725383e-03
15	0.831112	2.817515e-01	0.547586	1.774181e-03
20	-5.369910	5.919271e+00	0.550011	6.504848e-04
25	-5.399861	5.949221e+00	0.549360	3.946928e-07
30	153.797931	1.532486e+02	0.549322	3.866523e-05
35	173.880368	1.733310e+02	0.549357	3.234144e-06
40	-4912.416912	4.912966e+03	0.549365	4.506167e-06

分别作图如下:





4、讨论

第一部分的结果中，可以证实复化 Simpson 方法比复化梯形方法的更快收敛速度，与理论上分别为四阶、二阶相符合(由于 N 与误差在都取对数后近似线性，实际关系近似为确定的指数函数)，且在更细分时可能实际收敛速度更快。

第二部分的结果中，可发现拉格朗日函数估计积分结果的误差受插值点影响非常大，且虽然在插值点的近似较良好时可以得到很好的结果，但它的开销非常大，因此复化 Simpson 的估计更加易用。

附录-代码

```
fprintf('Task1:\n');
f = @(x) sin(x);
r = 4;
res = 1.653643620863612;
fprintf('N\t\terr1\t\t\tord1\t\t\terr2\t\t\tord2\n');
N = 2;
err1_old = abs(c5_simp(f, r, N) - res);
err2_old = abs(c5_trap(f, r, N) - res);
fprintf('2\t\t%e\t\t\t\t\t%e\n', err1_old, err2_old);
for i = 1:11
```

```
    N = N * 2;
    err1 = abs(c5_simp(f, r, N) - res);
    err2 = abs(c5_trap(f, r, N) - res);
    ord1 = log (err1_old / err1) / log(2);
    ord2 = log (err2_old / err2) / log(2);
    fprintf('%d\t\t%e\t\t%f\t\t%e\t\t%f\n', N, err1, ord1, err2, ord2);
    err1_old = err1;
    err2_old = err2;
end
fprintf('\nTask2:\n');
f = @(x)(1+25*x.^2).^(-1);
a = c5_lagrange;
res = 0.5493603067780064;
fprintf('N\tres1\t\t\terr1\t\t\tres2\t\t\terr2\n');
for i = 1:8
    res1 = a.lagrange_uni(f, 5*i);
    res2 = a.lagrange_ch(f, 5*i);
    err1 = abs(res - res1);
    err2 = abs(res - res2);
    fprintf('%d\t%f\t\t%e\t\t%f\t\t%e\n', 5*i, res1, err1, res2, err2);
end
```

```
function res = c5_simp(f, r, N)
   arr = f([0:r/N:r]);
   res = arr(1) + arr(N+1);
   for i = 1:N/2
       res = res + 2 * arr(2*i - 1) + 4 * arr(2 * i);
   end
   res = res * r / (3 * N);
end
```

```
function res = c5_trap(f, r, N)
   arr = f([0:r/N:r]);
   res = arr(1) + arr(N+1);
   for i = 2:N
       res = res + 2 * arr(i);
   end
   res = res * r / (2 * N);
end
```

```
function Funcollect = c5_lagrange
    Funcollect.lagrange_uni = @lagrange_uni;
    Funcollect.lagrange_ch = @lagrange_ch;
end

function res = lagrange_uni(f, N)
    X = [-1:2/N:1];
```

```
    Xin = f(X);
    res = count_res(X, Xin);
end

function res = lagrange_ch(f, N)
    X = zeros(1, N + 1);
    for i = 0:N
        X(i + 1) = -cos((i + 1) * pi / (N + 2));
    end
    Xin = f(X);
    res = count_res(X, Xin);
end

function res = count_res(X, Xin)
    syms x;
    N = length(Xin);
    p = 0;
    for i = 1:N
        temp = 1;
        for j = 1:N
            if j ~= i
                temp = temp * (x - X(j)) / (X(i) - X(j));
            end
        end
        p = p + temp * Xin(i);
    end
    res = double(int(p, x, -1, 1));
end
```

Homework 6 实验报告

PB20000296 郑滕飞

1、功能实现

实验要求：实现复化梯形积分与复化三点高斯积分并对比精度。

代码结构：

c6.m 主程序脚本[调用 c6_trap、c6_gauss]

c6_trap.m 计算复化梯形方法数值积分的结果

c6_gauss.m 计算复化三点高斯方法数值积分的结果

2、算法简述

复化梯形方法在分为 N 个区间时公式为

$$\int_a^b f(x)\mathrm{d}x \approx \frac{b-a}{N}\left(\sum_{i=1}^{N-1} f\left(a+i\frac{b-a}{N}\right)+\frac{f(a)}{2}+\frac{f(b)}{2}\right)$$

而任意区间的三点 Gauss 积分为(下方计算来自作业)：

$$\int_a^b f(x)\mathrm{d}x = \frac{b-a}{2}\int_{-1}^{1} f\left(\frac{b-a}{2}x+\frac{a+b}{2}\right)\mathrm{d}x$$

$$\approx \frac{b-a}{2}\left(\frac{5}{9}f\left(-\frac{b-a}{2}\sqrt{\frac{3}{5}}+\frac{a+b}{2}\right)+\frac{8}{9}f\left(\frac{a+b}{2}\right)+\frac{5}{9}f\left(\frac{b-a}{2}\sqrt{\frac{3}{5}}+\frac{a+b}{2}\right)\right)$$

于是将整个区间分为 N 个子区间后再对每个估计即得到复化的三点 Gauss 积分方法。

3、结果展示

控制台直接输出为：

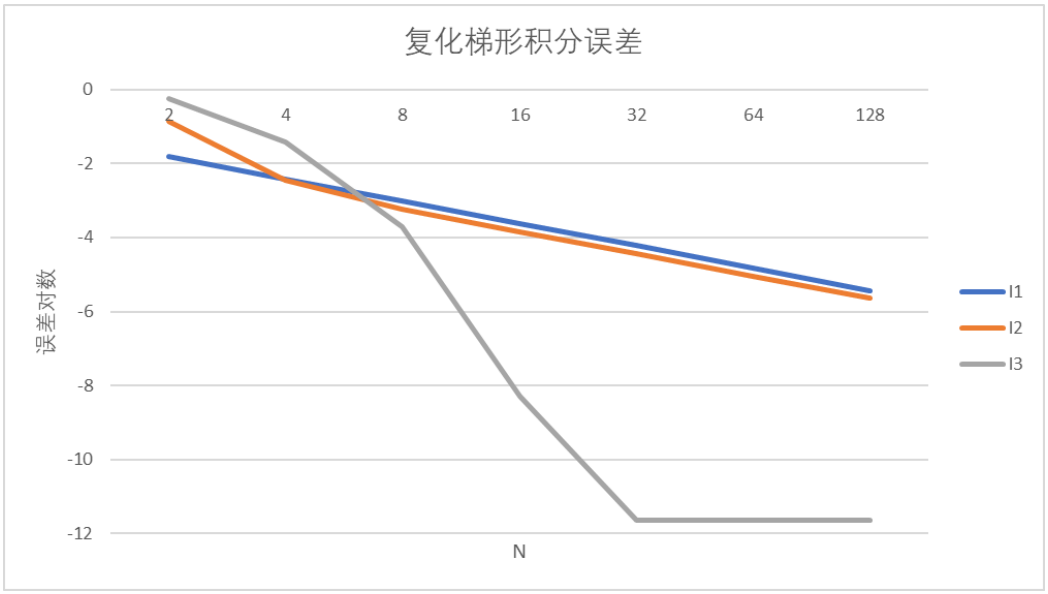
Trap:

