1.编写一个割线法程序，求解下面的方程组：

（1） $x^2 - e^x = 0$
（2） $xe^x - 1 = 0$
（3） $lgx + x - 2 = 0$

解：

为了避免计算导数值，使用差商来替代导数，即有：

$$x_{k+1} = x_k - \frac{f(x_k)}{f(x_k) - f(x_{k-1})}\ (k = 1,2,3 \ldots)$$

解题过程中，需要首先估计x_0, x_1，可以用几何法或估计法大致算出根的一个边界值，然后进行迭代，知道x_k趋近于某个值，并且与x_{k-1}的差小于允许误差eps即迭代完成

代码如下：

```
#include <bits/stdc++.h>
using namespace std;


double eps = 1e-5; //  允许误差范围


// (1)
double f1(double x)
{
	return pow(x, 2) - exp(x);
}

// (2)
double f2(double x)
{
	return x*exp(x) - 1;
}

// (3)
double f3(double x)
```

```
{
	return log10(x) - x - 2;
}

// 割线法迭代公式
double g(double x0, double x1, double (*f)(double))
{
	return x1 - f(x1) * (x1 - x0) / (f(x1) - f(x0));
}

int main()
{
	double x0 = 0, x1 = 1;
	int cnt = 0;
	while (fabs(x1 - x0) > eps)
	{
		double x2 = g(x0, x1, f2);
		x0 = x1;
		x1 = x2;
		cnt++;
		printf("迭代%d 次的结果为:%.5lf\n",cnt, x1);
	}
	return 0;
}
```

运行结果为:

```
迭代 1 次的结果为:0.36788
迭代 2 次的结果为:0.50331
迭代 3 次的结果为:0.57862
迭代 4 次的结果为:0.56653
迭代 5 次的结果为:0.56714
迭代 6 次的结果为:0.56714
```

2. 编写用改进的平方根法解方程组$\boldsymbol{Ax}=\boldsymbol{b}$的程序，并解下列方程组：

$$A=\begin{bmatrix} 0.5 & -0.5 & 0 & 0 & 0 & 0 \\ -0.5 & 1.5 & -0.5 & -0.25 & 0.25 & 0 \\ 0 & -0.5 & 1.5 & 0.25 & -0.25 & 0 \\ 0 & -0.25 & 0.25 & 1.5 & -0.5 & 0 \\ 0 & 0.25 & -0.25 & -0.5 & 1.5 & -0.5 \\ 0 & 0 & 0 & 0 & -0.5 & 0.5 \end{bmatrix}$$
$$b=(-1,0,0,0,0,0)^T$$

解：

当方程组的系数是对称正定时，可以使用改进平方根法改进LU矩阵分解，加速运行效率，对于方程组$Ax=b$，令$Ly=b$，$Ux=y$，其中U,L分别为上三角矩阵和下三角矩阵，则有：$A=LU$，再用回代即可求解

对于上三角矩阵U矩阵有：

$$u_{ki}=a_{ki}-\sum_{q=1}^{k-1}\frac{u_{qk}u_{qi}}{u_{qq}}\ (i=k,k+1,\dots n)$$

对于下三角矩阵L矩阵有：

$$l_{ik}=a_{ki}-\frac{\sum_{q=1}^{k-1}\frac{u_{qi}u_{qk}}{u_{qq}}}{u_{kk}}=\frac{u_{ki}}{u_{kk}}\ (i=k+1,k+2,\dots n)$$

若矩阵A为对称正定矩阵，则A一定能直接作LU分解，且$l_{ik}=\frac{u_{ki}}{u_{kk}}\ (k=1,2\dots n)$

代码如下：

```
#include <bits/stdc++.h>
using namespace std;

// LU decomposition using vector
void LU(vector<vector<double>> &A, vector<vector<double>> &L, vector<vector<double>> &U) {
	int n = A.size();
	for (int i = 0; i < n; i++) {
```

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

    // 打印 U 矩阵

    cout << "U:" << endl;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < U[i].size(); j++) {
            cout << U[i][j] << " ";
        }
        cout << endl;
    }

    // 打印 L 矩阵

    cout << "L:" << endl;
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < L[i].size(); j++) {
            cout << L[i][j] << " ";
        }
        cout << endl;
    }
}

// solve Ax = b using LU decomposition
vector<double> solve(vector<vector<double>> &A, vector<double> &b) {
    int n = A.size();
    vector<vector<double>> L(n, vector<double>(n, 0));
```

```
	vector<vector<double>> U(n, vector<double>(n, 0));
	LU(A, L, U);
	vector<double> y(n, 0);
	vector<double> x(n, 0);
	// Ly = b
	for (int i = 0; i < n; i++) {
		double sum = 0;
		for (int k = 0; k < i; k++) {
			sum += L[i][k] * y[k];
		}
		y[i] = b[i] - sum;
	}
	// Ux = y
	for (int i = n - 1; i >= 0; i--) {
		double sum = 0;
		for (int k = i + 1; k < n; k++) {
			sum += U[i][k] * x[k];
		}
		x[i] = (y[i] - sum) / U[i][i];
	}
	return x;
}

// main function
int main() {
	vector<vector<double>> A = {{0.5, -0.5, 0, 0, 0, 0}, {-0.5, 1.5, -0.5, -0.25, 0.25,
0},
	{0, -0.5, 1.5, 0.25, -0.25, 0}, {0, -0.25, 0.25, 1.5, -0.5, 0},
	{0, 0.25, -0.25, -0.5, 1.5, -0.5}, {0, 0, 0, 0, -0.5, 0.5}};
	vector<double> b = {-1, 0, 0, 0, 0, 0};
	vector<double> x = solve(A, b);

	cout << "x:" << endl;
	for (int i = 0; i < x.size(); i++) {
		cout << x[i] << endl;
	}
	return 0;
}
```

运行结果如下:

