LinearSolver

一、项目介绍

本项目利用 `C++` 实现了一个简单的线性规划求解器，总共提供了三种求解方法：

- 采用大M法求初始可行解的单纯形法
- 两阶段的单纯形法
- 对偶单纯形法

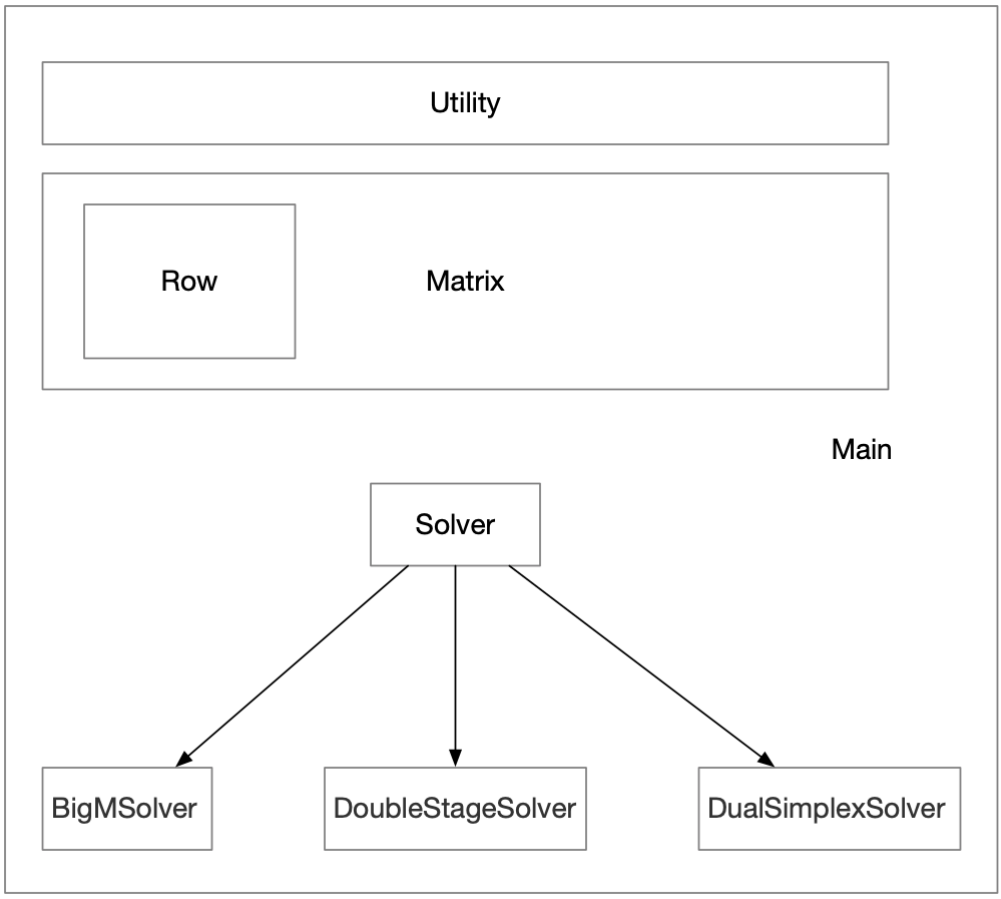
本项目能够用于求解一定规模（200个变量、500个约束）的规划问题，并将误差控制在 10^{-4} 以内，同时还考虑到了求解过程中的各种情况，如退化、无解、无有限最优解等，对这些情况均作出正确的处理。

此外，为进一步提升运行效率，本项目使用 `OpenMP` 支持多个线程并行计算。

使用方式见压缩包内 `readme.md` 文件。

二、总体设计

1. 整体架构

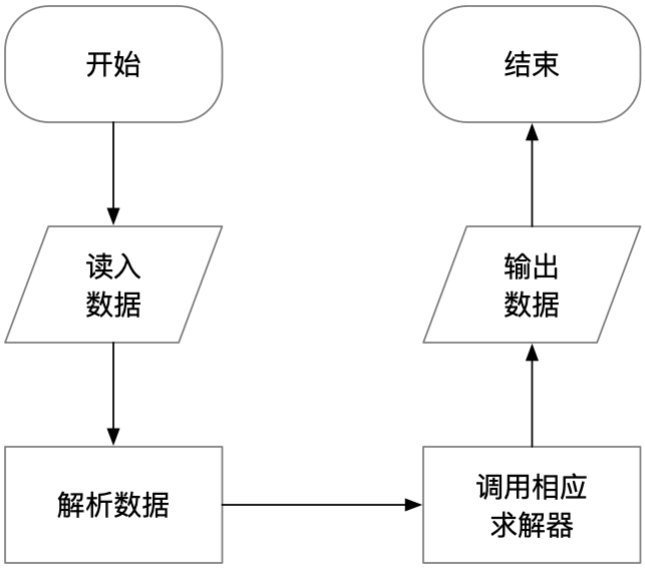


2. 模块划分

本项目各模块划分如下：

- utility.cpp / utility.h：定义一些公用的宏和函数。该模块中负责规定打印格式，提供打印各种信息的接口，指定程序运行的模式（是否输出DEBUG信息以及是否开启并行选项等），选择使用的求解方法等。
- matrix.cpp / matrix.h：实现矩阵类以及矩阵类的一些基本操作（算术运算、添加行和列等）。
- row.cpp / row.h：实现行类，用于表示矩阵的一行，可以从矩阵通过方括号下标运算得到，也支持通过下标得到某个具体元素。
- solver.cpp / solver.h：实现求解器类以及所有求解方法共享的操作，如松弛、标准化操作，也是所有求解器的父类。
- bigMSolver.cpp / bigMSolver.h：采用大M法求初始可行解的单纯形法求解器，继承自求解器类。
- doubleStageSolver.cpp / doubleStageSolver.h：两阶段的单纯形法求解器，继承自求解器类。
- dualSimplexSolver.cpp / dualSimplexSolver.h：对偶单纯形法求解器，继承自求解器类。
- main.cpp：程序入口。

