Lab 1 实验报告

PB20000296 郑滕飞

高斯消元：

1、上/下三角阵求解与高斯消元

```
void forward_subs(vector<vector<double>>& L, vector<double>& b)
{
    int n = (int)L.size();
    for (int j = 0; j < n; j++) {
        b[j] /= L[j][j];
        for (int k = j + 1; k < n; k++) b[k] -= b[j] * L[k][j];
    }
}
```

上下三角阵可直接通过除以主对角线元素后对应相减来求解，而主对角线元素为 1 时直接删去除以的过程即可，图示为针对上三角矩阵，针对下三角可类似实现。

高斯消元直接仿照书上实现即可：

```
int gauss_elim(vector<vector<double>>& A)
{
    int n = (int)A.size();
    for (int j = 0; j < n - 1; j++) {
        if (A[j][j] == 0) return -1;
        for (int k = j + 1; k < n; k++) A[k][j] /= A[j][j];
        for (int k = j + 1; k < n; k++)
            for (int i = j + 1; i < n; i++)
                A[k][i] -= A[k][j] * A[j][i];
    }
    return 0;
}
```

对全主元/列主元，可先找到最大值后再记录，如全主元寻找、交换部分的代码：

```
    int p = -1, q = -1;
    for (int k = j; k < n; k++)
        for (int i = j; i < n; i++)
            if (abs(A[k][i]) > max) {
                max = abs(A[k][i]);
                p = k;
                q = i;
            }
    if (p == -1) return;
    u[j] = p;
    v[j] = q;
```

此处先将 p，q 置为-1，这样若剩下矩阵都为 0 则直接返回。值得一提的是，由于三种高斯消元可能出现异常返回，用 int 类型事实上比 void 类型更加合适。

主元消去法需要进行一些置换矩阵的乘法，也即相当于交换 b 中的对应分量，具体代码如下：

```
void vector_pb(vector<int>& u, vector<double>& b)
{
    int n = (int)u.size();
    for (int i = 0; i < n; i++)
        swap(b[i], b[u[i]]);
}

void vector_qb(vector<int>& v, vector<double>& b)
{
    int n = (int)v.size();
    for (int i = n - 1; i >= 0; i--)
        swap(b[i], b[v[i]]);
}
```

由于构造过程，交换 q 的过程与交换 p 是相反的，首次写代码时此处出错过，后来重新仔细看书才意识到问题。

2、其他处理与计时方式

```
    int N = 84;
    vector<vector<double>> A0(N, vector<double>(N));
    vector<double> b0(N);
    vector<int> u(N - 1), v(N - 1);

    for (int i = 0; i < N - 1; i++)
    {
        A0[i][i] = 6;
        A0[i + 1][i] = 8;
        A0[i][i + 1] = 1;
        b0[i] = 15;
    }
    A0[N - 1][N - 1] = 6;
    b0[0] = 7;
    b0[N - 1] = 14;
```

此处为第一题构造矩阵的处理。由于构造的矩阵需要在不同方法下多次使用，而方法对矩阵具有破坏性，因此必须复制。

尝试可发现，对 vector 类型直接使用=即可达成复制，因此先初始化出 A0，b0，再每次用 A，b 复制即可。

```
vector<vector<double>> A = A0;
vector<double> b = b0;

start = GetTickCount64();
gauss_elim(A);
forward_subs1(A, b);
back_subs(A, b);
ende = GetTickCount64();
cout << "no pivoting result:" << endl;
for (int i = 0; i < N; i++) {
    cout << b[i] << ' ';
    if (i % 10 == 9) cout << endl;
}
cout << endl;
cout << "time: " << ende - start << "ms" << endl << endl;
```

关于计时，通过 GetTickCount 进行开始到结束部分的计时，打印与复制等不在计时范围内，从而达到测试算法时间的效果。

又如，全主元消去法的测试如下：

```
A = A0;
b = b0;

start = GetTickCount64();
gauss_elim_full_pivoting(A, u, v);
vector_pb(u, b);
forward_subs1(A, b);
back_subs(A, b);
vector_qb(v, b);
ende = GetTickCount64();
cout << "full pivoting result:" << endl;
for (int i = 0; i < N; i++) {
    cout << b[i] << ' ';
    if (i % 10 == 9) cout << endl;
}
cout << endl;
cout << "time: " << ende - start << "ms" << endl << endl;
```

3、题 1 结果展示

按照题目的要求构造 84 阶矩阵时，结果如下。

```
exercise 1:

no pivoting result:
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 0.999998 1 0.999992 1.00002 0.999969 1.00006
0.999878 1.00024 0.999512 1.00098 0.998047 1.00391 0.992188 1.01562 0.968752 1.0625
0.875008 1.24998 0.500031 1.99994 -0.999878 4.99976 -6.99951 16.999 -30.998 64.9961
-126.992 256.984 -510.969 1024.94 -2046.87 4096.74 -8190.47 16383.9 -32764.5 65531
-131055 262097 -524127 1.048e+06 -2.09498e+06 4.18586e+06 -8.35533e+06 1.66451e+07 -3.30281e+07 6.50077e+07
-1.25821e+08 2.34867e+08 -4.02629e+08 5.36838e+08
time: 0ms

col pivoting result:
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 0.999999 1 0.999997
time: 0ms

full pivoting result:
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 0.999999 1 0.999997
time: 15ms
```

可以发现，不选主元的偏差明显，行列主元则较为精确。由于阶数过低，时间性能的差别不明显，此处将阶数调整为 200 后观察时间性能和误差：

```
no pivoting result:
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 0.999998 1 0.999992 1.00002 0.999969 1.00006
0.999878 1.00024 0.999512 1.00098 0.998047 1.00391 0.992188 1.01562 0.968752 1.0625
0.875008 1.24998 0.500031 1.99994 -0.999878 4.99976 -6.99951 16.999 -30.998 64.9961
-126.992 256.984 -510.969 1024.94 -2046.87 4096.75 -8190.5 16384 -32765 65533
-131063 262129 -524255 1.04851e+06 -2.09702e+06 4.19405e+06 -8.38809e+06 1.67762e+07 -3.35524e+07 6.71048e+07
-1.3421e+08 2.68419e+08 -5.36838e+08 1.07368e+09 -2.14735e+09 4.29471e+09 -8.58941e+09 1.71788e+10 -3.43576e+10 6.87153e+10
-1.37431e+11 2.74861e+11 -5.49722e+11 1.09944e+12 -2.19889e+12 4.39778e+12 -8.79556e+12 1.75911e+13 -3.51822e+13 7.03644e+13
-1.40729e+14 2.81458e+14 -5.62916e+14 1.12583e+15 -2.25166e+15 4.50332e+15 -9.00665e+15 1.80133e+16 -3.60266e+16 7.20532e+16
-1.44106e+17 2.88213e+17 -5.76426e+17 1.15285e+18 -2.3057e+18 4.6114e+18 -9.22281e+18 1.84456e+19 -3.68912e+19 7.37825e+19
-1.47565e+20 2.9513e+20 -5.9026e+20 1.18052e+21 -2.36104e+21 4.72208e+21 -9.44416e+21 1.88883e+22 -3.77766e+22 7.55533e+22
-1.51107e+23 3.02213e+23 -6.04426e+23 1.20885e+24 -2.4177e+24 4.83541e+24 -9.67082e+24 1.93416e+25 -3.86833e+25 7.73665e+25
-1.54733e+26 3.09466e+26 -6.18932e+26 1.23786e+27 -2.47573e+27 4.95146e+27 -9.90292e+27 1.98058e+28 -3.96117e+28 7.92233e+28
-1.58447e+29 3.16893e+29 -6.33787e+29 1.26757e+30 -2.53515e+30 5.07029e+30 -1.01406e+31 2.02812e+31 -4.05623e+31 8.11247e+31
-1.62249e+32 3.24499e+32 -6.48997e+32 1.29799e+33 -2.59599e+33 5.19198e+33 -1.0384e+34 2.07679e+34 -4.15358e+34 8.30717e+34
-1.66143e+35 3.32287e+35 -6.64573e+35 1.32915e+36 -2.65829e+36 5.31659e+36 -1.06332e+37 2.12663e+37 -4.25327e+37 8.50654e+37
-1.70131e+38 3.40261e+38 -6.80521e+38 1.36104e+39 -2.72205e+39 5.44402e+39 -1.08877e+40 2.17741e+40 -4.35429e+40 8.70644e+40
-1.74044e+41 3.47747e+41 -6.94134e+41 1.38282e+42 -2.74387e+42 5.40063e+42 -1.04528e+43 1.9512e+43 -3.34491e+43 4.45988e+43

time: 94ms
```

```
col pivoting result:
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 0.999999 1 0.999994 1.00001 0.999978 1.00004 0.99991 1.00018 0.999642
1.00072 0.998568 1.00286 0.994271 1.01146 0.977083 1.04583 0.908333 1.18333 0.633333
1.73333 -0.466667 3.93333 -4.86667 12.7333 -22.4667 47.9333 -92.8667 188.733 -374.467
751.933 -1500.87 3004.73 -6006.47 12015.9 -24028.9 48060.7 -96118.5 192240 -384477
768957 -1.53791e+06 3.07582e+06 -6.15164e+06 1.23033e+07 -2.46066e+07 4.92132e+07 -9.84263e+07 1.96853e+08 -3.93705e+08
7.87411e+08 -1.57482e+09 3.14964e+09 -6.29929e+09 1.25986e+10 -2.51971e+10 5.03943e+10 -1.00789e+11 2.01577e+11 -4.03154e+11
8.06309e+11 -1.61262e+12 3.22523e+12 -6.45047e+12 1.29009e+13 -2.58019e+13 5.16037e+13 -1.03207e+14 2.06415e+14 -4.1283e+14
8.2566e+14 -1.65132e+15 3.30264e+15 -6.60528e+15 1.32106e+16 -2.64211e+16 5.28422e+16 -1.05684e+17 2.11369e+17 -4.22738e+17
8.45476e+17 -1.69095e+18 3.3819e+18 -6.76381e+18 1.35276e+19 -2.70552e+19 5.41104e+19 -1.08221e+20 2.16442e+20 -4.32884e+20
8.65767e+20 -1.73153e+21 3.46307e+21 -6.92614e+21 1.38523e+22 -2.77045e+22 5.54091e+22 -1.10818e+23 2.21636e+23 -4.43273e+23
8.86545e+23 -1.77309e+24 3.54617e+24 -7.09231e+24 1.41845e+25 -2.83686e+25 5.67355e+25 -1.13464e+26 2.269e+26 -4.5369e+26
9.06936e+26 -1.8121e+27 3.61711e+27 -7.20584e+27 1.42982e+28 -2.81425e+28 5.44694e+28 -1.01676e+29 1.74302e+29 -2.32403e+29

time: 94ms
```

```
full pivoting result:
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 0.999999 1 0.999994 1.00001 0.999978 1.00004 0.99991 1.00018 0.999642
1.00072 0.998568 1.00286 0.994271 1.01146 0.977083 1.04583 0.908333 1.18333 0.633333
1.73333 -0.466667 3.93333 -4.86667 12.7333 -22.4667 47.9333 -92.8667 188.733 -374.467
751.933 -1500.87 3004.73 -6006.47 12015.9 -24028.9 48060.7 -96118.5 192240 -384477
768957 -1.53791e+06 3.07582e+06 -6.15164e+06 1.23033e+07 -2.46066e+07 4.92132e+07 -9.84263e+07 1.96853e+08 -3.93705e+08
7.87411e+08 -1.57482e+09 3.14964e+09 -6.29929e+09 1.25986e+10 -2.51971e+10 5.03943e+10 -1.00789e+11 2.01577e+11 -4.03154e+11
8.06309e+11 -1.61262e+12 3.22523e+12 -6.45047e+12 1.29009e+13 -2.58019e+13 5.16037e+13 -1.03207e+14 2.06415e+14 -4.1283e+14
8.2566e+14 -1.65132e+15 3.30264e+15 -6.60528e+15 1.32106e+16 -2.64211e+16 5.28422e+16 -1.05684e+17 2.11369e+17 -4.22738e+17
8.45476e+17 -1.69095e+18 3.3819e+18 -6.76381e+18 1.35276e+19 -2.70552e+19 5.41104e+19 -1.08221e+20 2.16442e+20 -4.32884e+20
8.65767e+20 -1.73153e+21 3.46307e+21 -6.92614e+21 1.38523e+22 -2.77045e+22 5.54091e+22 -1.10818e+23 2.21636e+23 -4.43273e+23
8.86545e+23 -1.77309e+24 3.54617e+24 -7.09231e+24 1.41845e+25 -2.83686e+25 5.67355e+25 -1.13464e+26 2.269e+26 -4.5369e+26
9.06936e+26 -1.8121e+27 3.61711e+27 -7.20584e+27 1.42982e+28 -2.81425e+28 5.44694e+28 -1.01676e+29 1.74302e+29 -2.32403e+29

time: 125ms
```

由于矩阵构造，全主元与列主元的结果一致，均显著优于不选主元。而时间性能上，不选主元和列主元几乎一致(事实上，测试更大规模的数据时，列主元的速度甚至优于不选主元，如下方为 N=500 时三者的输出结果。)

```
time: 1813ms
```

```
time: 1797ms
```

```
time: 1891ms
```