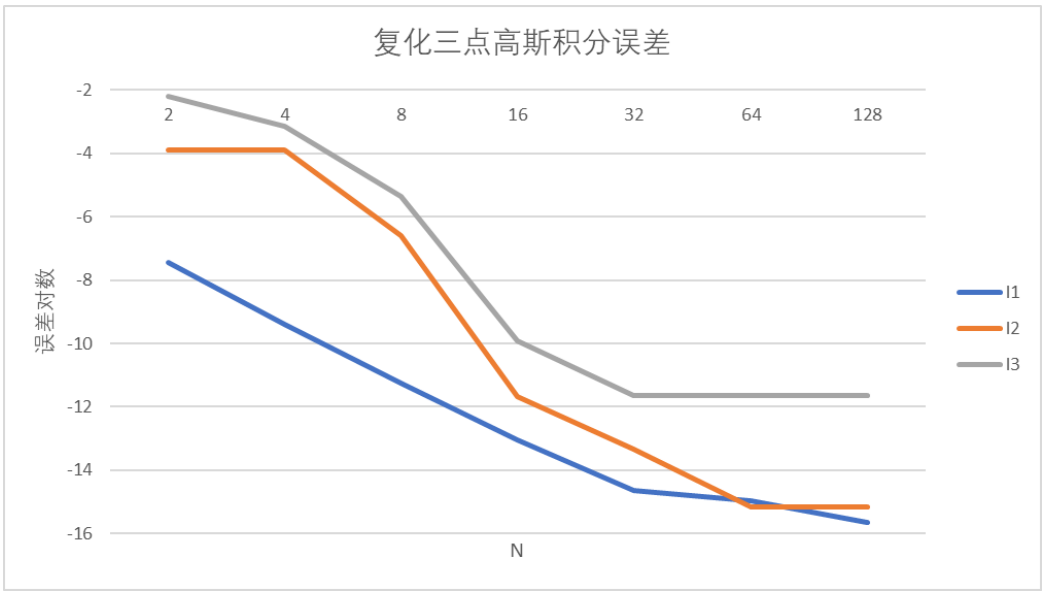
N	err1	ord1	err2	ord2	err3	ord3
2	1.545388e-02		1.330059e-01		5.611915e-01	
4	3.840035e-03	2.008778	3.594101e-03	5.209715	3.759270e-02	3.899969
8	9.585180e-04	2.002242	5.642612e-04	2.671196	1.927882e-04	7.607292
16	2.395360e-04	2.000563	1.440819e-04	1.969474	5.120245e-09	15.200444
32	5.987816e-05	2.000141	3.603799e-05	1.999299	2.331912e-12	11.100483
64	1.496917e-05	2.000035	9.010592e-06	1.999825	2.331912e-12	0.000000
128	3.742271e-06	2.000009	2.252716e-06	1.999956	2.331912e-12	0.000000

Gauss:

N	err1	ord1	err2	ord2	err3	ord3
2	3.611056e-08		1.267599e-04		6.116555e-03	
4	4.021533e-10	6.488531	1.259308e-04	0.009468	7.383276e-04	3.050386
8	5.743184e-12	6.129751	2.457990e-07	9.000937	4.326077e-06	7.415058
16	8.837375e-14	6.022089	2.071232e-12	16.856630	1.173559e-10	15.169881
32	2.220446e-15	5.314697	4.640732e-14	5.479993	2.329248e-12	5.654882
64	1.110223e-15	1.000000	6.661338e-16	6.122397	2.331468e-12	-0.001375
128	2.220446e-16	2.321928	6.661338e-16	0.000000	2.337686e-12	-0.003842

将误差取对数作图得到：





4、讨论

由于涉及到浮点数精度问题，第三个积分误差达到 1e-12，前两个积分误差达到 1e-15 即可视为收敛。从作图可直观看出，三点高斯估计的收敛速度要远快于梯形估计，这某种意义上可以解释为：三点高斯是用五次多项式进行近似的，而梯形估计基本是一次近似。当函数的性态波动不大时(例如第三个例子)，两种方法都能较快收敛。但对前两个波动较大的函数，高次近似会更为准确。

不过，从收敛阶的角度，梯形积分的收敛阶更加稳定，体现在误差对数图像上更接近直线。

附录-代码

```
f1 = @(x) exp(-x.^2);
res1 = 0.7468241328124279;
f2 = @(x) (1 + x.^2) .^(-1);
res2 = 1.325817663668032;
f3 = @(x) (2 + cos(x)) .^(-1);
res3 = 3.627598728470767;
fprintf("Trap:\n");
fprintf('N\t\terr1\t\t\tord1\t\t\terr2\t\t\tord2\t\t\terr3\t\t\tord3\n'
);
N = 2;
err1_old = abs(c6_trap(f1, 0, 1, N) - res1);
err2_old = abs(c6_trap(f2, 0, 4, N) - res2);
err3_old = abs(c6_trap(f3, 0, 2 * pi, N) - res3);
fprintf('2\t\t%e\t\t\t\t\t%e\t\t\t\t\t%e\n',     err1_old,     err2_old,
err3_old);
for i = 1:6
    N = N * 2;
    err1 = abs(c6_trap(f1, 0, 1, N) - res1);
    err2 = abs(c6_trap(f2, 0, 4, N) - res2);
    err3 = abs(c6_trap(f3, 0, 2 * pi, N) - res3);
    ord1 = log (err1_old / err1) / log(2);
    ord2 = log (err2_old / err2) / log(2);
    ord3 = log (err3_old / err3) / log(2);
    fprintf('%d\t\t%e\t\t%f\t\t%e\t\t%f\t\t%e\t\t%f\n', ...
        N, err1, ord1, err2, ord2, err3, ord3);
    err1_old = err1;
    err2_old = err2;
    err3_old = err3;
end
fprintf("\nGauss:\n");
fprintf('N\t\terr1\t\t\tord1\t\t\terr2\t\t\tord2\t\t\terr3\t\t\tord3\n'
);
N = 2;
err1_old = abs(c6_gauss(f1, 0, 1, N) - res1);
err2_old = abs(c6_gauss(f2, 0, 4, N) - res2);
err3_old = abs(c6_gauss(f3, 0, 2 * pi, N) - res3);
fprintf('2\t\t%e\t\t\t\t\t%e\t\t\t\t\t%e\n',     err1_old,     err2_old,
err3_old);
for i = 1:6
    N = N * 2;
    err1 = abs(c6_gauss(f1, 0, 1, N) - res1);
    err2 = abs(c6_gauss(f2, 0, 4, N) - res2);
    err3 = abs(c6_gauss(f3, 0, 2 * pi, N) - res3);
    ord1 = log (err1_old / err1) / log(2);
```

```
    ord2 = log (err2_old / err2) / log(2);
    ord3 = log (err3_old / err3) / log(2);
    fprintf('%d\t\t%e\t\t%f\t\t%e\t\t%f\t\t%e\t\t%f\n', ...
        N, err1, ord1, err2, ord2, err3, ord3);
    err1_old = err1;
    err2_old = err2;
    err3_old = err3;
end
```

```
function res = c6_trap(f, a, b, N)
    val = f([a:(b-a)/N:b]);
    val(1) = val(1) / 2;
    val(N+1) = val(N+1) / 2;
    res = sum(val) * (b - a) / N;
end
```

```
function res = c6_gauss(f, a, b, N)
    point = [a:(b-a)/N:b];
    res = 0;
    con = sqrt(3/5);
    for i = 1:N
        x = point(i);
        y = point(i+1);
        c = (x + y) / 2;
        d = (y - x) / 2;
        res = res + f(c + d * con) * 5 + f(c - d * con) * 5 + f(c) * 8;
    end
    res = res * (b - a) / (18 * N);
end
```