3. 运行流程



4. 数据结构

Matrix

```
int row;
int column;
double *data;
```

`Matrix` 的成员变量及其含义如下：

- row：矩阵的行数。
- column：矩阵的列数。
- data：矩阵的内容，是一个指向double类型指针，在构造对象的时候动态分配，长度为row * column。

Solver

```
int n, m;
Matrix c;
Matrix a, b, d;
Matrix e;
vector<int> negative;  // xi -> -xi, if xi <= 0
vector<pair<int, int> > noConstraints;  // xi -> xi - xj, if no constraints on xi
```

`Solver` 的成员变量及其含义如下：

- n：变量个数。
- m：约束个数。
- a, b, c, d, e：与题目描述一致。
- negative：用于存储小于等于0的变量下标，初始为空，在标准化一步中被添加。
- noConstraints：用于存储替换无限制变量的一对变量下标，初始为空，在标准化一步中被添加。

三、原理说明

1. 松弛和标准化

由于约束和变量限制种类多样，不利于用统一的方法求解，因此在求解前，应先对数据进行一些处理。

主要分为两步：

- 松弛：对不等式的约束，引进松弛变量，将其变为等号。
- 标准化：对不是限制非负的变量，进行替换，使得所有变量的限制都转为非负。

2. 求解单纯形表

这里的单纯形表需要先换成基本形式（即引入松弛变量、人工变量和归一化）。这一部分将在下面部分介绍。同时此处的求解对象是$\max$型。单纯形表的运算步骤如下：

1. 停止检验：检验是否所有的检验数均≤ 0。若是，则迭代停止，通知其余模块进行可行解检验。否则进行下一步。
2. 无界检验：检测是否存在某检验数$\sigma_i > 0$，有$a_{ij} \leq 0$。若是，则为无界解，迭代停止。否则进行下一步。
3. 令$\sigma_k = \max\limits_j\{\sigma_j\}$。对所有$a_{ik} > 0$，计算$\theta_i = b_i / a_{ik}$。令$\theta_s = \min\limits_i\{\theta_i\}$。
4. 迭代运算：首先更换第s行的基变量为x_k。对于第s行，归一化使得$a_{sk} = 1$。对于第k列，使用初等行变换令其余列为0。对于检验行和目标值，使用初等行变换使得c_k为0。
5. 回到第1步进行下一轮迭代。

3. 大M法

对每条约束都增加一个对应的人工变量 $\bar{x}_i$，并引入较大的惩罚项 M。

原问题转化为

$$
\begin{aligned}
\max_{x} \quad & z = c^T x - M(\bar{x}_1 + \bar{x}_2 + \cdots + \bar{x}_m) \\
\text{s.t.} \quad & Ax + \bar{x} = b \\
& x, \bar{x} \geq 0
\end{aligned}
$$

惩罚项 M 的作用是迫使 $\bar{x}=0$。

在实际实现中，引入惩罚项M后还需要对检验数进行归一化（令基变量为0）。之后调用上一步求解单纯形表。若单纯形表正常停止，则检验是否所有的人工变量检验数均为0（即存在人工变量是基变量）。若是，则算法无解，否则输出答案。

4. 两阶段法

对每条约束都增加一个对应的人工变量 $\bar{x}_i$，将原问题转化为

$$\begin{aligned} \min_{x} \quad & \textstyle\sum_{i=1}^{m} \bar{x}_i \\ \text{s.t.} \quad & Ax + \bar{x} = b \\ & x, \bar{x} \geq 0 \end{aligned}$$

若$\sum_{i=1}^{m}=0$，则得到原问题的一个可行解。

若$\sum_{i=1}^{m}>0$，则原问题无解。

两阶段需要求解两次单纯形表。第一阶段，引入系数为1的人工变量，归一化后调用解单纯形表的函数。停止后，检验目标函数值是否为0以进行有解/无解判断。

第二阶段，恢复之前的系数并去除人工变量。归一化后再次调用解单纯形表的函数（单纯形表其余部分可以不动）。停止后，输出最终答案。

5. 对偶单纯形法

对偶单纯形法将从对偶可行性逐步搜索出原始问题最优解。

下面以$\max$问题为例，说明对偶单纯形法的迭代过程：

1. 添加松弛变量，将原问题化为标准形式，并且绘制如式(1)的单纯形表。
2. 首先确定出基变量，选取b中负值最小的行（记为i行），将其对应的变量x定为出基变量
3. 然后确定入基变量，选取行（记为j行）对应的检验变量c_j与系数$a_{i,j}$（要求为负，如果所有的a均为非负，则原问题无解）比值的绝对值最小（绝对值比值最小，保证变化后检验数依然小于0）。
4. 替换形表左侧的基变量，通过行变换将系数$a_{i,j}$化为1且该列其他系数化为0。
5. 重复步骤1-4直到b中均为非负。

