学生学号		实验课成绩	

学 生 实 验 报 告 书

武漢理工大學

实验课程名称	计算机数值分析
开课学院	计算机与人工智能学院
指导教师姓名	
学生姓名	
学生专业班级	

2024 -- 2025 学年 第 1 学期

实验课程名称：计算机数值分析

实验项目名称	用 C 语言实现几个多项式插值的程序。(Lagrange 插值、Newton 插值)			实验成绩	
实验者		专业班级		组别	
同组者				实验日期	年 月 日

第一部分：实验分析与设计（可加页）

一、实验内容描述（问题域描述）

问题 已知函数的一组数据，考虑使用合适的数值方法，求出一个未知节点处的函数值近似。

解决 考虑利用多项式来拟合函数，此时涉及到的是 Lagrange 插值。在提供了已知函数数据后，根据 Lagrange 插值的相关原理，套用 Lagrange 插值公式求出相应节点处的插值。

二、实验基本原理与设计

根据 Lagrange 插值公式

$$p_n(x) = \sum_{k=0}^{n} y_k l_k(x) = \sum_{k=0}^{n} \left(\prod_{\substack{j=0 \\ j \neq k}}^{n} \frac{x - x_j}{x_k - x_j} \right) y_k$$

由于其形式较为简单（累乘后求和），考虑使用嵌套的循环来实现。其中，

- ✧ 内部的循环负责构造出 Lagrange 插值基函数的值
- ✧ 外部的循环负责将这些基函数与已知的 y_k 组合后相加。

三、主要仪器设备及耗材

1．PC 机

2．C++集成环境

第二部分：实验调试与结果分析（可加页）

一、调试过程（包括调试方法描述、实验数据记录，实验现象记录，实验过程发现的问题等）

实验过程发现的问题

利用 C++语言实现 Lagrange 插值函数，算法如下

参数：

- ✧ std::vector<std::pair<*int*, *int*>> data 是一组已知的函数值，其中 pair 的第一项为 x_k，第二项为 y_k。
- ✧ *int* x 是要求插值的点

```
double LagrangeInterpolation(const std::vector<std::pair<int, int>> &data, int x) {
    double y = 0;
    int k = 0;
    while (k < data.size()) {
        double t = 1;
        for (int j = 0; j < data.size(); j++) {
            if (j == k) continue;
            t = t * (x - data[j].first) / (data[k].first - data[j].first);
        }
        y += t * data[k].second;
        k++;
    }
    return y;
}
```

该函数内部的 for 循环实现了求 Lagrange 插值基函数，而外部循环根据 data 的大小（给出点的多少）来对 Lagrange 插值基函数与 y_k 的乘积进行求和。

二、实验结果及分析（包括结果描述、实验现象分析、影响因素讨论、综合分析和结论等）

实验过程未遇到 Warning 或者 Error，但有以下考虑过程：由于 Lagrange 插值需要基础数据，因此选择合适的基础数据提供方法就显得很重要。由于 x、y 是两两相匹配的，因此使用 std::pair<*int*, *int*> 这一数据结构来处理。

准确性验证代码如下：

```
cout << (LagrangeInterpolation({{1,9}, {2,6}, {4,0}}, 5) == -3) << endl;
cout << (LagrangeInterpolation({{1,9}, {2,6}, {4,0}}, 2) == 6) << endl;
cout << (LagrangeInterpolation({{1,9}, {2,6}, {4,0}}, 8) == -12) << endl;
cout << (abs(LagrangeInterpolation({{2, 3}, {7, 2}, {9, 12}}, -3) - 41.142857142857146) < 0.001) << endl;
cout << (LagrangeInterpolation({{2, 3}, {7, 2}, {9, 12}}, 9) == 12) << endl;
cout << (abs(LagrangeInterpolation({{2, 3}, {7, 2}, {9, 12}}, -10) - 156.94285714285715) < 0.001) << endl;
```

输出为：

```
1
1
```

1
1
1
1

代表了测试通过。

第三部分：实验小结、建议及体会

1. 本次实验的内容是一项基本的插值方法——Lagrange 插值。Lagrange 插值易于编写算法，因为它已经给出了明确的公式，且公式的形式并不复杂，适合直接计算
2. 借助于 pair、vector 数据结构，可以高效地传递 Lagrange 插值过程所需的数据
3. 虽然插值基函数的构造具有一定特殊性，但在代码中通过 *if* (j == k) *continue*;这一判断，即可实现。

