### 1 数值误差的避免

说明：

封装了三种求结果的方法，并在main函数里调用。

直接展开法求出前七项后不再工作，猜测是数值过大，但无力debug了

源代码：

#include <cmath>

#include <iostream>

using namespace std;

//直接展开法求e^-x

double exp\_direct(int x){

    double sum;

    double jingdu = (1e-16)/2;

    for(int n=0;;n++){

        double x\_n = pow(x, n);

        double x\_factorial = tgamma(n+1);

        int sign = (n%2==0)? 1 : -1 ;

        double s = sign \* x\_n / x\_factorial;

        sum += s;

        if(abs(s/sum) < jingdu){

            break;

        }

    }

    return sum;

}

//递归法求e^-x

double exp\_recursion(int x){

    double sum=1;

    double s=1;

    double jingdu = (1e-16)/2;

    for(int n=1;;n++){

        s = (-1 \* s \* x) / n;

        sum += s;

        if(abs(s/sum) < jingdu){

            break;

        }

    }

    return sum;

}

//递归法求e^x

double exp\_recursion\_pos(int x){

    double sum = 1;

    double s =1;

    double jingdu = (1e-16)/2;

    for(int n=1;;n++){

        s = (s \* x) / n;

        sum += s;

        if(abs(s/sum) < jingdu){

            break;

        }

    }

    return sum;

}

int main(){

    for(int x=0;x<101;x+=10){

        cout<<exp(-1 \* x)<<' ';

    }

    cout<<endl;

    for(int x=0;x<101;x+=10){

        cout<<exp\_recursion(x)<<' ';

    }

    cout<<endl;

    for(int x=0;x<101;x+=10){

        cout<<1/exp\_recursion\_pos(x)<<' ';

    }

    cout<<endl;

    for(int x=0;x<101;x+=10){

        cout<<exp\_direct(x)<<' ';

    }

    cout<<endl;

    return 0;

}

### 3.Hilbert矩阵

## C

说明：

因为需要多次计算Cn，且计算某个Cn时可能会用到之前的结果，所以设计了一个数组用于存放已经计算过的Cn

源代码：

#include<iostream>

#include<vector>

#include <cmath>

//递归计算，用c\_n数组存储Cn以避免大量重复计算

std::vector<double> c\_n(20, 1);

double cn\_rec(int n){

    if(n==1 || n==2) return 1;

    if(c\_n[n-1] != 1) return c\_n[n-1];

    double res = cn\_rec(n-1) \* std::tgamma(n);

    c\_n[n-1] = res;

    return res;}

//计算det（Hn）

double detH\_n(int n){

    double cn = cn\_rec(n);

    double c2n = cn\_rec(2\*n);

    double det = std::pow(cn,4) / c2n;

    return det;}

int main(){

    //n从1到10，计算det（Hn）

    for(int i =1;i<11;i++){

        std::cout << detH\_n(i) << " ";

    }}

## D

说明：

封装了以下几个方法：

构建希尔伯特矩阵

Gem解方程组

将矩阵Cholesky分解的递归函数

Cholesky解方程组

源代码：

#include<iostream>

#include<cmath>

#include<vector>

//创建希尔伯特矩阵

std::vector<std::vector<double>> hilbertMat(int n){

    std::vector<std::vector<double>> matrix(n, std::vector<double>(n,0));

    for (int i = 0; i < n; i++) {

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

            double value = 1.0/(i+j+1);

            matrix[i][j] = value;}}

    return matrix;}

//高斯消元法解方程组

std::vector<double> gem(std::vector<std::vector<double>> matrix){

    int n = matrix.size();//矩阵阶数

    for(int i =0;i<n;i++){

        matrix[i].push\_back(1);//加上b，变成增广矩阵}

    for(int i=0;i<n;i++){   //高斯消元，变成上三角矩阵

        for(int j =i+1;j<n;j++){

            double multip = matrix[i][j] / matrix[i][i] ;

            for(int k = i;k<n+1;k++){

                matrix[j][k] -= matrix[i][k] \* multip;}}}

    std::vector<double> solution(n,0);  //初始化解向量

    for(int i=n-1;i>-1;i--){    //反代解方程

        for(int j = i+1;j<n;j++){

            matrix[i][n] -= matrix[i][j] \* solution[j];}

        solution[i] = matrix[i][n] / matrix[i][i];}

return solution;}

//cholesky的递归函数

std::vector<std::vector<double>>  cholesky\_rec(int depth,std::vector<std::vector<double>>  matrix){