Homework 7 实验报告

PB20000296 郑滕飞

1、功能实现

实验要求：实现龙贝格方法估算积分。
代码结构：
c7.m 主程序脚本[调用 c6_R]
c7_R.m 计算龙贝格矩阵

2、算法简述

龙贝格算法通过动态规划的思想进行积分的近似。由 MATLAB 下标为 1 开始，M+1 行龙贝格算法矩阵构造应为：

$$R(1,1)=\frac{f(a)+f(b)}{2}$$

$$R(n,1)=\frac{R(n-1,1)}{2}+\frac{b-a}{2^{n-1}}\sum_{i=1}^{2^{n-1}}f\left(a+(2i-1)\frac{b-a}{2^{n-1}}\right),1<n\le M+1$$

$$R(n,m)=R(n,m-1)+\frac{1}{4^{m-1}-1}\big(R(n,m-1)-R(n-1,m-1)\big),1<m\le n\le M+1$$

对于要求计算的三个积分，前两个利用极限

$$\lim_{x\to 0}\frac{\sin x}{x}=1$$

$$\lim_{x\to 0}\frac{\cos x-\exp x}{\sin x}=-1$$

定义不连续点的值。

而第三个，变换可得

$$\int_1^{\infty}(x\exp x)^{-1}\mathrm{d}\,x=\int_0^1\frac{\exp(-1/t)}{t}\mathrm{d}\,t$$

而不连续处有

$$\lim_{t\to 0^+}\frac{\exp(-1/t)}{t}=0$$

于是可进行积分计算。

3、结果展示

控制台直接输出为：

```
Task1:
-0.079265
0.439793    0.612813
0.694514    0.779420    0.790527
0.820691    0.862750    0.868305    0.869540
0.883485    0.904416    0.907194    0.907811    0.907962
0.914809    0.925250    0.926639    0.926947    0.927022    0.927041
0.930452    0.935666    0.936361    0.936515    0.936553    0.936562    0.936564
```

Task2:

-2.383394						
-2.191697	-2.127798					
-2.182764	-2.179787	-2.183253				
-2.205099	-2.212544	-2.214728	-2.215228			
-2.223417	-2.229523	-2.230655	-2.230908	-2.230969		
-2.234395	-2.238054	-2.238623	-2.238750	-2.238780	-2.238788	
-2.240341	-2.242323	-2.242607	-2.242671	-2.242686	-2.242690	-2.242691

Task3:

0.183940						
0.227305	0.241760					
0.219834	0.217344	0.215716				
0.219351	0.219190	0.219313	0.219370			
0.219384	0.219394	0.219408	0.219410	0.219410		
0.219384	0.219384	0.219383	0.219383	0.219383	0.219383	
0.219384	0.219384	0.219384	0.219384	0.219384	0.219384	0.219384

将与精确结果的误差对数列表分析：

Task1

0.01087						
-0.2956	-0.4772					
-0.5993	-0.7782	-0.8081				
-0.9017	-1.0792	-1.1091	-1.1161			
-1.2034	-1.3802	-1.4102	-1.4171	-1.4188		
-1.5048	-1.6812	-1.7112	-1.7182	-1.7199	-1.7203	
-1.806	-1.9823	-2.0122	-2.0192	-2.0209	-2.0213	-2.0214

Task2

-0.8639						
-1.2605	-0.9252					
-1.195	-1.1752	-1.1983				
-1.382	-1.4679	-1.4967	-1.5036			
-1.635	-1.7678	-1.7976	-1.8045	-1.8062		
-1.9138	-2.0687	-2.0986	-2.1056	-2.1073	-2.1077	
-2.2041	-2.3697	-2.3996	-2.4066	-2.4083	-2.4087	-2.4088

Task3

-1.4505						
-2.1012	-1.6502					
-3.3468	-2.6903	-2.4355				
-4.4818	-3.7123	-4.1495	-4.8611			
-6.4502	-4.978	-4.6171	-4.5907	-4.5881		
-8.7006	-6.9372	-6.2381	-6.013	-5.9686	-5.9581	
-9.9032	-9.3031	-8.1443	-8.724	-8.244	-8.1702	-8.1531