由于对偶单纯形法只适用于对偶可行解较容易找到的问题，但大多数情况下，对偶可行解找起来都并不是那么容易。因而我们对问题加了一些限制，要求使用对偶单纯形法的问题必须满足：

- 所有的初始检验数都非负（即 $c_i \geq 0$）。
- 所有的约束均为大于等于（即 $d_j = 1$）。
- 所有的变量均限制为非负（即 $e_i = 1$）。

当满足这三个条件时，松弛后得到的必定是一个对偶可行的解，且最后 m 个变量均为松弛变量，可以作为一组初始基；否则，对偶单纯形法求解器将返回 $k = -2$，表示认定该数据不适合选择对偶单纯形法求解，而拒绝做出回答。

四、程序实现

1. 松弛和标准化

松弛操作的实现位于 `Solver.cpp` 文件的 `relax` 函数中，目的是将每个约束都转变为等号。

程序逻辑如下：

- 逐一检查每个约束，检查左右两侧关系，分为三种情况：
  - 如果已经是等号，满足要求，不作任何处理
  - 如果是大于，增加松弛变量，矩阵A右侧新增该变量对应的一列，且只有当前约束中该松弛变量的系数为-1，其余都为0，同时将大于变为等号。
  - 如果是小于，增加松弛变量，矩阵A右侧新增该变量对应的一列，且只有当前约束中该松弛变量的系数为1，其余都为0，同时将小于变为等号。

该部分代码如下：

```
void Solver::relax() {
    Matrix zero = Matrix(1, 1);
    Matrix one = Matrix(1, 1, 1);
    for (int i = 0; i < m; i++) {
        if (equal(d[i][0], 0)) continue;  // 已经满足条件，不做任何处理
        if (equal(d[i][0], 1)) {
            n++;
            c.appendColumn(zero);  // 松弛变量的检验数为0
            e.appendColumn(one);  // 松弛变量的约束为 x >= 0
            // 在单纯形表右侧添加一列
            Matrix newColumn(m, 1);
            newColumn[i][0] = -1;
            a.appendColumn(newColumn);
            d[i][0] = 0;  // 约束变为等式
            continue;
        }
        if (equal(d[i][0], -1)) {
            n++;
            c.appendColumn(zero);  // 松弛变量的检验数为0
            e.appendColumn(one);  // 松弛变量的约束为 x >= 0
            // 在单纯形表右侧添加一列
            Matrix newColumn(m, 1);
            newColumn[i][0] = 1;
            a.appendColumn(newColumn);
            d[i][0] = 0;  // 约束变为等式
            continue;
        }
    }
}
```

标准化操作的实现位于 `Solver.cpp` 文件的 `normalize` 函数中，目的是使得等式右侧非负，且变量的约束都大于等于0。

程序逻辑如下：

- 逐一检查每个约束，查看其右侧，如果已经非负则不作处理，否则将该行取反。
- 逐一检查每个变量，分为三种情况：
  - 该变量大于等于0，这正是我们想要的，跳过。
  - 该变量小于等于0，这不是标准形式，需要做以下替换，令 $x_i = -x_i'$。同时将该变量记录下来，将其下标i push到 `negative` 的末尾。
  - 该变量无限制，这也是标准形式，需要做以下替换，令 $x_i = x_i' - x_j'$。不仅需要新增一个变量 x_j'，同时还要将这一对变量下标 (i, j) 记录下来，将其下标push到 `noConstraints` 的末尾。

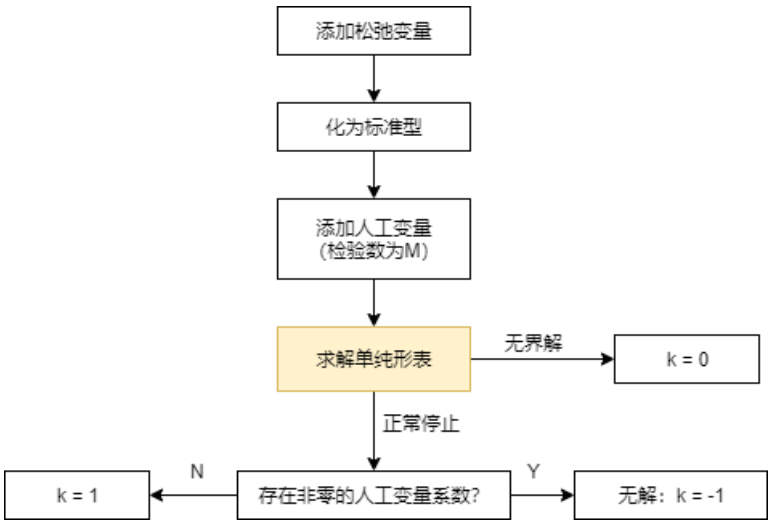
对被替换的变量下标进行记录是为了在求解后，能够利用所记录的下标数据，恢复初始的变量值。

