

**工科硕士生数值分析上机报告**

|  |  |
| --- | --- |
| **编号：** | **C53** |
| **学号：** | **193206** |
| **姓名：** | **汪 涛** |
| **指导老师：** | **杜 睿** |

写在前面：本上机实验报告主要参考教材与《常用算法程序集（C语言）》（第五版），代码书写环境是Microsoft Visual Studio Code。

1. **对数的近似计算**

1. ，计算出，并回答作为的近似值有几位有效数字？

解：的高阶导为： ，当时有：

编写计算和估计误差的程序源码：

#include "stdio.h"

#include "stdlib.h"

#include "math.h"

// 定义计算S10\_1的函数

double S10\_1(double x){

double m = 0.0;

m = 1.0\*x-x\*x/2 + x\*x\*x/3-pow(x, 4)/4+ pow(x, 5)/5- pow(x, 6)/6 + pow(x, 7)/7 - pow(x, 8)/8 + pow(x, 9)/9 - pow(x, 10)/10;

return m;

}

// 定义误差估计函数直接计算

double y1(double x){

double z;

z = fabs(log(1.0+x)-

(1.0\*x- x\*x/2 + x\*x\*x/3 - pow(x, 4)/4 + pow(x, 5)/5 - pow(x, 6)/6 + pow(x, 7)/7 - pow(x, 8)/8 + pow(x, 9)/9 - pow(x, 10)/10));

return (z);

}

// 主函数对结果进行输出

void main(){

printf("THE VALUE S10\_1(1) IS: %13.10e\n",S10\_1(1.0));

double x1=0.0, step=0.0000001, temp1=0.0;

while(x1<=1.0){

if(temp1<=y1(x1))

temp1=y1(x1);

x1 = x1 + step;

}

printf("THE VALUES OF MAX DELT(1): %13.10e\n", temp1);

printf("\n");

}

编译输出结果：

THE VALUE S10\_1(1) IS: 6.4563492063e-001

THE VALUES OF MAX DELT(1): 4.7512259800e-002

已知，准确值= 0.69314718055995，若将作为近似值，那么：

即有1位有效数字。

1. ，计算出，并回答作为的近似值有几位有效数字？

解：的高阶导为： ，当时有：

编写计算和估计误差的程序源码：

#include "stdio.h"

#include "stdlib.h"

#include "math.h"

// 定义计算S10\_2的函数

double S10\_2(double x){

double m = 0.0;

m = -1.0\*x + x\*x/2 - x\*x\*x/3 + pow(x, 4)/4 - pow(x, 5)/5 + pow(x, 6)/6 - pow(x, 7)/7 + pow(x, 8)/8 - pow(x, 9)/9 + pow(x, 10)/10;

return m;

}

// 定义误差估计函数直接计算

double y2(double x){

double z;

z = fabs(log(1.0/(1.0+x)) -

(-1.0\*x + x\*x/2 - x\*x\*x/3 + pow(x, 4)/4 - pow(x, 5)/5 +

pow(x, 6)/6 - pow(x, 7)/7 + pow(x, 8)/8 - pow(x, 9)/9 + pow(x, 10)/10));

return (z);

}

// 主函数对结果进行输出

void main(){

printf("THE VALUE S10\_2(-0.5) IS: %13.10e\n",S10\_2(-0.5));

double x2=-0.5, step=0.0000001, temp2=0.0;

while(x2<=0.5){

if(temp2<=y2(x2))

temp2=y2(x2);

x2 = x2 + step;

}

printf("THE VALUES OF MAX DELT(-0.5): %13.10e\n", temp2);

printf("\n");

}

编译输出结果：

THE VALUE S10\_2(-0.5) IS: 6.9306485615e-001

THE VALUES OF MAX DELT(-0.5): 8.2324409152e-005

已知，准确值= 0.69314718055995，若将作为近似值，那么：

即有3位有效数字。

1. ，计算出，并回答作为的近似值有几位有效数字？

解：的高阶导为： ，当时有：

编写计算和估计误差的程序源码：

#include "stdio.h"

#include "stdlib.h"

#include "math.h"

// 定义计算S10\_3的函数

double S10\_3(double x){

double n = 0.0;

n = 2.0\*x + 2.0\*x\*x\*x/3 + 2.0\*pow(x, 5)/5 + 2.0\*pow(x, 7)/7 + 2.0\*pow(x, 9)/9;

return (n);

}

// 定义误差估计函数直接计算

double y3(double x){

double z;

z = fabs(log((1.0+x)/(1.0-x))-

(2.0\*x + 2.0\*x\*x\*x/3 + 2.0\*pow(x, 5)/5 + 2.0\*pow(x, 7)/7 + 2.0\*pow(x, 9)/9));

return (z);

}

// 主函数对结果进行输出

void main(){

printf("THE VALUE S10\_3(0.333) IS: %13.10e\n", S10\_3(1.0/3));

double x3=-1.0/3, temp3=0.0;

while(x3<=1.0/3){

if(temp3<=y3(x3))

temp3=y3(x3);

x3 = x3 + step;

}

printf("THE VALUES OF MAX DELT(0.333): %13.10e\n", temp3);

printf("\n");

}

编译输出结果：

THE VALUE S10\_3(0.333) IS: 6.9314604739e-001

THE VALUES OF MAX DELT(0.333): 1.1331691180e-006

已知，准确值= 0.69314718055995，若将作为近似值，那么：

即有5位有效数字。

1. 比较上述三种算法的计算精度，体会编程中要注意的“尽量避免大数吃小数”、“尽量避免相近数相减”的准则。

解：

以上三种算法的计算精度不断提高。具体地考查算法执行的情况，发现：

算法Ⅰ中S10\_1内部，m = 1.0\*x-x\*x/2 + x\*x\*x/3-pow(x, 4)/4+ pow(x, 5)/5- pow(x, 6)/6 + pow(x, 7)/7 - pow(x, 8)/8 + pow(x, 9)/9 - pow(x, 10)/10; 泰勒展开式的项有正有负，即出现“相近的数相减”，导致计算精度较其他算法极差；

算法Ⅱ中S10\_2内部，m = (-1.0\*x + x\*x/2)+( - x\*x\*x/3 + pow(x, 4)/4 )- pow(x, 5)/5 + pow(x, 6)/6 - pow(x, 7)/7 + pow(x, 8)/8 - pow(x, 9)/9 + pow(x, 10)/10;相对算法Ⅰ有所优化，将相减改为相加，但仍存在“大数吃小数”现象；

