**北京邮电大学软件学院**

**\_\_2019-2020\_\_学年第\_1\_学期实验报告**

**课程名称： 数值计算与分析**

**实验名称： Hilbert矩阵的求解分析问题**

**实验完成人：**

**姓名：** 卜文杭 **学号：**\_2017211969\_**成绩：**\_\_\_\_\_\_\_\_

**指导教师：**\_\_\_\_\_\_\_ \_\_\_\_ 漆涛\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**日 期： 2019 年 12 月 25 日**

1. **实验内容**

实验要求利用Python语言编程环境，运用各种方法求解方程组

其中，为n阶Hilbert矩阵，矩阵元素;x,b为1行n列列向量，,即原方程的准确解为.

方法要求：

（1）方法选用列选主元高斯消去法,完全选主元高斯消去法,Householder变换对角变换法,行平衡辅助法.

（2）选用不同的精度计算误差，比较不同方法所带来的误差并讨论误差的产生原因.

1. **实验环境**
2. Python编译环境：Python 3.7.3
3. 程序编程环境：Jetbeans Pycharm Community 2019.1.2
4. 数字格式支持库：Numpy 1.18.10
5. **实验算法分析**
6. 实验辅助相关数据结构的实现

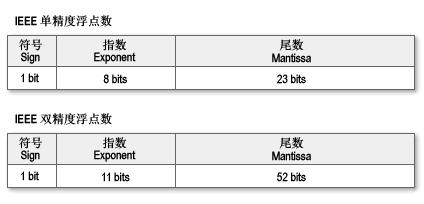
关于矩阵的数值计算需要三大部分数据结构：数值精度、矩阵计算、具体算法的实现.

* 1. 数值精度的实现：

本次实验主要使用两种数值表示实现：以两整数之商表示的有理数，以数值浮点数表示的单精度、双精度表示.

有理数使用自定义python类实现，使用两整数之商表示有理数，包含将有理数类转化为实际实数数值表示的方法，该表示Python默认精度，即Double精度；程序在运算时均保留有理数的分子与分母进行运算，在运算过程中实时对结果进行分母最简化处理，以减少计算规模导致的有效数字位过多导致的系统有效位移除带来的误差错误;

计算机内的浮点数包含单精度，双精度表示，它们大多使用2进制的浮点方法进行标识:



IEEE单精度使用4字节，1符号位8位指数23位位数表示数据;

而双精度使用8字节1位号位11位指数52位位数表示数据.

由于以上浮点数定义较为清晰且方便计算的需要，这里使用numpy的float32(对应IEEE单精度Single)和float64（对应IEEE双精度Double进行计算）库，可以计算出正确结果.

* 1. 矩阵计算

矩阵主要采用Python多维可变长有序list进行.由于python语言没有为list实现矩阵的必要运算，如矩阵四则运算，矩阵的各类变换，这里在自定义库MyMatrix中手动编写实现了这些方法.

* 1. 具体算法的实现:

为屏蔽具体的算法实施细节、使用的数据结构对算法的影响，这里采用了统一的算法实施结构：

首先是生成Hilbert矩阵的方法getHilbert(type,n);

再根据生成的Hilbert矩阵生成对应的解B方法getMatrixB(Hilbert)

最后使用solve方法解决问题即可.

这些solve方法主要包括两部分：根据方法对原有矩阵进行变换使得左部Hilbert矩阵变为三角矩阵；根据三角矩阵求解向量.

对于一个上三角矩阵有求解方法

这个方法是一个算法复杂度为的通用方法，以下略称该方法为**上三角矩阵求解.**实际对于一个下三角矩阵也有类似求解方程方法,称为**下三角矩阵求解**,二者实质是等价的,以下使用的三角矩阵求解方程法,包括具体的程序计算设计默认采用上三角矩阵求解方法,在本文中简称为**三角矩阵求解**.

1. 列选主元高斯消去法

列选主元高斯消去法对于欲求解矩阵H的每一列i，选取当前矩阵i行及以后的行中该列绝对值最大的元素为该列的主元并调整其为第n行，对n行以下的行进行消元使得该行以下的该列元素均为0，以此构造上三角矩阵.

三角矩阵构造伪代码为：

for i in range(0,n):

选每一行的最大主元所在的行maxIndex

h1[maxIndex],h1[i] = h1[i],h1[maxIndex]

b1[maxIndex],b1[i] = b1[i],b1[maxIndex]

#现在最大主元在第i行第i列

for j in range (i,hilbert.n):

h1[i][j] /= temp

b1[i][0] = (hilbert.type)(b1[i][0] / temp)

for j in range (i+1,hilbert.n):

#进行行变换使得该行以下行该列元素为0

1. 完全选主元高斯消去法

完全选主元高斯消去法类似于列选主元高斯消去法，只是主元的选取不限制于该行单行而是整个n-i行n-i列的矩阵中,允许行之间互相交换.为记录行所对应的实际向量顺序,需要单开一个数组以记录实际的变量顺序.