该部分代码如下：

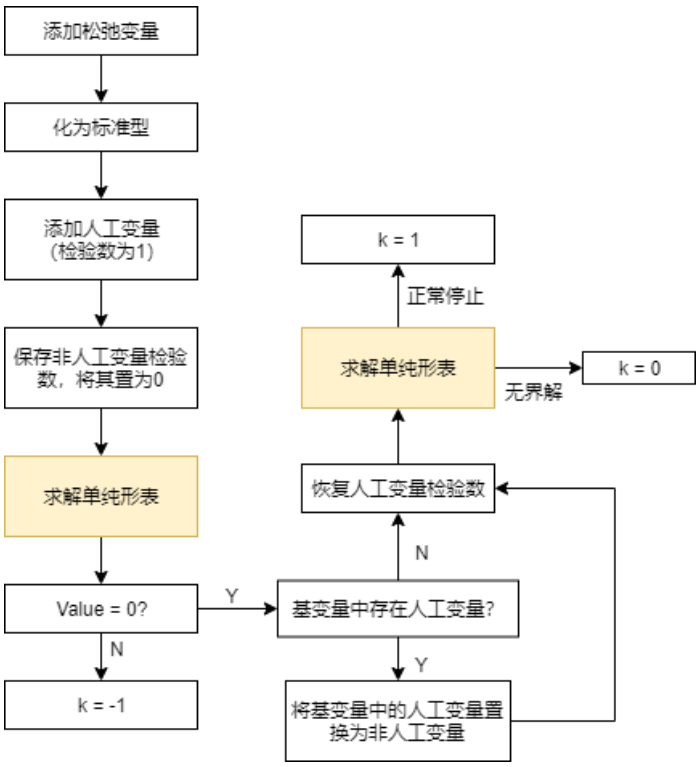
```
void Solver::normalize() {
    Matrix one = Matrix(1, 1, 1);
    relax();  // 进行松弛
    for (int i = 0; i < m; i++) {
        if (b[i][0] >= 0) continue;  // 约束本身就满足条件，跳过
        // 不满足条件的约束，对一整行取反
        b[i][0] *= -1;
        for (int j = 0; j < n; j++) {
            a[i][j] *= -1;
        }
    }
    for (int j = 0; j < n; j++) {
        if (equal(e[0][j], 1)) continue;  // 变量本身满足非负，跳过
        // 处理小于等于0的变量，用其相反变量取代
        if (equal(e[0][j], -1)) {
            c[0][j] *= -1;
            e[0][j] = 1;
            for (int i = 0; i < m; i++) {
                a[i][j] *= -1;
            }
            negative.push_back(j);
            continue;
        }
        // 处理无限制的变量，用一对变量的差取代
        if (equal(e[0][j], 0)) {
            n++;
            c.appendColumn(one);
            c[0][n - 1] = c[0][j] * -1;
            e[0][j] = 1;
            e.appendColumn(one);
            Matrix newColumn = a.getColumn(j) * -1;
            a.appendColumn(newColumn);
            noConstraints.emplace_back(j, n - 1);
            continue;
        }
    }
}
```

2. 大M法求解单纯性表

本次实验同时实现了大M法和两阶段法。其中大M法的算法流程图如下所示：

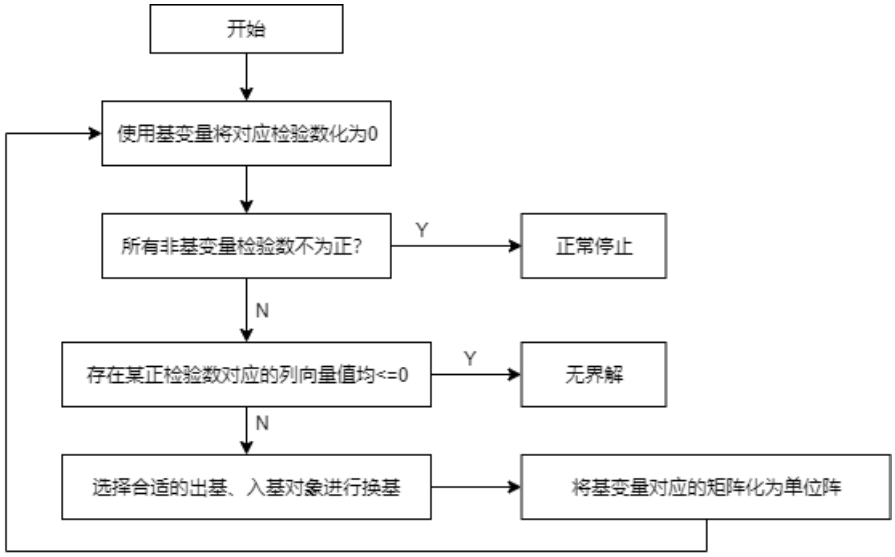


两阶段法的过程较为复杂，流程图如下所示：



这两种方法都调用了同一个模块“求解单纯形表”（见黄底部分）。该模块是单纯形法的核心模块，具有两个终止状态：无界解和正常停止。注意正常停止并非表示这是有限最优解，这需要顶层模块进行进一步判断。

求解单纯形表的算法流程图如下：



大M法的实现位于 `BigMSolver` 类中，两阶段法的实现位于 `DoubleStageSolver` 中。两个类的头文件基本一致，定义如下：

```
class VanillaSimplexSolver : public Solver {
private:
    // 存放单纯形表的当前结果
    double value = 0;
    // 存放非人工变量的个数
    int nonManualVariableCount = 0;
    // 存放第i条约束对应的基变量序号
    vector<int> baseIndex;

    // 置换基变量的方法
    void exchange(int inIndex, int outIndex);
```

```
public:
    VanillaSimplexSolver(int n, int m, Matrix c, Matrix a, Matrix b, Matrix d, Matrix e);

    ~VanillaSimplexSolver();

    // 求解方法
    void solve(int &k, double &y, Matrix &x) override;

    // 调试方法，用于打印单纯形表
    void printSimplexTable();

};
```