4、讨论

从误差列表可以看出，在函数性态不好时，龙贝格方法的收敛速度会较有限。不过，对第三个较好的函数就有很不错的收敛性，且其动态规划思想也有效节省了时间复杂度，

附录-代码

```
f1 = @(x) sin(x) / ((x ~= 0) * x + (x == 0)) + (x == 0);
res1 = 0.9460830703671841;
f2 = @(x) (cos(x) - exp(x)) / ((x ~= 0) * sin(x) + (x == 0)) - (x == 0);
res2 = -2.246591720728612;
f3 = @(x) exp(-1/x) / ((x ~= 0) * x + (x == 0));
res3 = 0.2193839343983438;
N = 6;
fprintf("Task1:\n");
R = c7_R(f1, 0, 1, N);
for i = 1:N+1
    for j = 1:i
        fprintf("%f\t", R(i,j));
    end
    fprintf("\n");
end
fprintf("\nTask1 Error:\n");
for i = 1:N+1
    for j = 1:i
        fprintf("%e\t", abs(res1 - R(i,j)));
    end
    fprintf("\n");
end
fprintf("\nTask2:\n");
R = c7_R(f2, -1, 1, N);
for i = 1:N+1
    for j = 1:i
        fprintf("%f\t", R(i,j));
    end
    fprintf("\n");
end
fprintf("\nTask2 Error:\n");
for i = 1:N+1
    for j = 1:i
        fprintf("%e\t", abs(res2 - R(i,j)));
    end
    fprintf("\n");
end
fprintf("\nTask3:\n");
R = c7_R(f3, 0, 1, N);
for i = 1:N+1
    for j = 1:i
        fprintf("%f\t", R(i,j));
```

```
    end
    fprintf("\n");
end
fprintf("\nTask3 Error:\n");
for i = 1:N+1
    for j = 1:i
        fprintf("%e\t", abs(res3 - R(i,j)));
    end
    fprintf("\n");
end
```

```
function R = c7_R(f, a, b, M)
    R = zeros(M + 1);
    h = (b - a) / 2;
    R(1, 1) = h * (f(b) - f(a));
    for i = 2:M+1
        sum = 0;
        for j = a+h : 2*h : b-h
            sum = sum + f(j);
        end
        R(i, 1) = R(i-1, 1) / 2 + h * sum;
        h = h / 2;
    end
    for i = 2:M+1
        t = 1 / (4 ^ (i - 1) - 1);
        for j = i:M+1
            R(j, i) = R(j, i-1) * (t + 1) - R(j-1, i-1) * t;
        end
    end
end
```

Homework 8 实验报告

PB20000296 郑滕飞

1、功能实现

实验要求：利用四阶龙格-库塔方法估算常微分方程数值解。
代码结构：
c8.m 主程序脚本[调用 c8_heun]
c8_heun.m 龙格-库塔方法与误差估算的实现

2、算法简述

四阶龙格-库塔方法本质是在问题$x'(t)=f(t,x)$中通过 x(t)与 f 的信息估算 x(t+h)，并不断循环得到整个区间的估算，单步估算公式为

$$F_1 = hf\big(t, x(t)\big)$$

$$F_2 = hf\left(t+\frac{h}{2}, x(t)+\frac{F_1}{2}\right)$$

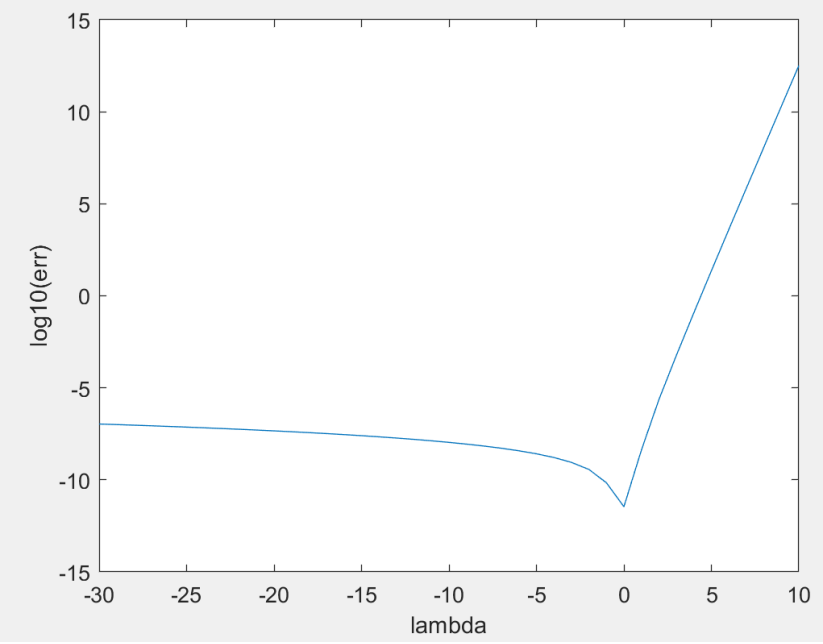
$$F_3 = hf\left(t+\frac{h}{2}, x(t)+\frac{F_2}{2}\right)$$

$$F_4 = hf(t+h, x(t)+F_3)$$

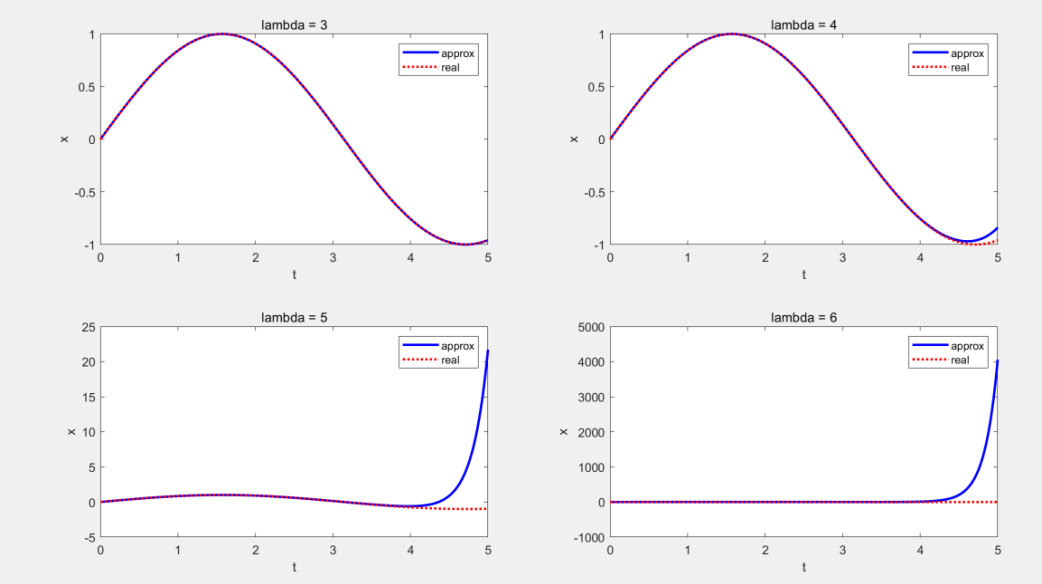
$$x(t+h) \approx x(t)+\frac{F_1+2F_2+2F_3+F_4}{6}$$

3、结果展示

对问题$x'(t)=\lambda x-cos\,t+\lambda \sin t, x(0)=0$，其精确解为$x=\sin t$，与数值解对比并计算最大误差，可得到误差取对数与$\lambda$的关系是：



可以发现，当λ = 0时误差最小，达到了 e-12 量级，而无论向左向右，误差的值都会增大。但是，λ < 0时增大非常平缓，而λ > 0则极为陡峭。下面四张图来自λ = 3,4,5,6时：