三角矩阵构造伪代码为：

x = [i for i in range(0,hilbert.n)]

#变量顺序表

for i in range (0,hilbert.n):

maxIndex0,maxIndex1 = i,i

for j in range (i,hilbert.n):

for k in range (i,hilbert.n):

#选每一行的最大主元所在的行maxIndex

h1[maxIndex0],h1[i] = h1[i],h1[maxIndex0]

b1[maxIndex0],b1[i] = b1[i],b1[maxIndex0]

#行交换

for j in range (0,hilbert.n):

h1[j][maxIndex1],h1[j][i] = h1[j][i],h1[j][maxIndex1]

x[maxIndex1],x[i] = x[i],x[maxIndex1]

#列交换

#现在最大主元在第i行第i列

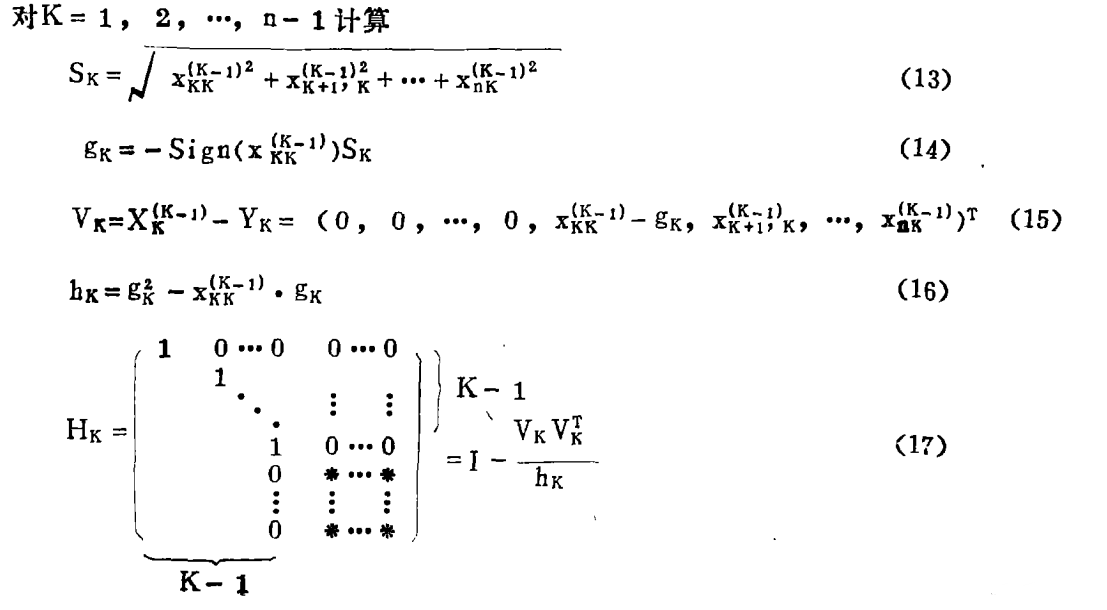
for j in range (i+1,hilbert.n):

#进行行变换使得该行以下行该列元素为0

1. Householder变换对角变换法

Householder变换是一种反射变换方法，它基于一个事实，即每一个范量非0的非奇异矩阵都可以表示为一个上三角矩阵和下三角矩阵之乘积.于是，可以对Hilbert矩阵（该矩阵显然是非奇异矩阵）进行Hoseholder反射变换以将其改变为上三角矩阵.

Householder变换方法如下：





具体相关证明[1]在此简略，以下给出HouseHolder变换所指示的三角矩阵构造伪代码：

for i in range (0,hilbert.n):

for j in range (i,hilbert.n):

sk += (h1[j][i] \* h1[j][i])

sk = sqrt(sk)

gk = -sk \* sign(h1[i][i])

vk = trans([[0 if j < i else (h1[j][i] - gk if j == i else h1[j][i] )for j in range(0,hilbert.n)]])

hk = gk\* gk - h1[i][i] \* gk

Hk = submi(oneMatrix(hilbert.n),div(multi(vk,trans(vk)),hk))

h1 = multi(Hk,h1)

b1 = multi(Hk,b1)

1. 行平衡辅助法

行平衡辅助法主要是在以上提到的三种消元或变换过程中途多次乘一个矩阵范数非1的对角矩阵实现某行某列的数据整体放缩或扩张，以使得每次进行行间运算时行间的关键值数量级均相近.这样会减少误差.

本次实验中采用的行平衡方法是结合列选主元高斯消去法的行平衡方法，该方法是在每行进行主元选取前，通过行调整将每行的首元素的数量级强制调整为1，即每个选取数都应当在1和10之间.