各个部分的用途见注释。该类是 `Solver` 类的子类，因此存放了 `a` 、 `b` 、 `c` 、 `e` 等数据。

算法开始时，将为每条约束都添加一个人工变量，同时将对应的检验数置为 `M` 。核心部分代码如下：

```
nonManualVariableCount = n;
// add artificial variables
for (int i = 0; i < m; i++) {
    a.appendColumn(zero);
    a[i][n] = 1;
    c.appendColumn(M);
    e.appendColumn(one);
    baseIndex.push_back(n);
    n++;
}
```

由于实验中的参考资料大部分是用于求解max型的线性规划，因此此处也需要将检验数取反，将min化为max。采用这种设计时 `value` 的最终值即为目标函数值（无需取反）。代码略。

之后，需要使用初等行变换，将基变量对应的检验数化为0。显然基变量全为人工变量，其对应的检验数是$-M$（已取反），因此将每一行乘以M加至检验数即可。代码如下：

```
// normalize checked number of mannual variables
for (int i = 0; i < m; i++) {
    auto rowi = a[i];
    checkedArray.addRow(rowi, M);
    double delta = M * b[i][0];
    value += delta;
}
```

之后就是求解单纯形表。求解单纯形表的部分位于一个 `whilte(true)` 循环内。

循环开始时，首先遍历每一个检验数，找到大于0的检验数。之后，首先判断该检验数对应的列向量是否全部小于等于0。若是，则输出无界解；否则，对比目前的最大检验数值并更新。代码如下：

```
// get max
int inIndex = -1;
double maxCheckedNum = 0;
for (int i = 0; i < n; i++) {
    double num = checkedArray[i];
    if (num > 0 && !equal(num, 0)) {
        // judge unbound
        bool flag = true;
        for (int j = 0; j < m; j++) {
            double aji = a[j][i];
```

```
            if (aji > 0 && !equal(aji, 0)) {
                flag = false;
                break;
            }
        }
        if (flag) {
            // All numbers in column i are <= 0: unbound!
            k = 0;
            return;
        }
        // get max index
        if (num > maxCheckedNum && !equal(num, maxCheckedNum)) {
            inIndex = i;
            maxCheckedNum = num;
        }
    }
}
```

为避免浮点运算精度问题，此处大量使用了 `equal` 浮点比较方法。采用按从小到大的序号扫描方式同时也可以解决潜在的**退化**问题。

之后，倘若所有检验数都为负，则停止迭代过程。大M法中此处同时判断了是否有解。代码如下：

```
if (inIndex == -1) {
    // all of checked numbers are <= 0
    // judge manual variables
    for (int i = 0; i < m; i++) {
        if (baseIndex[i] >= nonManualVariableCount && b[i][0] != 0) {
            // no feasible solution!
            k = -1;
            return;
        }
    }
    // output answer
    k = 1;
    for (int i = 0; i < m; i++) {
        int base = baseIndex[i];
        x[0][base] = b[i][0];
    }
    y = value;
    return;
}
```

若以上条件都不成立，算法将找到合适的outIndex作为出基对象。代码如下：

```
int outIndex = -1;
int minOutIndex = INT_MAX;
double minRatio = 0;
for (int i = 0; i < m; i++) {
    double aik = a[i][inIndex];
    if (aik <= 0 || equal(aik, 0)) continue;
    double ratio = b[i][0] / aik;
    if (outIndex == -1 || minRatio > ratio || (equal(minRatio, ratio) && baseIndex[i] <
minOutIndex)) {
        minRatio = ratio;
        outIndex = i;
        minOutIndex = baseIndex[i];
    }
}
// outIndex == -1 is impossible, due to this case is unbound!
```

```
exchange(inIndex, outIndex);
```

同样为了避免退化问题，此处用了minOutIndex来记录最小的同值index。使用 `equal` 函数同样解决了浮点精度问题。

最终，算法调用了exchange函数并进入下一循环。exchange用于更换基操作，核心代码如下所示：

```
void VanillaSimplexSolver::exchange(int inIndex, int outIndex) {
    baseIndex[outIndex] = inIndex;
    Row pivot = a[outIndex];
    Row pivotB = b[outIndex];
    // normalize to one
    double toOneRatio = 1 / a[outIndex][inIndex];
    pivot.multi(toOneRatio);
    pivotB.multi(toOneRatio);
    // normalize other rows to zero
#ifdef PARALLEL
#pragma omp parallel for default(none) shared(pivot) shared(pivotB) shared(inIndex)
shared(outIndex)
#endif
    for (int i = 0; i < m; i++) {
        if (i != outIndex) {
            Row another = a[i];
            Row anotherB = b[i];
            double ratio = -another[inIndex];
            another.addRow(pivot, ratio);
            anotherB.addRow(pivotB, ratio);
        }
    }
    // transform checked number to zero
    Row cRow = c[0];
    double ratio = -cRow[inIndex];
    cRow.addRow(pivot, ratio);
    value += pivotB[0] * ratio;
}
```