4、讨论

从结果上来看，λ > 0对误差的影响是非常大的。λ = 3时，误差在 e-4 量级，肉眼不可见，但更大时立刻就能看出明显的误差，如λ = 4的末尾处。这也体现了误差累积对远处值的影响可能非常之大。

附录-代码

```
x_real = @(t) sin(t);
max_err = zeros(1, 41);
figure();
for lambda = -30:10
    f = @(t, x) lambda * x + cos(t) - lambda * sin(t);
    flag = ismember(lambda, 3:6);
    if flag
        subplot(2, 2, lambda - 2);
    end
    max_err(lambda + 31) = c8_heun(f, 0.01, 0, 5, 0, x_real, flag);
    if flag
        title("lambda = " + num2str(lambda));
    end
end
figure();
plot(-30:10, log(max_err) / log(10));
xlabel('lambda');
```

```
ylabel('log10(err)');
```

```
function e = c8_heun(f, h, a, b, xa, x_real, if_plot)
    t = a:h:b;
    M = length(t);
    x = zeros(1, M);
    x(1) = xa;
    for i = 1:M-1
        s = t(i);
        y = x(i);
        F1 = h * f(s, y);
        F2 = h * f(s + h / 2, y + F1 / 2);
        F3 = h * f(s + h / 2, y + F2 / 2);
        F4 = h * f(s + h, y + F3);
        x(i + 1) = y + (F1 + 2 * F2 + 2 * F3 + F4) / 6;
    end
    xr = x_real(t);
    e = max(abs(x - xr));
    if (if_plot)
        plot(t, x, 'b', 'linewidth', 2);
        hold on;
        plot(t, xr, 'r:', 'linewidth', 2);
        legend('approx', 'real');
        xlabel('t');
        ylabel('x');
        hold off;
    end
end
```

Homework 9 实验报告

PB20000296 郑滕飞

1、功能实现

实验要求：利用 RKF 与 Adams-Bashforth 公式估算常微分方程数值解。
代码结构：
c9.m 主程序脚本[调用 c9_RKF45、c9_AB]
c9_RKF45.m 自适应 RKF 方法的实现
c9_AB.m Adams-Bashforth 公式的实现

2、算法简述

a. RKF45 估算数值解

RKF 方法通过一列 F1 至 F6 给出了四阶精度与五阶精度两个公式，分别为(F1 至 F6 的计算过于复杂，这里不列出)：

$$y(x+h)=y(x)+\frac{25}{216}F_1+\frac{1408}{2565}F_3+\frac{2197}{4104}F_4-\frac{1}{5}F_5$$

$$y_c(x+h)=y(x)+\frac{16}{135}F_1+\frac{6656}{12825}F_3+\frac{28561}{56430}F_4-\frac{9}{50}F_5+\frac{2}{55}F_6$$

将上方的四阶公式作为标准值，下方对比误差，就得到了 RKF45 算法，为了使步长 h 能自适应更新，取更新公式为

$$h_{new}=0.9h\left(\frac{\delta}{|y-y_c|}\right)^{0.2}$$

此处 0.2 为$\frac{1}{4+1}$，是由于第一个公式是 4 阶。

对于实际问题$y'=\exp(xy)+\cos(y-x),y(1)=3$，经测试可取$\delta=10^{-10}$，迭代 12000 步左右达到溢出。

b. Adams-Bashforth 公式估算数值解

Adams-Bashforth 公式是如下的显式多步方法：

$$y_{n+1}=y_n+\frac{h}{720}(1901f_n-2774f_{n-1}+2616f_{n-2}-1274f_{n-3}+251f_{n-4})$$

为了得到初值，需要先用龙格-库塔方法计算前五项，从第六项开始可以以此计算。

3、结果展示

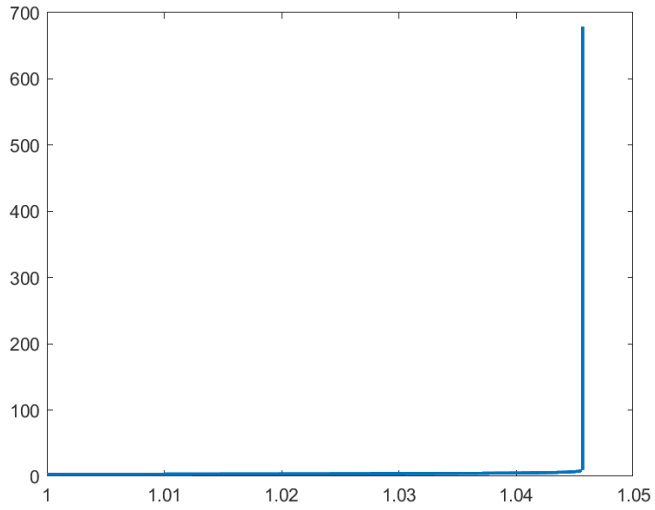
a.

最终得出的发散位置为：1.045644。

直接运行程序 c9 中间会出现提示输入环节，示例输入输出(标黄为输入)：

```
input x in range above: 1.03
y = 3.974092
```

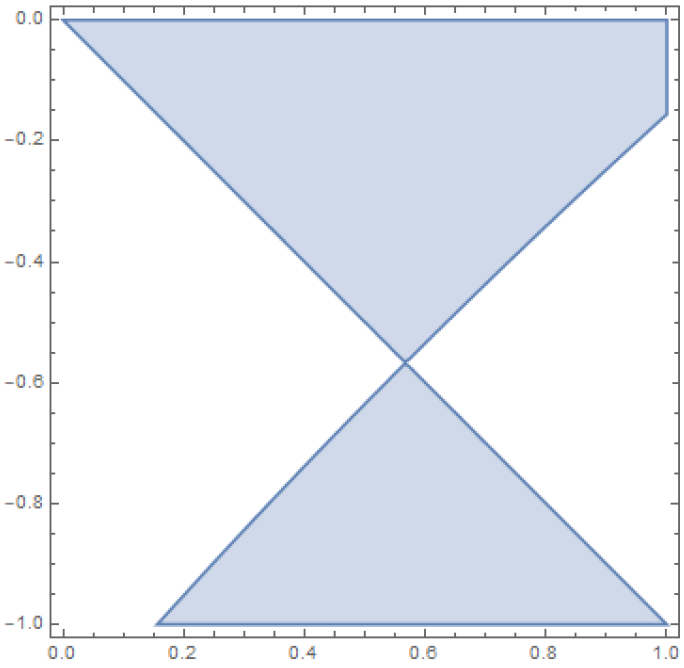
作图可以发现，此函数在之前变化平缓，1.04 附近突然呈现极快的增长：



b.

