实验报告4

BY 刘家骥 PB20071417

一, 实验目的:

理论上的分析表明, 求解病态的线性方程组是有困难的. 编写程序研究实际情况是否如此，发现具体计算过程会产生的现象.

二, 实验内容:

考虑线性方程组 $Hx=b$，其中H 为n阶 Hilbert 矩阵，即

$$H=(h_{ij})_{n\times n}\,,\quad h_{ij}=\frac{1}{i+j-1},\ i,\ j=1,2,...,n$$

这是一个著名的病态问题. 通过先给定解, 在这里是取$x_i=1$, 再计算出右端向量b的办法给出一个精确解已知的问题.

系数矩阵H取不同的阶数n, 分别用Doolittle LU分解法、Jacobi迭代、Gauss-Seidel迭代三种方法来求解方程组, 并给出两种迭代法的迭代步数和误差, Doolittle LU 分解法中的上三角阵U和下三角阵L.

三, 计算方法和计算结果:

1, 分别编写Doolittle LU 分解法、Jacobi 迭代、Gauss-Seidel 迭代的一般程序(见报告附录)；

2, 先取阶数n=6，分别用 LU 分解法、Jacobi 迭代、Gauss-Seidel 迭代去求解上述的病态方程组$Hx=b$；分别报告它们的数值结果(包括数值解、迭代步数)以及它们在1-范数下的计算误差.

已知原方程组的精确解就是$x_i^*=1\,,i=1\,,2\,,...,n$, 因此这里的计算误差就是

$$E=\|x^*-x\|_1$$

n=6 时, 三种方法结果如下:

LU分解法:

$$x_1=1.00000000$$

$$x_2=1.00000000$$

$$x_3=1.00000000$$

$$x_4=1.00000000$$

$$x_5=1.00000000$$

$$x_6=1.00000000$$

计算误差为

$$E_{\mathrm{LU}} = 0.00000000$$

Jacobi迭代失败, 无法得到结果.

Gauss-Seidel 迭代:

$$x_1 = 0.99961883$$

$$x_2 = 1.00578281$$

$$x_3 = 0.98166532$$

$$x_4 = 1.01131287$$

$$x_5 = 1.01643398$$

$$x_6 = 0.98491751$$

迭代步数

$$N = 6948$$

计算误差为

$$E_{\text{G-S}} = 0.06732800$$

3, 再取阶数n=9，仍然用上述的三种计算方法去求解它们，请分别报告各自的数值结果(含数值解、迭代步数)以及计算过程中可能出现的问题.

n=9 时, 三种方法结果如下:

LU分解法:

$$x_1 = 1.00000000$$

$$x_2 = 1.00000000$$

$$x_3 = 0.99999996$$

$$x_4 = 1.00000025$$

$$x_5 = 0.99999914$$

$$x_6 = 1.00000166$$

$$x_7 = 0.99999818$$

$$x_8 = 1.00000105$$

$$x_9 = 0.99999975$$

计算误差为

$$E_{\mathrm{LU}}=0.00000594$$

Jacobi迭代失败, 无法得到结果.

Gauss-Seidel 迭代:

$$x_1=1.00015805$$

$$x_2=0.99573768$$

$$x_3=1.02133512$$

$$x_4=0.97328986$$

$$x_5=0.98984582$$

$$x_6=1.01271805$$

$$x_7=1.01935944$$

$$x_8=1.00720628$$

$$x_9=0.98005216$$

迭代步数

$$N=4669$$

计算误差为

$$E_{\text{G-S}}=0.12185143$$

可以看到n=9时, 采用LU分解法的结果没有n=6时那样精确, 而且Gauss-Seidel迭代的计算误差也明显增大了.

这可能是因为n=9时的一部分系数h_{ij}可以取到比较小的值, 例如

$h_{99}=0.05882353$, 因而导致之后计算出的矩阵U的部分元素值很小（可以看第4部分的计算结果，例如$u_{88}=0.00000001$), 小数作除数，使得计算误差增大.