1. **实验模拟结果与结果分析**
2. 实验模拟结果

对不同数制下矩阵规模的Hilbert矩阵进行求解，求解结果如下：

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 规模n | 数制 | 求解方法 | 解误差 | 目标b误差 |
| 2 | Single | 列选主元 | 3.9984e-7 | 0.0 |
| 完全选主元 | 3.9984e-7 | 0.0 |
| Householder | 4.4511e-7 | 3.00e-8 |
| 行平衡列选主元 | 3.9984e-7 | 0.0 |
| Double | 列选主元 | 8.0059e-16 | 0.0 |
| 完全选主元 | 8.0059e-16 | 0.0 |
| Householder | 1.2009e-15 | 2.2204e-16 |
| 行平衡列选主元 | 8.0059e-16 | 0.0 |
| Rational | 列选主元 | 0.0 | 0.0 |
| 完全选主元 | 0.0 | 0.0 |
| 3 | Single | 列选主元 | 7.7302e-6 | 5.9604e-8 |
| 完全选主元 | 6.0280e-6 | 0.0 |
| Householder | 1.3478 | 0.0036 |
| 行平衡列选主元 | 2.9406e-6 | 0.0 |
| Double | 列选主元 | 1.7634e-14 | 2.22.4e-16 |
| 完全选主元 | 1.5413e-14 | 0.0 |
| Householder | 1.3478 | 0.0036 |
| 行平衡列选主元 | 1.9574e-14 | 2.2204e-16 |
| Rational | 列选主元 | 0.0 | 0.0 |
| 完全选主元 | 0.0 | 0.0 |
| 5 | Single | 列选主元 | 0.0094 | 5.9604e-8 |
| 完全选主元 | 0.0121 | 2.6656e-7 |
| Householder | 3.3754 | 0.0253 |
| 行平衡列选主元 | 0.0076 | 2.3841 |
| Double | 列选主元 | 1.3596e-12 | 3.3306e-16 |
| 完全选主元 | 6.0618e-12 | 2.4825e-16 |
| Householder | 3.3754 | 0.0253 |
| 行平衡列选主元 | 1.3889 | 4.0878e-16 |
| Rational | 列选主元 | 0.0 | 0.0 |
| 完全选主元 | 0.0 | 0.0 |
| 7 | Single | 列选主元 | 0.6463 | 1.6858e-7 |
| 完全选主元 | 0.0638 | 5.9605e-8 |
| Householder | 5.4632 | 0.0572 |
| 行平衡列选主元 | 0.8751 | 1.8849e-7 |
| Double | 列选主元 | 3.2882e-8 | 4.0030e-16 |
| 完全选主元 | 3.2205e-8 | 2.2204e-16 |
| Householder | 5.4632 | 0.0572 |
| 行平衡列选主元 | 6.2741 | 2.7195e-16 |
| Rational | 列选主元 | 0.0 | 0.0 |
| 完全选主元 | 0.0 | 0.0 |
| 10 | Single | 列选主元 | 34.7270 | 2.3842e-7 |
| 完全选主元 | 14.5699 | 2.2302e-7 |
| Householder | 8.6368 | 0.1110 |
| 行平衡列选主元 | 34.0330 | 1.0532e-15 |
| Double | 列选主元 | 0.0005 | 5.6610e-16 |
| 完全选主元 | 0.0007 | 5.8747 |
| Householder | 8.6368 | 0.1110 |
| 行平衡列选主元 | 0.0001 | 1.0533e-15 |
| Rational | 列选主元 | 2.0171e-6 | 1.0532e-15 |
| 完全选主元 | 4.4767e-6 | 1.2412e-15 |
| 15 | Single | 列选主元 | 585 | 7.0268e-6 |
| 完全选主元 | 113.6793 | 1.4709e-6 |
| Householder | 13.9260 | 0.2103 |
| 行平衡列选主元 | 13.9660 | 7.6563e-7 |
| Double | 列选主元 | 18.0097 | 1.3643e-15 |
| 完全选主元 | 20.6146 | 2.1384e-13 |
| Householder | 13.9260 | 0.2012 |
| 行平衡列选主元 | 10.3754 | 2.6944e-15 |
| Rational | 列选主元 | 14319 | 3.1072e-13 |
| 完全选主元 | 23689 | 7.0194e-13 |
| 20 | Single | 列选主元 | 68.7250 | 7.3243e-7 |
| 完全选主元 | 39.1079 | 7.8624e-7 |
| Householder | 19.1782 | 0.2853 |
| 行平衡列选主元 | 99.1347 | 1.7821e-6 |
| Double | 列选主元 | 61.7647 | 1.9420e-15 |
| 完全选主元 | 55.2942 | 2.6084e-15 |
| Householder | 19.1782 | 0.2853 |
| 行平衡列选主元 | 57.1656 | 9.2536e-15 |
| Rational | 列选主元 | 853702 | 1.5912e-11 |
| 完全选主元 | 5852208 | 2.6082e-10 |
| 30 | Single | 列选主元 | 10486 | 7.6421e-5 |
| 完全选主元 | 356.5514 | 3.5146e-6 |
| Householder | 29.5830 | 0.4333 |
| 行平衡列选主元 | 153.2991 | 1.9101e-6 |
| Double | 列选主元 | 432.2398 | 5.3614e-15 |
| 完全选主元 | 636.5675 | 1.5199e-14 |
| Householder | 29.5830 | 0.4332 |
| 行平衡列选主元 | 120.7359 | 9.1968e-15 |
| Rational | 列选主元 | 8.8860e9 | 1.5726e-7 |
| 完全选主元 | 196451 | 3.8697e-12 |

注：有理数数制不支持Householder变换及行平衡列选主元方法.