先将精确解作图：

```
In[4]:= RegionPlot[x^2 - t^2 + 2 E^x - 2 E^(-t) > 0, {t, 0, 1}, {x, -1, 0}]
        绘制区域                                  自…   自然常数
```



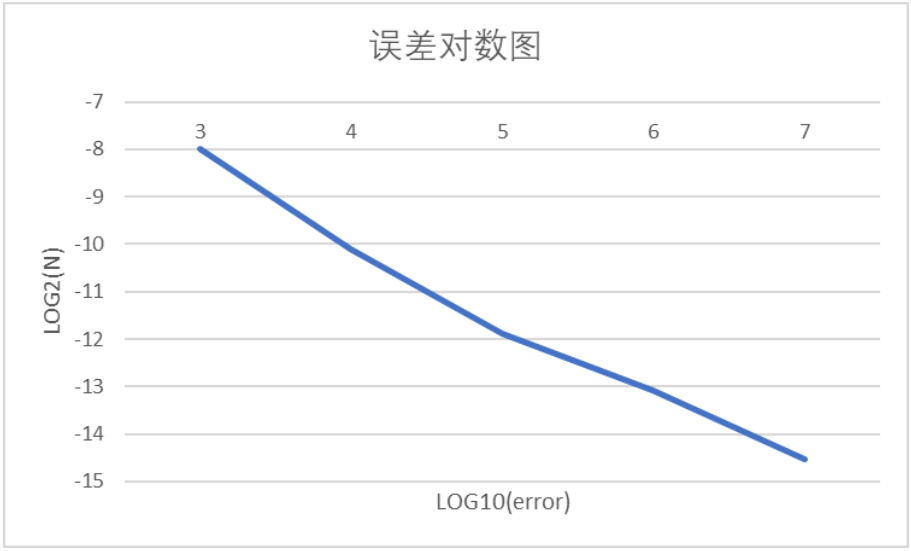
可以发现，-1 处有两个可能解，-1 与-0.15 左右的解。其中，-1 对应的是$x = -t$的一支，而在$t = exp(-t)$处会产生一个奇点，导致两个分支的出现。

由于需要验证数值精度，若按题目要求从 0 开始，会始终收敛到$x = -t$一支，并且误差为 0，因此，我改动了要求，从 t=0.6 开始，由初值-0.534416297250012 确定在上半支后对最终结果-0.155782424588460 进行估算，并对比精度，结果如下：

N	err	ord
8	1.002742e-08	
16	7.892278e-11	6.989294

```
32  1.263656e-12   5.964766
64  8.107404e-14   3.962220
128 2.942091e-15   4.784326
```

将误差取对数作图得到：



可看出其一般拥有四阶以上精度。

4、讨论

这两个部分分别给出了两种使得数值估计出现原则性错误的情况：发散与分支。若微分方程某点处发散，不加自适应检测地取 h 会导致其后的情况无法估算，而若微分方程存在多解，在完全无其他信息时给定的初值也无法决定去向哪个分支。实际应用中，可能需要更精确的策略来确定真实情况。

附录-代码

```
fprintf("Task1:\n");
f = @(x, y) exp(x * y) + cos(y - x);
res = c9_RKF45(f, 3, 1, 0.01, 1e-10);
plot(res(1,:), res(2,:), "linewidth", 2);
fprintf("result range: [1, %f]\n", res(1, end));
x = input("input x in range above: ");
if (x >= 1 && x < res(1, end))
    i = 1;
    while res(1, i) < x
        i = i + 1;
    end
    if i == 1
        fprintf("y = 3\n");
    else
        x_b = res(1, i-1);
```

```
        x_a = res(1, i);
        y_b = res(2, i-1);
        y_a = res(2, i);
        y = ((y_b-y_a) * x + x_b*y_a - y_b*x_a) / (x_b-x_a);
        fprintf("y = %f\n", y);
    end
else
    fprintf("ERROR: number not in range\n");
end

fprintf("\nTask2:\n");
fprintf("N \terr          \tord\n");
f = @(t,x) (t - exp(-t)) / (x + exp(x));
x_1 = -0.155782424588460;
x0 = -0.534416297250012;
t0 = 0.6;
h = 0.4 / 8;
r = c9_AB(f, x0, t0, h, 1);
err_old = abs(r(2, end) - x_1);
fprintf("8 \t%e\n", err_old);
for i = 1:4
    h = h / 2;
    r = c9_AB(f, x0, t0, h, 1);
    err = abs(r(2, end) - x_1);
    ord = log (err_old / err) / log(2);
    fprintf("%d\t%e\t%f\n", int32(0.4/h), err, ord);
    err_old = err;
end
```

```
function res = c9_RKF45(f, y0, x0, h0, delta, max_iter)
    if (~exist('max_iter', 'var'))
        max_iter = -1;
    end
    x = [];
    y = [];
    x_now = x0;
    y_now = y0;
    h = h0;
    count = 0;
    while isfinite(y_now)
        x = [x x_now];
        y = [y y_now];
        F1 = h * f(x_now, y_now);
        F2 = h * f(x_now + h/4, y_now + F1/4);
        F3 = h * f(x_now + h*3/8, y_now + F1*3/32 + F2*9/32);
```

```
        F4 = h * f(x_now + h*12/13, y_now + F1*1932/2197 - F2*7200/2197 ...
            + F3*7296/2197);
        F5 = h * f(x_now + h, y_now + F1*439/216 - F2*8 + F3*3680/513 ...
            - F4*845/4104);
        F6 = h * f(x_now + h/2, y_now - F1*8/27 + F2*2 - F3*3544/2565 ...
            + F4*1859/4104 - F5*11/40);
        x_now = x_now + h;
        y_cmp = y_now + F1*16/135 + F3*6656/12825 + F4*28561/56430 ...
            - F5*9/50 + F6*2/55;
        y_now = y_now + F1*25/216 + F3*1408/2565 + F4*2197/4104 - F5/5;
        err = abs(y_now - y_cmp);
        h = 0.9 * h * (delta / err) ^ 0.2;
        count = count + 1;
        if count == max_iter
            break;
        end
    end
    res(1,:) = x;
    res(2,:) = y;
end
```

```
function res = c9_AB(f, y0, x0, h, x_max)
    x = x0:h:x_max;
    y = zeros(1, length(x));
    y(1) = y0;
    for i = 1:4
        F1 = h * f(x(i), y(i));
        F2 = h * f(x(i) + h/4, y(i) + F1/4);
        F3 = h * f(x(i) + h*3/8, y(i) + F1*3/32 + F2*9/32);
        F4 = h * f(x(i) + h*12/13, y(i) + F1*1932/2197 - F2*7200/2197 ...
            + F3*7296/2197);
        F5 = h * f(x(i) + h, y(i) + F1*439/216 - F2*8 + F3*3680/513 ...
            - F4*845/4104);
        y(i + 1) = y(i) + F1*25/216 + F3*1408/2565 + F4*2197/4104 - F5/5;
    end
    for i = 5:length(x) - 1
        y(i + 1) = y(i) + h / 720 * ( f(x(i), y(i)) * 1901 ...
            - f(x(i-1), y(i-1)) * 2774 + f(x(i-2), y(i-2)) * 2616 ...
            - f(x(i-3), y(i-3)) * 1274 + f(x(i-4), y(i-4)) * 251 );
    end
    res(1,:) = x;
    res(2,:) = y;
end
```