并且迭代速度也会受其影响而变慢, 这导致在$\|x^{(k+1)}-x^{(k)}\|_1\approx\varepsilon=10^{-5}$时, $E_{\text{G-S}}=\|x^*-x^{(k)}\|_1$仍然比较大.(迭代速度慢时, 相邻两次迭代结果很接近, 并不意味着迭代结果和精确解很接近!)

4, 对LU 分解，请分别报告n=6和9时的LU 分解的分解结果，即给出对应的三角矩阵L和U.

n=6时,

$$L = \begin{pmatrix} 1.00000000 &&&&& \\ 0.50000000 & 1.00000000 &&&& \\ 0.33333333 & 1.00000000 & 1.00000000 &&& \\ 0.25000000 & 0.90000000 & 1.50000000 & 1.00000000 && \\ 0.20000000 & 0.80000000 & 0.71428571 & 2.00000000 & 1.00000000 & \\ 0.16666667 & 0.71428571 & 1.78571429 & 2.77777778 & 2.50000000 & 1.00000000 \end{pmatrix}$$

$$U = \begin{pmatrix} 1.00000000 & 0.50000000 & 0.33333333 & 0.25000000 & 0.20000000 & 0.16666667 \\ & 0.08333333 & 0.08333333 & 0.07500000 & 0.06666667 & 0.05952381 \\ && 0.00555556 & 0.00833333 & 0.00952381 & 0.00992063 \\ &&& 0.00035714 & 0.00071429 & 0.00099206 \\ &&&& 0.00002268 & 0.00005669 \\ &&&&& 0.00000143 \end{pmatrix}$$

n=9时,

$$L =$$

$$\begin{pmatrix} 1.00000000 &&&&&&&& \\ 0.50000000 & 1.00000000 &&&&&&& \\ 0.33333333 & 1.00000000 & 1.00000000 &&&&&& \\ 0.25000000 & 0.90000000 & 1.50000000 & 1.00000000 &&&&& \\ 0.20000000 & 0.80000000 & 0.71428571 & 2.00000000 & 1.00000000 &&&& \\ 0.16666667 & 0.71428571 & 1.78571429 & 2.77777778 & 2.50000000 & 1.00000000 &&& \\ 0.14285714 & 0.64285714 & 1.78571429 & 3.33333333 & 4.09090909 & 3.00000000 & 1.00000000 && \\ 0.12500000 & 0.58333333 & 1.75000000 & 3.71212121 & 5.56818182 & 5.65384615 & 3.50000000 & 1.00000000 & \\ 0.11111111 & 0.53333333 & 1.69696970 & 3.95959596 & 6.85314685 & 8.61538462 & 7.46666667 & 3.99999993 & 1.00000000 \end{pmatrix}$$

$$U =$$

$$\begin{pmatrix} 1.00000000 & 0.50000000 & 0.33333333 & 0.25000000 & 0.20000000 & 0.16666667 & 0.14285714 & 0.12500000 & 0.11111111 \\ & 0.08333333 & 0.08333333 & 0.07500000 & 0.06666667 & 0.05952381 & 0.05357143 & 0.04861111 & 0.04444444 \\ && 0.00555556 & 0.00833333 & 0.00952381 & 0.00992063 & 0.00992063 & 0.00972222 & 0.00942761 \\ &&& 0.00035714 & 0.00071429 & 0.00099206 & 0.00119048 & 0.00132576 & 0.00141414 \\ &&&& 0.00002268 & 0.00005669 & 0.00009276 & 0.00012626 & 0.00015540 \\ &&&&& 0.00000143 & 0.00000429 & 0.00000809 & 0.00001233 \\ &&&&&& 0.00000009 & 0.00000032 & 0.00000067 \\ &&&&&&& 0.00000001 & 0.00000002 \\ &&&&&&&& 0.00000000 \end{pmatrix}$$

四，实验结论:

适当地分析并比较三种计算方法，你能得出什么结论或经验教训.