1. 实验结果误差比较与分析
   1. 数制选择对计算结果误差的影响

一般而言，对于浮点数，精度越高会导致计算结果的误差减小。但是不同的精度受到不同算法优化调整的能力是不同的，有些时候精度提高反而会因为留下计算误差而导致计算误差扩大化.例如高斯消去法在规模较小时Double精度的计算结果要大于Single精度，但在规模较大时，Double精度留下的更多精度差距累计聚集导致表现反而低于低精度的Single.

而Rational有理数的计算结果误差完全取决于存储有理数的分子与分母的精度数；在分子分母精度较高的时候，可以做到零误差；但分子分母精度较低的时候，舍入误差会极大影响结果的正确性.此外，有理数的计算不适用于一些需要开方运算的算法,这也极大的影响了有理数精度计算的可用性.

* 1. 方法选择对计算结果误差的影响

不同方法对于计算结果的误差影响与使用的数制和运算规模有很大关系.

对于高斯消去法而言，精度的提升会极大减小误差;但完全选主元相对于列选主元法而言,对于低精度数据的准确度提升会显著大于高数据元素,高精度数据在某些时刻使用完全选主元方法反而会降低数据准确度;

而对于基于变换的方法，如Householder方法而言，由于计算迭代次数相对固定，所以对于高阶矩阵而言误差可以保持稳定，并且相对于高斯消去法而言无论在时间效率还是精度效率都相对出色.

可以认为，方法选择对计算结果产生的计算舍入误差的影响与算法的实际复杂度成正比.低复杂度的算法在处理高阶问题时，在阶数增加时,最终准确度会占据优势.

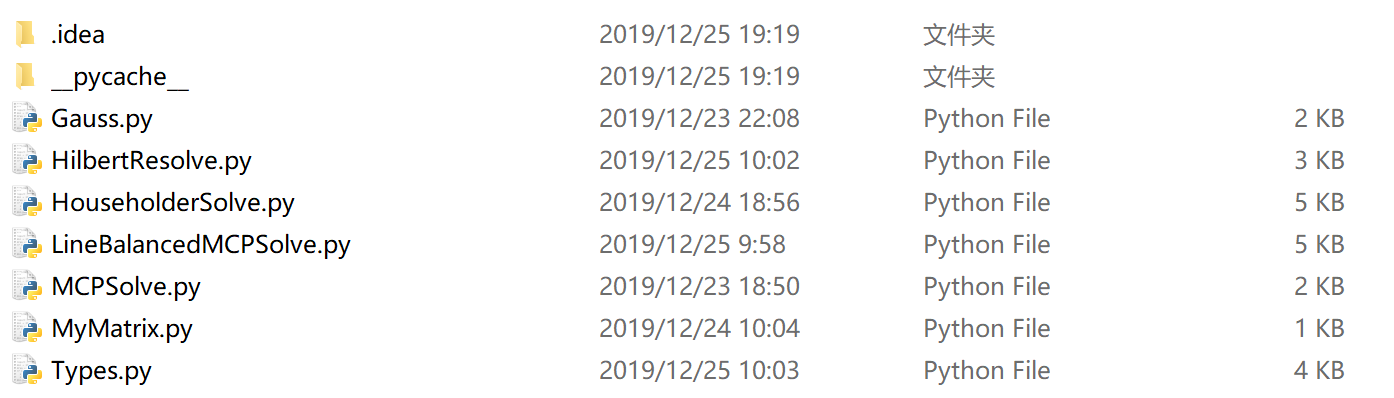
* 1. 程序设计及其他因素对计算结果误差的影响

所要求解的矩阵为Hilbert矩阵，该矩阵的条件数随规模指数型增加，代表该矩阵病态程度随规模提升程度较高.因此，任何基于消元的计算方法在规模增加时都会出现较明显的误差度增大问题.

这个矩阵的行平衡方法对于实际算法的优化在维数较高的时候会有明显体现，但维度较低时，由于引入了进一步的运算会导致产生误差明显大于算法提升效果，因此只适用于某些较高维度的Hilbert矩阵计算.