Homework 10 实验报告

PB20000296 郑滕飞

1、功能实现

实验要求：利用 Mathematica 绘制线性多步法的绝对稳定性区域。
代码结构：
plot_stable.nb 主程序笔记本

2、算法简述

若线性多步法写为

$$\sum_{i=0}^{k} a_{k-i}\, x_{n-i} = \mathrm{h} \sum_{i=0}^{k} b_{k-i}\, f_{n-i}$$

记

$$p(z) = \sum_{i=0}^{k} a_i\, z^i, q(z) = \sum_{i=0}^{k} b_i z^i$$

则绝对稳定性区域是指满足使 z 的多项式

$$p(z) - \omega q(z) = 0$$

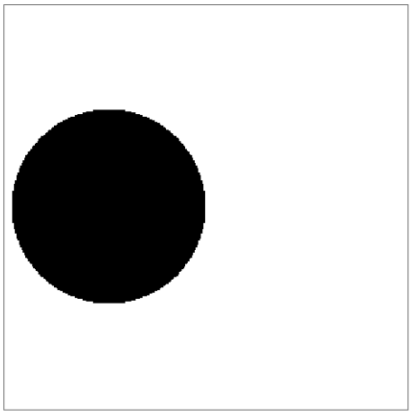
的所有根模长小于 1 的ω所构成的区域。

为了绘制区域，我实现了 PlotStable 函数，其输入的四个参数分别为：多项式 p、多项式 q、绘制精度 n（实际绘制的是$(n+1)\times(n+1)$像素的图片）与绘制范围 range（实际绘制范围是实部和虚部绝对值都小于等于 range 的情况）。实现方式为，在绘制区域内以实部、虚部分别$\frac{2\mathrm{range}}{n}$的步长取点并对每点分析是否满足条件得到布尔型矩阵，最后满足条件的点以黑色表示，不满足以白色表示，即得到绝对稳定性区域。

3、结果展示

对欧拉方法的示例结果如下：

```
p[z_] := z - 1; q[z_] := 1;
PlotStable[p, q, 300, 2]
```

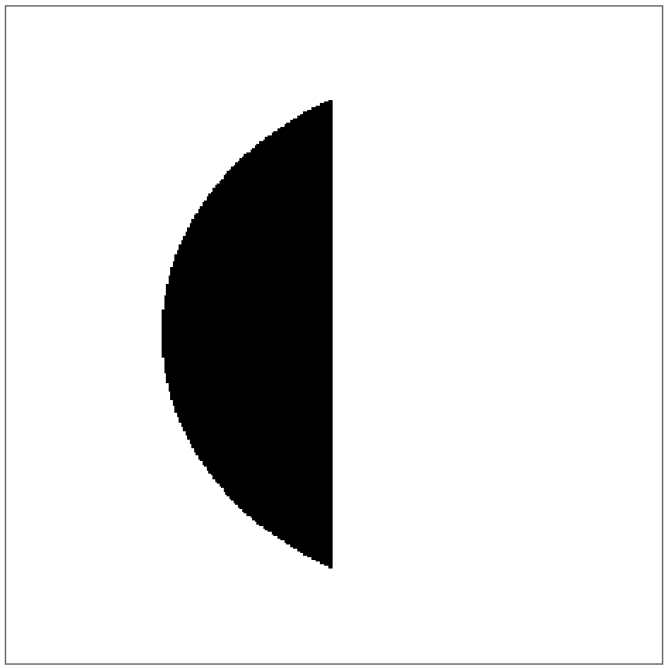


注意第四个参数表示实部虚部的范围都是±2，满足书上推导的以-1 为中心 1 为半径的圆。

对 Adams – Bashforth 公式、Adams – Moulton 公式分别效果如下：

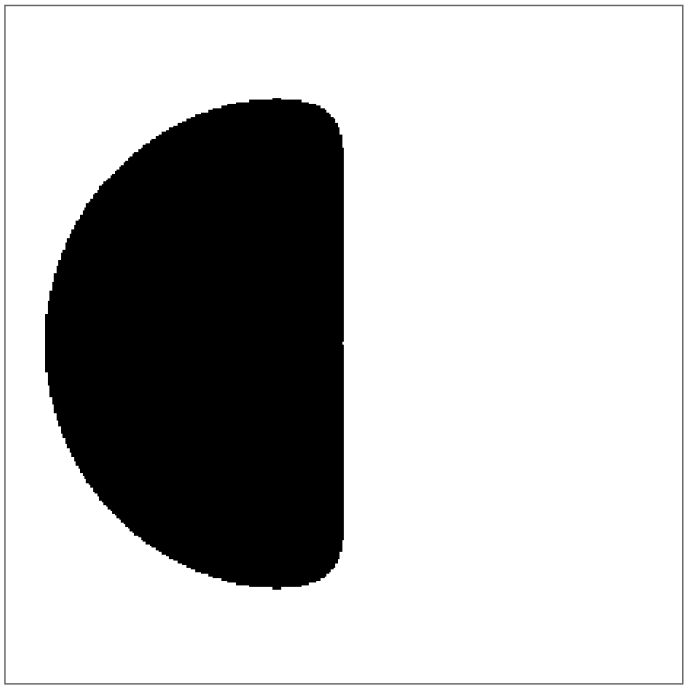
```
p[z_] := z^5 - z^4; q[z_] := (1901 z^4 - 2774 z^3 + 2616 z^2 - 1274 z + 251) / 720;

PlotStable[p, q, 300, 0.3]
```



```
p[z_] := z^4 - z^3; q[z_] := (251 z^4 + 646 z^3 - 264 z^2 + 106 z - 19) / 720;

PlotStable[p, q, 300, 2]
```



4、讨论

注意到第二张图的 range 只有 0.3，而第三张图有 2，这代表 Adams – Bashforth 公式的绝对稳定性区域事实上比 Adams – Moulton 公式小很多，这也符合隐式方法远比显式方法稳定性强的基本结果。

附录-代码

```
PlotStable[p_, q_, n_, range_] :=
  Module[{A, a, b, z, list}, A = Table[True, {i, n + 1}, {j, n + 1}];
   Do[a = -range + 2 range*(j - 1)/n; b = range - 2 range*(i - 1)/n;
    list = z /. NSolve[p[z] - (a + b I) q[z] == 0, z];
    A[[i, j]] = Max[Abs[list]] < 1, {i, n + 1}, {j, n + 1}];
   ArrayPlot[Boole[A]]];

p[z_] := z - 1; q[z_] := 1;
PlotStable[p, q, 300, 2]

p[z_] := z^5 - z^4;
q[z_] := (1901 z^4 - 2774 z^3 + 2616 z^2 - 1274 z + 251)/720;
PlotStable[p, q, 300, 0.3]

p[z_] := z^4 - z^3;
q[z_] := (251 z^4 + 646 z^3 - 264 z^2 + 106 z - 19)/720;
PlotStable[p, q, 300, 2]
```