首先比较Jacobi迭代法和Gauss-Siedel迭代法.

可以看到两次计算中, Jacobi法都迭代失败了(我在程序中设定的时迭代次数>1000000次视为迭代失败), 在取$x_i = 0, i = 1, 2, ..., n$时, 可发现Jacobi法不收敛, 因为在调试程序时, 我尝试让其输出每一次迭代后的计算误差$E = \|x^* - x^{(k)}\|_1$, 发现这个误差是在不断地增大.

事实上也可证明迭代矩阵的谱半径$\rho(I - D^{-1}H) > 1$, 因此Jacobi法并不收敛.

然后是Gauss-Siedel迭代法, 两次计算中迭代都是收敛的, 都能迭代成功, 但是由于迭代速度慢, 迭代次数是偏多的, 有超过4000次. 另外,
这也使得在达到停止条件$\|x^{(k+1)}-x^{(k)}\|_1<\varepsilon=10^{-5}$, 误差仍然较大.

但是如果不管这个停止条件, 例如我强制让其迭代1000000次, 得到的迭代结果确实可以达到高精度, $n=6$时可以达到$E<10^{-16}$(双精度浮点数无法表示).

再看Doolittle LU分解法, 显然这种方法在n比较小的时候(n值一般小于10), 是计算量最小的, 而且误差也很小. 但是在这种病态线性方程组问题中, 该方法表现出了不足: 随着n的增大, 计算误差也会增大.

总的来说, 在n比较小且没有出现稀疏矩阵时, Doolittle LU分解法更好, 但是随着n的增大, 计算误差也会增大. 而Gauss-Siedel迭代法计算量大, 尤其是在求解病态方程组时, 迭代速度明显放慢, 计算量更大, 但是可以通过增大迭代次数, 得到很精确的结果.

而且在多数情况下, Gauss-Siedel迭代法比Jacobi迭代法更容易实现收敛.