1. **附录**
2. 算法程序包结构与文件说明

实验文件夹../HilbertResolve/



算法主程序:HilbertResolve.py

相关数据结构定义：矩阵运算支持MyMatrix.py，数据类定义Types.py

算法程序：MCPSolve.py,Gauss.py,HouseholderSolve.py,LineBalancedMCPSolve.py

1. 算法程序代码

b.1 MyMatrix.py

class MyMatrix:

def \_\_init\_\_(self, type, n):

self.type = type

self.n = n

self.\_\_matrix = [ [] for i in range (0,n) ]

def \_\_getitem\_\_(self, item):

if (item < 0 or item >= self.n):

raise IndexError("MyMatrix: Line OutOfBounds exception")

return self.\_\_matrix[item]

def \_\_setitem\_\_(self, key, value):

self.\_\_matrix[key] = value

return self.\_\_matrix[key]

def \_\_len\_\_(self):

return self.n

def print(self):

for i in range (0,self.n):

print("[",end='')

for j in range(0,self.n):

print(self.\_\_matrix[i][j],end='')

if (j != self.n - 1):

print (",",end=' ')

print("]")

b.2 Types.py

# coding=utf-8

from numpy import float32 as Single

from numpy import float64 as Double

#Single和Double类型使用numpy扩展包的float32和float64shix

#有理数使用自己实现的类实现

class Rational:

"""自定义有理数类"""

def \_\_init\_\_(self,numerator,denominator):

if denominator == 0:

raise RuntimeError("Rational: denominator cannot be zero.")

#创建有理数

self.\_\_numerator = numerator #分子

self.\_\_denominator = denominator #分母

self.simplify()

def simplify(self):

#化简

isMinus = False

minusPoints = 0

if self.\_\_numerator == 0:

self.\_\_denominator = 1

return

if self.\_\_numerator < 0:

minusPoints += 1

self.\_\_numerator = - self.\_\_numerator

if self.\_\_denominator < 0:

minusPoints += 1

self.\_\_denominator = - self.\_\_denominator

isMinus = (True if (minusPoints % 2 == 1) else False)

rationgcd = Rational.gcd(self.\_\_numerator,self.\_\_denominator)

self.\_\_numerator /= rationgcd

self.\_\_denominator /= rationgcd

if isMinus:

self.\_\_numerator = - self.\_\_numerator

def abs(self):

return self.getReal()

def getReal(self):

#返回实数数值

return self.\_\_numerator / self.\_\_denominator

def \_\_str\_\_(self):

return '[' + str(self.\_\_numerator) + "," + str(self.\_\_denominator) + ']'

def printValue(self):

print(self.\_\_str\_\_())

def \_\_add\_\_(self, other):

#有理数相加

if isinstance(other,Rational):

return Rational(self.\_\_numerator \* other.\_\_denominator + self.\_\_denominator \* other.\_\_numerator,self.\_\_denominator \* other.\_\_denominator)

else:

return NotImplemented

def \_\_sub\_\_(self, other):

# 有理数相减

if isinstance(other, Rational):

return Rational(self.\_\_numerator \* other.\_\_denominator - self.\_\_denominator \* other.\_\_numerator,

self.\_\_denominator \* other.\_\_denominator)

else:

return NotImplemented

def \_\_mul\_\_(self, other):

#有理数乘法

if isinstance(other,Rational):

return Rational(self.\_\_numerator \* other.\_\_numerator , self.\_\_denominator \* other.\_\_denominator)

else:

return NotImplemented

def \_\_truediv\_\_(self, other):

#有理数除法

if isinstance(other,Rational):

return Rational(self.\_\_numerator \* other.\_\_denominator , self.\_\_denominator \* other.\_\_numerator)

else:

return NotImplemented

def gcd(a, b):

"""辗转相除求最大公因数"""

while True:

if a < b:

a, b = b, a

a = a % b

if a == 0:

break

return b

if \_\_name\_\_ == "\_\_main\_\_":

#测试函数

a = Rational(2,3)

b = Rational(2,4)

c = Rational(4,2)

d = a \* b

e = Single(0)

f = Double(1)

print(a)

print(b)

b.3 MCPSolve.py

from Types import Rational,Single,Double

def getReal(number):

if isinstance(number,Rational):

return number.getReal()

elif isinstance(number,Single) or isinstance(number,Double):

return number

else:

raise NotImplemented

def MCPsolve(hilbert,b):

#列选主元

if not isinstance(b[0][0],hilbert.type):

raise NotImplemented

h1 = hilbert

b1 = b

for i in range (0,hilbert.n):

maxIndex = i

for j in range (i,hilbert.n):

if abs(getReal(h1[j][i])) > abs(getReal(h1[maxIndex][i])):

maxIndex = j

#选每一行的最大主元所在的行maxIndex

h1[maxIndex],h1[i] = h1[i],h1[maxIndex]

b1[maxIndex],b1[i] = b1[i],b1[maxIndex]