综合以上可发现，一般来说列主元的确是最优的思路。

平方根法：

1、平方根法与改进

仿照书上代码可实现平方根法与改进的平方根法:

```cpp
void cholesky_decomp(vector<vector<double>>& A)
{
    int n = (int)A.size();
    for (int k = 0; k < n; k++) {
        A[k][k] = sqrt(A[k][k]);
        for (int i = k + 1; i < n; i++) A[i][k] /= A[k][k];
        for (int j = k + 1; j < n; j++)
            for (int i = j; i < n; i++)
                A[i][j] -= A[i][k] * A[j][k];
    }
    for (int i = 0; i < n - 1; i++)
        for (int j = i + 1; j < n; j++)
            A[i][j] = A[j][i];
}
```

```cpp
void modified_cholesky_decomp(vector<vector<double>>& A)
{
    int n = (int)A.size();
    vector<double> v(n);
    for (int j = 0; j < n; j++) {
        for (int i = 0; i < j; i++) v[i] = A[j][i] * A[i][i];
        for (int i = 0; i < j; i++) A[j][j] -= A[j][i] * v[i];
        for (int k = j + 1; k < n; k++) {
            for (int i = 0; i < j; i++) A[k][j] -= A[k][i] * v[i];
            A[k][j] /= A[j][j];
        }
    }
}
```

在平方根法的最后，我将 A 的上三角部分存储了 L 的转置，方便之后调用前代、回代进行方程的求解，而改进平方根法的这一步则由单独的函数完成：

```cpp
void matrix_DLT(vector<vector<double>>& A)
{
    int n = (int)A.size();
    for (int j = 0; j < n; j++)
        for (int i = j + 1; i < n; i++)
            A[j][i] = A[i][j] * A[j][j];
}
```

3、题 2 结果展示

对第一问，随机数采用 `rand()%10` 生成了随机一位数，而为了对比结果与精确解的区别，代码中将解出的 x 对应的 Ax 也算了出来，以下为 100 阶时示例的 b 对应的解：

```
b:
4 0 9 1 1 4 5 3 4 3
6 4 1 0 9 6 4 6 7 3
3 0 0 1 7 9 8 7 2 2
3 1 7 9 3 6 4 3 5 0
2 0 2 7 0 5 0 1 5 7
7 2 4 2 7 0 2 2 7 3
9 3 9 6 2 7 1 4 0 0
4 1 1 6 6 0 4 6 0 9
7 0 5 8 6 2 5 9 4 2
7 7 4 1 1 3 4 1 3 4
```

```
cholesky result:
0.413263 -0.132631 0.913042 0.00220996 0.0648584 0.349206 0.44308 0.21999 0.357024 0.209775
0.545231 0.33792 0.0755736 -0.0936558 0.860985 0.48381 0.300912 0.507069 0.628396 0.208969
0.281912 -0.0280901 -0.00101135 0.0382036 0.618976 0.77204 0.660621 0.62175 0.121874 0.159508
0.283048 0.010016 0.616793 0.822056 0.16265 0.551441 0.322936 0.219203 0.485034 -0.0695428
0.210394 -0.0343952 0.133558 0.698812 -0.121678 0.517971 -0.0580333 0.0623618 0.434416 0.593482
0.630769 0.0988307 0.380924 0.0919294 0.699782 -0.0897475 0.197694 0.112812 0.674186 0.145326
0.872555 0.12912 0.836249 0.508389 0.0798621 0.69299 -0.00976523 0.404662 -0.0368548 -0.036114
0.397995 0.0561668 0.0403368 0.540465 0.555015 -0.0906157 0.351142 0.579195 -0.143096 0.851761
0.625481 -0.106573 0.440248 0.704091 0.518843 0.107478 0.406381 0.828714 0.306483 0.106454
0.628981 0.603737 0.333647 0.0597942 0.0684116 0.25609 0.370687 0.0370429 0.258885 0.374112

Ax:
4 2.22045e-16 9 1 1 4 5 3 4 3
6 4 1 -2.22045e-16 9 6 4 6 7 3
3 -6.87384e-17 -6.93889e-18 1 7 9 8 7 2 2
3 1 7 9 3 6 4 3 5 0
2 -2.77556e-17 2 7 -2.22045e-16 5 -5.55112e-17 1 5 7
7 2 4 2 7 -1.11022e-16 2 2 7 3
9 3 9 6 2 7 1 4 -7.63278e-17 -1.11022e-16
4 1 1 6 6 0 4 6 -2.22045e-16 9
7 -3.88578e-16 5 8 6 2 5 9 4 2
7 7 4 1 1 3 4 1 3 4

time: 16ms
```

```
modified cholesky result:
0.413263 -0.132631 0.913042 0.00220996 0.0648584 0.349206 0.44308 0.21999 0.357024 0.209775
0.545231 0.33792 0.0755736 -0.0936558 0.860985 0.48381 0.300912 0.507069 0.628396 0.208969
0.281912 -0.0280901 -0.00101135 0.0382036 0.618976 0.77204 0.660621 0.62175 0.121874 0.159508
0.283048 0.010016 0.616793 0.822056 0.16265 0.551441 0.322936 0.219203 0.485034 -0.0695428
0.210394 -0.0343952 0.133558 0.698812 -0.121678 0.517971 -0.0580333 0.0623618 0.434416 0.593482
0.630769 0.0988307 0.380924 0.0919294 0.699782 -0.0897475 0.197694 0.112812 0.674186 0.145326
0.872555 0.12912 0.836249 0.508389 0.0798621 0.69299 -0.00976523 0.404662 -0.0368548 -0.036114
0.397995 0.0561668 0.0403368 0.540465 0.555015 -0.0906157 0.351142 0.579195 -0.143096 0.851761
0.625481 -0.106573 0.440248 0.704091 0.518843 0.107478 0.406381 0.828714 0.306483 0.106454
0.628981 0.603737 0.333647 0.0597942 0.0684116 0.25609 0.370687 0.0370429 0.258885 0.374112

Ax:
4 1.11022e-16 9 1 1 4 5 3 4 3
6 4 1 0 9 6 4 6 7 3
3 4.14165e-17 0 1 7 9 8 7 2 2
3 1 7 9 3 6 4 3 5 -1.38778e-16
2 -2.77556e-17 2 7 1.11022e-16 5 -5.55112e-17 1 5 7
7 2 4 2 7 -1.66533e-16 2 2 7 3
9 3 9 6 2 7 1 4 -6.93889e-17 0
4 1 1 6 6 5.55112e-17 4 6 -2.22045e-16 9
7 -1.11022e-16 5 8 6 2 5 9 4 2
7 7 4 1 1 3 4 1 3 4

time: 16ms
```

示例情况中，结果与时间都是几乎一致的，并且在考虑 double 误差时精确程度极高。

当阶数变为 1000 时，两者的结果依然十分相近，且也有极高的精确程度(下图为 Ax 的部分，都与对应的 b 几乎相同)：

```
5 6 6 1 -2.22045e-16 6 4 6 -4.44089e-16 8
6 2 2 2 6 -3.33067e-16 8 2 5 2
3 9 8 7 2 3 6 8 6 2
6 6 -5.55112e-17 4 7 9 6 3 7 -2.22045e-16
4 4 9 8 5 2 9 -5.55112e-17 4 8
6 -3.33067e-16 5 8 4 1 7 9 5 1
9 5 3 4 6 3 3 2 9 1
6 7 5 6 3 -2.22045e-16 8 6 8 1.11022e-16
9 4 -5.55112e-17 2 7 7 3 -2.22045e-16 7 4
-1.66533e-16 5 3 6 4 7 4 9 3 6
7 8 -1.11022e-16 9 8 9 3 1 1 1
2 3 8 7 4 -1.11022e-16 7 8 -1.11022e-16 4
8 5 5 5 2 -1.66533e-16 5 2 3 7
```

但是，此时直接平方根消耗的时间显著多于改进的平方根法：

```
time: 6188ms
```

```
time: 3532ms
```

而第二问，二者就表现出了更明显的差别，直接平方根法直接显示了-nan(经测试，当阶数达到 14 时其即会显示-nan，只能在不超过 13 时求解，推测与浮点数快速指数运算的原理有关)，而改进的平方根法虽然数字亦在有限几项后变得非常不准确，但能算出完整的结果。

```
exercise 2_2:

cholesky result:
-nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind)
-nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind)
-nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind)
-nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind)

time: 0ms

modified cholesky result:
1.00003 0.994927 1.20016 -2.25844 27.1908 -104.419 147.985 396.609 -1891.08 2621.03
-804.517 174.334 -2800.91 937.812 3240.17 1201.7 -1723.22 -6837.18 4279.32 -856.212
7574.24 -4346.32 -2330.18 1450.92 -1525.86 411.367 1415.31 -3264.24 5006.02 -1051.14
-4128.5 3052.79 3279.77 -3579.81 -1404.79 -486.237 1820.35 2397.91 -3053.46 792.299

time: 0ms
```

4、题 3 结果展示

将题 1 的程序运用到题 2 后，可以得到两方法的对比。

第一问在 100 的规模结果如下：

```
b:
5 4 3 3 2 1 2 2 5 2
8 6 6 4 6 7 7 1 7 7
8 7 0 8 1 9 3 5 3 0
8 0 4 4 5 6 9 8 7 6
3 4 4 1 1 6 9 0 4 1
2 6 8 2 9 4 7 4 6 8
9 0 1 5 9 5 5 0 2 7
4 5 2 0 2 6 9 6 1 1
3 7 1 8 2 9 0 1 3 0
0 0 8 3 9 1 1 9 4 6

no pivoting result:
0.467083 0.329175 0.24117 0.259127 0.167564 0.0652371 0.180065 0.134111 0.478826 0.0776257
0.744917 0.473205 0.523029 0.296506 0.511913 0.584362 0.64447 -0.0290623 0.646153 0.567528
0.678564 0.64683 -0.146864 0.821811 -0.071241 0.8906 0.165244 0.456957 0.265185 -0.10881
0.82291 -0.120294 0.38003 0.319991 0.420059 0.479421 0.785731 0.663264 0.581627 0.520469
0.213688 0.342655 0.359763 0.059716 0.0430766 0.509518 0.861748 -0.126997 0.408226 0.0447353
0.144421 0.511052 0.745059 0.0383597 0.871344 0.248196 0.646697 0.284831 0.504993 0.665243
0.842582 -0.0910586 0.068004 0.411019 0.821806 0.370919 0.469002 -0.0609387 0.140385 0.657086
0.288755 0.455368 0.157565 -0.0310172 0.152607 0.50495 0.797894 0.516106 0.041044 0.0734542
0.224414 0.682402 -0.0484334 0.801932 0.0291105 0.906962 -0.0987333 0.0803709 0.295024 -0.0306146
0.0111213 -0.0805988 0.794866 0.131935 0.885781 0.0102576 0.0116429 0.873314 0.255221 0.574478

Ax:
5 4 3 3 2 1 2 2 5 2
8 6 6 4 6 7 7 1 7 7
8 7 2.22045e-16 8 1 9 3 5 3 -1.11022e-16
8 -2.22045e-16 4 4 5 6 9 8 7 6
3 4 4 1 1 6 9 0 4 1
2 6 8 2 9 4 7 4 6 8
9 -2.77556e-17 1 5 9 5 5 5.55112e-17 2 7
4 5 2 -2.77556e-17 2 6 9 6 1 1
3 7 1 8 2 9 -1.249e-16 1 3 -3.46945e-17
0 1.11022e-16 8 3 9 1 1 9 4 6

time: 15ms
```

```
col pivoting result:
0.467083 0.329175 0.24117 0.259127 0.167564 0.0652371 0.180065 0.134111 0.478826 0.0776257
0.744917 0.473205 0.523029 0.296506 0.511913 0.584362 0.64447 -0.0290623 0.646153 0.567528
0.678564 0.64683 -0.146864 0.821811 -0.071241 0.8906 0.165244 0.456957 0.265185 -0.10881
0.82291 -0.120294 0.38003 0.319991 0.420059 0.479421 0.785731 0.663264 0.581627 0.520469
0.213688 0.342655 0.359763 0.059716 0.0430766 0.509518 0.861748 -0.126997 0.408226 0.0447353
0.144421 0.511052 0.745059 0.0383597 0.871344 0.248196 0.646697 0.284831 0.504993 0.665243
0.842582 -0.0910586 0.068004 0.411019 0.821806 0.370919 0.469002 -0.0609387 0.140385 0.657086
0.288755 0.455368 0.157565 -0.0310172 0.152607 0.50495 0.797894 0.516106 0.041044 0.0734542
0.224414 0.682402 -0.0484334 0.801932 0.0291105 0.906962 -0.0987333 0.0803709 0.295024 -0.0306146
0.0111213 -0.0805988 0.794866 0.131935 0.885781 0.0102576 0.0116429 0.873314 0.255221 0.574478

Ax:
5 4 3 3 2 1 2 2 5 2
8 6 6 4 6 7 7 1 7 7
8 7 2.22045e-16 8 1 9 3 5 3 -1.11022e-16
8 -2.22045e-16 4 4 5 6 9 8 7 6
3 4 4 1 1 6 9 0 4 1
2 6 8 2 9 4 7 4 6 8
9 -2.77556e-17 1 5 9 5 5 5.55112e-17 2 7
4 5 2 -2.77556e-17 2 6 9 6 1 1
3 7 1 8 2 9 -1.249e-16 1 3 -3.46945e-17
0 1.11022e-16 8 3 9 1 1 9 4 6

time: 16ms
```

```
full pivoting result:
0.467083 0.329175 0.24117 0.259127 0.167564 0.0652371 0.180065 0.134111 0.478826 0.0776257
0.744917 0.473205 0.523029 0.296506 0.511913 0.584362 0.64447 -0.0290623 0.646153 0.567528
0.678564 0.64683 -0.146864 0.821811 -0.071241 0.8906 0.165244 0.456957 0.265185 -0.10881
0.82291 -0.120294 0.38003 0.319991 0.420059 0.479421 0.785731 0.663264 0.581627 0.520469
0.213688 0.342655 0.359763 0.059716 0.0430766 0.509518 0.861748 -0.126997 0.408226 0.0447353
0.144421 0.511052 0.745059 0.0383597 0.871344 0.248196 0.646697 0.284831 0.504993 0.665243
0.842582 -0.0910586 0.068004 0.411019 0.821806 0.370919 0.469002 -0.0609387 0.140385 0.657086
0.288755 0.455368 0.157565 -0.0310172 0.152607 0.50495 0.797894 0.516106 0.041044 0.0734542
0.224414 0.682402 -0.0484334 0.801932 0.0291105 0.906962 -0.0987333 0.0803709 0.295024 -0.0306146
0.0111213 -0.0805988 0.794866 0.131935 0.885781 0.0102576 0.0116429 0.873314 0.255221 0.574478

Ax:
5 4 3 3 2 1 2 2 5 2
8 6 6 4 6 7 7 1 7 7
8 7 1.11022e-16 8 1 9 3 5 3 4.44089e-16
8 1.11022e-16 4 4 5 6 9 8 7 6
3 4 4 1 1 6 9 4.996e-16 4 1
2 6 8 2 9 4 7 4 6 8
9 8.32667e-17 1 5 9 5 5 1.66533e-16 2 7
4 5 2 0 2 6 9 6 1 1
3 7 1 8 2 9 -1.38778e-17 1 3 -3.64292e-17
0 1.11022e-16 8 3 9 1 1 9 4 6

time: 31ms
```