Homework 11 实验报告

PB20000296 郑滕飞

1、功能实现

实验要求：利用 RKF 与 Adams-Bashforth 公式估算常微分方程数值解。
代码结构：
c11.m 主程序脚本[调用 c11_dif]
c11_dif.m 有限差分方法实现

2、算法简述

有限差分法中，通过将一阶导数、二阶导数近似为差分来构造方程组求解。具体来说，设

$$y(a)=\alpha, y(b)=\beta, y''=u(x)+v(x)y+w(x)y'$$

则令

$$h=\frac{b-a}{N+1}, x_i=a+ih, u_i=u(x_i), v_i=v(x_i), w_i=w(x_i)$$

可构造方程

$$\begin{pmatrix} d_1 & c_1 & 0 & \cdots & 0 \\ a_1 & d_2 & c_2 & \ddots & \vdots \\ 0 & a_2 & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & c_{n-1} \\ 0 & \cdots & 0 & a_{n-1} & d_n \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_{n-1} \\ y_n \end{pmatrix} = \begin{pmatrix} b_1-a_0\alpha \\ b_2 \\ \vdots \\ b_{n-1} \\ b_n-c_n\beta \end{pmatrix}$$

其中

$$a_i=-1-\frac{1}{2}hw_{i+1}, d_i=2+h^2v_i, c_i=-1+\frac{1}{2}hw_i, b_i=-h^2u_i$$

3、结果展示

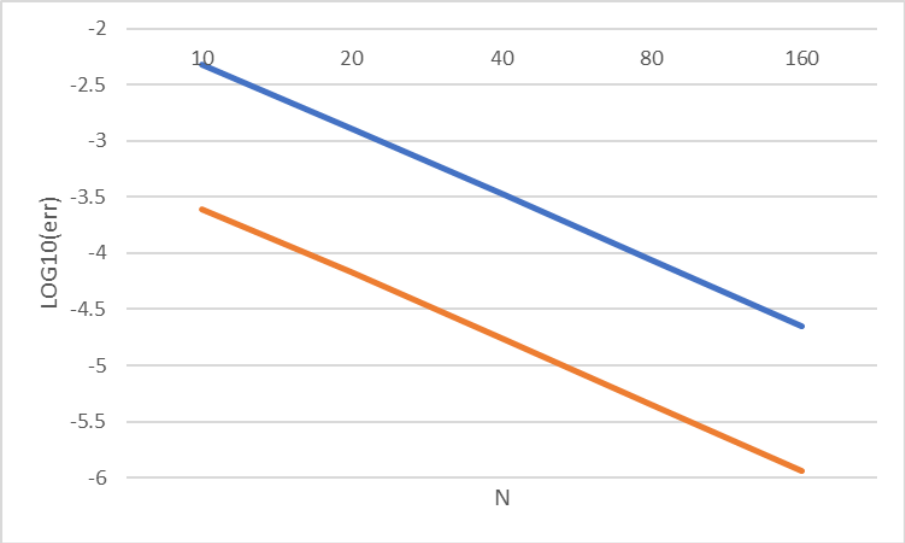
控制台直接输出如下：

```
Task1:
N   err           ord
10  4.728096e-03
20  1.297929e-03   1.865048
40  3.402235e-04   1.931656
80  8.717387e-05   1.964515
160 2.206366e-05   1.982224

Task2:
N   err           ord
10  2.440064e-04
20  6.712021e-05   1.862100
40  1.760674e-05   1.930619
80  4.511445e-06   1.964467
```

```
160 1.141885e-06    1.982172
```

将误差取对数作图得到：



从图中也能看出较稳定的二阶收敛性。

4、讨论

通过多点差分可以将连续的微分方程转化为离散的代数方程组，且对线性微分方程可以离散出线性结果，是常用的求解方法。

附录-代码

```
fprintf("Task1:\n");
u = @(x) 0 * x;
v = @(x) -1 + 0 * x;
w = @(x) 0 * x;
f_real = @(x) 7 * sin(x) + 3 * cos(x);
fprintf('N\terr         \tord\n');
N = 10;
y = c11_dif(0, 3, pi/2, 7, N, u, v, w);
x = 0:(pi/(2*N+2)):(pi/2);
y_real = f_real(x);
err_old = max(abs(y - y_real));
fprintf('10\t%e\n', err_old);
for i = 1:4
    N = N * 2;
    y = c11_dif(0, 3, pi/2, 7, N, u, v, w);
    x = 0:(pi/(2*N+2)):(pi/2);
    y_real = f_real(x);
    err = max(abs(y - y_real));
    ord = log (err_old / err) / log(2);
    fprintf('%d\t%e\t%f\n', N, err, ord);
```

```
    err_old = err;
end

fprintf("\nTask2:\n");
u = @(x) 2 * exp(x);
v = @(x) -1 + 0 * x;
w = @(x) 0 * x;
f_real = @(x) exp(x) + cos(x);
fprintf('N\terr        \tord\n');
N = 10;
y = c11_dif(0, 2, 1, exp(1)+cos(1), N, u, v, w);
x = 0:(1/(N+1)):1;
y_real = f_real(x);
err_old = max(abs(y - y_real));
fprintf('10\t%e\n', err_old);
for i = 1:4
    N = N * 2;
    y = c11_dif(0, 2, 1, exp(1)+cos(1), N, u, v, w);
    x = 0:(1/(N+1)):1;
    y_real = f_real(x);
    err = max(abs(y - y_real));
    ord = log (err_old / err) / log(2);
    fprintf('%d\t%e\t%f\n', N, err, ord);
    err_old = err;
end
```

```
function y = c11_dif(xa, ya, xb, yb, N, u, v, w)
    h = (xb - xa) / (N + 1);
    x = xa+h:h:xb-h;
    u = u(x);
    v = v(x);
    w = w(x);
    A = zeros(N);
    b = zeros(N, 1);
    for i = 1:N-1
        A(i, i) = 2 + v(i) * h^2;
        A(i, i+1) = h * w(i) / 2 - 1;
        A(i+1, i) = -h * w(i+1) / 2 - 1;
        b(i) =  -u(i) * h^2;
    end
    b(1) = b(1) + ya * (h * w(1) / 2 + 1);
    A(N, N) = 2 + v(N) * h^2;
    b(N) = -u(N) * h^2 - yb * (h * w(N) / 2 - 1);
    y = [ya, (A \ b)', yb];
end
```