#现在最大主元在第i行第i列

temp = h1[i][i]

for j in range (i,hilbert.n):

h1[i][j] /= temp

b1[i][0] = (hilbert.type)(b1[i][0] / temp) if (hilbert.type != Rational) else (b1[i][0] / temp)

#本行首元素归1化

for j in range (i+1,hilbert.n):

temp = h1[j][i]

for k in range(i,hilbert.n):

h1[j][k] -= h1[i][k] \* temp

b1[j][0] -= b1[i][0] \* temp

for i in range (hilbert.n - 1, 0, -1):

for j in range(i - 1, -1 , -1):

b1[j][0] = (hilbert.type)(b1[j][0] - b1[i][0] \* h1[j][i]) if (hilbert.type != Rational) else (b1[j][0] - b1[i][0] \* h1[j][i])

#以上对角化

return b1

b.4 Gauss.py

from Types import Rational,Single,Double

def getReal(number):

if isinstance(number,Rational):

return number.getReal()

elif isinstance(number,Single) or isinstance(number,Double):

return number

else:

raise NotImplemented

def Gausssolve(hilbert,b):

#全选主元

if not isinstance(b[0][0],hilbert.type):

raise NotImplemented

h1 = hilbert

b1 = b

x = [i for i in range(0,hilbert.n)]

for i in range (0,hilbert.n):

maxIndex0,maxIndex1 = i,i

for j in range (i,hilbert.n):

for k in range (i,hilbert.n):

if abs(getReal(h1[j][k])) > abs(getReal(h1[maxIndex0][maxIndex1])):

maxIndex0,maxIndex1 = j,k

#选每一行的最大主元所在的行maxIndex

h1[maxIndex0],h1[i] = h1[i],h1[maxIndex0]

b1[maxIndex0],b1[i] = b1[i],b1[maxIndex0]

#行交换

for j in range (0,hilbert.n):

h1[j][maxIndex1],h1[j][i] = h1[j][i],h1[j][maxIndex1]

x[maxIndex1],x[i] = x[i],x[maxIndex1]

#列交换

#现在最大主元在第i行第i列

temp = h1[i][i]

for j in range (i,hilbert.n):

h1[i][j] /= temp

b1[i][0] = (hilbert.type)(b1[i][0] / temp) if (hilbert.type != Rational) else (b1[i][0] / temp)

#本行首元素归1化

for j in range (i+1,hilbert.n):

temp = h1[j][i]

for k in range(i,hilbert.n):

h1[j][k] -= h1[i][k] \* temp

b1[j][0] -= b1[i][0] \* temp

for i in range (hilbert.n - 1, 0, -1):

for j in range(i - 1, -1 , -1):

b1[j][0] = (hilbert.type)(b1[j][0] - b1[i][0] \* h1[j][i]) if (hilbert.type != Rational) else (b1[j][0] - b1[i][0] \* h1[j][i])

#以上对角化

for i in range (0,hilbert.n):

for j in range (i,hilbert.n):

if x[i] > x[j]:

x[i],x[j]=x[j],x[i]

b1[i][0],b1[j][0]=b1[j][0],b1[i][0]

return b1

b.5 HouseholderSolve.py

from Types import Rational,Single,Double

from numpy import sign,sqrt

def getReal(number):

if isinstance(number,Rational):

return number.getReal()

elif isinstance(number,Single) or isinstance(number,Double):

return number

else:

raise NotImplemented

def oneMatrix(n):

#n位单位矩阵

return [ [1 if i==j else 0 for j in range(0,n)] for i in range(0,n)]

def submi(matrix1,matrix2):

#矩阵减法

if len(matrix1) != len(matrix2) or len(matrix1[0]) != len(matrix2[0]):

raise TypeError("Matrix submission ERROR")

result = [[] for i in range(0,len(matrix1))]

for i in range(0, len(matrix1)):

for j in range(0, len(matrix2[0])):

result[i].append(matrix1[i][j] - matrix2[i][j])

return result

def mul(matrix,number):

#矩阵乘法

result = [[] for i in range(0,len(matrix))]

for i in range (0,len(matrix[0])):

for j in range(0, len(matrix)):

result[j].append(matrix[i][j] \* number)

return result

def div(matrix,number):

#矩阵乘法

result = [[] for i in range(0,len(matrix))]

for i in range (0,len(matrix[0])):

for j in range(0, len(matrix)):

result[j].append(matrix[i][j] / number)

return result

def multi(matrix1,matrix2):

#矩阵乘法

if len(matrix1[0]) != len(matrix2):

raise TypeError("Matrix multiply ERROR")

type1 = type(matrix1[0][0])

result = [ [] for i in range (0,len(matrix1))]

for i in range (0,len(matrix1)):

for j in range (0,len(matrix2[0])):

sum = (type1)(0) if type1 != Rational else Rational(0,1)

for k in range (0,len(matrix1[0])):

sum += matrix1[i][k] \* matrix2[k][j]

result[i].append(sum)

return result

def trans(matrix):