实验课程名称：计算机数值分析

实验项目名称	用 C 语言实现几个求常微分方程初值问题解的程序。(Euler 方法及其改进、龙格-库塔(Runge-Kutta)方法)			实验成绩	
实验者		专业班级		组别	
同组者				实验日期	年 月 日

第一部分：实验分析与设计（可加页）

一、实验内容描述（问题域描述）

本次实验所要解决的是利用数值方法来近似解决常微分方程的初值问题：

$$y' = f(x, y)$$
$$y(x_0) = y_0$$

其基本思想是利用差分方法，通过合适的离散化手段，给出近似解的递推公式。根据这一思想，引出了 Euler 方法、Runge-Kutta 方法、Adams 方法等。

本次实验尝试使用 Runge-Kutta 方法来提供近似解。

二、实验基本原理与设计

Runge-Kutta 方法的基本原理，同样用到了利用差商来近似导数的离散化手段，并利用微分中值定理引入变量 ξ 和平均斜率的概念。其基础计算格式为

$$y(x_{n+1}) = y(x_n) + hf(\xi, y(\xi))$$

其中，记

$$K^* = f(\xi, y(\xi))$$

它被称为**平均斜率**。Runge-Kutta 方法就是在区间内寻找到合适的点，并与其它点处的斜率进行加权获得 K^*，代入基础的计算格式得到具有 n 阶精度的 Runge-Kutta 方法。

经过一系列的推导，可以得到被称为**经典格式**的 **4 阶 Runge-Kutta 方法**，其表达式为

$$\begin{cases} y_{n+1} = y_n + \dfrac{h}{6}(K_1 + 2K_2 + 2K_3 + K_4) \\ K_1 = f(x_n, y_n) \\ K_2 = f(x_{n+\frac{1}{2}}, y_n + \dfrac{h}{2}K_1) \\ K_3 = f(x_{n+\frac{1}{2}}, y_n + \dfrac{h}{2}K_2) \\ K_4 = f(x_{n+1}, y_n + hK_3) \end{cases}$$

三、主要仪器设备及耗材

1．PC 机

2．C++集成环境

第二部分：实验调试与结果分析（可加页）

三、调试过程（包括调试方法描述、实验数据记录，实验现象记录，实验过程发现的问题等）

实验过程发现的问题

经典格式的算法实现如下。

参数：

- ✧ *double* x0 是初值条件的 x 值
- ✧ *double* y0 是初值条件的 y 值
- ✧ *double* h 是 Runge-Kutta 方法的步长
- ✧ *int* N 是 Runge-Kutta 方法的循环次数
- ✧ *double*(*f)(*double*, *double*) 是初值问题中的 f(x,y) 的编程等价

```
std::vector<std::pair<double, double>> RungeKutta4th(double x0, double y0, double h, int N, double (*f)(double, double)) {
    std::vector<std::pair<double, double>> result;

    int n = 1;
    double x1, y1, k1, k2, k3, k4;
    while (n <= N) {
        x1 = x0 + h;
        k1 = f(x0, y0);
        k2 = f(x0 + (h / 2), y0 + (h / 2) * k1);
        k3 = f(x0 + (h / 2), y0 + (h / 2) * k2);
        k4 = f(x1, y0 + h * k3);
        y1 = h * (k1 + 2 * k2 + 2 * k3 + k4) / 6 + y0;

        result.emplace_back(x1, y1);

        n++;
        x0 = x1;
        y0 = y1;
    }
    return result;
}
```

四、实验结果及分析（包括结果描述、实验现象分析、影响因素讨论、综合分析和结论等）

利用下列测试代码，输出 4 阶 Runge-Kutta 方法求解的过程数据。

```
cout << "RungeKutta4th: f3, y(0)=1, h=0.2, N=5" << endl;
vector<pair<double, double>> RK = RungeKutta4th(0, 1, 0.2, 5, f3);
for (auto& p : RK) {
    cout << "x: " << p.first << ", y: " << p.second << endl;
}
```

其中，f3 是下面的函数：

```
double f3(double x, double y) {
    return y - (2 * x) / y;
}
```

给出的输出为

```
RungeKutta4th: f3, y(0)=1, h=0.2, N=5
x: 0.2, y: 1.18323
x: 0.4, y: 1.34167
x: 0.6, y: 1.48328
x: 0.8, y: 1.61251
x: 1, y: 1.73214
```