算法Ⅲ中S10\_3内部，n = 2.0\*x + 2.0\*x\*x\*x/3 + 2.0\*pow(x, 5)/5 + 2.0\*pow(x, 7)/7 + 2.0\*pow(x, 9)/9;偶数项均为0，仅为奇数项相加，加法次数更少，相对算法Ⅱ又进行了优化，因此计算精度更高。

1. **方程求根的Newton法**
2. 分析方程存在几个实根，给出每个根所在的区间。

解：由，，用区间二分法查找实根，根据一次函数和指数函数的连续递增特性，初拟区间[0,100]，编写程序源码：

#include "math.h"

#include "stdio.h"

// 二分法搜索根函数

int dhrt(a, b, h, eps, x, m, f)

int m;

double a, b, h, eps, x[], (\*f)();

{

int n, js;

double z, y, z1, y1, z0, y0;

n = 0;

z = a;

y = (\*f)(z);

while((z<=b+h/2.0)&&(n!=m)){

if(fabs(y)<eps){

n = n + 1;

x[n-1] = z;

z = z + h/2.0;

y = (\*f)(z);

}else{

z1 = z + h;

y1 = (\*f)(z1);

if(fabs(y1)<eps){

n = n + 1;

x[n-1] = z1;

z = z1 + h/2.0;

y = (\*f)(z);

}else if(y\*y1>0.0){

y = y1;

z = z1;

}else{

js = 0;

while(js==0){

if(fabs(z1-z)<eps){

n = n + 1;

x[n-1] = (z1 + z)/2.0;

z = z1 + h/2.0;

y = (\*f)(z);

js = 1;

}else{

z0 = (z1+z)/2.0;

y0 = (\*f)(z0);

if(fabs(y0)<eps){

x[n] = z0;

n = n + 1;

js = 1;

z = z0 + h/2.0;

y = (\*f)(z);

}else if((y\*y0)<0.0){

z1 = z0;

y1 = y0;

}else{

z = z0;

y = y0;

}

}

}

}

}

}

return(n);

}

// 主函数输出结果

void main(){

int i, n;

static int m = 5;

static double x[5];

double g(double);

n = dhrt(0.0, 100.0, 0.02, 0.00000005, x, m, g);

printf("M=%d\n", n);

for(i=0; i<=n-1; i++){

printf("x(%d)=%13.7e\n", i, x[i]);

}

printf("\n");

}

double g(x)

double x;

{

double z;

z = 6.0\*x-exp(x);

return(z);

}

编译输出结果：

M=2

x(0)=2.0448145e-001

x(1)=2.8331479e+000

即得存在两个根，分别在区间[0, 1]与[2, 3]内部。

1. 研究用Newton法求方程的根，分析迭代初值在什么范围内选取时，迭代序列收敛到相应的根。用Newton法求出具有8个有效数字的近似根。

解：分析迭代值的取值范围，可以采用大范围收敛定理进行判定。

，若要保证前提条件，则以划分区间进行分析：

当，存在闭区间，，，，，保号，且，，当时，收敛条件失效；同理可证时若要保证Newton迭代格式收敛，则，。

只要迭代初值非十分接近或等于，迭代有限次取任意初始值，均能收敛到两个区间的根。Newton迭代法的程序源码：

#include "stdio.h"

#include "math.h"

// Newton迭代法函数

int newt(x, eps, js, f)

int js;

double \*x, eps;

void (\*f)();

{

int k, l;

double y[2],d,p,x0,x1;

l=js;

k=1;

x0=\*x;

(\*f)(x0,y);

d=eps+1.0;

while ((d>=eps)&&(l!=0)){

if (fabs(y[1])+1.0==1.0){

printf("err\n");

return(-1);

}

// y[0]表示目标函数，y[1]表示目标函数的一阶导函数

x1=x0-y[0]/y[1];

(\*f)(x1,y);

d=fabs(x1-x0); p=fabs(y[0]);

if (p>d) d=p;

x0=x1; l=l-1;

}

\*x=x1;

k=js-l;

return(k);

}

// 定义好目标函数

void newtf(x, y)

double x, y[2];

{

y[0] = 6.0\*x-exp(x);

y[1] = 6.0-exp(x);

return;

}

// 主函数输出结果

void main(){

int js, k;

double x1, x2, eps;

void newtf(double, double []);

eps = 0.00000005; // 控制精度要求

js = 1000; // 最大迭代次数

x1 = log(5); // 迭代初值x1

k = newt(&x1, eps, js, newtf); // 输出迭代的次数

if(k>=0){

printf("k=%d x1=%13.7e\n", k, x1);

}

x2 = log(10); // 迭代初值x2

k = newt(&x2, eps, js, newtf);

if(k>=0){

printf("k=%d x2=%13.7e\n", k, x2);

}

printf("\n");}

编译输出结果：

k=6 x1=2.0448145e-001

k=6 x2=2.8331479e+000

即得具有8位有效数字的近似根为：和

1. 分析方程存在几个实根，给出每个根所在的区间。

解：，按照(Ⅰ)中的思路继续使用二分法查找根所在的区间。由的根知在与左右凹凸性发生变化，故将搜索区间定为，进行根的个数与大致值搜索程序源码：

#include "math.h"

#include "stdio.h"

#include "dhrt.c" // 封装二分查找的函数

void main(){

int i, n;

static int m = 5;

static double x[5];

double f(double);

n = dhrt(-10.0, 10.0, 0.2, 0.00000005, x, m, f);

printf("M=%d\n", n);

for(i=0; i<=n-1; i++){

printf("x(%d)=%13.7e\n", i, x[i]);

}

printf("\n");

}

double f(x)

double x;

{

double z;

z = 3.0\*x\*x-exp(x);

return(z);

}

编译输出结果：

M=3

x(0)=-4.5896227e-001

x(1)=9.1000757e-001

x(2)=3.7330790e+000

即得有三个根，它们所在的区间分别为：，和。

1. 研究用Newton法求方程的根，分析迭代初值在什么范围内选取时，迭代序列收敛到相应的根。用Newton法求出具有8个有效数字的近似根。

解：与(Ⅱ)中问题相似，分析迭代初值的取值范围，由的两个根和，显然与不能作迭代格式的分母。用大范围收敛定理进行判定，需要将区间分为三部分：，和。

1. 当时，可以在闭区间上，，，，，保号，且，，当时，即的第一个实数根时，收敛条件失效；
2. 当时，可以在闭区间上，，，，，保号，且，，当时，即的第二个实数根时，收敛条件失效；
3. 当时函数形状复杂变化较多，不易通过大范围区间收敛证明，可以通过区间内循环迭代法证明除了两端点时发散，其他初值均收敛。