可以发现，三个算法都得到了基本准确的结果，除了全主元消去外与平方根法效率差别不大。但是，`1000` 的规模即出现了明显差别：

```
time: 9969ms
```
```
time: 8719ms
```
```
time: 13375ms
```

不选主元与列主元的速度都在改进平方根法的三倍左右，与理论计算的结果一致。不过，在精度上，三种算法都达到了较高的精度(由于其正定且主对角线比起周围大很多，三种算法的实际结果几乎无区别)。

而第二问的情况如下：

```
no pivoting result:
1 0.999782 1.00951 0.823416 2.7167 -8.59692 32.6272 -56.877 40.7007 48.6039
-118.234 106.696 -74.517 42.1086 15.4826 15.6782 -28.4538 -78.642 99.0434 -61.0471
93.381 -61.9504 -9.04398 91.1563 -65.0043 43.8346 -175.183 38.7058 153.483 -0.694875
-65.6605 51.3813 33.1039 -194.959 75.1504 100.395 -73.5576 60.1729 -52.6182 16.785
```

```
col pivoting result:
0.999998 1.00026 0.989834 1.17152 -0.51087 8.60896 -21.5428 39.5435 -36.2332 31.751
-41.5928 13.7476 65.5399 -15.4594 -82.5762 -26.8622 125.456 -13.6811 -9.97322 -7.33972
-115.648 200.059 -38.9658 -116.495 -112.56 175.276 184.997 -54.0737 -164.972 -228.971
377.105 -22.3954 -83.7536 -57.8397 11.529 123.567 -106.404 87.1689 -82.9998 32.3404
```

```
full pivoting result:
1 0.999962 1.00123 0.984795 1.06783 1.12204 -1.38031 10.3504 -14.0553 4.39969
15.8301 -2.76889 3.04308 -58.8846 63.8396 23.1708 -8.45386 -52.0645 8.79042 18.8575
45.6256 -94.6778 59.6831 -16.438 114.032 -120.982 17.5541 -83.5912 140.61 -22.2632
-74.1716 8.92977 136.012 -81.5255 -22.9813 -25.0029 61.3632 -46.4293 43.0047 -15.6017
```

可以发现，这样的高斯消元比起平方根法虽然笨重，但是稳定程度要高了很多，虽然仍然十分不准确，但偏差的量级更小。而观察也可发现，相对来说，全主元的偏差小于列主元的偏差小于直接消去的偏差，与理论分析结果一致。

总结：

本次实验最大收获：在数学实验与数值代数的夹逼下终于还是下了 vs 并且配上了 Qt、OpenCV 和 Eigen 三个库。

此处的结果分析已经能较清晰看出效率的差别，而在今后利用条件数，则能更准确地估算误差的情况。

Lab 2 实验报告

PB20000296 郑滕飞

估算无穷范数：

1、准备工作

```
void forward_subs(vector<vector<double>> L, vector<double>& b);//前代法

void forward_subs1(vector<vector<double>> L, vector<double>& b);//对角元为1的前代法

void back_subs(vector<vector<double>> U, vector<double>& b);//回代法

void back_subs1(vector<vector<double>> U, vector<double>& y);//对角元为1的回代法

void gauss_elim_col_pivoting(vector<vector<double>>& A, vector<int>& u);//列主元Gauss消去法

void vector_pb(vector<int> u, vector<double>& b);//计算向量Pb

void vector_qb(vector<int> v, vector<double>& b);//计算向量Qb

void matrix_vector_times(vector<vector<double>> A, vector<double>& b);//计算矩阵向量乘法

void transpose(vector<vector<double>>& A); //计算转置

double inf_norm(vector<vector<double>> A); //计算矩阵无穷范数

double inf_norm(vector<double> b); //计算向量无穷范数

double inf_norm(vector<double> b, int& n); //计算向量无穷范数，返回行数

double inverse_inf_norm(vector<vector<double>> A); //估计逆的无穷范数
```

上图即为估计无穷范数所用到的基本函数。在估计逆的无穷范数前，除了上次实验已经实现的高斯消去相关内容外，还需要实现转置、矩阵向量乘法与无穷范数(此处针对矩阵还是向量、是否需要返回行数进行了不同的重载)。这部分的函数都较为简单，如无穷范数部分的实现，直接遍历比较即可：

```
double inf_norm(vector<vector<double>> A) {
    int n = (int)A.size();
    double max;
    double ret = 0;
    for (int i = 0; i < n; i++) {
        max = 0;
        for (int j = 0; j < n; j++)
            max += abs(A[i][j]);
        if (max > ret) ret = max;
    }
    return ret;
}

double inf_norm(vector<double> b) {
    int n = (int)b.size();
    double ret = 0;
    for (int i = 0; i < n; i++)
        if (abs(b[i]) > ret) ret = abs(b[i]);
    return ret;
}

double inf_norm(vector<double> b, int& max_pos) {
    int n = (int)b.size();
    max_pos = 0;
    double ret = 0;
    for (int i = 0; i < n; i++)
        if (abs(b[i]) > ret) {
            ret = abs(b[i]);
            max_pos = i;
        }
    return ret;
}
```

2、主体代码

接下来需要实现的是主体部分的代码，其中的一个核心难点是，在列主元消去后如何得到$A^T x = b$的解。

当已知$PA = LU$时，可发现上面的方程化为$(PA)^T PX = b$，从而解得$x = P^{-1}L^{-T}U^{-T}b$。于是，当A分解为LU后，先作转置，再利用前代与对角元为 1 的回代，最后类似全主元计算Qb的方式处理P^{-1}，即可得到解。由此，计算x, w, z, v的过程为：

```
gauss_elim_col_pivoting(A, u);
vector<vector<double>> LUT = A;
transpose(LUT);
```

```
w = x;
forward_subs(LUT, w);
back_subs1(LUT, w);
vector_qb(u, w);
for (int i = 0; i < n; i++)
    v[i] = (w[i] > 0) ? 1 : -1;
z = v;
vector_pb(u, z);
forward_subs1(A, z);
back_subs(A, z);
```

其余部分的代码利用之前已准备的小函数容易实现，如下图为循环中剩下的部分：

```
temp = 0;
for (int i = 0; i < n; i++)
    temp += z[i] * x[i];
if (inf_norm(z, max_pos) <= temp) {
    for (int i = 0; i < n; i++)
        ret += abs(w[i]);
    return ret;
}
else {
    for (int i = 0; i < n; i++) x[i] = 0;
    x[max_pos] = 1;
}
```

由于 C++的特性，无需利用 flag 标识退出循环，只要识别到结果后直接返回即可。在之前的重载中，max_pos 用于储存最大的位置，这里按 max_pos 的结果操作 x 即可。

3、题 1 结果展示

题 1 的相关代码如下：

```
double Hilbert(int N)
{
    vector<vector<double>> A(N, vector<double>(N));
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++)
            A[i][j] = 1.0 / (i + j + 1);
    return inf_norm(A) * inverse_inf_norm(A);
}
```

通过对 N 进行循环，即可得到各阶希尔伯特矩阵的条件数估计。

输出结果如下：

```
exercise 1:

Hilber matrix of 5 rows: 943656
Hilber matrix of 6 rows: 2.90703e+07
Hilber matrix of 7 rows: 9.85195e+08
Hilber matrix of 8 rows: 3.38728e+10
Hilber matrix of 9 rows: 1.09965e+12
Hilber matrix of 10 rows: 3.53538e+13
Hilber matrix of 11 rows: 1.23062e+15
Hilber matrix of 12 rows: 3.83175e+16
Hilber matrix of 13 rows: 4.63576e+17
Hilber matrix of 14 rows: 1.409e+19
Hilber matrix of 15 rows: 1.10237e+18
Hilber matrix of 16 rows: 1.97409e+18
Hilber matrix of 17 rows: 1.84597e+18
Hilber matrix of 18 rows: 9.70942e+19
Hilber matrix of 19 rows: 3.98036e+19
Hilber matrix of 20 rows: 2.99872e+18
```

可以看到，希尔伯特矩阵的条件数在低阶时即达到了很大的数值，这也是为什么对其相关方程的估算误差极大。

估算解误差：

1、代码实现

在实现了无穷范数的估计后，估算解误差的代码很简单：

```
double estimate_err_col(vector<vector<double>> A, vector<double> b, vector<double>& result)
{
    int n = (int)A.size();
    vector<vector<double>> A0 = A;
    double mu = inf_norm(A);
    double beta = inf_norm(b);
    vector<int> u(n - 1);
    vector<double> x(n);
    gauss_elim_col_pivoting(A0, u);
    x = b;
    vector_pb(u, x);
    forward_subs1(A0, x);
    back_subs(A0, x);
    result = x;
    double nu = inverse_inf_norm(A);
    matrix_vector_times(A, x);
    for (int i = 0; i < n; i++) x[i] -= b[i];
    double gamma = inf_norm(x);
    return nu * mu * gamma / beta;
}
```

此处利用 `A0` 存储 `A` 后进行消元，并解出列主元消去法的 `x`，存在引用调用的 `result` 中。另一方面，按照书上的公式计算理论误差的值，并作为返回值。

值得注意的是，由于列主元消去会破坏原矩阵，而求解会破坏原向量，如果之后仍要利用其进行验证(如计算$||A\tilde{x}-b||_\infty$时)，必须先进行保存(这次主要遇到的 bug 就来自于这里)。

2、题 2 结果展示

题 2 的相关代码如下，之后只需要对 N 进行循环即可。

```
void random(int N)
{
    vector<vector<double>> A(N, vector<double>(N));
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++) {
            if (i > j) A[i][j] = -1;
            else if (i == j) A[i][j] = 1;
            else if (j == N - 1) A[i][j] = 1;
        }
    vector<double> x(N), b(N), x_count(N);
    for (int i = 0; i < N; i++)
        x[i] = 1.0/(rand() % 100 + 1);
    b = x;
    matrix_vector_times(A, b);
    double err = estimate_err_col(A, b, x_count);
    for (int i = 0; i < N; i++) x_count[i] -= x[i];
    cout << "rows: " << N << endl;
    cout << "inf norm of difference between x_count and x_real: ";
    cout << inf_norm(x_count) << endl;
    cout << "estimated error: ";
    cout << err << endl << endl;
}
```

这里的随机是在 1 到 100 的倒数中进行随机，这是由于起初利用 rand 生成整数时，发现估计误差与实际误差均为 0，没有意义，因此改为了小数，部分输出结果如下：

```
rows: 27
inf norm of difference between x_count and x_real: 1.48753e-11
estimated error: 8.29049e-10

rows: 28
inf norm of difference between x_count and x_real: 1.43526e-11
estimated error: 3.81142e-10

rows: 29
inf norm of difference between x_count and x_real: 3.36861e-10
estimated error: 1.10754e-08

rows: 30
inf norm of difference between x_count and x_real: 4.50956e-10
estimated error: 1.09168e-08
```

从结果中可以发现，估算的误差比实际的误差要高出一到两个量级，并且在阶数升高时偏差更大。因此，其最好只作为理论界使用。

```
rows: 59
inf norm of difference between x_count and x_real: 0.0344828
estimated error: 1.67048

rows: 60
inf norm of difference between x_count and x_real: 0.166667
estimated error: 6.69611
```

当阶数进一步升高时，偏差的比例没有变化太多，但估计与实际的数量差别更加明显(奇怪的观察：估算误差与实际误差的比例与阶数的数值接近)。

总结：

使用矩阵/向量的值前一定注意有没有被破坏过!
使用矩阵/向量的值前一定注意有没有被破坏过!
使用矩阵/向量的值前一定注意有没有被破坏过!