代码首先将当前行向量归一化（使得新基值为1），之后对矩阵进行遍历的行变换操作，保证基向量对应矩阵为单位阵。最终，利用该单位阵将基向量的检验数化为0。

为了加速代码运行速度，在化为单位阵的部分使用了OpenMP库实现了并行化算法。

3. 两阶段法求解单纯性表

两阶段法的开始部分与大M法相似，区别在于此处检验数取1。同时也对检验数取反，表示内部计算用的是max系统。该部分代码忽略。

在第一阶段，首先记录下检验数列表，并将非人工变量的检验数置为0。之后使用基变量（人工变量）进行初等行变换，更新对应的检验数为0。代码如下：

```
// 1st stage: min sum of artificial variables
Matrix oldCheckedMat(c);
for (int i = 0; i < nonManualVariableCount; i++) {
    checkedArray[i] = 0;
}
// normalize checked number of artificial variables
for (int i = 0; i < m; i++) {
    auto rowi = a[i];
    checkedArray.addRow(rowi, 1);
    double delta = b[i][0];
    value += delta;
}
```

之后调用了 `solveInternal()` 函数求解单纯形表。求解代码与大M法基本一致，此处略去介绍。

接着代码做了两个判断：判断 `value` 是否为0、判断是否有人工变量在基变量中。前者直接返回 `k = 1` ，后者则调用 `exchange` 函数进行换基操作。代码如下：

```
if (!equal(value, 0)) {
    k = -1;
    return;
}
// check if there is an artificial variable in base
for (int i = 0; i < m; i++) {
    if (baseIndex[i] >= nonManualVariableCount) {
        for (int j = 0; j < nonManualVariableCount; j++) {
            if (!equal(checkedArray[j], 0)) {
                exchange(j, i);
                break;
            }
        }

    }
}
```

接着，对检验数进行恢复（代码略），进入第二阶段。因为检验数发生了改变，该阶段首先使用初等行变换，将基变量的检验数重新置为0。代码如下：

```
for (int i = 0; i < m; i++) {
    int base = baseIndex[i];
    double baseCheckedNum = checkedArray[base];
    if (baseCheckedNum != 0) {
        auto currentRow = a[i];
        checkedArray.addRow(currentRow, -baseCheckedNum);
        value -= b[i][0] * baseCheckedNum;
    }
}
```

最后在调用了一次 `solveInternal` 函数，判断是否为无界解、有限解。代码如下：

```
// solve!
bool notUnbound = solveInternal();
if (!notUnbound) {
    k = 0;
    return;
}
// output answer
k = 1;
for (int i = 0; i < m; i++) {
    int base = baseIndex[i];
    x[0][base] = b[i][0];
}
y = value;
```

4. 对偶单纯形法

对偶单纯形法的实现位于 `dualSimplexSolver.cpp` 。

其中， `solve` 函数是主要的求解函数，首先检查各检验数是否保证非负，否则认为不适合用对偶单纯形法。

```
// check if dual is feasible
for (auto j = 0; j < n; j++) {
    if (c[0][j] < -EPS) {
        k = -2;
        return;
    }
}
```

接着将问题转化为最大化问题（其实不是特别有必要，只是为了更符合直觉），对检验数取反。

检查倒数 m 个变量，在满足前面提到的三个条件的情况下，该 m 个变量可以作为一组基，且检验数为0。如果遇到某个检验数不为0，同样认为不适合用对偶单纯形法。

将剩下的变量加入非基变量中。

```
// make it a max problem
auto constrains = c[0];
constrains.multi(-1.0);
// the problem is relaxed and normalized
for (auto j = n - m; j < n; j++) {
    if (equal(constrains[j], 0.0)) {
        // if it's slack variable
        baseIndices.push_back(j);
    } else {
        k = -2;
        return;
    }
}
for (auto j = 0; j <= n - m - 1; j++) {
    verifyIndices.push_back(j);
}
for (auto i = 0; i < m; i++) { // for each row
    // reverse the row
    if (equal(a[i][n - m + i], -1.0)) {
        auto rowA = a[i];
        auto rowB = b[i];
        rowA.multi(-1.0);
        rowB.multi(-1.0);
    }
```

```
}
```

接着进入循环部分，每次挑选一个出基变量，一个入基变量。

出基变量挑选的标准是选择绝对值最大且小于0的 b_i 对应的变量，若找不到这样的变量，说明已经原问题可行，同时对偶也可行，均为最优解。

入基变量挑选的标准是在系数为负数的变量中挑选，保证在对应检验数变为0后，其他检验数要全部小于等于0（注意问题已经转为最大化）。若没有系数为负数的变量，则原线性规划无解。