与参考计算结果（下图）比对无误，表示方法实现成功。

表 3－3

x_n	y_n	$y(x_n)$	x_n	y_n	$y(x_n)$
0.2	1.183 2	1.183 2	0.8	1.612 5	1.612 5
0.4	1.341 7	1.341 6	1.0	1.732 1	1.732 1
0.6	1.483 3	1.483 2			

第三部分：实验小结、建议及体会

1. 当方法内部需要用到外部的函数时，可以使用 C++中的函数指针特性，用 *double* (*f)(*double*, *double*)) 来表示带有双精度输入和输出的二元函数。同理，一元函数可以用 *double* (*f)(*double*) 来表示。
2. 同样地，为了展示 Runge-Kutta 方法的过程数据，可将每一步的结果都放入一个 std::vector<std::pair<*double*, *double*>> 后输出。

实验课程名称：计算机数值分析

实验项目名称	用 C 语言实现几个非线性方程求根的程序。(二分法、迭代法、迭代过程的加速)			实验成绩	
实验者		专业班级		组别	
同组者				实验日期	年 月 日

第一部分：实验分析与设计（可加页）

一、实验内容描述（问题域描述）

对一个非线性方程组进行近似求解的一大基本方法是**迭代法**。通过从原方程构造出相应的迭代函数，写出迭代公式进行迭代求解。一个迭代公式要具有实用价值，首先必须是**收敛的**。但是，单凭收敛并不能称之为完美，还需要有良好的收敛速度。

本实验将探究迭代过程的加速，给出 Aitken 算法的具体实现。

二、实验基本原理与设计

为了加速迭代过程，可以对迭代公式进行加工：将原值与利用普通迭代公式的结果加权平均后，作为迭代的下一个值。经过推导可以得到迭代公式

$$x_{k+1} = \frac{1}{1-L}[\varphi(x_k) - Lx_k]$$

但这一方法涉及到对导数值的估计 L 因而造成计算上一定程度的麻烦。按照同样的思想对迭代一次后的值继续迭代，再同样利用 Lagrange 微分中值定理，对下列比例式进行化简：

$$\frac{x^* - \bar{x}_{k+1}}{x^* - \tilde{x}_{k+1}} \approx \frac{x^* - x_k}{x^* - \bar{x}_{k+1}}$$

就得到了

$$x^* \approx \tilde{x}_{k+1} - \frac{(\tilde{x}_{k+1} - \bar{x}_{k+1})^2}{\tilde{x}_{k+1} - 2\bar{x}_{k+1} + x_k}$$

取这一近似值作为第 k+1 次的迭代值，就得到了 **Aitken 加速方法**。

三、主要仪器设备及耗材

1．PC 机

2．C++集成环境

第二部分：实验调试与结果分析（可加页）

四、调试过程（包括调试方法描述、实验数据记录，实验现象记录，实验过程发现的问题等）

实验过程发现的问题

为了方便查看过程，定义结构

```
typedef struct {
    int steps;
    bool ok;
    double res;
    vector<vector<double> > iter;
} AitkenResult;
```

作为 Aitken 算法的返回值，其中

- steps 表示 Aitken 算法的迭代次数
- ok 表示 Aitken 算法是否迭代成功
- res 是最终满足精度要求的结果
- iter 为每一次迭代过程中，第二步的 $\bar{x}$、$\tilde{x}$ 和第一步的 x（按此顺序存储）

Aitken 算法实现如下。

参数：

- *double* x0 是迭代初值
- *double* e 是要求的精度
- *int* N 是限定的最大迭代次数
- *double*(*phi)(*double*) 是迭代函数

```
AitkenResult AitkenAcceleration(double x0, double e, int N, double(*phi)(double)) {
    int k = 1;
    double x1, x2;
    AitkenResult result;
    vector<vector<double> > iter;
    while (true) {
        vector<double> step;
        x1 = phi(x0);
        step.push_back(x1);
        x2 = phi(x1);
        step.push_back(x2);
        x2 = x2 - pow((x2 - x1), 2) / (x2 - 2 * x1 + x0);
        step.push_back(x2);

        iter.push_back(step);

        if (abs(x2 - x0) < e) {
            return {k, true, x2, iter};
        }
        if (k == N) {
            return {k, false, 0.0, iter};
```

```
            }
            k++;
            x0 = x2;
        }
}
```