具体使用Newton迭代法求解的根程序源码：

#include "math.h"

#include "stdio.h"

#include "newt.c" // 封装Newton迭代法的函数

void main(){

int js, k;

double x3, x4, x5, eps;

void newtf(double, double []);

eps = 0.00000005;

js = 1000;

x3 = -0.835;

k = newt(&x3, eps, js, newtf); // 记录收敛时的迭代次数

if(k>=0){

printf("k=%d x3=%13.7e\n", k, x3);

}

x4 = 0.953;

k = newt(&x4, eps, js, newtf);

if(k>=0){

printf("k=%d x4=%13.7e\n", k, x4);

}

x5 = 3.657;

k = newt(&x5, eps, js, newtf);

if(k>=0){

printf("k=%d x5=%13.7e\n", k, x5);

}

}

void newtf(x, y)

double x, y[2];

{

y[0] = 3.0\*x\*x-exp(x);

y[1] = 6.0\*x-exp(x);

return;

}

编译输出结果：

k=5 x3=-4.5896227e-001

k=4 x4=9.1000757e-001

k=4 x5=3.7330790e+000

即得具有8个有效数字的近似根分别为：，和。

1. **Hilbert矩阵的条件数**
2. 编程计算的无穷范数的程序。

解：已知Hilbert矩阵的元素为 程序源代码：

#include "stdio.h"

#include "stdlib.h"

#include "math.h"

// 生成Hilbert矩阵的函数Hilb

double Hilb(double \*h, int n)

{

for(int i=0;i<n; i ++)

for(int j=0;j<n; j ++)

{

h[j+ n\*i]=1/((i+j+1)\*1.0);

}

return \*h;

}

// 计算Hilbert矩阵无穷范数的函数InfinNorm\_Hilb

double InfinNorm\_Hilb(double \*h, int n)

{

double norm, nor=0.0;

for(int i=0; i<n; i++){

norm = 0.0;

for(int j=0; j<n; j ++){

norm = norm + fabs(h[i\*n +j]);

}

if(nor<=norm)

nor = norm;

}

return nor;

}

// 主函数执行输出

void main(){

double hilb[100000];

Hilb(hilb, 4); // 以4阶Hilbert矩阵的无穷范数为例，任意正整数阶可选

printf("Infinite Norm IS:%13.2e\n", InfinNorm\_Hilb(hilb, 4));

}

编译输出结果：Infinite Norm IS: 2.08e+000

1. 编写计算的无穷范数条件数的程序（可调用求逆函数，在Mathematica中为Inverse[H]，在Matlab中为inv(H)，其他语言中自行查找）

解：Hilbert矩阵是一个n阶对称正定矩阵，则程序源代码为：

#include "stdio.h"

#include "stdlib.h"

#include "math.h"

/\*\*

\* 实矩阵相乘函数trmul

\* double a[m][n] 存放矩阵A的元素

\* double b[n][k] 存放矩阵B的元素

\* int m 矩阵A与乘积矩阵C的行数

\* int n 矩阵A的列数，矩阵B的行数

\* int k 矩阵B与乘积矩阵C的列数

\* double c[m][k] 返回乘积矩阵C=AB的元素

\* \*/

void trmul(a, b, m, n, k, c)

int m, n, k;

double a[], b[], c[];

{

int i, j, l, u;

for(i=0; i<=m-1; i++)

for(j=0; j<=k-1; j ++){

u = i\*k+ j;

c[u] = 0.0;

for(l=0; l<=n-1; l++)

c[u] = c[u] + a[i\*n+ l]\*b[l\*k +j];

}

return;

}

/\*\*

\* 求n阶对称正定矩阵A的逆矩阵 (也是一个对称正定矩阵)

\* double a[n][n]

\* int n 矩阵阶数

\* int ssgj() 函数返回标志值，若返回标志值小于0，则表示程序工作失败；

\* 若返回标志值大于0，则表示正常返回

\* \*/

int ssgj(double a[], int n){

int i, j, k, m;

double w, g,\*b;

b=malloc(n\*sizeof(double));

for (k=0; k<=n-1; k++)

{ w=a[0];

if (fabs(w)+1.0==1.0)

{ free(b); printf("fail\n"); return(-2);}

m=n-k-1;

for (i=1; i<=n-1; i++)

{ g=a[i\*n]; b[i]=g/w;

if (i<=m) b[i]=-b[i];

for (j=1; j<=i; j ++)

a[(i-1)\*n+j-1]=a[i\*n +j]+g\*b[j];

}

a[n\*n-1]=1.0/w;

for (i=1; i<=n-1; i++)

a[(n-1)\*n+i-1]=b[i];

}

for (i=0; i<=n-2; i++)

for (j=i+1; j<=n-1; j ++)

a[i\*n +j]=a[j\*n +i];

free(b);

return(2);

}

// 求Hilbert矩阵无穷范数条件数的函数InfinCond\_Hilb

double InfinCond\_Hilb(double \*h, int n){

double cond, h\_[100000];

for(int i=0; i<n; i++)

for(int j=0; j<n; j ++)

h\_[i\*n +j] =h[i\*n +j]; // 以h\_矩阵保留原始的h矩阵

ssgj(h, n); // 对h矩阵求逆并覆盖到原h矩阵上

cond = InfinNorm\_Hilb(h\_, n)\*InfinNorm\_Hilb(h, n); // 计算无穷范数条件数

return cond;

}

// 主函数执行输出

void main(){

double hilb[10000];

Hilb(hilb, 5); // 以5阶Hilbert矩阵的无穷范数为例，任意正整数阶可选

printf("n= 5 Infinite Cond IS: %13.6e\n", InfinCond\_Hilb(hilb, 5));

}

编译输出结果：n= 5 Infinite Cond IS: 9.436560e+005

1. 对，计算的无穷范数；画出的关系图。

解：调用（Ⅱ）中编译好的函数InfinCond\_Hilb，循环计算不同维数的Hilbert矩阵无穷范数条件数，并使用Origin Pro 9.0绘制折线图。程序源代码：

void main(){

for(int k=10; k<101; k+=10){

double hilb[10000];