当出基变量和入基变量都选好后，调用 `pivot` 函数，替换基变量，更新单纯形表。

```
while (true) {
    // find the base
    int outBaseIndex = -1;
    int inBaseIndex = -1; // index in verifyIndices vector
    double tb = 0.0;
    int minOutBaseIndex = -1;
    for (auto i = 0; i < m; i++) {
        // always return the smallest one that is smaller than 0
        if (GRE(tb, b[i][0]) || (equal(b[i][0], tb) && baseIndices[i] < minOutBaseIndex)) {
            tb = b[i][0];
            outBaseIndex = i;
            minOutBaseIndex = baseIndices[i];
        }
    }
    if (outBaseIndex == -1) {
        // no negative value in b
        // It's the optimal solution ?
        k = 1;
        y = targetValue;
        for (auto i = 0; i < m; i++) {
            x[0][baseIndices[i]] = b[i][0];
        }
        return;
    }
    // at least one smaller than 0
    // find the smallest absolute ratio
    double ratio = 0;
    int minInBaseIndex = -1;
    for (auto j = 0; j < verifyIndices.size(); j++) {
        // check A_i
        if (GEQ(a[outBaseIndex][verifyIndices[j]], 0.0)) continue;
        double nRatio = std::abs(c[0][verifyIndices[j]] / a[outBaseIndex]
[verifyIndices[j]]);
        if (inBaseIndex == -1 || GRE(ratio, nRatio) ||
            (equal(nRatio, ratio) && verifyIndices[j] < minInBaseIndex)) {
            inBaseIndex = j;
            ratio = nRatio;
            minInBaseIndex = verifyIndices[j];
        }
    }
    if (inBaseIndex == -1) {
        // A_i >= 0
        k = -1;
        return;
    }
    // the inBase index is found, swap and update
    pivot(outBaseIndex, inBaseIndex);
}
```

`pivot` 函数实现如下：

```
void DualSimplexSolver::pivot(int outBaseIndex, int inBaseIndex) {
    // update the rows
    double scaleRatio = 1 / a[outBaseIndex][verifyIndices[inBaseIndex]];
    // if want parallel update out base row in advance
    auto rowA = a[outBaseIndex];
    auto rowB = b[outBaseIndex];
    rowA.multi(scaleRatio);
    rowB.multi(scaleRatio);

    for (auto i = 0; i < m; i++) {
        if (i != outBaseIndex) {
            auto anotherA = a[i];
            auto anotherB = b[i];
            double ratio = -anotherA[verifyIndices[inBaseIndex]];
            anotherA.addRow(rowA, ratio);
            anotherB.addRow(rowB, ratio);
        }
    }

    // update the constrains
    auto rowC = c[0];
    double ratio = -rowC[verifyIndices[inBaseIndex]];
    rowC.addRow(rowA, ratio);
    targetValue += ratio * rowB[0];
    // swap the basis
    int temp = verifyIndices[inBaseIndex];
    verifyIndices[inBaseIndex] = baseIndices[outBaseIndex];
    baseIndices[outBaseIndex] = temp;
}
```

五、测试结果

1. 测试用例说明

测试数据主要来自著名的线性规划问题的测试集 NETLIB，从其中选取一些规模符合要求的数据集，并预处理为本实验要求的输入格式。选取的数据集规模、特性和最优目标函数值展示如下：

Name	Rows	Cols	Nonzeros	Bytes	BR	Optimal Value
ADLITTLE	57	97	465	3690		2.2549496316E+05
AFIRO	28	32	88	794		-4.6475314286E+02
AGG3	517	302	4531	32570		1.0312115935E+07
BLEND	75	83	521	3227		-3.0812149846E+01
BRANDY	221	249	2150	14028		1.5185098965E+03
SC50A	51	48	131	1615		-6.4575077059E+01
SC50B	51	48	119	1567		-7.0000000000E+01
SHARE1B	118	225	1182	8380		-7.6589318579E+04
SHARE2B	97	79	730	4795		-4.1573224074E+02
STOCFOR1	118	111	474	4247		-4.1131976219E+04

在此基础上，加入了一些小规模的测试集，用于调试分析，另设计了一些特殊情形的数据，用于测试我们的实现能否正确处理退化、无界解、多重最优解和无解的情况。除以上来自 NETLIB 的数据，其余数据的规模都显示在下文给出的测试结果中。

需要注明的是，本实验中作为基准的最优目标函数值，由目前较流行的用于线性规划问题求解的优化器 Gurobi 的 Python API 求解得到。

另由于对偶单纯形法限制求解 c 非负和约束类型为 1 （大于等于）的问题，对上述数据进行修改，在保证数据规模的情况下修改约束类型，后使用 Gurobi 求解验证，保留有有限可行解的数据集用于对偶单纯形的测试。

单纯形的测试数据置于 ./data/input 和 ./data/solution 下，对偶单纯形的测试数据置于 ./data/input-dual 和 ./data/solution-dual 下。

下面给出用于测试四种特殊情形的数据：

退化情形

$$
\begin{aligned}
\text{Maximize} \quad & 2x_1 + x_2 \\
\text{Subject to:} \quad & 4x_1 + 3x_2 \leq 12 \\
& 4x_1 + x_2 \leq 8 \\
& 4x_1 + 2x_2 \leq 8 \\
& x_1, x_2 \geq 0
\end{aligned}
$$

```
2 3
-2 -1
4 3 12 -1
4 1 8 -1
4 2 8 -1
1 1
```

该情形中，第 2 个约束实际上是多余的，出现了退化，在单纯形法求解的过程中可能出现迭代计算的循环。

无界解情形

$$
\begin{aligned}
\text{Maximize} \quad & 2x_1 + x_2 \\
& x_1 - x_2 \leq 10 \\
\text{Subject to:} \quad & 2x_1 - x_2 \leq 40 \\
& x_1, x_2 \geq 0
\end{aligned}
$$

```
2 2
-2 -1
1 -1 10 -1
2 -1 40 -1
1 1
```

该情形中，对任意非负 δ, 解 $(x_1, x_2, s_1, s_2) = (30 + \delta, 20 + 2\delta, \delta, 0)$ 都是可行解，因此无有限最优解。