Lab 3 实验报告

PB20000296 郑滕飞

代码实现：

1、QR 分解

此处针对书上的代码做了两处优化：

首先，在书上的做法中，需要储存 x 与 v 两个向量，而实际应用时完全可以就地进行存储，直接对 A[j : m][j]进行操作。此外，在题目中所涉及的数据范围，无需利用无穷范数防止向上溢出，由此代码中计算 Householder 变换的部分是这么实现的：

```
if (j == m - 1) break;
/*
double inf = 0;
for (int i = j; i < m; i++)
    if (abs(A[i][j]) > inf) inf = abs(A[i][j]);
if (inf < 1e-10) {
    d[j] = 0;
    continue;
}
for (int i = j; i < m; i++) A[i][j] /= inf;
*/
double sum2 = 0;
for (int i = j + 1; i < m; i++) sum2 += A[i][j] * A[i][j];
if (sum2 < 1e-10) {
    d[j] = 0;
    continue;
}
double norm2 = sqrt(sum2 + A[j][j] * A[j][j]);
if (A[j][j] <= 0) v1 = A[j][j] - norm2;
else v1 = -sum2 / (A[j][j] + norm2);
d[j] = 2 * v1 * v1 / (sum2 + v1 * v1);
for (int i = j + 1; i < m; i++) A[i][j] /= v1;
A[j][j] = norm2;
/*
A[j][j] *= inf;
*/
```

其中，如果需要用无穷范数进行归一化，将注释的部分加上即可。当判断出 d[j]为 0 时，无论 v 为何都不会影响计算，因此直接进行下一次循环。由于过程中破坏了 A[j][j]的值，这里需要补上，利用正交变换不改变二范数，A[j][j]的值应为其下方向量二范数的结果，即书上算法中的 α。

其次，由于这里已经完成了第 j 列的计算，之后的过程直接从第 j+1 列开始即可：

```
for (int i = j; i < m; i++) for (int k = j + 1; k < n; k++) {
    temp[i][k] = 0;
    for (int t = j; t < m; t++)
        temp[i][k] += d[j] * (i == j ? 1 : A[i][j]) * (t == j ? 1 : A[t][j]) * A[t][k];
}
for (int i = j; i < m; i++) for (int k = j + 1; k < n; k++)
    A[i][k] -= temp[i][k];
```

由于没有单独存储 v，作乘法时需要注意 v[1]对应的 A[j][j]并不是取其中的值，而

是直接取 1。

在 j 循环结束之后，还有一个重要的步骤：这样得到的上三角阵在 m<=n 时对角线最后一个元素未必非负，可能需要利用 Householder 方阵 diag(1,…,1,-1)进行最后操作，也即：

```
if (m <= n && A[m - 1][m - 1] < 0) {
    A[m - 1][m - 1] = -A[m - 1][m - 1];
    d[m - 1] = 2;
}
```

2、其他操作

为了从 QR 分解得到最小二乘结果，还需要计算 Qb 并解上三角方程。注意到，储存了 d 与 v 后，计算 Qb 事实上即为将这些 Householder 方阵用相同的顺序左乘 b，因此过程为：

```
void QTb(vector<vector<double>> A, vector<double> d, vector<double>& b) {
    int m = (int)A.size(), n = (int)A[0].size();
    vector<double> temp = b;
    for (int j = 0; j < n; j++) {
        for (int i = j; i < m; i++){
            temp[i] = 0;
            for (int t = j; t < m; t++)
                temp[i] += d[j] * (i == j ? 1 : A[i][j]) * (t == j ? 1 : A[t][j]) * b[t];
        }
        for (int i = j; i < m; i++) b[i] -= temp[i];
    }
}
```

为了得到结果，还需要将回代法作出改进：

```
void back_subs(vector<vector<double>> U, vector<double>& y)
{
    int n = (int)U[0].size();
    for (int j = n - 1; j >= 0; j--) {
        y[j] /= U[j][j];
        for (int k = j - 1; k >= 0; k--) y[k] -= y[j] * U[k][j];
    }
}
```

这样，即使对不是方阵的 U，也可以只取其右上三角的部分进行回代。而这个过程中，也自然忽略了 b 超过 n 的分量。

将这些方法整合，得到最终结果：

```
void min_square(vector<vector<double>>& A, vector<double>& b) {
    int m = (int)A.size(), n = (int)A[0].size();
    vector<double> d(min(m, n));
    houseQR(A, d);
    QTb(A, d, b);
    back_subs(A, b);
}
```

在操作后，m 维向量 b 的前 n 个位置也就存储着最后的结果。

结果展示：

1、题 1

```
excercise 1_1:

result:
-nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind)
-nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind)
-nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind)
-nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind)
-nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind)
-nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind)
-nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind)
-nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind) -nan(ind)
-nan(ind) -nan(ind) -inf inf
time: 609ms
```

```
excercise 1_2:

b:
6 0 5 9 4 4 7 9 3 9
9 9 3 6 3 1 5 1 9 4
7 0 1 4 0 6 8 2 9 5
6 4 4 2 8 1 8 7 6 0
6 6 7 3 3 3 8 7 8 9
1 0 0 5 4 0 4 0 5 6
2 1 4 2 4 6 6 2 4 8
8 6 5 2 5 4 3 0 2 5
6 8 1 2 3 2 6 9 9 4
9 4 9 4 5 7 6 0 7 9

result:
0.610379 -0.103789 0.427513 0.828664 0.285852 0.312821 0.585937 0.827811 0.135956 0.81263
0.737745 0.80992 0.163053 0.559547 0.241472 0.0257294 0.501234 -0.0380666 0.879432 0.243744
0.683129 -0.0750368 0.0672383 0.402654 -0.0937781 0.535127 0.742512 0.0397535 0.859953 0.360719
0.532862 0.310663 0.360509 0.084246 0.797031 -0.0545512 0.748482 0.569732 0.5542 -0.111731
0.563113 0.480605 0.630835 0.211045 0.258719 0.201761 0.723667 0.561572 0.660614 0.832292
0.0164637 0.00307059 -0.0471696 0.468626 0.360911 -0.0777397 0.416486 -0.0871156 0.45467 0.540413
0.141195 0.0476386 0.382419 0.128167 0.335915 0.512679 0.537291 0.114415 0.318556 0.700023
0.681212 0.487861 0.440178 0.110358 0.456243 0.327216 0.271602 -0.0432346 0.160744 0.435795
0.48131 0.751102 0.00767253 0.172173 0.270598 0.121848 0.510921 0.768944 0.799638 0.23468
0.853564 0.229678 0.849651 0.273809 0.412256 0.603631 0.551432 -0.117951 0.628076 0.837192

Ax:
6 0 5 9 4 4 7 9 3 9
9 9 3 6 3 1 5 1 9 4
7 0 1 4 0 6 8 2 9 5
6 4 4 2 8 1 8 7 6 0
6 6 7 3 3 3 8 7 8 9
1 0 0 5 4 0 4 0 5 6
2 1 4 2 4 6 6 2 4 8
8 6 5 2 5 4 3 0 2 5
6 8 1 2 3 2 6 9 9 4
9 4 9 4 5 7 6 0 7 9

time: 1157ms
```

```
excercise 1_3:

result:
-0.210223 59.4304 -676.214 3099.89 -6280.67 4871.25 15.6828 0.496093 -1514.47 3.49574
1.414 0.459829 -1.44043 2.83076 707.94 -5.01204 -3.85466 -2.73752 -1.96982 -1.42699
-1.0037 -0.603112 2.01004 -259.257 -86.955 -13.3489 -5.2997 -2.8456 -1.75936 -1.18023
-0.836207 -0.617289 -0.471505 -0.371598 -0.302234 -0.25444 -0.222994 -0.205214 -0.200602 -0.211338
-0.244376 -0.318165 -0.487634 -0.976718 -3.80691 14.8158 -6.27787 -1.24683 513.16 32.7192
15.3314 9.82339 7.27336 5.91707 5.18772 4.87103 4.90667 5.34999 6.43182 8.83595
14.9878 39.4038 -4555.64 4048.15 7.27801 2.28421 1.09283 0.636689 0.417388 0.296312
0.222875 0.175182 0.142575 0.119388 0.102405 0.0897035 0.0800965 0.0728355 0.0674577 0.0637066
0.0614993 0.0609408 0.0623993 0.0666994 0.0756134 0.0932277 0.130501 0.225787 0.609901 -92.2372
87.1556 0.429272 0.148759 0.0788266 0.0508674 0.0368079 0.0286698 0.02348 0.0199254 20.8785

time: 281ms
```

对比之前的五种方法(见第一次实验报告)时间几乎在 15-16ms，QR 分解的时间显著提升，并且精度也并没有明显的提升。

关于第一题的方程组，在 84 阶时无法得到解，经测试最高能在 53 阶时算出解：

```
result:
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1 0.999999
1 0.999997 1.00001 0.999989 1.00002 0.999954 1.00009 0.999817 1.00037 0.999268
1.00146 0.997071 1.00586 0.988293 1.02339 0.953308 1.09302 0.81543 1.36328 0.296875
2.3125 -1.25 4
time: 109ms
```

可以发现，误差积累的速度是非常快的，后方项的偏差巨大。

分析其原因，这里的三个方程组都极为特殊。前两个是整元素三对角阵，无论是 LU 还是 LLT、LDLT 的形式都十分简单，而 QR 多次矩阵乘法会涉及多次开根，导致精度无优势。而希尔伯特方程的条件数过大，亦无法得到好的结果。

2、题 2-3

题 2 的生成过程如下：

```
cout << "excercise 2:" << endl << endl;

vector<double> t = { -1,-0.75,-0.5,0,0.25,0.5,0.75 };
vector<double> y = { 1,0.8125,0.75,1,1.3125,1.75,2.3125 };
vector<vector<double>> A(7, vector<double>(3));
for (int i = 0; i < 7; i++) {
    A[i][0] = 1;
    A[i][1] = t[i]*t[i];
    A[i][2] = t[i];
}
min_square(A, y);
for (int i = 0; i < 3; i++) cout << y[i] << ' ';
cout << endl << endl;
```

此为题 2、3 的最终结果：

```
excercise 2:

1 1 1

excercise 3:

2.07752 0.718888 9.6802 0.153506 13.6796 1.98683 -0.958225 -0.484023 -0.0736469 1.0187 1.44352 2.90279
```

对于最小二乘问题，由于其并非直接求解方程，QR 分解体现出了更强的适用性。

总结：

不要半夜写代码，会 de 不出 bug……

Lab 4 实验报告

PB20000296 郑滕飞

原理分析：

1、PDE G-S 迭代

由于$(D-L)x^{(k+1)}=Ux^{(k)}+b$，按行还原可得

$$b_i=\sum_{j=1}^{i}A_{ij}x_j^{(k+1)}+\sum_{j=i+1}^{n}A_{ij}x_j^{(k)}$$

于是

$$-u_{i-1,j}^{(k+1)}-u_{i,j-1}^{(k+1)}+\left(4+h^2g(ih,jh)\right)u_{i,j}^{(k+1)}-u_{i+1,j}^{(k)}-u_{i,j+1}^{(k)}=h^2f(ih,jh)$$

2、PDE SOR 迭代

由于$(D-\omega\,L)x^{(k+1)}=\left((1-\omega)\mathrm{D}+\omega U\right)x^{(k)}+\omega\mathrm{b}$，按行还原可得

$$b_i=\sum_{j=1}^{i-1}A_{ij}x_j^{(k+1)}+\sum_{j=i+1}^{n}A_{ij}x_j^{(k)}+\frac{1}{\omega}A_{ii}x_i^{(k+1)}+\frac{\omega-1}{\omega}A_{ii}x_i^{(k)}$$

于是

$$-u_{i-1,j}^{(k+1)}-u_{i,j-1}^{(k+1)}+\left(4+h^2g(ih,jh)\right)\left(\frac{1}{\omega}u_{i,j}^{(k+1)}+\frac{\omega-1}{\omega}u_{i,j}^{(k)}\right)-u_{i+1,j}^{(k)}-u_{i,j+1}^{(k)}=h^2\mathrm{f}(ih,jh)$$

代码实现：

1、ODE

由于三种迭代方式都是单步线性定常迭代，主要的差异在构造矩阵、向量的部分，按照书上的思路求逆、运算即可：

Jacobi 迭代较为简单：

```
for (int i = 0; i < n; i++) {
    double temp = A[i][i];
    b[i] /= temp;
    for (int j = 0; j < n; j++) {
        if (i == j) B[i][j] = 0;
        else B[i][j] = -A[i][j] / temp;
    }
}
```