Hilb(hilb, k);

printf("n=%d Infinite Cond IS: %13.6e\n", k, log(InfinCond\_Hilb(hilb, k)));

}

}

编译输出结果：

n=10 Infinite Cond IS: 3.119641e+001

n=20 Infinite Cond IS: 4.161372e+001

n=30 Infinite Cond IS: 4.171846e+001

n=40 Infinite Cond IS: 4.178703e+001

n=50 Infinite Cond IS: 4.183732e+001

n=60 Infinite Cond IS: 4.187669e+001

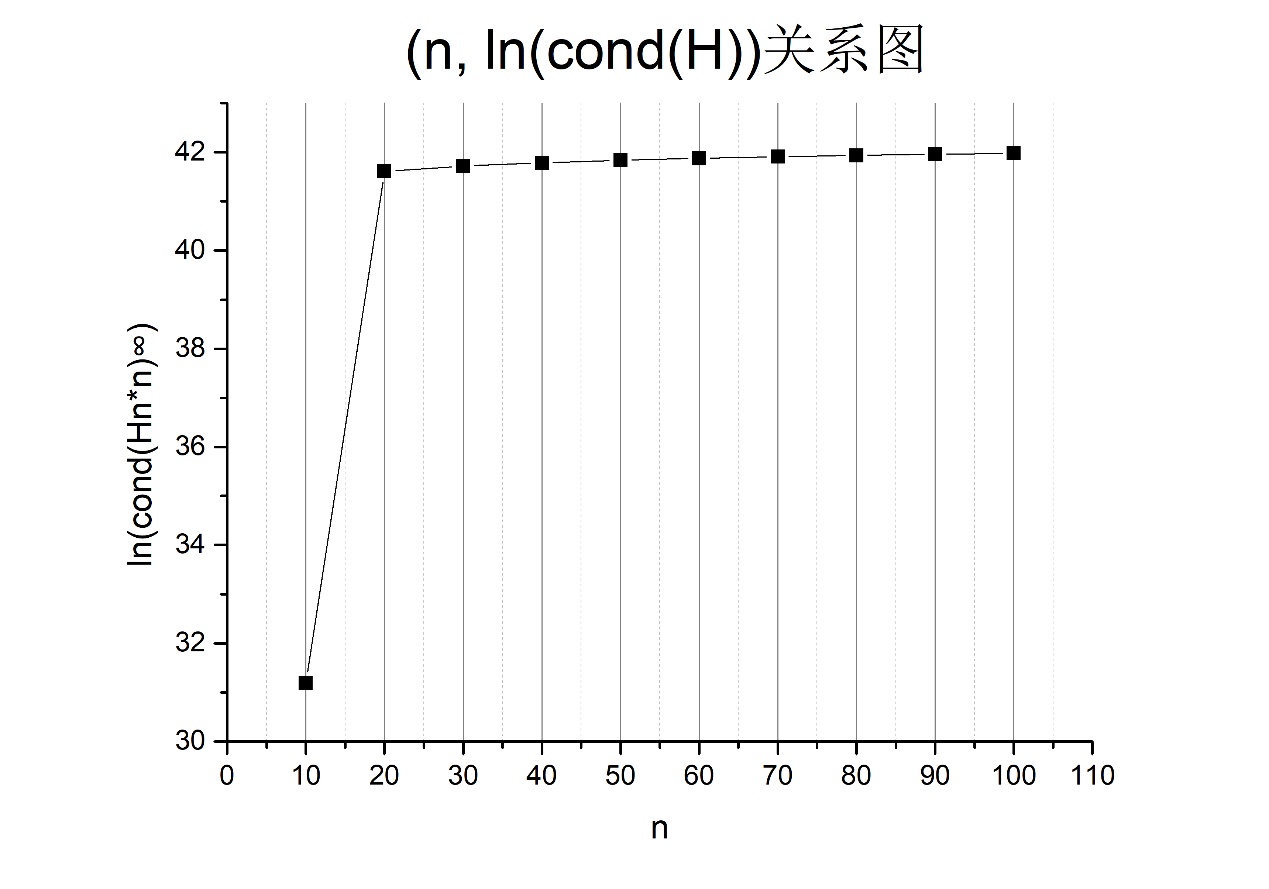
n=70 Infinite Cond IS: 4.190885e+001

n=80 Infinite Cond IS: 4.193593e+001

n=90 Infinite Cond IS: 4.195923e+001

n=100 Infinite Cond IS: 4.197964e+001

绘出的图线如下：



1. 令，，对，求解，并计算和以及它们的无穷范数；

解：将解向量放入对应大小的数组中，程序源代码：

#include "stdio.h"

#include "stdlib.h"

#include "math.h"

#include "trmul.c" // // 封装实矩阵相乘的函数

/\*\*

\* 求解主对角线占优系数矩阵的高斯-赛德尔迭代法AX=B

\* double a[n][n] 存放对称正定矩阵A

\* int n 方程组的阶数

\* double b[n] 存放方程组右端的常数向量

\* double eps 控制精度要求

\* double x[n] 返回方程组的解向量

\* int gsdl() 函数返回值。若返回的标志值小于0，

\* 则表示系数矩阵不具有主对角占优的绝对优势；

\* 若返回的标志值大于0，则比奥斯古正常返回

\* \*/

void gsdl(double a[], double b[], int n, double x[], double eps){

int i, j;

double p, t, s, q;

p=eps+1.0;

while (p>=eps)

{ p=0.0;

for (i=0; i<=n-1; i++)

{ t=x[i]; s=0.0;

for (j=0; j<=n-1; j ++)

if (j!=i) s=s+ a[i\*n +j]\*x[j];

x[i]=(b[i]-s)/a[i\*n +i];

q=fabs(x[i]-t)/(1.0+fabs(x[i]));

if (q>p) p=q;

}

}

return;

}

// 分别定义n=10, 50和100时的求解向量函数

void SolveVector\_10(double \*h, double \*s){

int p, q, k;

double x[10], S[10], H[10][10], eps = 0.000001;

for(p=0; p<10; p++)

for(q=0; q<10; q++)