```
上三角U矩阵:
0.5 -0.5 0 0 0 0
0 1 -0.5 -0.25 0.25 0
0 0 1.25 0.125 -0.125 0
0 0 0 1.425 -0.425 0
0 0 0 0 1.29825 -0.5
0 0 0 0 0 0.307432
下三角L矩阵:
1 0 0 0 0 0
-1 1 0 0 0 0
0 -0.5 1 0 0 0
0 -0.25 0.1 1 0 0
0 0.25 -0.1 -0.298246 1 0
0 0 0 0 -0.385135 1
x:
-3.25275
-1.25275
-0.373626
-0.0879121
0.175824
0.175824
```

3. 编写一个用牛顿前插公式计算函数值的程序，要求先输出差分表，再计算$\boldsymbol{x}$点的函数值，并应用于下面的问题：

x_i	20	21	22	23	24
y_i	1.30103	1.32222	1.34242	1.36173	1.38021

求$x = 21.4$时的三次插值多项式的值

解：

先迭代求出k次差分表，再代入牛顿前插公式中：

$$N_n(x_0 + th) = \sum_{k=0}^{n} \frac{\Delta^k f_0}{k!} \prod_{j=0}^{k-1} (t - j)$$

代码如下：

```
#include <iostream>
#include <vector>

double newtonInterpolation(double x, const std::vector<double>& xi, const std::vector<double>& yi) {
	int n = xi.size();
	std::vector<std::vector<double>> diffTable(n, std::vector<double>(n));
	for (int i = 0; i < n; i++) {
		diffTable[i][0] = yi[i];
	}

	for (int j = 1; j < n; j++) {
		for (int i = 0; i < n - j; i++) {
			diffTable[i][j] = (diffTable[i + 1][j - 1] - diffTable[i][j - 1]) / (xi[i + j] - xi[i]);
		}
	}
	std::cout << "差分表：" << std::endl;

	for (int i = 0; i < n; i++) {
		std::cout << xi[i] << "\t";
		for (int j = 0; j <= i; j++) {
			std::cout << diffTable[i - j][j] << "\t";
		}
```

```
            std::cout << std::endl;
        }
        std::cout << std::endl;
        double result = 0.0;
        double prod = 1.0;

        for (int i = 0; i < n; i++) {
            prod = diffTable[i][i];
            for (int j = 0; j < i; j++) {
                prod *= (x - xi[j]);
            }
            result += prod;
        }

        return result;
}
int main() {
        std::vector<double> xi = { 20, 21, 22, 23, 24 };
        std::vector<double> yi = { 1.30103, 1.32222, 1.34242, 1.36173, 1.38021 };

        double x = 21.4;
        double interpolationResult = newtonInterpolation(x, xi, yi);


        std::cout << "在  x = " << x << " 时的三次插值多项式的值为: " <<
interpolationResult << std::endl;

        return 0;
}
```

运行结果如下:

```
差分表:

20      1.30103

21      1.32222  0.02119

22      1.34242  0.0202   -0.000495

23      1.36173  0.01931  -0.000445        1.66667e-05

24      1.38021  0.01848  -0.000415        1e-05    -1.66667e-06

在 x = 21.4 时的三次插值多项式的值为: 1.32908
```

4. 编写求超定方程组的最小二乘法解的程序，并解下列方程组：

$$\begin{cases} 2x+4y=10 \\ 3x-5y=-13 \\ 10x-12y=-26 \\ 4x+11y=25 \end{cases}$$

解:

对于方程组$Ax=b$，可以写成$A^TAx=A^Tb$，可以证明如果A是满秩的，则方程组$A^TAx=A^Tb$存在唯一解，并把该方程组的解称为超定方程组的最小二乘法解

代码如下:

```
#include <bits/stdc++.h>
using namespace std;

// 最小二乘法解超定方程组

vector<double> leastSquare(vector<vector<double>> &A, vector<double> &b) {
	int n = A.size();
	int m = A[0].size();
	vector<vector<double>> AT(m, vector<double>(n, 0));
	vector<vector<double>> ATA(m, vector<double>(m, 0));
	vector<double> ATb(m, 0);
	for (int i = 0; i < m; i++) {
		ATb[i] = 0;
		for (int j = 0; j < n; j++) {
			AT[i][j] = A[j][i];
			ATb[i] += A[j][i] * b[j];
		}
	}
	for (int i = 0; i < m; i++) {
		for (int j = i; j < m; j++) {
			ATA[i][j] = 0;
			for (int k = 0; k < n; k++) {
				ATA[i][j] += AT[i][k] * A[k][j];
			}
			ATA[j][i] = ATA[i][j];
```

```
        }
    }
    vector<double> x(m, 0);
    // solve ATA * x = ATb
    for (int i = 0; i < m; i++) {
        double sum = 0;
        for (int k = 0; k < i; k++) {
            sum += ATA[i][k] * x[k];
        }
        x[i] = (ATb[i] - sum) / ATA[i][i];
    }
    return x;
}

// main function
int main() {
    int n, m;
    cin >> n >> m;
    vector<vector<double>> A = {{2, 4}, {8, 4}, {2, 1}, {7, -1}, {4, 0}};
    vector<double> b = {10, -13, -26, 25};
    vector<double> x = leastSquare(A, b);
    for (int i = 0; i < x.size(); i++) {
        cout << x[i] << " ";
    }
    cout << endl;
    return 0;
}
```

运行结果如下:

```
0.284672 -2.14599
```