#矩阵转置运算

result = [[] for i in range(0,len(matrix[0]))]

for i in range (0,len(matrix)):

for j in range(0, len(matrix[0])):

result[j].append(matrix[i][j])

return result

def HouseholderSolve(hilbert,b):

if not isinstance(b[0][0],hilbert.type):

raise NotImplemented

if hilbert.type == Rational:

raise ArithmeticError("cannot use householder with type Rational")

h1 = hilbert

b1 = b

sk = (hilbert.type)(0)

for i in range (0,hilbert.n):

for j in range (i,hilbert.n):

sk += (h1[j][i] \* h1[j][i])

sk = sqrt(sk)

gk = -sk \* sign(h1[i][i])

vk = trans([[0 if j < i else (h1[j][i] - gk if j == i else h1[j][i] )for j in range(0,hilbert.n)]])

hk = gk\* gk - h1[i][i] \* gk

Hk = submi(oneMatrix(hilbert.n),div(multi(vk,trans(vk)),hk))

h1 = multi(Hk,h1)

b1 = multi(Hk,b1)

#主元归一化！

for i in range (0,hilbert.n):

temp = h1[i][i]

for j in range (i,hilbert.n):

h1[i][j] /= temp

b1[i][0] /= temp

for i in range (hilbert.n - 1, 0, -1):

for j in range(i - 1, -1 , -1):

b1[j][0] = (hilbert.type)(b1[j][0] - b1[i][0] \* h1[j][i]) if (hilbert.type != Rational) else (b1[j][0] - b1[i][0] \* h1[j][i])

#以上对角化

return b1

if \_\_name\_\_ == "\_\_main\_\_":

a = [[Rational(1,1),Rational(2,1)],[Rational(3,1),Rational(4,1)]]

b = [[Rational(1,1),Rational(2,1)],[Rational(3,1),Rational(4,1)]]

c = multi(a,b)

for i in range (0,len(c)):

print('[',end="")

for j in range (0,len(c[0])):

print(c[i][j],end=',' if j != len(c[0]) - 1 else '')

print(']')

c = trans(c)

for i in range (0,len(c)):

print('[',end="")

for j in range (0,len(c[0])):

print(c[i][j],end=',' if j != len(c[0]) - 1 else '')

print(']')

a = [[Rational(1, 1), Rational(2, 1)], [Rational(3, 1), Rational(4, 1)]]

b = [[Rational(-1, 1), Rational(2, 1)], [Rational(3, 1), Rational(4, 1)]]

c = submi(a,b)

for i in range(0, len(c)):

print('[', end="")

for j in range(0, len(c[0])):

print(c[i][j], end=',' if j != len(c[0]) - 1 else '')

print(']')

num = 0

print(sign(num))

b.6 LineBalancedMCPSolve.py

from Types import Rational,Single,Double

def getReal(number):

if isinstance(number,Rational):

return number.getReal()

elif isinstance(number,Single) or isinstance(number,Double):

return number

else:

raise NotImplemented

def oneMatrix(n):

#n位单位矩阵

return [ [1 if i==j else 0 for j in range(0,n)] for i in range(0,n)]

def submi(matrix1,matrix2):

#矩阵减法

if len(matrix1) != len(matrix2) or len(matrix1[0]) != len(matrix2[0]):

raise TypeError("Matrix submission ERROR")

result = [[] for i in range(0,len(matrix1))]

for i in range(0, len(matrix1)):

for j in range(0, len(matrix2[0])):

result[i].append(matrix1[i][j] - matrix2[i][j])

return result

def mul(matrix,number):

#矩阵乘法

result = [[] for i in range(0,len(matrix))]

for i in range (0,len(matrix[0])):

for j in range(0, len(matrix)):

result[j].append(matrix[i][j] \* number)

return result

def div(matrix,number):

#矩阵乘法

result = [[] for i in range(0,len(matrix))]

for i in range (0,len(matrix[0])):

for j in range(0, len(matrix)):

result[j].append(matrix[i][j] / number)

return result

def multi(matrix1,matrix2):

#矩阵乘法

if len(matrix1[0]) != len(matrix2):

raise TypeError("Matrix multiply ERROR")

type1 = type(matrix1[0][0])

result = [ [] for i in range (0,len(matrix1))]

for i in range (0,len(matrix1)):

for j in range (0,len(matrix2[0])):

sum = (type1)(0) if type1 != Rational else Rational(0,1)

for k in range (0,len(matrix1[0])):

sum += matrix1[i][k] \* matrix2[k][j]

result[i].append(sum)

return result

def trans(matrix):

#矩阵转置运算

result = [[] for i in range(0,len(matrix[0]))]

for i in range (0,len(matrix)):

for j in range(0, len(matrix[0])):

result[j].append(matrix[i][j])

return result

def getReal(number):

if isinstance(number,Rational):

return number.getReal()

elif isinstance(number,Single) or isinstance(number,Double):

return number

else:

raise NotImplemented

def LineBalancedMCPsolve(hilbert,b):