H[p][q] = h[p\*10+q];

for(k=0; k<10; k++)

x[k] = 1.0;

trmul(H, x, 10, 10, 1, S);

gsdl(h, S, 10, s, eps);

for(int j=0; j<10; j ++)

printf("x(%d)=%13.10e\n", j, s[j]);

return;

}

void SolveVector\_50(double \*h, double \*s){

int p, q, k;

double x[50], S[50], H[50][50], eps = 0.000001;

for(p=0; p<50; p++)

for(q=0; q<50; q++)

H[p][q] = h[p\*50+q];

for(k=0; k<50; k++)

x[k] = 1.0;

trmul(H, x, 50, 50, 1, S);

gsdl(h, S, 50, s, eps);

for(int j=0; j<50; j ++)

printf("x(%d)=%13.10e\n", j, s[j]);

return;

}

void SolveVector\_100(double \*h, double \*s){

int p, q, k;

double x[100], S[100], H[100][100], eps=0.000001;

for(p=0; p<100; p++)

for(q=0; q<100; q++)

H[p][q] = h[p\*100+q];

for(k=0; k<100; k++)

x[k] = 1.0;

trmul(H, x, 100, 100, 1, S);

gsdl(h, S, 100, s, eps);

for(int j=0; j<100; j ++)

printf("x(%d)=%13.10e\n", j, s[j]);

return;

}

// 主函数执行输出

void main(){

double norm;

// 当n=10时，求解

double nor=0.0, Nor=0.0, X[10], Y[10], A[100], S[10], H[10][10];

Hilb(A, 10);

SolveVector\_10(A, S);

for(int c=0; c<10; c ++){

norm = 0.0;

X[c] = 1.0-S[c];

norm = fabs(X[c]);

if(nor <= norm)

nor = norm;

}

printf("the norm of x-x' is: %13.10e\n", nor);

double x[10], b[10], b\_[10];

for(int k=0; k<10; k++)

x[k] = 1.0;

Hilb(A, 10);

for(int p=0; p<10; p++)

for(int q=0; q<10; q++)

H[p][q] = A[p\*10+q];

trmul(H, x, 10, 10, 1, b);

trmul(H, S, 10, 10, 1, b\_);

for(int e=0; e<10; e++){

norm = 0.0;

Y[e] = b[e]-b\_[e];

norm = fabs(Y[e]);

if(Nor <= norm)

Nor = norm;

}

printf("the norm of b-b\_' is: %13.10e\n", Nor);

printf("\n");

// 当n=50时，求解

double nor2=0.0, Nor2=0.0, X2[50], Y2[50], A2[2500], S2[50], M[50][50];

Hilb(A2, 50);

SolveVector\_50(A2, S2);

for(int c=0; c<50; c ++){

norm = 0.0;

X2[c] = 1.0-S2[c];

norm = fabs(X2[c]);

if(nor2 <= norm)

nor2 = norm;

}

printf("the norm of x2-x2' is: %13.10e\n", nor2);

double x2[50], b2[50], b2\_[50];

for(int k=0; k<50; k++)

x2[k] = 1.0;

Hilb(A2, 50);

for(int p=0; p<50; p++)

for(int q=0; q<50; q++)

M[p][q] = A2[p\*50+q];

trmul(M, x2, 50, 50, 1, b2);

trmul(M, S2, 50, 50, 1, b2\_);

for(int e=0; e<50; e++){

norm = 0.0;

Y2[e] = b2[e]-b2\_[e];

norm = fabs(Y2[e]);

if(Nor2 <= norm)

Nor2 = norm;

}

printf("the norm of b2-b2\_' is: %13.10e\n", Nor2);

printf("\n");

// 当n=100时，求解

double nor3=0.0, Nor3=0.0, X3[100], Y3[100], A3[10000], S3[100], N[100][100];

Hilb(A3, 100);

SolveVector\_100(A3, S3);

for(int c=0; c<100; c ++){

norm = 0.0;

X3[c] = 1.0-S3[c];

norm = fabs(X3[c]);

if(nor3 <= norm)

nor3 = norm;

}

printf("the norm of x3-x3' is: %13.10e\n", nor3);

double x3[100], b3[100], b3\_[100];

for(int k=0; k<100; k++)

x3[k] = 1.0;

Hilb(A3, 100);

for(int p=0; p<100; p++)

for(int q=0; q<100; q++)

N[p][q] = A3[p\*100+q];

trmul(N, x3, 100, 100, 1, b3);

trmul(N, S3, 100, 100, 1, b3\_);

for(int e=0; e<100; e++){

norm = 0.0;

Y3[e] = b3[e]-b3\_[e];

norm = fabs(Y3[e]);

if(Nor3 <= norm)

Nor3 = norm;

}

printf("the norm of b3-b3\_' is: %13.10e\n", Nor3);

printf("\n");