五、实验结果及分析（包括结果描述、实验现象分析、影响因素讨论、综合分析和结论等）

采用如下测试代码进行测试：

```
cout << "---------AitkenAcc x0=0.5, e=0.0001, N=100" << endl;
auto aitken = AitkenAcceleration(0.5, 0.0001, 100, phi1);
if (!aitken.ok) cout << "Aitken Failed after " << aitken.steps << " steps" << endl;
else cout << "Aitken OK after " << aitken.steps << " steps" << endl;
cout << "x_bar \t x_tilde \t x_k" << endl;
for (auto &it: aitken.iter) {
    cout << it[0] << '\t' << it[1] << '\t' << it[2] << endl;
}
```

得到如下结果：

```
---------AitkenAcc x0=0.5, e=0.0001, N=100
Aitken OK after 3 steps
x_bar        x_tilde     x_k
0.606531  0.545239  0.567624
0.566871  0.567298  0.567143
0.567143  0.567143  0.567143
```

与标准结果进行比对，表明算法实现准确。

k	$\bar{x}_k$	$\tilde{x}_k$	x_k
1	0.606 53	0.545 24	0.567 12
2	0.566 87	0.567 30	0.567 14
3	0.567 14	0.567 14	

第三部分：实验小结、建议及体会

实验通过实现 Aitken 加速算法，验证了其在迭代法中的高效性和准确性。Aitken 算法通过对普通迭代结果进行二次加工，有效提高了收敛速度，减少了计算步骤。除了本实验所进行的任务以外，还可以进一步对比 Aitken 加速与其他加速方法的性能，探讨其优劣势。

实验课程名称：计算机数值分析

实验项目名称	用 C 语言实现几个求线性方程组解的程序。(Gauss 消去法)			实验成绩	
实验者		专业班级		组别	
同组者				实验日期	年 月 日

第一部分：实验分析与设计（可加页）

一、实验内容描述（问题域描述）

在用直接法求解线性方程组时，一大常用的方法是**消去法**，主要包括 Jordan 消去法和 Gauss 消去法。其中，Jordan 消去法的计算量较大，但其过程较为简单；Gauss 消去的计算量相比 Jordan 消去法减少了约 50%，但其过程较为复杂。

本实验将探讨 Gauss 消去法的本质，并给出相应算法的具体实现和测试。

二、实验基本原理与设计

说明：本部分的内容参考了清华大学出版社出版，梁鑫等编著的《线性代数入门》书籍。

从线性代数的视角来看，对线性方程组求解的过程是在对线性方程组对应的增广矩阵 [A | b] 做**初等变换**，最终得到一个形式简单、易于求解且与原方程同解的增广矩阵。

Jordan 消去法之所以到最后的计算比较简单，是因为它的步骤将增广矩阵化为了一个**行简化阶梯型矩阵**。本教材里的 Jordan 消去法并没有产生完全的行简化阶梯型矩阵（相应消去法*应该是？*被称为 Gauss-Jordan 消去法），最后一个未知数对应的列不满足主元以上均为 0 的性质，但也足够简单，可以被看做是行简化阶梯型。行简化阶梯型矩阵的一大简单之处就在于不需要进行回代。

而 Gauss 消去法则简化了 Jordan 消去法的矩阵变换过程，使得计算量减小。它将增广矩阵化为一个普通的**阶梯型矩阵**，在这种矩阵上可以应用**回代法**进行求解。

然而，从计算机的角度考虑，Gauss 消去法的过程只会确保矩阵上三角区域的元素中的主元非 0，但当它们非常小的时候，会导致严重的舍入误差积累，于是又需要进行 Gauss 消元法的主元选择，进行适当的行对换操作（也是初等变换），确保能够在电算过程得到正确的结果。