对 G-S 和 SOR，利用了第一章作业题中下三角矩阵求逆的算法：

```
for (int i = 0; i < n; i++) {
    double temp = A[i][i];
    for (int j = 0; j <= i; j++)
        A[i][j] /= temp;
    L[i][i] = 1.0 / temp;
}
for (int j = 0; j < n; j++) {
    for (int i = j + 1; i < n; i++) {
        double temp = A[i][j];
        for (int t = 0; t <= j; t++)
            L[i][t] -= temp * L[j][t];
    }
}
```

值得一提的是，通过适当的顺序，迭代向量可以直接用 b 原地构建(这里 A 的下三角部分存储了逆矩阵)：

```
for (int i = n - 1; i >= 0; i--) {
    b[i] *= A[i][i];
    for (int j = 0; j < i; j++)
        b[i] += A[i][j] * b[j];
}
```

构建完矩阵后，迭代的过程是相似的，以 SOR 为例：

```
gen_sor(A, L, b, omega);
vector<double> x(n), x_next(n), diff(n);
int count = 0;
do {
    x = x_next;
    matrix_vector_times(L, x_next);
    for (int i = 0; i < n; i++)
        x_next[i] += b[i];
    count++;
    if (count == MAX_ITER) return -1;
} while (max_diff(x, x_next) > ODETOL);
b = x_next;
return count;
```

此处设定了两个终止条件，和前一次的差达到目标值或达到最大迭代次数。正常返回的返回值为迭代次数，而达到最大次数时返回-1，视为不收敛。最终，开始的向量 b 中存储了结果。

构建 ODE 矩阵的过程如下：

```
const int N = b.size() + 1;
const double h = 1.0 / N;
A[0][0] = -2 * epsilon - h;
for (int i = 1; i < N - 1; i++) {
    A[i - 1][i] = epsilon + h;
    A[i][i - 1] = epsilon;
    A[i][i] = -2 * epsilon - h;
    b[i - 1] = h * h / 2;
}
b[N - 2] = h * h / 2 - epsilon - h;
```

由于边界条件一个是 0，一个是 1，相当于 y[0]=0,y[n]=1，由此构建出正确的 b。由于可以直接知道精确值的结果，可以计算出后对比：

```
for (int i = 1; i < N; i++)
    acc[i - 1] = (1 - exp(-i * h / epsilon)) / (2 * (1 - exp(-1 / epsilon))) + i * h / 2;
cout << "jacobi  \t";
construct_matrix(epsilon, A, b);
start = GetTickCount64();
cout << "iter: " << jacobi(A, b) << '\t';
ende = GetTickCount64();
cout << "time: " << ende - start << "ms" << '\t';
cout << "error: " << max_diff(b, acc) << endl;
```

2、PDE

具体的计算方式已经在第一部分说明过，接下来需要构造迭代。为了方便过程中的下标转换与边界的控制，我引入了一些宏定义：

```
#define BORDER 1.0
#define MAX_ITER 20000

//带越界检测的获取元素，填充与获取都转化为下标1开始
#define GET(U, i, j) ((i > 0 && i <= U.size() && j > 0 && j <= U.size()) ? U[i-1][j-1] : BORDER)
#define PUT(d, U, i, j) (U[i-1][j-1] = d)
```

首先，此处的 `i` 与 `j` 为坐标`(ih,jh)`的位置，对应到矩阵减 `1` 的下标，且 `GET` 方法在位于边界后直接返回边界值。

具体的迭代过程，由于不管哪种迭代方式，确定新的 `u[i][j]`最多会用到新的 `u[i][j-1]`与 `u[i-1][j]`，只要自左向右、自上至下计算新的值即可。以 `Jacobi` 与 `SOR` 迭代为例(此处 `Nm1` 为 `N` 的值减 `1`)：

```
int count = 0;
do {
    U = U_next;
    for (int i = 1; i <= Nm1; i++) for (int j = 1; j <= Nm1; j++) {
        double temp = GET(U, i - 1, j) + GET(U, i, j - 1) + GET(U, i + 1, j) + GET(U, i, j + 1);
        temp += h * h * f(i * h, j * h);
        temp /= 4 + h * h * g(i * h, j * h);
        PUT(temp, U_next, i, j);
    }
    count++;
    if (count == MAX_ITER) return -1;
} while (max_diff(U, U_next) > PDETOL);
U = U_next;
return count;
```

```
int count = 0;
do {
    U = U_next;
    for (int i = 1; i <= Nm1; i++) for (int j = 1; j <= Nm1; j++) {
        double temp = GET(U_next, i - 1, j) + GET(U_next, i, j - 1) + GET(U, i + 1, j) + GET(U, i, j + 1)
        temp += h * h * f(i * h, j * h);
        temp *= omega / (4 + h * h * g(i * h, j * h));
        temp -= (omega - 1) * GET(U, i, j);
        PUT(temp, U_next, i, j);
    }
    count++;
    if (count == MAX_ITER) return -1;
} while (max_diff(U, U_next) > PDETOL);
U = U_next;
return count;
```

结果输出方式如下：

```
cout << "sor[" << omega << "]\t";
start = GetTickCount64();
cout << "iter: " << sor_PDE(U, f, g, omega) << '\t';
ende = GetTickCount64();
cout << "time: " << ende - start << "ms" << endl;
cout << endl;

cout << '{';
for (int i = 1; i < N; i++) for (int j = 1; j < N; j++) {
    cout << '{' << double(i)/N << ',' << double(j) / N << ',' << U[i-1][j-1] << "},";
}
cout << "\b}" << endl;
```

除了一般输出的时间与迭代次数外，还可以选择将结果的坐标输出为 `Mathematica` 列表的格式，方便绘制散点图。

结果展示：

1、题 1

输出如下:

```
exercise 1:

epsilon: 1
jacobi          iter: 13172     time: 3593ms    error: 0.00124648
gs              iter: 6574      time: 1891ms    error: 0.00128724
sor[1.2]        iter: 4659      time: 1266ms    error: 0.000957347
sor[1.5]        iter: 2570      time: 687ms     error: 0.000627502
sor[1.8]        iter: 984       time: 266ms     error: 0.000406372

epsilon: 0.1
jacobi          iter: 5926      time: 1516ms    error: 0.00906186
gs              iter: 2981      time: 781ms     error: 0.00908583
sor[1.2]        iter: 2083      time: 563ms     error: 0.00899946
sor[1.5]        iter: 1130      time: 312ms     error: 0.00891191
sor[1.8]        iter: 422       time: 110ms     error: 0.00885277

epsilon: 0.01
jacobi          iter: 569       time: 172ms     error: 0.0660664
gs              iter: 333       time: 78ms      error: 0.0660671
sor[1.2]        iter: 236       time: 78ms      error: 0.0660643
sor[1.5]        iter: 101       time: 47ms      error: 0.0660603
sor[1.8]        iter: 250       time: 78ms      error: 0.0660614

epsilon: 0.0001
jacobi          iter: 118       time: 31ms      error: 0.00495075
gs              iter: 109       time: 31ms      error: 0.00495052
sor[1.2]        iter: -1        time: 5282ms    error: 0.995952
sor[1.5]        iter: -1        time: 5421ms    error: 0.997463
sor[1.8]        iter: -1        time: 5141ms    error: 0.998982
```

可以发现，在 epsilon 越小时，由于矩阵变得更加接近上三角阵，Jacobi 与 G-S 迭代的迭代矩阵就更加接近幂零，从而收敛更快。总体来说，G-S 迭代的性能显著优于 Jacobi 迭代。而关于误差，则是经历了先增大后减小的过程。

对 SOR 迭代，当 epsilon 减小时，可以发现最优松弛因子随之减小，而对理论极限情况，当 epsilon 为 0 时，任何松弛因子大于 1 的迭代都无法收敛。

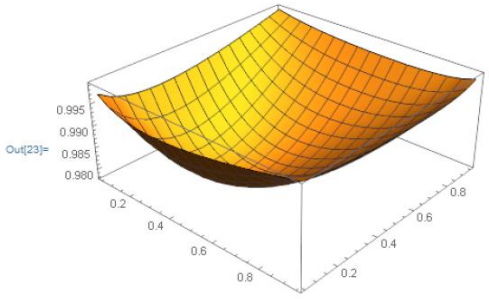
2、题 2

输出如下，输出中的 SOR 后的数为精确到小数点后两位的最优松弛因子。当阶数变高时，最优松弛因子增大，符合理论结果。

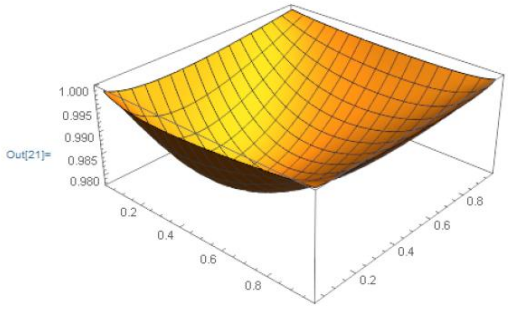
```
exercise 2:

N = 20
jacobi          iter: 930       time: 78ms
gs              iter: 492       time: 47ms
sor[1.73]       iter: 62        time: 16ms

N = 40
jacobi          iter: 3305      time: 984ms
gs              iter: 1759      time: 532ms
sor[1.85]       iter: 122       time: 31ms

N = 60
jacobi          iter: 6884      time: 3906ms
gs              iter: 3681      time: 2078ms
sor[1.9]        iter: 177       time: 109ms
```

N = 20、40、60 的迭代解与 Mathematica 计算的数值解解作图如下:

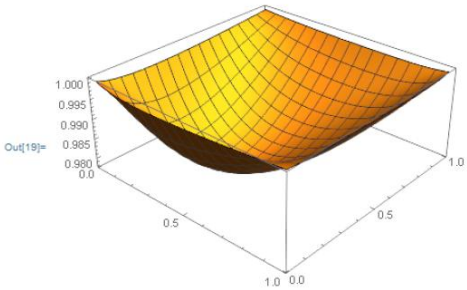
In[23]:= ListPlot3D[A]
点集三维图



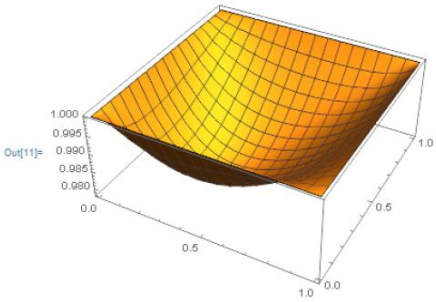
In[21]:= ListPlot3D[A]
点集三维图



In[19]:= ListPlot3D[A]
点集三维图



In[11]:= Plot3D[Evaluate[u[x, y] /. sol], {x, 0, 1}, {y, 0, 1}]
绘… 计算



在整体形状大致相同的情况下，当 N 增大时图像的顶点处形态更加接近真实结果。

Lab 5 实验报告

PB20000296 郑滕飞

代码实现：

1、共轭梯度法

直接仿照书上实现算法即可：

```
int congrade(vector<vector<double>> A, vector<double>& b) {
    int n = (int)A.size();
    vector<double> r = b, p(n), w(n);
    b = w;
    double rho = dot(r, r), alpha, beta, rhot;
    int count = 0;
    do {
        count++;
        if (count == 1)
            p = r;
        else {
            beta = rho / rhot;
            for (int i = 0; i < n; i++)
                p[i] = beta * p[i] + r[i];
        }
        w = p;
        matrix_vector_times(A, w);
        alpha = rho / dot(p, w);
        for (int i = 0; i < n; i++) {
            b[i] += alpha * p[i];
            r[i] -= alpha * w[i];
        }
        rhot = rho;
        rho = dot(r, r);
        if (count == MAX_ITER) return -1;
    } while (abs(alpha) * max_diff(p) > TOL);
    return count;
}
```

此处 dot 和 matrix vector times 是之前实验已写过的矩阵、向量相乘与向量点乘，最终结果在 b 中存储。由于 b 每次更新的量来自 alpha 与 p，p 的无穷范数乘 alpha 的绝对值即为判定终止所需的无穷范数。

[function 部分除这个算法外都是 Lab 4 已经实现的迭代算法]

[由于本实验中的时间较短，计时方案采用 Query Performance 进行微秒精度计时]

2、差分矩阵构造

题 1 中构造差分矩阵的思路为每行考虑。假设此行周围四个元素中没有边界，则它们都是变量，需要在 A 中求解更新。若有边界元素，则不在 A 中，而是将对应的边界值增加到 b 中，以保证边界的恒定。