}

编译输出结果：

x(0)=1.0004187379e+000

x(1)=9.9243525638e-001

x(2)=1.0285568150e+000

x(3)=9.7629422157e-001

x(4)=9.8142305108e-001

x(5)=1.0030900315e+000

x(6)=1.0174256368e+000

x(7)=1.0174590720e+000

x(8)=1.0036653858e+000

x(9)=9.7892521428e-001

the norm of x-x' is: 2.8556815009e-002

the norm of b-b\_' is: 2.7123996782e-007

x(0)=1.0000607976e+000

x(1)=1.0007697305e+000

x(2)=9.8777107223e-001

x(3)=1.0280396264e+000

x(4)=1.0008901925e+000

x(5)=9.8273707212e-001

x(6)=9.8205941902e-001

x(7)=9.9048404134e-001

x(8)=1.0005778867e+000

x(9)=1.0084735115e+000

x(10)=1.0129223111e+000

x(11)=1.0140872567e+000

x(12)=1.0127210067e+000

x(13)=1.0097093749e+000

x(14)=1.0058539636e+000

x(15)=1.0017903181e+000

x(16)=9.9797537393e-001

x(17)=9.9470623079e-001

x(18)=9.9215000204e-001

x(19)=9.9037474929e-001

x(20)=9.8937707847e-001

x(21)=9.8910492029e-001

x(22)=9.8947539105e-001

x(23)=9.9038827836e-001

x(24)=9.9173588215e-001

x(25)=9.9340995235e-001

x(26)=9.9530638045e-001

x(27)=9.9732820601e-001

x(28)=9.9938737629e-001

x(29)=1.0014056037e+000

x(30)=1.0033146033e+000

x(31)=1.0050558899e+000

x(32)=1.0065803015e+000

x(33)=1.0078473462e+000

x(34)=1.0088244606e+000

x(35)=1.0094862279e+000

x(36)=1.0098135955e+000

x(37)=1.0097931218e+000

x(38)=1.0094162591e+000

x(39)=1.0086786934e+000

x(40)=1.0075797389e+000

x(41)=1.0061217846e+000

x(42)=1.0043098057e+000

x(43)=1.0021509206e+000

x(44)=9.9965400682e-001

x(45)=9.9682935693e-001

x(46)=9.9368837833e-001

x(47)=9.9024334226e-001

x(48)=9.8650714947e-001

x(49)=9.8249314359e-001

the norm of x2-x2' is: 2.8039626429e-002

the norm of b2-b2\_' is: 2.5686291449e-007

x(0)=1.0000351684e+000

x(1)=9.9694447856e-001

x(2)=1.0233046964e+000

x(3)=9.5895912360e-001

x(4)=9.9304902959e-001

x(5)=1.0204648218e+000

x(6)=1.0252498845e+000

x(7)=1.0165620682e+000

x(8)=1.0039684267e+000

x(9)=9.9295875574e-001

x(10)=9.8569543519e-001

x(11)=9.8241321387e-001

x(12)=9.8245170855e-001

x(13)=9.8486186048e-001

x(14)=9.8871446981e-001

x(15)=9.9323196863e-001

x(16)=9.9782537557e-001

x(17)=1.0020853598e+000

x(18)=1.0057544705e+000

x(19)=1.0086945778e+000

x(20)=1.0108561375e+000

x(21)=1.0122519823e+000

x(22)=1.0129362631e+000

x(23)=1.0129882529e+000

x(24)=1.0125003254e+000

x(25)=1.0115693310e+000

x(26)=1.0102906408e+000

x(27)=1.0087542314e+000

x(28)=1.0070422885e+000

x(29)=1.0052279098e+000

x(30)=1.0033746065e+000

x(31)=1.0015363370e+000

x(32)=9.9975790259e-001

x(33)=9.9807556922e-001

x(34)=9.9651781770e-001

x(35)=9.9510615326e-001

x(36)=9.9385591849e-001

x(37)=9.9277709610e-001

x(38)=9.9187505662e-001

x(39)=9.9115126062e-001

x(40)=9.9060390058e-001

x(41)=9.9022846059e-001

x(42)=9.9001824572e-001

x(43)=9.8996481562e-001

x(44)=9.9005839062e-001

x(45)=9.9028818722e-001

x(46)=9.9064270655e-001

x(47)=9.9110999845e-001

x(48)=9.9167786043e-001

x(49)=9.9233401590e-001

x(50)=9.9306626972e-001

x(51)=9.9386262343e-001

x(52)=9.9471136743e-001

x(53)=9.9560117274e-001

x(54)=9.9652113968e-001

x(55)=9.9746084411e-001

x(56)=9.9841037894e-001

x(57)=9.9936037226e-001

x(58)=1.0003019924e+000

x(59)=1.0012269740e+000

x(60)=1.0021275970e+000

x(61)=1.0029966920e+000

x(62)=1.0038276325e+000

x(63)=1.0046143157e+000

x(64)=1.0053511512e+000

x(65)=1.0060330430e+000

x(66)=1.0066553676e+000

x(67)=1.0072139629e+000

x(68)=1.0077050902e+000

x(69)=1.0081254364e+000

x(70)=1.0084720686e+000

x(71)=1.0087424320e+000

x(72)=1.0089343175e+000

x(73)=1.0090458453e+000

x(74)=1.0090754454e+000

x(75)=1.0090218320e+000

x(76)=1.0088839965e+000

x(77)=1.0086611729e+000

x(78)=1.0083528375e+000

x(79)=1.0079586760e+000

x(80)=1.0074785799e+000

x(81)=1.0069126231e+000

x(82)=1.0062610529e+000

x(83)=1.0055242709e+000

x(84)=1.0047028261e+000

x(85)=1.0037973940e+000

x(86)=1.0028087754e+000

x(87)=1.0017378736e+000

x(88)=1.0005856976e+000

x(89)=9.9935333995e-001

x(90)=9.9804197437e-001

x(91)=9.9665284711e-001

x(92)=9.9518726437e-001

x(93)=9.9364659062e-001

x(94)=9.9203223845e-001

x(95)=9.9034566060e-001

x(96)=9.8858835013e-001

x(97)=9.8676182799e-001

x(81)=1.0069126231e+000

x(82)=1.0062610529e+000

x(83)=1.0055242709e+000

x(84)=1.0047028261e+000

x(85)=1.0037973940e+000

x(86)=1.0028087754e+000

x(87)=1.0017378736e+000

x(88)=1.0005856976e+000

x(89)=9.9935333995e-001

x(90)=9.9804197437e-001

x(91)=9.9665284711e-001

x(92)=9.9518726437e-001

x(93)=9.9364659062e-001

x(94)=9.9203223845e-001

x(95)=9.9034566060e-001

x(96)=9.8858835013e-001

x(97)=9.8676182799e-001

x(98)=9.8486764256e-001

x(99)=9.8290735749e-001

the norm of x3-x3' is: 4.1040876400e-002

the norm of b3-b3\_' is: 2.5711252860e-007

1. 通过以上的数值实验，你有哪些体会？

解：从上述实验中我体会到：

1. Hilbert矩阵是高度病态的对称正定矩阵，条件数极大；
2. 一般而言，愈大，病态愈严重；
3. 和之间并不是线性关系，当时，在30-45范围内波动；
4. 采用Gauss-Seidel迭代法求解Hilbert线性方程组，迭代总是收敛的，收敛速度与矩阵的阶数无直接关系，即对于病态的线形方程组，采用迭代求解可以得到较准确的解，无需考虑的病态性质。
5. **圆的拟合**
6. 用最小二乘法推导形如（3）的拟合公式中的系数的计算公式。

解：已知，其中是圆心坐标，是半径；

则，

记，，，

有，，

已有数据集点到圆心的距离：

令，则：

求解以上各式，先消去

得：

，即得：

得：

，即得：

其中，可以令：

上式可以记为：

进一步可以解得：

1. 利用Ⅰ中的计算公式，根据给定的数据，编写程序求得，和。

解：程序源代码：

#include "stdio.h"