三、主要仪器设备及耗材

1．PC 机

2．C++集成环境

第二部分：实验调试与结果分析（可加页）

四、调试过程（包括调试方法描述、实验数据记录，实验现象记录，实验过程发现的问题等）

实验过程发现的问题

按照 Gauss 消去法的原理，选择主元，然后进行相应的变换，可以得到相应算法的实现如下：

参数：

为了表达方便，约定下标从 0 开始。

- ✧ vector<vector<*double*> > a 方程组的系数矩阵
- ✧ vector<*double*> b 方程组的常数项矩阵

```
vector<double> GaussElimination(vector<vector<double> > a, vector<double> b) {
    int k = 1;
    int n = static_cast<int>(b.size());

    int l, i, j;

    while (k < n + 1) {
        double d = a[k - 1][k - 1];
        l = k;
        i = k + 1;

        while (i < n + 1) {
            if (abs(a[i - 1][k - 1]) > abs(d)) {
                d = a[i - 1][k - 1];
                l = i;
            }
            i++;
        }

        if (d == 0) {
            return {};
        }

        if (l != k) {
            double t;
            for (j = k; j <= n; j++) {
                t = a[l - 1][j - 1];
                a[l - 1][j - 1] = a[k - 1][j - 1];
                a[k - 1][j - 1] = t;
            }
            t = b[k - 1];
            b[k - 1] = b[l - 1];
            b[l - 1] = t;
        }
```

```
            for (j = k + 1; j <= n; j++) {
                a[k - 1][j - 1] = a[k - 1][j - 1] / a[k - 1][k - 1];
            }
            b[k - 1] = b[k - 1] / a[k - 1][k - 1];
            for (i = k + 1; i <= n; i++) {
                for (j = k + 1; j <= n; j++) {
                    a[i - 1][j - 1] = a[i - 1][j - 1] - a[i - 1][k - 1] * a[k - 1][j - 1];
                }
            }
            for (i = k + 1; i <= n; i++) {
                b[i - 1] = b[i - 1] - a[i - 1][k - 1] * b[k - 1];
            }
            k++;
        }

        for (i = n - 1; i >= 1; i--) {
            double sum = 0;
            for (j = i + 1; j <= n; j++) {
                sum += a[i - 1][j - 1] * b[j - 1];
            }
            b[i - 1] = b[i - 1] - sum;
        }

        return b;
}
```

五、实验结果及分析（包括结果描述、实验现象分析、影响因素讨论、综合分析和结论等）

进行两次测试，分别为一般的线性方程组和含有小系数值的线性方程组。

测试 1

```
cout << "---------Gauss Test 1" << endl;
vector<vector<double> > a = {{2, -1, 3},
                             {4, 2,  5},
                             {1, 2,  0}};
vector<double> b = {1, 4, 7};
for (int i = 0; i < a.size(); i++) {
    cout << a[i][0] << "x1 + " << a[i][1] << "x2 + " << a[i][2] << "x3 = " << b[i] << endl;
}
auto gaussRes = GaussElimination(a, b);
cout << "RESULT: " << endl;
for (int i = 0; i < gaussRes.size(); i++) {
    cout << "x" << i + 1 << "=" << gaussRes[i] << " ";
}
```

结果 1

```
---------Gauss Test 1
2x1 + -1x2 + 3x3 = 1
```

```
4x1 + 2x2 + 5x3 = 4
1x1 + 2x2 + 0x3 = 7
RESULT:
x1=9 x2=-1 x3=-6
```

求解准确。

测试 2

```
cout << "---------Gauss Test 2" << endl;
cout << fixed << setprecision(5);

a = {{1e-5, 1},
     {1,      1}};
b = {1, 2};
for (int i = 0; i < a.size(); i++) {
     cout << a[i][0] << "x1 + " << a[i][1] << "x2 + " << a[i][2] << "x3 = " << b[i] << endl;
}
gaussRes = GaussElimination(a, b);
cout << "RESULT: " << endl;
for (int i = 0; i < gaussRes.size(); i++) {
     cout << "x" << i + 1 << "=" << gaussRes[i] << " ";
}
```

结果 2

```
---------Gauss Test 2
0.00001x1 + 1.00000x2 + 3.00000x3 = 1.00000
1.00000x1 + 1.00000x2 + 5.00000x3 = 2.00000
RESULT:
x1=1.00001 x2=0.99999
```

求解依然准确。可以断定以上 GaussElimination 算法实现准确。

第三部分：实验小结、建议及体会

本实验实现并测试了 Gauss 消去法的算法并验证了其对线性方程组的求解能力。在两次测试中，无论是一般情况还是含有小系数的情况，算法均能正确求解。除了这两种情况外，其实还可以增加对更多特殊矩阵（如稀疏矩阵）的测试。

实验过程加深了对矩阵变换和消去法原理的理解，同时体会到数值计算中舍入误差处理的重要性。