            if(depth == 0) return {{1.0}};  //一阶分解返回1

            auto matrix\_H = cholesky\_rec(depth-1, matrix);  //递归调用

            std::vector<double> solution(depth, 0); //h向量

            double sum=0;   //为了方便beta的计算

            for(int i =0;i<depth;i++){  //反代解h

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

                    matrix[i][depth] -= solution[j] \* matrix\_H[i][j];}

                solution[i] = matrix[i][depth] / matrix\_H[i][i];

                sum += solution[i] \* solution[i];}

            double beta =  std::sqrt(matrix[depth][depth] - sum);   //beta

            //创建新的H

            solution.push\_back(beta);

            matrix\_H.push\_back(solution);

            return matrix\_H;}

//cholesky解方程组

std::vector<double> cholesky(std::vector<std::vector<double>> matrix)

{

    int n = matrix.size();

    //递归地cholesky分解hilbert矩阵，返回下三角矩阵L；

    std::vector<std::vector<double>> L = cholesky\_rec(n-1, matrix);

    //向量y初始化

    std::vector<double> solution\_y(n,0);

    //反代法解向量y

    for(int i=0;i<n;i++){

        double b=1.0;

        for(int j=0;j<i;j++){

            b -= solution\_y[j] \* L[i][j];

        }

        solution\_y[i] = b / L[i][i];

    }

    //x向量

    std::vector<double> solution\_x(n, 0);

    //转置下三角L，获得上三角U

    std::vector<std::vector<double>> U(n, std::vector<double>(n,0));

    for(int i =0;i<n;i++){

        for(int j =0;j<n;j++){

            U[i][j] = L[j][i];

        }

    }

    //反代法解x

    for(int i=n-1;i>-1;i--){

        for(int j = i+1;j<n;j++){

            solution\_y[i] -= U[i][j] \* solution\_x[j];

        }

        solution\_x[i] = solution\_y[i] / U[i][i];

    }

    return solution\_x;

}

int main() {

    int n;

    std::cout << "Enter the number of rows/columns: ";

    std::cin >> n;

    auto matrix = hilbertMat(n);

    auto solution\_gem = gem(matrix);

    auto solution\_cho = cholesky(matrix);

    //输出解

    for(int i=0;i<n;i++){

        std::cout<< solution\_gem[i] << " ";

    }

    std::cout<< std::endl;

    for(int i=0;i<n;i++){

        std::cout<< solution\_cho[i] << " ";

    }

    return 0;

}

### 4 矩阵与二次型

说明：在jupyter中画出的图是静态的，如果想要改变观察图像的角度，需要在ax.view\_init中手动设置θ和φ。且从jupyter复制的代码没有清晰的结构和颜色提示。

源代码：

import numpy as np

import matplotlib.pyplot as plt

from mpl\_toolkits.mplot3d import Axes3D

from IPython import display

# 画函数

def f1(x, y):

z = y + np.sqrt(2 - (x-y)\*\*2)

return z

def f2(x, y):

z = y - np.sqrt(2 - (x-y)\*\*2)

return z

# 生成数据

x = np.linspace(-2, 2, 100)

y = np.linspace(-2, 2, 100)

x, y = np.meshgrid(x, y)

z1 = f1(x, y)

z2 = f2(x, y)

# 绘制图像

fig = plt.figure()

ax = fig.add\_subplot(111, projection='3d')

ax.plot\_surface(x, y, z1, cmap='viridis')

ax.plot\_surface(x, y, z2, cmap='plasma')

# 添加轴标签

ax.set\_xlabel('X')

ax.set\_ylabel('Y')

ax.set\_zlabel('Z')

ax.view\_init(elev=45, azim=90)

plt.show()

#画特征向量

fig = plt.figure()

ax = fig.add\_subplot(111, projection='3d')

ax.quiver(0, 0, 0, 1, 1, 1, length=2, color='red')

ax.quiver(0, 0, 0, 1, 0, -1, length=2, color='red')

ax.quiver(0, 0, 0, 1, -2, 1, length=2, color='red')

ax.set\_xlabel('X')

ax.set\_ylabel('Y')

ax.set\_zlabel('Z')

ax.set\_xlim([-2, 2])

# 设置Y轴范围

ax.set\_ylim([-2, 2])

# 设置Z轴范围

ax.set\_zlim([-2, 2])

ax.view\_init(elev=45, azim=90)

plt.show()