#include "stdlib.h"

#include "math.h"

#define PI 3.14159265358979

void main(){

double X[100],Y[100];

double X1=0;

double Y1=0;

double X2=0;

double Y2=0;

double X3=0;

double Y3=0;

double X1Y1=0;

double X1Y2=0;

double X2Y1=0;

double eps=0.05;

for(int i=1; i<101; i++){

X[i] = 1.0 + pow(-1.0, i)\*eps + 3\*cos(i\*PI/50.0);

Y[i] = 10.0 + 3 \* sin((i/50.0+eps)\*PI);

// printf("The Value of X(%d)=%13.10e\n", i, X[i]);

// printf("The Value of Y(%d)=%13.10e\n", i, Y[i]);

X1 = X1 + X[i];

Y1 = Y1 + Y[i];

X2 = X2 + X[i]\*X[i];

Y2 = Y2 + Y[i]\*Y[i];

X3 = X3 + X[i]\*X[i]\*X[i];

Y3 = Y3 + Y[i]\*Y[i]\*Y[i];

X1Y1 = X1Y1 + X[i]\*Y[i];

X1Y2 = X1Y2 + X[i]\*Y[i]\*Y[i];

X2Y1 = X2Y1 + X[i]\*X[i]\*Y[i];

}

printf("The Value of X1=%13.10e\n", X1);

printf("The Value of Y1=%13.10e\n", Y1);

printf("The Value of X2=%13.10e\n", X2);

printf("The Value of Y2=%13.10e\n", Y2);

printf("The Value of X3=%13.10e\n", X3);

printf("The Value of Y3=%13.10e\n", Y3);

printf("The Value of X1Y1=%13.10e\n", X1Y1);

printf("The Value of X1Y2=%13.10e\n", X1Y2);

printf("The Value of X2Y1=%13.10e\n", X2Y1);

double A,D,E,G,H;

double a, b, c, x, y, r;

double N = 100;

A = N\*X2 - X1\*X1;

D = N\*X1Y1 - X1\*Y1;

E = N\*X3 + N\*X1Y2 - (X2+Y2)\*X1;

G = N\*Y2 - Y1\*Y1;

H = N\*X2Y1 + N\*Y3 - (X2+Y2)\*Y1;

printf("The Value of A = %13.10e\n", A);

printf("The Value of D = %13.10e\n", D);

printf("The Value of E = %13.10e\n", E);

printf("The Value of G = %13.10e\n", G);

printf("The Value of H = %13.10e\n", H);

b = (E\*G-D\*H)/(A\*G-D\*D);

c = (A\*H-D\*E)/(A\*G-D\*D);

a = (X2+Y2-b\*X1-c\*Y1)/N;

x = b/2.0;

y = c/2.0;

r = sqrt(a\*4.0 + b\*b + c\*c)/2.0;

printf("The Value of a = %13.10e\n", a);

printf("The Value of b = %13.10e\n", b);

printf("The Value of c = %13.10e\n", c);

printf("The Value of x\* = %13.10e\n", x);

printf("The Value of y\* = %13.10e\n", y);

printf("The Value of r = %13.10e\n", r);

}

编译输出结果：

The Value of X1=1.0000000000e+002

The Value of Y1=1.0000000000e+003

The Value of X2=5.5025000000e+002

The Value of Y2=1.0450000000e+004

The Value of X3=1.4507500000e+003

The Value of Y3=1.1350000000e+005

The Value of X1Y1=1.0703955093e+003

The Value of X1Y2=1.1857910185e+004

The Value of X2Y1=5.6432910185e+003

The Value of A = 4.5025000000e+004

The Value of D = 7.0395509268e+003

The Value of E = 2.3084101854e+005

The Value of G = 4.5000000000e+004

The Value of H = 9.1407910185e+005

The Value of a = -9.1997500000e+001

The Value of b = 2.0000000000e+000

The Value of c = 2.0000000000e+001

The Value of x\* = 1.0000000000e+000

The Value of y\* = 1.0000000000e+001

The Value of r = 3.0004166377e+000

即得：，和

1. **数值积分**
2. 取不同的步长分别采用复化梯形公式，复化Simpson公式、Romberg求积公式编程计算积分I的近似值，并与积分的准确值比较，是否存在一个，当从继续减少时，计算精度将不能再被改善？

解：给定一个积分精度eps=0.00000000001，改变不同步长时的求积公式的近似值，，，程序源代码：

1. 复化梯形公式

#include "math.h"

#include "stdio.h"

double f(x)

double x;

{

double y;

y=sqrt(x)\*log(x+0.00000000000001); // 排除log(0)不存在的情况

return(y);

}

void main(){

double a, b, eps;

a=0.0; b=1.0; eps=0.00000000001;

int n, k;

double fa,fb,h,t1,p,s,x,t,fm;

fa=(\*f)(a); fb=(\*f)(b);

n=2;h=(b-a)/n; // 对应m=1开始的步长

fm=(\*f)(a +h);

t1=h\*(fa+2\*fm+ fb)/2.0;

p=eps+1.0;

while (p>=eps)

{

s=0.0;

for (k=0;k<=n-1;k++){

x=a+(k+0.5)\*h;

s=s+(\*f)(x);

}

t=(t1+h\*s)/2.0;

p=fabs(t1-t);

t1=t; n=n +n; h=h/2.0;

}

printf("The Value of n = %d\n", n);

printf("The Value of h = %13.10e\n", 1.0/n);

printf("t=%13.10e\n", t);

printf("ABS of |(-4/9)-t|=%13.10e\n", fabs((-4/9.0)-t)); // 积分准确值误差比较

}

编译输出结果：

The Value of n = 134217728

The Value of h = 7.4505805969e-009

t=-4.4444444444e-001

ABS of |(-4/9)-t|=2.1124768601e-012

1. 复化Simpson公式

#include "math.h"

#include "stdio.h"

double f(x)

double x;

{

double y;

y=sqrt(x)\*log(x+0.00000000000001);

return(y);

}

void main(){

double a, b, eps;

a=0.0; b=1.0; eps=0.00000000001;

int n, k;

double h,t1,t2,s1,s2,ep,p,x;

n=2;h=(b-a)/n; // 对应m=1开始的步长

t1=h\*((\*f)(a)+2\*(\*f)(a +h)+(\*f)(b))/2.0;

s1=t1;

ep=eps+1.0;

while (ep>=eps)