此外，与上个实验相同，这里也增加了 Mathematica 格式的输出，用于之后画图进行比较。

构造差分矩阵代码如下：

```
for (int i = 0; i < Nm1; i++) {
    for (int j = 0; j < Nm1; j++) {
        b[Nm1 * i + j] = h * h * sin((i + 1) * h * (j + 1) * h) / 4;
        A[Nm1 * i + j][Nm1 * i + j] = 1 + h * h / 4;
        if (j != 0) A[Nm1 * i + j][Nm1 * i + j - 1] = -1.0 / 4;
        else b[Nm1 * i + j] += (i + 1) * (i + 1) * h * h / 4;
        if (j != Nm1 - 1) A[Nm1 * i + j][Nm1 * i + j + 1] = -1.0 / 4;
        else b[Nm1 * i + j] += ((i + 1) * (i + 1) * h * h + 1) / 4;
        if (i != 0) A[Nm1 * i + j][Nm1 * (i - 1) + j] = -1.0 / 4;
        else b[Nm1 * i + j] += (j + 1) * (j + 1) * h * h / 4;
        if (i != Nm1 - 1) A[Nm1 * i + j][Nm1 * (i + 1) + j] = -1.0 / 4;
        else b[Nm1 * i + j] += ((j + 1) * (j + 1) * h * h + 1) / 4;
    }
}
```

结果展示：

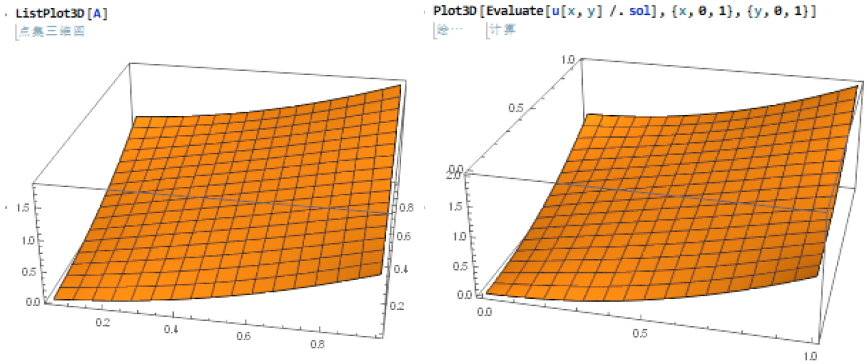
1、题 1

```
10000000 ticks per second
the following timings are all in ticks

exercise 1:

congrade          iter: 55        time: 2033283
sor[1.73]         iter: 63        time: 9661836
```

SOR 迭代找到的最优松弛因子在精确到两位小数意义下为 1.73，具体结果如上。

可以发现，无论是单次迭代的速度还是所需的迭代次数，共轭梯度法都明显优于 SOR，并且，它不需要确定最优的因子，方便操作。

迭代结果与 Mathematica 得到的数值解作图如下：



左侧为迭代解，右侧为软件得到的数值解，可以发现是一致的。

2、题 2

```
exercise 2:

N = 20    iter: 8         time: 4388      error: 0.00133152
N = 40    iter: 12        time: 13927     error: 0.000440042
N = 60    iter: 15        time: 30722     error: 0.000315032
N = 80    iter: 15        time: 46256     error: 0.000511366
```

如图可以发现，对其他算法难以求解的希尔伯特矩阵，共轭梯度法有非常良好的表现。事实上，当 N 的值很大时，共轭梯度法依然可以有较好的结果：

```
N = 500  iter: 23       time: 1762339   error: 0.000579831
N = 800  iter: 22       time: 4019647   error: 0.00104675
N = 1000         iter: 22       time: 5196127   error: 0.00144875
N = 1500         iter: 38       time: 18789495  error: 0.000381725
```

由此可见，共轭梯度法对对称正定阵的计算结果是非常优秀的。

3、题 3

```
exercise 3:

congrade          iter: 6        time: 3689      error: 2.22045e-16
jacobi            iter: 77       time: 8829      error: 4.0728e-07
gs                iter: 43       time: 5757      error: 1.94088e-07
```

比起其他两种迭代算法，共轭梯度法的次数只有六次，并且误差几乎为 0[此题容易得到精确解为{1,-2,3,-2,1}，误差是与精确解比较的]，而其他两种方法的误差都是与迭代结束条件量级相同。这是由于共轭梯度法事实上是直接求解的方法。由于此为五阶方阵，迭代五次后得到精确解，第六次发现没有更新即退出。

Lab 6 实验报告

PB20000296 郑滕飞

代码实现：

1、幂法

按照伪代码容易实现：

```
int power_method(vector<vector<double>>A, vector<double>& u, double& ret) {
    int n = (int)u.size(), count = 0;
    double before;
    ret = 0;
    do {
        count++;
        before = ret;
        matrix_vector_times(A, u);
        ret = 0;
        for (int i = 0; i < n; i++)
            if (abs(u[i]) > abs(ret))
                ret = u[i];
        for (int i = 0; i < n; i++) u[i] /= ret;
    } while (abs(ret - before) > 1e-6 && count < MAX_ITER);
    return count < MAX_ITER ? count : -1;
}
```

此处 ret 记录的是模最大元素的值，比较时再取绝对值即可。

2、就地 Householder 变换与部分矩阵乘法

在实际进行计算时，经常需要操作向量 Householder 变换，或是部分矩阵的乘法，为了方便书写代码，先实现了这两者：

```
double householder(vector<double>& v, int begin, int end) {
    double sigma = 0;
    for (int i = begin + 1; i < end; i++)
        sigma += v[i] * v[i];
    if (sigma < 1e-6) return 0;
    double alpha = sqrt(sigma + v[begin] * v[begin]);
    double v0 = (v[begin] <= 0) ? (v[begin] - alpha) : (-sigma / (v[begin] + alpha));
    for (int i = begin + 1; i < end; i++) v[i] /= v0;
    v[begin] = 1;
    double ret = 2 * v0 * v0 / (sigma + v0 * v0);
}
```

```
int matrix_times(vector<vector<double>>& A, int A_row_begin, int A_row_end, int A_col_begin, int A_col_end,
    vector<vector<double>> B, int B_row_begin, int B_row_end, int B_col_begin, int B_col_end,
    vector<vector<double>> C, int C_row_begin, int C_row_end, int C_col_begin, int C_col_end) {
    if (A_row_begin >= A_row_end || B_row_begin >= B_row_end || C_row_begin >= C_row_end) return -1;
    if (A_col_begin >= A_col_end || B_col_begin >= B_col_end || C_col_begin >= C_col_end) return -2;
    if (A_row_begin < 0 || B_row_begin < 0 || C_row_begin < 0) return -3;
    if (A_col_begin < 0 || B_col_begin < 0 || C_col_begin < 0) return -4;
    if (A_row_end > A.size() || B_row_end > B.size() || C_row_end > C.size()) return -5;
    if (A_col_end > A[0].size() || B_col_end > B[0].size() || C_col_end > C[0].size()) return -6;
    int row = A_row_end - A_row_begin, col = A_col_end - A_col_begin, mid = B_col_end - B_col_begin;
    if (row != B_row_end - B_row_begin) return 1;
    if (col != C_col_end - C_col_begin) return 2;
    if (mid != C_row_end - C_row_begin) return 3;
    double temp;
    for (int i = 0; i < row; i++) for (int j = 0; j < col; j++) {
        temp = 0;
        for (int k = 0; k < mid; k++)
            temp += B[B_row_begin + i][B_col_begin + k] * C[C_row_begin + k][C_col_begin + j];
        A[A_row_begin + i][A_col_begin + j] = temp;
    }
    return 0;
}
```

第一张图以 v 作为输入向量与输出向量的就地 Householder 变换，而 begin 与 end 指示可用的部分；第二张图则表示类似 MATLAB 中

```
A(Arb:Are-1,Acb:Ace-1) = B(Brb:Bre-1,Bcb:Bce-1) . C(Crb:Cre-1,Ccb:Cce-1)
```

形式的矩阵乘法，方便后续计算。

3、上 Hessenberg 化与收敛判定

利用上方两个函数，上 Hessenberg 化容易实现：

```
int upper_hessenberg(vector<vector<double>>& A) {
    int n = (int)A.size();
    vector<vector<double>> temp(n, vector<double>(n));
    vector<double> v(n);
    double beta;
    for (int k = 0; k < n - 2; k++) {
        for (int i = k + 1; i < n; i++) v[i] = A[i][k];
        beta = householder(v, k + 1, n);
        for (int i = k + 1; i < n; i++)
            for (int j = k + 1; j < n; j++)
                temp[i][j] = (i == j ? 1 : 0) - beta * v[i] * v[j];
        if (matrix_times(A, k + 1, n, k, n, temp, k + 1, n, k + 1, n, A, k + 1, n, k, n)) return -1;
        if (matrix_times(A, 0, n, k + 1, n, A, 0, n, k + 1, n, temp, k + 1, n, k + 1, n)) return -2;
    }
    return 0;
}
```

这里 matrix_times 的返回值非 0 即代表调用过程出错，直接中止。

更为复杂的是判定终止条件部分。为了找到最大的 m 和最小的 l，事实上需要从矩阵下方向上看，利用状态机的思路进行判定：

```
int flag = 0;
for (int i = n - 1; i > 0; i--) {
    if (A[i][i - 1] != 0) {
        if (flag == 0) flag = 1;
        else if (flag == 1) {
            m = n - i - 2;
            flag = 2;
        }
    }
    else {
        if (flag == 1) flag = 0;
        else if (flag == 2) {
            l = i;
            break;
        }
    }
}
if (m == n) return 1;
return 0;
```

flag 为 0 代表在 m 的范围内且只碰到对角元，为 1 代表在 m 的范围内且已经碰到了次对角元，再碰到次对角元即结束 m，进入 flag = 2，直到不存在次对角元后设置 l 并退出。由于 m 与 l 都有自始至终不被赋值的可能，需要设定 m 的初值为 n，l 的初值为 0。

4、双重步位移迭代

这部分整体就是仿照书上的伪代码，不过当最后右下角为二阶非上三角阵且有两个实特征值时，Householder 变换是无法处理的，因此起初我加上了对最终情况可以单独使用 Givens 变换的判定：

```
double a = A[end - 2][end - 2],
    b = A[end - 2][end - 1],
    c = A[end - 1][end - 2],
    d = A[end - 1][end - 1],
    delta = (a - d) * (a - d) + 4 * b * c;
if (delta >= 0 && abs(A[end - 2][end - 3]) < 1e-6) {
    t = (d - a + sqrt(delta)) / (2 * b);
    temp[0][0] = temp[1][1] = 1 / sqrt(1 + t * t);
    temp[0][1] = t / sqrt(1 + t * t);
    temp[1][0] = -temp[0][1];
    matrix_times(A, end - 2, end, begin, end, temp, 0, 2, 0, 2, A, end - 2, end, begin, end);
    temp[0][1] = temp[1][0];
    temp[1][0] = -temp[0][1];
    matrix_times(A, begin, end, end - 2, end, A, begin, end, end - 2, end, temp, 0, 2, 0, 2);
    matrix_times(P, begin, end, end - 2, end, P, begin, end, end - 2, end, temp, 0, 2, 0, 2);
}
```

这里的 `delta` 即为判别式，若其对应了实特征值，会在此被处理掉。不过，如果想用这个方法，需要让终止条件也进行对应改变，否则在检测到只有一阶到两阶对角块时仍然会直接终止，因此，后来我选用了更简单的办法。

5、特征值计算

```
int implicit_qr(vector<vector<double>>& A) {
    int n = (int)A.size();
    upper_hessenberg(A);
    int m, l, count = 0;
    while (!judge_and_modify(A, m, l)) {
        count++;
        vector<vector<double>> P = two_step_qr(A, m, l);
        matrix_times(A, 0, l, l, n - m, A, 0, l, l, n - m, P, l, n - m, l, n - m);
        for (int i = l; i < n - m; i++) for (int j = i + 1; j < n - m; j++)
            swap(P[i][j], P[j][i]);
        matrix_times(A, l, n - m, n - m, n, P, l, n - m, l, n - m, A, l, n - m, n - m, n);
        if (count == MAX_ITER) break;
    }
    return count;
}
```

在有了上面这些准备后，计算 Schur 分解的代码就较为简单了，只需要将传出的 P 在对应位置进行矩阵乘法(不过这里对应确实也不那么好想)。而下方即为输出所有特征值的代码：

```
for (int i = 0; i < n; i++) {
    if (i == n - 1 || A[i + 1][i] == 0) cout << A[i][i] << endl;
    else {
        double a = A[i][i], b = A[i][i + 1], c = A[i + 1][i], d = A[i + 1][i + 1];
        double delta = (a - d) * (a - d) + 4 * b * c;
        if (delta < 0) {
            cout << (a + d) / 2 << " + " << sqrt(-delta) / 2 << " I" << endl;
            cout << (a + d) / 2 << " - " << sqrt(-delta) / 2 << " I" << endl;
        }
        else {
            cout << (a + d + sqrt(delta)) / 2 << endl;
            cout << (a + d - sqrt(delta)) / 2 << endl;
        }
        i++;
    }
}
```

在输出中，对二阶对角块计算时，只需要注意判别式正负就能确定是实根还是复根，这就避免了在之前为了最后的次对角元进行复杂的矩阵运算。

结果展示：

1、题 1

```
10000000 ticks per second
the following timings are all in ticks

exercise 1:

iter: 19        time: 4188      result: -3
iter: 79        time: 8308      result: 1.87939
iter: 11        time: 2588      result: -100
```

对于幂法，我取的初值均为最后一个分量为 1，其他为 0 的单位向量。简单尝试可以发现，幂法的收敛速度与初值有很大的关系，而对多项式友方阵，这个初值相对安全，总体也体现出了不错的(十分之一毫秒级)时间性能。