附: 实验程序源代码:

```
#include <stdio.h>
#include <math.h>
double norm_1(double array[],int n);
double error(double array[], int n);
int main() {
        //The number of unknowns is set at most 12, so the order of matrix is not greater than 12.
        double h[13][13] = {0}, x[13] = {0}, x1[13] = {0}, y[13] = {0}, y1[13] = {0}, b[13] = {0}, l[13][13] = {0}, u[13][13] = {0}, err[13]={0};
        int i, j, k, n;
        printf("Please input the value of n (n<=12): \n");
        scanf("%d",&n);

        //Doolittle's Method is as follows
        for( i = 1; i <= n ; i ++) {
                for( j = 1; j <= n ; j ++) {
                        h[i][j] = 1.0 / ( i + j - 1);
                        b[i] = b[i] + h[i][j];
                }
        }
        printf("Vector b is as follows:\n");
        for( i = 1; i <= n ; i ++) {
                printf("%.8lf\n", b[i]);
        }
        printf("Matrix H is as follows\n");
        for( i = 1; i <= n ; i ++) {
                for( j = 1; j <= n ; j ++) {
                        printf("%.8lf\t", h[i][j]);
```

```
            }
            printf("\n");
    }


    for ( i = 1; i <= n; i++) {
            l[i][i] = 1;
            for (j = i ; j <= n ;j++) {
                    u[i][j] = h[i][j];
                    if ( i != 1 ) {

                            for ( k = 1; k < i; k++) {
                                    u[i][j] = u[i][j] - l[i][k]
* u[k][j];
                            }
                    }

            }//Here it is aimed to get u in each row
            for( j = i+1; j <= n; j++) {
                    l[j][i] = h[j][i];
                    if ( i != 1) {
                            for ( k = 1; k < i; k++) {
                                    l[j][i] = l[j][i] - l[j][k]
* u[k][i];
                            }
                    }
                    l[j][i] = l[j][i] / u[i][i];
            }//Here it is aimed to get l in each column
    }
    //It gives matrix L and U
    printf("The result of Doolittle's Method is as follows\n");
    printf("L = \n");
    for( i = 1; i <= n; i++) {
            for( j = 1 ; j <= n ; j++) {
                    printf("%.8f\t", l[i][j]);
            }
            printf("\n");
    }
    printf("U = \n");
  for( i = 1; i <= n; i++) {
    for( j = 1 ; j <= n ; j++) {
        printf("%.8f\t", u[i][j]);
    }
    printf("\n");
  }
    for(i=1;i<=n;i++) {
```

```
            y[i] = b[i];
            for(j=1;j<i;j++) {
                    y[i] = y[i] - l[i][j] * y[j];
            }
    }//Get vector y which satisfies Ly=b and Ux=y
    for(i=n;i>=1;i-- ) {
            x[i] = y[i];
            for(j=i+1;j<=n;j++) {
                    x[i] = x[i] - u[i][j] * x[j];
            }
            x[i] = x[i] / u[i][i];
    } //Get vector x

    for(i=1;i<=n;i++) {
            printf("x[%d] = %.8lf\n", i, x[i]);
    }
    printf("The error is %.8lf\n", error(x, n));

    //Jacobi iteration is as follows
    k = 0;
    for(i=1;i<=n;i++) {
            err[i] =1;
            x1[i] = 0;
            x[i] = 0;
    }
    while( norm_1(err,n)>=0.00001&&k<1000000) {
            for(i=1;i<=n;i++){

                    for(j=1;j<=n;j++) {
                            if(j == i){
                                    continue;
                            }
                            x1[i] = x1[i] - h[i][j] / h[i][i]
*x[j];
                    }
                    x1[i] = x1[i] + b[i] / h[i][i];

            }

            for(i=1;i<=n;i++) {
                    err[i] = x1[i] - x[i];
                    x[i] = x1[i];
                    x1[i] = 0;
            }
            k++;
    }
```

```
	if(k>=1000000) {
		printf("Jacobi's iteration failed!\n");
	}
	else {
		printf("The result of Jacobi's iteration is as
follows:\n");
		for(i=1;i<=n;i++) {
			printf("x[%d] = %.8lf\n", i, x[i]);

		}
		printf("There are %d steps.\nThe error is %.8lf",
k, error(x,n));
	}



	//G-S iteration is as follows
	k = 0;
	for(i=1;i<=n;i++) {
		err[i] =1;
		x1[i] = 0;
		x[i] = 0;
	}
	while( norm_1(err,n)>=0.00001&& k<1000000) {
		for(i=1;i<=n;i++){

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

				x1[i] = x1[i] - h[i][j] / h[i][i]
*x1[j];
			}
			for(j=i+1;j<=n;j++) {

				x1[i] = x1[i] - h[i][j] / h[i][i]
*x[j];
			}
			x1[i] = x1[i] + b[i] / h[i][i];

		}
		/*
		for(i=1;i<=n;i++) {
			printf("\nx[%d] = %lf\n k = %d", i,
x1[i],k);
			}
		*/
		for(i=1;i<=n;i++) {
```

```
                        err[i] = x1[i] - x[i];
                        x[i] = x1[i];
                        x1[i] = 0;
                }
                k++;
        }
        if(k>=1000000) {
                printf("G-S iteration failed!\n");
        }
        else {
                printf("The result of G-S iteration is as
follows:\n");
                for(i=1;i<=n;i++) {
                        printf("x[%d] = %.8lf\n", i, x[i]);

                }
                printf("There are %d steps.\nThe error is %.8lf",
k, error(x,n));
        }


        return 0;

}

double norm_1(double array[],int n) {
        double sum;
        int i;
        sum = 0;
        for(i=0;i<n;i++) {
                sum = sum + fabs(array[i+1]);
        }
        //printf("the error is %lf\n", sum);
        return sum;

}
double error(double array[], int n) {
        double sum, err1[13];
        int i;
        for(i=1;i<=n; i++) {
                err1[i] = array[i] - 1.0;
        }
        sum = norm_1(err1, n);
        return sum;
}
```