{ p=0.0;

for (k=0;k<=n-1;k++)

{ x=a+(k+0.5)\*h;

p=p+(\*f)(x);

}

t2=(t1+h\*p)/2.0;

s2=(4.0\*t2-t1)/3.0;

ep=fabs(s2-s1);

t1=t2; s1=s2; n=n +n; h=h/2.0;

}

printf("The Value of n = %d\n", n);

printf("The Value of h = %13.10e\n", 1.0/n);

printf("t=%13.10e\n", s2);

// 积分准确值误差比较

printf("ABS of |(-4/9)-S|=%13.10e\n", fabs((-4/9.0)-s2));

}

编译输出结果：

The Value of n = 67108864

The Value of h = 1.4901161194e-008

t=-4.4444444444e-001

ABS of |(-4/9)-t|=2.8956836928e-012

1. Romberg公式

#include "math.h"

#include "stdio.h"

double f(x)

double x;

{

double y;

y=sqrt(x)\*log(x+0.00000000000001);

return(y);

}

void main(){

double a, b, eps;

a=0.0; b=1.0; eps=0.0000000000000001;

int m, n, i, k;

double y[10], h, ep, p, x, s, q;

n=2; h=(b-a)/n; // 对应m=1开始的步长

y[0]=h\*((\*f)(a)+2\*(\*f)(a +h)+(\*f)(b))/2.0;

m=1; ep=eps+1.0;

while ((ep>=eps)&&(m<=9))

{ p=0.0;

for (i=0;i<=n-1;i++)

{ x=a+(i+0.5)\*h;

p=p+(\*f)(x);

}

p=(y[0]+h\*p)/2.0;

s=1.0;

for (k=1;k<=m; k++)

{ s=4.0\*s;

q=(s\*p-y[k-1])/(s-1.0);

y[k-1]=p; p=q;

}

ep=fabs(q-y[m-1]);

m=m+1; y[m-1]=q; n=n +n; h=h/2.0;

}

printf("The Value of n = %d\n", n);

printf("The Value of h = %13.10e\n", 1.0/n);

printf("t=%13.10e\n", q);

// 积分准确值误差比较

printf("ABS of |(-4/9)-R|=%13.10e\n", fabs((-4/9.0)-q));

}

编译输出结果：

The Value of n = 1024

The Value of h = 9.7656250000e-004

t=-4.4442240878e-001

ABS of |(-4/9)-R|=2.2035665875e-005

理论上排除计算机存储限制，同时选取的精度无限接近与零，求积公式的近似值极限逼近精确值。可考虑到实际运算时机器的特性，存在某一个，当从继续减少时，计算精度将不能再被改善。

1. 记

对于积分做变量代换，可得，给定。确定正数使得

解：变量代换之后，被积函数可以利用Newton-Leibniz公式找到原函数，则：

采用Newton迭代法进行求解：当时

程序源代码：

#include "math.h"

#include "stdio.h"

// Newton迭代求解的函数

int newt(x, eps, js, f)

int js;

double \*x, eps;

void (\*f)();

{

int k, l;

double y[2],d,p,x0,x1;

l=js;

k=1;

x0=\*x;

(\*f)(x0,y);

d=eps+1.0;

while ((d>=eps)&&(l!=0)){

if (fabs(y[1])+1.0==1.0){

printf("err\n");

return(-1);

}

x1=x0-y[0]/y[1];

(\*f)(x1,y);

d=fabs(x1-x0); p=fabs(y[0]);

if (p>d) d=p;

x0=x1; l=l-1;

}

\*x=x1;

k=js-l;

return(k);

}

void main(){

int js, k;

double x0, eps;

void g(double, double []);

eps = 0.000000005;

js = 1000;

x0 = 10.0; // 迭代初值取10.0

k = newt(&x0, eps, js, g);

if(k>=0){

printf("a=%13.10e\n", x0);

}

}

void g(a, x)

double a, x[2];

{

x[0] = (2/3.0)\*exp(-3\*a/2.0)\*(a+2/3.0)-0.25\*pow(10, -8); // 原函数

x[1] = (-1.0\*a)\*exp(-3\*a/2.0); // 一阶导函数

return;

}

编译输出结果：a=1.4758333031e+001

故正数a的取值为：14.758333。

1. 记

取，用Romberg求积公式计算的近似值，使得，并计算，指出所用节点数。

解：程序源代码：

#include "math.h"

#include "stdio.h"

void main(){

double a, b, eps, f(double);

a=0.0; b=14.758333; eps=0.00000005;

int m, n, i, k;

double y[10], h, ep, p, x, s, q;

n=1; h=b-a;

y[0]=h\*((\*f)(a)+(\*f)(b))/2.0;

m=1; ep=eps+1.0;

while ((ep>=eps)&&(m<=9))

{ p=0.0;

for (i=0;i<=n-1;i++)

{ x=a+(i+0.5)\*h;

p=p+(\*f)(x);

}

p=(y[0]+h\*p)/2.0;

s=1.0;

for (k=1;k<=m; k++)

{

s=4.0\*s;

q=(s\*p-y[k-1])/(s-1.0);

y[k-1]=p; p=q;

}

ep=fabs(q-y[m-1]);

m=m+1; y[m-1]=q; n=n +n; h=h/2.0;

}

printf("The Number of Key Points = %d\n", n+1); // 计算所用节点数

printf("K=%13.12e\n", q);

printf("ABS of |(-4/9)-R2n|=%13.10e\n", fabs((-4/9.0)-q)); // 计算近似值与准确值之间的误差

}

double f(double a){

double y;

y = (-1.0\*a)\*exp(-3\*a/2.0);

return(y);

}

编译输出结果：

The Number of Key Points = 65

K=-4.444438267819e-001

ABS of |(-4/9)-R2n|=6.1766252168e-007

即，所用节点数为65。

1. 观察你的计算结果，谈谈你的体会。

答：

1. 复化梯形公式、复化Simpson公式和Romberg公式在计算一定精度前提下的定积分时，可以得到与准确值非常接近的结果；
2. 复化梯形公式算法简单，n = 134217728；复化Simpson公式算法是复化梯形公式的外推n = 67108864；Romberg公式算法复杂，n = 1024，所用节点数愈来愈少，精度愈来愈高，理论上可以无限提高误差量级；
3. Romberg方法在达到相同目标精度的前提下计算量小，速度快。