2、题 2.2

```
solving:

iter: 61
eigenvalues:
1.01427 + 0.0809094 I
1.01427 - 0.0809094 I
0.98722 + 0.240331 I
0.98722 - 0.240331 I
0.933657 + 0.392562 I
0.933657 - 0.392562 I
0.855163 + 0.532585 I
0.855163 - 0.532585 I
0.753722 + 0.655393 I
0.753722 - 0.655393 I
0.63234 + 0.753394 I
0.63234 - 0.753394 I
0.507606 + 0.810547 I
0.507606 - 0.810547 I
0.417028 + 0.871033 I
0.417028 - 0.871033 I
0.289893 + 0.946498 I
0.289893 - 0.946498 I
0.139166 + 0.992482 I
```

```
0.139166 - 0.992482 I
-0.0197298 + 1.00935 I
-0.0197298 - 1.00935 I
-0.180153 + 0.997987 I
-0.180153 - 0.997987 I
-0.337067 + 0.959186 I
-0.337067 - 0.959186 I
-0.485239 + 0.894551 I
-0.485239 - 0.894551 I
-0.620783 + 0.805861 I
-0.620783 - 0.805861 I
-0.738999 + 0.695936 I
-0.738999 - 0.695936 I
-0.836861 + 0.56782 I
-0.836861 - 0.56782 I
-0.910481 + 0.425519 I
-0.910481 - 0.425519 I
-0.956335 + 0.273781 I
-0.956335 - 0.273781 I
-0.968102 + 0.120724 I
-0.968102 - 0.120724 I
-0.952627
```

如图即为输出，与 Mathematica 对比可以发现至少精确到小数点后三位部分一致，与设置的精度相符。

3、题 2.3

直接计算：

```
comparing:

iter: 3
eigenvalues:
17.4389
2.87153 + 0.642579 I
2.87153 - 0.642579 I
6.81809

iter: 3
eigenvalues:
17.4755
2.86869 + 0.688311 I
2.86869 - 0.688311 I
6.78713

iter: 3
eigenvalues:
17.5119
2.86611 + 0.731749 I
2.86611 - 0.731749 I
6.75589
```

从结果中可以发现，在 x 从 0.9 到 1.1 的增加中较大的实特征值增加了，较小的减少了，而复特征值则是虚部增加实部减小，计算得模长亦增加。

总结：

写代码时间 3AM-9AM，感觉最阴间的地方在于 MATLAB 格式的伪代码转换到 C 产生的各种各样的冲突，于是稍有不慎就会报个下标溢出……

Lab 7 实验报告

PB20000296 郑滕飞

代码实现：

1、单次 Jacobi 迭代

为了避免繁琐的矩阵乘法，这里只需要关注改变的部分，而容易发现右乘只会改变两列，左乘只会改变两行，因此用 Givens 方阵相似后的处理可以写为：

```
void jacobi_once(vector<vector<double>>& A, vector<vector<double>>& P, int p, int q) {
    int n = (int)A.size();
    double tau = (A[q][q] - A[p][p]) / (2 * A[p][q]);
    double t = - tau + (tau >= 0 ? 1 : -1) * sqrt(tau * tau + 1);
    double c = 1 / sqrt(t * t + 1);
    double s = t * c;
    for (int i = 0; i < n; i++) {
        double t1 = A[i][p] * c - A[i][q] * s;
        double t2 = A[i][p] * s + A[i][q] * c;
        A[i][p] = t1;
        A[i][q] = t2;
        t1 = P[i][p] * c - P[i][q] * s;
        t2 = P[i][p] * s + P[i][q] * c;
        P[i][p] = t1;
        P[i][q] = t2;
    }
    for (int i = 0; i < n; i++) {
        double t1 = A[p][i] * c - A[q][i] * s;
        double t2 = A[p][i] * s + A[q][i] * c;
        A[p][i] = t1;
        A[q][i] = t2;
    }
}
```

此处算出 pq 平面对应的 c 与 s 后，即用对应的矩阵乘法更新了元素值。

2、过关 Jacobi

在单次迭代的基础上，过关 Jacobi 的写法是简单的：

```
int jacobi(vector<vector<double>>& A, vector<vector<double>>& P) {
    int n = (int)A.size();
    double err = 0;
    for (int i = 0; i < n; i++)
        for (int j = i + 1; j < n; j++)
            err += 2 * A[i][j] * A[i][j];
    err = sqrt(err);
    int count = 0;
    while (err > 1e-7) {
        int flag = 0;
        for (int i = 0; i < n; i++) {
            for (int j = i + 1; j < n; j++) {
                if (abs(A[i][j]) > err) {
                    flag = 1;
                    jacobi_once(A, P, i, j);
                }
            }
        }
        if (flag == 0) err /= n;
        count++;
    }
    return count;
}
```

这里引入了 flag 确定这轮中有没有发现不到 err 的，如果没有发现则使 err 下降，直到达到精度要求。此外，迭代次数按照全体扫描的轮数计算。

此外，由于需要输出的元素过多，最终的 A 与 P 的结果是放在代码同目录下的 output 文件夹中的：

```
if (if_write) {
    ofstream outfile;
    outfile.open(filepath, ios::out);
    outfile << "A:" << endl;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++)
            outfile << A[i][j] << '\t';
        outfile << endl;
    }
    outfile << endl;
    outfile << "Q:" << endl;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++)
            outfile << P[i][j] << '\t';
        outfile << endl;
    }
    cout << "file written done" << endl;
}
```

3、二分法

二分法过程的主要步骤是确定变号次数，而这直接由书上的伪代码可写出：

```
int count_change(vector<vector<double>> A, double mu) {
    int n = (int)A.size(), res = 0;
    double q = A[0][0] - mu;
    for (int i = 0; i < n; i++) {
        if (q < 0) res++;
        if (i < n - 1) {
            if (q == 0) q = abs(A[i + 1][i]) * 1e-6;
            q = A[i + 1][i + 1] - mu - A[i + 1][i] * A[i + 1][i] / q;
        }
    }
    return res;
}
```

剩下的二分过程也较为简单：

```
double bisection(vector<vector<double>>A, int m) {
    double r = inf_norm(A), l = -r;
    int count = 0;
    while (r - l > 1e-6) {
        double t = (r + l) / 2;
        if (count_change(A, t) >= m) r = t;
        else l = t;
        count++;
    }
    cout << "bisection iter: " << count << endl;
    return (l + r) / 2;
}
```

4、寻找特定特征值

此处需要结合反幂法，而反幂法又涉及求解方程组，因此我引用了第一章中列主元高斯消去的程序。

完整的过程如下：

```
double find_eigen(vector<vector<double>> A, int m, vector<double>& x) {
    int n = (int)A.size();
    double value = bisection(A, m);
    for (int i = 0; i < n; i++) A[i][i] -= value;
    vector<int> u(n - 1);
    gauss_elim_col_pivoting(A, u);
    int count = 0;
    vector<double> xold(n);
    do {
        xold = x;
        vector_pb(u, x);
        forward_subs1(A, x);
        back_subs(A, x);
        double sum = 0;
        for (int i = 0; i < n; i++) sum += x[i] * x[i];
        sum = sqrt(sum);
        for (int i = 0; i < n; i++) x[i] /= sum;
        count++;
        if (count == 10) break;
    } while (max_diff(xold, x) > 1e-6);
    cout << "inverse power iter: " << count << endl;
    return value;
}
```

由于反幂法的部分涉及一些奇异性，最高迭代次数控制在 10 次，由书上知识与自己测试也能发现，大部分时候两三次迭代就足够了。

结果展示：

1、题 1

各阶时特征值输出如下：

```
10000000 ticks per second
the following timings are all in ticks

exercise 1:

n: 50
iter: 16
eigenvalues:
2.00379 2.01516 2.03405 2.06041 2.09412 2.13506 2.18307 2.23798 2.29957 2.36761
2.44184 2.52198 2.60773 2.69876 2.79473 2.89527 3       3.10852 3.22043 3.33529
3.45267 3.57213 3.69322 3.81546 3.93841 4.06159 4.18454 4.30678 4.42787 4.54733
4.66471 4.77957 4.89148 5       5.10473 5.20527 5.30124 5.39227 5.47802 5.55816
5.63239 5.70043 5.76202 5.81693 5.86494 5.90588 5.93959 5.96595 5.98484 5.99621
file written done
time: 1226825

n: 60
iter: 17
eigenvalues:
2.00265 2.0106  2.02382 2.04229 2.06594 2.09473 2.12857 2.16737 2.21103 2.25943
2.31245 2.36994 2.43176 2.49774 2.5677  2.64145 2.71881 2.79957 2.88351 2.97041
3.06005 3.15217 3.24654 3.34292 3.44103 3.54062 3.64144 3.7432  3.84565 3.9485
4.0515  4.15435 4.2568  4.35856 4.45938 4.55897 4.65708 4.75346 4.84783 4.93995
5.02959 5.11649 5.20043 5.28119 5.35855 5.4323  5.50226 5.56824 5.63006 5.68755
5.74057 5.78897 5.83263 5.87143 5.90527 5.93406 5.95771 5.97618 5.9894  5.99735
file written done
time: 1864890
```

```
n: 70
iter: 17
eigenvalues:
2.00196 2.00783 2.01759 2.03124 2.04875 2.07007 2.09517 2.124   2.1565  2.19261
2.23226 2.27537 2.32186 2.37163 2.42458 2.48063 2.53964 2.60152 2.66613 2.73335
2.80306 2.8751  2.94935 3.02565 3.10387 3.18383 3.2654  3.3484  3.43268 3.51806
3.6044  3.6915  3.77921 3.86735 3.95576 4.04424 4.13265 4.22079 4.3085  4.3956
4.48194 4.56732 4.6516  4.7346  4.81617 4.89613 4.97435 5.05065 5.1249  5.19694
5.26665 5.33387 5.39848 5.46036 5.51937 5.57542 5.62837 5.67814 5.72463 5.76774
5.80739 5.8435  5.876   5.90483 5.92993 5.95125 5.96876 5.98241 5.99217 5.99804
file written done
time: 2902867

n: 80
iter: 19
eigenvalues:
2.0015  2.00601 2.01352 2.02402 2.03749 2.05391 2.07326 2.0955  2.12061 2.14855
2.17927 2.21273 2.24888 2.28767 2.32902 2.3729  2.41921 2.46791 2.51891 2.57214
2.62752 2.68496 2.74438 2.80568 2.86879 2.93359 3       3.06791 3.13723 3.20784
3.27964 3.35253 3.42639 3.50112 3.57659 3.6527  3.72934 3.80638 3.88371 3.96122
4.03878 4.11629 4.19362 4.27066 4.3473  4.42341 4.49888 4.57361 4.64747 4.72036
4.79216 4.86277 4.93209 5       5.06641 5.13121 5.19432 5.25562 5.31504 5.37248
5.42786 5.48109 5.53209 5.58079 5.6271  5.67098 5.71233 5.75112 5.78727 5.82073
5.85145 5.87939 5.9045  5.92674 5.94609 5.96251 5.97598 5.98648 5.99399 5.9985
file written done
time: 4249714
```

```
n: 90
iter: 19
eigenvalues:
2.00119 2.00477 2.01072 2.01904 2.02972 2.04275 2.05812 2.07579 2.09576 2.118
2.14249 2.16918 2.19806 2.22909 2.26222 2.29743 2.33467 2.37389 2.41505 2.45809
2.50298 2.54965 2.59804 2.64811 2.69979 2.75302 2.80774 2.86387 2.92136 2.98013
3.04013 3.10126 3.16346 3.22667 3.29079 3.35576 3.4215  3.48792 3.55496 3.62252
3.69054 3.75893 3.8276  3.89648 3.96548 4.03452 4.10352 4.1724  4.24107 4.30946
4.37748 4.44504 4.51208 4.5785  4.64424 4.70921 4.77333 4.83654 4.89874 4.95987
5.01987 5.07864 5.13613 5.19226 5.24698 5.30021 5.35189 5.40196 5.45035 5.49702
5.54191 5.58495 5.62611 5.66533 5.70257 5.73778 5.77091 5.80194 5.83082 5.85751
5.882   5.90424 5.92421 5.94188 5.95725 5.97028 5.98096 5.98928 5.99523 5.99881
file written done
time: 5488313

n: 100
iter: 20
eigenvalues:
2.00097 2.00387 2.0087  2.01546 2.02414 2.03473 2.04722 2.0616  2.07786 2.09597
2.11593 2.13771 2.16129 2.18665 2.21377 2.24261 2.27316 2.30537 2.33922 2.37468
2.41172 2.45029 2.49035 2.53188 2.57483 2.61916 2.66482 2.71178 2.75998 2.80938
2.85994 2.91159 2.9643  3.01801 3.07267 3.12823 3.18463 3.24182 3.29975 3.35835
3.41757 3.47736 3.53765 3.59839 3.65951 3.72097 3.7827  3.84463 3.90672 3.9689
4.0311  4.09328 4.15537 4.2173  4.27903 4.34049 4.40161 4.46235 4.52264 4.58243
4.64165 4.70025 4.75818 4.81537 4.87177 4.92733 4.98199 5.0357  5.08841 5.14006
5.19062 5.24002 5.28822 5.33518 5.38084 5.42517 5.46812 5.50965 5.54971 5.58828
5.62532 5.66078 5.69463 5.72684 5.75739 5.78623 5.81335 5.83871 5.86229 5.88407
5.90403 5.92214 5.9384  5.95278 5.96527 5.97586 5.98454 5.9913  5.99613 5.99903
file written done
time: 5926616
```