#列选主元

if not isinstance(b[0][0],hilbert.type):

raise NotImplemented

h1 = hilbert

b1 = b

for i in range (0,hilbert.n):

for j in range (i,hilbert.n):

if abs(h1[j][i].getReal() > 1 if type(h1[j][i]) == Rational else abs(getReal(h1[j][i]))) > 1:

while (abs(h1[j][i].getReal() > 1 if type(h1[j][i]) == Rational else abs(getReal(h1[j][i]))) > 1):

for k in range(i,hilbert.n):

h1[j][k] /= (10 if type(h1[j][k]) != Rational else Rational(10,1))

b1[j][0] /= (10 if type(b1[j][0]) != Rational else Rational(10,1))

elif abs(h1[j][i].getReal()) < 1 if type(h1[j][i]) == Rational else abs(getReal(h1[j][i])) < 1:

while (abs(h1[j][i].getReal()) < 1 if type(h1[j][i]) == Rational else abs(getReal(h1[j][i])) < 1):

for k in range(i,hilbert.n):

h1[j][k] \*= (10 if type(h1[j][k]) != Rational else Rational(10,1))

b1[j][0] \*= (10 if type(b1[j][0]) != Rational else Rational(10,1))

#以上先每行平衡

maxIndex = i

for j in range (i,hilbert.n):

if abs(getReal(h1[j][i])) > abs(getReal(h1[maxIndex][i])):

maxIndex = j

#选每一行的最大主元所在的行maxIndex

h1[maxIndex],h1[i] = h1[i],h1[maxIndex]

b1[maxIndex],b1[i] = b1[i],b1[maxIndex]

#现在最大主元在第i行第i列

temp = h1[i][i]

for j in range (i,hilbert.n):

h1[i][j] /= temp

b1[i][0] = (hilbert.type)(b1[i][0] / temp) if (hilbert.type != Rational) else (b1[i][0] / temp)

#本行首元素归1化

for j in range (i+1,hilbert.n):

temp = h1[j][i]

for k in range(i,hilbert.n):

h1[j][k] -= h1[i][k] \* temp

b1[j][0] -= b1[i][0] \* temp

for i in range (hilbert.n - 1, 0, -1):

for j in range(i - 1, -1 , -1):

b1[j][0] = (hilbert.type)(b1[j][0] - b1[i][0] \* h1[j][i]) if (hilbert.type != Rational) else (b1[j][0] - b1[i][0] \* h1[j][i])

#以上对角化

return b1

b.7 HilbertResolve.py

from Types import Rational,Single,Double

from MyMatrix import MyMatrix

from MCPSolve import MCPsolve,getReal

from Gauss import Gausssolve

from HouseholderSolve import HouseholderSolve,multi,submi

from LineBalancedMCPSolve import LineBalancedMCPsolve

def getHilbert(type, n):

hilbert = MyMatrix(type,n)

if type == Rational:

for i in range (1, n+1):

for j in range (0, n):

hilbert[i-1].append(Rational(1,i+j))

else:

for i in range (1, n+1):

for j in range (0 , n):

hilbert[i-1].append((type)(1/(i+j)))

return hilbert

def getMatrixB(hilbert):

if isinstance(hilbert,MyMatrix):

b = [ [] for i in range(0,hilbert.n)]

for i in range (0, hilbert.n):

sum = (Rational(0,1) if hilbert.type == Rational else 0)

for j in range (0, hilbert.n):

sum += hilbert[i][j]

if hilbert.type == Rational:

b[i].append(sum)

else:

b[i].append((hilbert.type)(sum))

return b

def absoluteError(x):

error = 0

for i in range (0,len(x)):

error += (getReal(x[i][0]) - 1) \*\* 2

return error \*\* (1/2)

def aimError(x):

type1 = type(x[0][0])

b = getMatrixB(getHilbert(type1,len(x)))

destination = multi(getHilbert(type1,len(x)),x)

error = 0

for i in range (0,len(x)):

error += (getReal(b[i][0]) - getReal(destination[i][0])) \*\* 2

return error \*\* (1/2)

if \_\_name\_\_ == "\_\_main\_\_":

for n in (2,3,5,7,10,15,20,25,30):

print('n=',n,':')

for solve in [MCPsolve,Gausssolve,HouseholderSolve,LineBalancedMCPsolve]:

for T in [Single,Double,Rational]:

if (solve == LineBalancedMCPsolve or solve == HouseholderSolve) and T == Rational:

continue

else:

print('solve:',solve,' type:',T,end='')

hilbert = getHilbert(T,n)

b = getMatrixB(hilbert)

x = solve(hilbert,b)

print('absolute error:',absoluteError(x),' aim error:',aimError(x))