[矩阵在 output 文件夹中]

可以发现，总体来说过关 Jacobi 在少量扫描次数时能保证不错的收敛性，时间性能也较好。

2、题 2

```
bisection iter: 23
inverse power iter: 10
smallest: 0.000967503
time: 210028
vector:
0.0044  0.0087  0.0131  0.0175  0.0218  0.0261  0.0304  0.0347  0.0389  0.0431
0.0472  0.0513  0.0554  0.0594  0.0633  0.0672  0.0710  0.0747  0.0784  0.0820
0.0855  0.0890  0.0923  0.0956  0.0987  0.1018  0.1048  0.1076  0.1104  0.1131
0.1156  0.1181  0.1204  0.1226  0.1247  0.1266  0.1285  0.1302  0.1318  0.1333
0.1346  0.1358  0.1369  0.1379  0.1387  0.1393  0.1399  0.1403  0.1406  0.1407
0.1407  0.1406  0.1403  0.1399  0.1393  0.1387  0.1379  0.1369  0.1358  0.1346
0.1333  0.1318  0.1302  0.1285  0.1266  0.1247  0.1226  0.1204  0.1181  0.1156
0.1131  0.1104  0.1076  0.1048  0.1018  0.0987  0.0956  0.0923  0.0890  0.0855
0.0820  0.0784  0.0747  0.0710  0.0672  0.0633  0.0594  0.0554  0.0513  0.0472
0.0431  0.0389  0.0347  0.0304  0.0261  0.0218  0.0175  0.0131  0.0087  0.0044
```

```
bisection iter: 23
inverse power iter: 4
largest: 3.99903
time: 151381
vector:
0.0044  -0.0087 0.0131  -0.0175 0.0218  -0.0261 0.0304  -0.0347 0.0389  -0.0431
0.0472  -0.0513 0.0554  -0.0594 0.0633  -0.0672 0.0710  -0.0747 0.0784  -0.0820
0.0855  -0.0890 0.0923  -0.0956 0.0987  -0.1018 0.1048  -0.1076 0.1104  -0.1131
0.1156  -0.1181 0.1204  -0.1226 0.1247  -0.1266 0.1285  -0.1302 0.1318  -0.1333
0.1346  -0.1358 0.1369  -0.1379 0.1387  -0.1393 0.1399  -0.1403 0.1406  -0.1407
0.1407  -0.1406 0.1403  -0.1399 0.1393  -0.1387 0.1379  -0.1369 0.1358  -0.1346
0.1333  -0.1318 0.1302  -0.1285 0.1266  -0.1247 0.1226  -0.1204 0.1181  -0.1156
0.1131  -0.1104 0.1076  -0.1048 0.1018  -0.0987 0.0956  -0.0923 0.0890  -0.0855
0.0820  -0.0784 0.0747  -0.0710 0.0672  -0.0633 0.0594  -0.0554 0.0513  -0.0472
0.0431  -0.0389 0.0347  -0.0304 0.0261  -0.0218 0.0175  -0.0131 0.0087  -0.0044
```

结果如上，两边的迭代次数都并不多，综合性能也较好。

总结：

难度而言总还是比 Lab 6 好写，不过看到最后一个是 SVD，估计还得继续折磨……

另：排序特征值逼我把算法课写的代码搬了出来，很难评价.jpg

Lab 8 实验报告

PB20000296 郑滕飞

代码实现：

1、二对角化

仿照书上伪代码实现即可，分为两步，每轮对行用 householder 变换并更新 U:

```
for (int j = k; j < m; j++) v[j] = A[j][k];
double beta = householder(v, k, m);
for (int i = k; i < n; i++) {
    u[i] = 0;
    for (int j = k; j < m; j++)
        u[i] += beta * v[j] * A[j][i];
}
for (int i = k; i < m; i++) for (int j = k; j < n; j++) {
    A[i][j] -= v[i] * u[j];
}
for (int i = 0; i < m; i++) {
    u[i] = 0;
    for (int j = k; j < m; j++)
        u[i] += beta * U[i][j] * v[j];
}
for (int i = 0; i < m; i++) for (int j = k; j < m; j++) {
    U[i][j] -= u[i] * v[j];
}
```

接着对列并更新 V:

```
if (k == n - 1) break;
for (int j = k + 1; j < n; j++) v[j] = A[k][j];
beta = householder(v, k + 1, n);
for (int i = k; i < m; i++) {
    u[i] = 0;
    for (int j = k + 1; j < n; j++)
        u[i] += beta * A[i][j] * v[j];
}
for (int i = k; i < m; i++) for (int j = k + 1; j < n; j++) {
    A[i][j] -= u[i] * v[j];
}
for (int i = 0; i < n; i++) {
    u[i] = 0;
    for (int j = k + 1; j < n; j++)
        u[i] += beta * V[i][j] * v[j];
}
for (int i = 0; i < n; i++) for (int j = k + 1; j < n; j++) {
    V[i][j] -= u[i] * v[j];
}
```

2、收敛判定

```
for (int i = 0; i < n; i++) {
    if (A[i][i] < 1e-10) A[i][i] = 0;
    if (i == n - 1) break;
    if (A[i][i + 1] < 1e-10) A[i][i + 1] = 0;
}
```

在如上方将值降为 0 后，进行收敛判定：

```
int now = n - 1;
while (A[now - 1][now] == 0)
    if (--now == 0) return 1;
end = now + 1;
while (A[now - 1][now] != 0)
    if (--now == 0)
        return (begin = 0);
begin = now;
return 0;
```

当返回值为 1 时说明已收敛，否则 begin 和 end 确定了中间块的位置。

3、零对角元处理

为方便计算进行了宏定义，：

```
#define sq(x) (x * x)
#define l2(x,y) sqrt(x * x + y * y)
```

仿照书上过程，进行检测，若有零对角元则从其开始到结束依次利用 Givens 变换，然后返回 0，进行下一轮，否则返回 1，正常进行 QR 迭代：

```
int zero_diag(vector<vector<double>>& A, vector<vector<double>>& U, int begin, int end) {
    int m = (int)U.size();
    for (int i = begin; i < end - 1; i++) if (A[i][i] == 0) {
        for (int j = i + 1; j < end; j++) {
            double a = A[i][j], b = A[j][j];
            double c = b / l2(a, b), s = a / l2(a, b);
            A[i][j] = 0;
            A[j][j] = l2(a, b);
            for (int t = 0; t < m; t++) {
                double t1 = c * U[t][i] - s * U[t][j];
                double t2 = s * U[t][i] + c * U[t][j];
                U[t][i] = t1;
                U[t][j] = t2;
            }
            if (j < end - 1) {
                A[i][j + 1] = -s * A[j][j + 1];
                A[j][j + 1] *= c;
            }
        }
        return 0;
    }
    return 1;
}
```

4、Wilkinson 位移迭代

这部分按照伪代码执行，先计算初始值：

```
int m = (int)A.size();
int n = (int)A[0].size();
double alpha = sq(A[end - 1][end - 1]) + sq(A[end - 2][end - 1]),
    delta = (sq(A[end - 2][end - 2]) + sq((end - 3) < begin ? 0 : A[end - 3][end - 2]) - alpha) / 2,
    beta = A[end - 2][end - 2] * A[end - 2][end - 1],
    mu = alpha - sq(beta) / (delta + (delta > 0 ? 1 : -1) * l2(delta, beta)),
    y = sq(A[begin][begin]) - mu,
    z = A[begin][begin] * A[begin][begin + 1];
```

在每轮迭代中，先对列处理：

```
double c = y / l2(y, z), s = -z / l2(y, z);
for (int t = 0; t < n; t++) {
    t1 = c * V[t][k] - s * V[t][k + 1];
    t2 = s * V[t][k] + c * V[t][k + 1];
    V[t][k] = t1;
    V[t][k + 1] = t2;
}
if (k > begin) A[k - 1][k] = l2(y, z);
y = c * A[k][k] - s * A[k][k + 1];
A[k][k + 1] = s * A[k][k] + c * A[k][k + 1];
z = -s * A[k + 1][k + 1];
A[k + 1][k + 1] *= c;
```

接着对行处理：

```
c = y / l2(y, z), s = -z / l2(y, z);
for (int t = 0; t < m; t++) {
    t1 = c * U[t][k] - s * U[t][k + 1];
    t2 = s * U[t][k] + c * U[t][k + 1];
    U[t][k] = t1;
    U[t][k + 1] = t2;
}
A[k][k] = l2(y, z);
if (k < end - 2) {
    y = c * A[k][k + 1] - s * A[k + 1][k + 1];
    A[k + 1][k + 1] = s * A[k][k + 1] + c * A[k + 1][k + 1];
    z = -s * A[k + 1][k + 2];
    A[k + 1][k + 2] *= c;
}
else {
    t1 = c * A[k][k + 1] - s * A[k + 1][k + 1];
    t2 = s * A[k][k + 1] + c * A[k + 1][k + 1];;
    A[k][k + 1] = t1;
    A[k + 1][k + 1] = t2;
}
```

5、完整 SVD

核心过程如下，U 与 V 是输入的全 0 矩阵，以单位阵进入迭代，返回计数值：

```
int SVD(vector<vector<double>>& A, vector<vector<double>>& U, vector<vector<double>>& V) {
    int m = A.size();
    int n = A[0].size();
    if (m < n) return -1;

    for (int i = 0; i < m; i++) U[i][i] = 1;
    for (int i = 0; i < n; i++) V[i][i] = 1;

    bidiag(A, U, V);
    int begin, end, count = 0;
    while (!judge_and_modify(A, begin, end)) {
        if (zero_diag(A, U, begin, end)) wilkinson(A, U, V, begin, end);
        count++;
    }

    return count;
}
```

结果展示：

```
10000000 ticks per second
the following timings are all in ticks

iter: 10
time: 12301
```

```
ep: 6.66134e-16
eq: 1.11022e-15
et: 0.090278
SVD:
0.382658
0.759848
0.835333
1.09577
1.89472
2.60196
3.1445
4.98101
5.94702
8.66648
32.2979
214.31
```

```
A:
214.310 0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
0.000   32.298  0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
0.000   0.000   8.666   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
0.000   0.000   0.000   5.947   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
0.000   0.000   0.000   0.000   4.981   0.000   0.000   0.000   0.000   0.000   0.000   0.000
0.000   0.000   0.000   0.000   0.000   3.144   0.000   0.000   0.000   0.000   0.000   0.000
0.000   0.000   0.000   0.000   0.000   0.000   2.602   0.000   0.000   0.000   0.000   0.000
0.000   0.000   0.000   0.000   0.000   0.000   0.000   1.895   0.000   0.000   0.000   0.000
0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.383   0.000   0.000   0.000
0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.760   0.000   0.000
0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   1.096   0.000
0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.835
0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000   0.000
```

```
V:
0.023   -0.042  0.010   -0.028  0.124   -0.089  0.060   -0.047  0.717   0.624   0.252   0.000
0.162   -0.644  -0.522  0.224   -0.385  -0.211  0.117   0.098   0.031   -0.066  0.102   -0.071
0.029   -0.094  -0.063  0.017   -0.043  0.185   0.105   0.001   -0.589  0.732   -0.088  0.220
0.145   -0.543  0.759   -0.227  -0.188  -0.029  -0.117  0.074   -0.016  -0.001  0.015   -0.006
0.035   -0.116  -0.019  0.057   -0.006  0.212   0.207   0.098   0.331   -0.055  -0.772  0.419
0.031   -0.087  -0.135  0.156   0.191   -0.053  -0.891  0.266   0.020   0.077   -0.093  0.175
0.157   -0.363  -0.050  0.090   0.642   0.257   0.098   -0.347  -0.053  -0.205  0.282   0.317
0.078   -0.176  -0.048  0.060   0.297   0.432   0.006   0.160   0.013   0.087   -0.245  -0.768
0.956   0.284   0.011   0.023   -0.056  -0.028  0.002   0.015   -0.004  -0.002  -0.004  -0.000
0.054   -0.084  -0.348  -0.924  0.035   0.046   -0.088  -0.001  0.005   -0.026  -0.040  0.022
0.026   -0.083  0.035   -0.030  0.436   -0.779  0.118   -0.096  -0.149  0.084   -0.341  -0.156
0.008   -0.029  -0.030  0.042   -0.261  0.050   -0.302  -0.864  0.046   0.065   -0.237  -0.158
```

*U 因为过大并不展示，具体可以在运行时看见

总结：

数 值 代 数 实 验 再 见

感觉这辈子之后的时间里应该不至于再用 C++生写矩阵运算了吧……带个 Eigen 或者 Python 加 Numpy 或者 MATLAB 不都比这强(

最折磨的地方应该是把 MATLAB 伪代码转成 C 的地方，毕竟下标和习惯都不一样，还有 C++的一大堆细节处理……

总之希望期末没事.jpg

[更新：期末已经有事了，悲]