无穷多解情形

$$
\begin{aligned}
\text{Maximize} \quad & 4x_1 + 14x_2 \\
& 2x_1 + 7x_2 \leq 21 \\
\text{Subject to:} \quad & 7x_1 + 2x_2 \leq 21 \\
& x_1, x_2 \geq 0
\end{aligned}
$$

```
2 2
-4 -14
2 7 21 -1
7 2 21 -1
1 1
```

该情形中，目标函数平行于非冗余的紧约束，最优目标函数值 42 可以在一个线性区间内取到，即，最优解可以表示为

$$(x_1, x_2) = \delta \times (0, 3) + (1 - \delta) \times (7/3, 7/3)$$

解数量无穷多。

无解情形

$$\begin{aligned} \text{Maximize} \quad & x_1 + x_2 \\ \text{Subject to:} \quad & x_1 + 2x_2 \leq 8 \\ & 3x_1 + 2x_2 \leq 12 \\ & 1x_1 + 3x_2 \geq 13 \\ & x_1, x_2 \geq 0 \end{aligned}$$

```
2 3
-1 -1
1 2 8 -1
3 2 12 -1
1 3 13 1
1 1
```

该情形中，约束间存在冲突使不存在基本可行解。

2. 测试结果

单纯形 - 大M法

测试中使用 $M = 10^6$。

```
Name       #Vars     #Cstrs    Optimal             Result              Error     Time

adlittle   97        56        225494.963162       225495              0.000000  78
afiro      32        27        -464.753143         -464.753            0.000000  34
agg3       302       516       10312115.935089     1.03121e+07         0.000002  6043
blend      83        74        -30.81215           -30.8121            0.000002  77
brandy     249       220       1518.509896         1518.51             0.000000  627
degeneracy2          3         -4                  -4                  0.000000  62
emptyest   11        16        Unbounded           Unbounded           NaN       36
example    10        12        3.236842            3.23684             0.000001  36
infeasible2          3         Infeasible                              NaN       40
maros      5         7         128.333333          128.333             0.000003  34
muloptimal2          2         -42                 -42                 0.000000  34
nazareth   3         3         Unbounded                               NaN       30
nexample   4         5         7.0                 7                   0.000000  35
sc50a      48        50        -64.575077          -64.5751            0.000000  42
sc50b      48        50        -70.0               -70                 0.000000  74
share1b    225       117       -76589.318579       -76589.3            0.000000  254
share2b    79        96        -415.732241         -415.732            0.000001  183
stocfor1   111       117       -41131.976219       -41132              0.000001  143
testprob   3         6         54.0                54                  0.000000  39
unbounded  2         2         Unbounded                               NaN       30
```

单纯形 - 两阶段法

```
Name       #Vars     #Cstrs    Optimal             Result              Error     Time
```

```
adlittle  97       56       225494.963162       225494.963162       0.000000  82
afiro     32       27       -464.753143         -464.753143         0.000000  55
agg3      302      516      10312115.935089     10312115.935089     0.000000  5376
blend     83       74       -30.81215           -30.812150          0.000000  110
brandy    249      220      1518.509896         1518.509896         0.000000  1097
degeneracy2        3        -4                  -4.000000           0.000000  58
emptyest  11       16       Unbounded                               NaN       31
example   10       12       3.236842            3.236842            0.000000  33
infeasible2        3        Infeasible                              NaN       37
maros     5        7        128.333333          128.333333          0.000000  32
muloptimal2        2        -42                 -42.000000          0.000000  35
nazareth  3        3        Unbounded                               NaN       29
nexample  4        5        7.0                 7.000000            0.000000  36
sc50a     48       50       -64.575077          -64.575077          0.000000  45
sc50b     48       50       -70.0               -70.000000          0.000000  73
share1b   225      117      -76589.318579       -76589.318579       0.000000  308
share2b   79       96       -415.732241         -415.732241         0.000000  156
stocfor1  111      117      -41131.976219       -41131.976219       0.000000  179
testprob  3        6        54.0                54.000000           0.000000  55
unbounded 2        2        Unbounded                               NaN       30
```

对偶单纯形

```
Name      #Vars     #Cstrs    Optimal             Result              Error     Time

adlittle  97       56       886684.636819       886684.636819       0.000000  75
afiro     32       27       0.0                 0.000000                      60
agg3      302      516      Infeasible                              NaN       1003
bandm     472      305      515.893222          517.443695          0.003005  1207
emptyest  11       6        0.0                 0.000000            0.000000  75
example   10       5        0.0                 0.000000            0.000000  46
maros     5        3        134.444444          134.444444          0.000000  31
nazareth  3        2        4.0                 4.000000            0.000000  39
nexample  4        3        2.0                 2.000000            0.000000  35
sc50a     48       50       74.666667           74.666667           0.000000  56
sc50b     48       50       70.0                70.000000           0.000000  44
stocfor1  111      117      32711.883383        32711.883383        0.000000  107
testprob  3        3        68.0                68.000000           0.000000  45
```

由以上测试数据可以看出，三种实现在我们的数据集上基本都达到了小于等于 0.1^4 的误差目标，一般规模的问题都能在 1 秒内给出解，对其中最大的一份数据 agg3（规模为302个决策变量，516个约束），两种实现的单纯形法都能在 5-6 秒给出正确解；对我们设计的几个特殊情况的测试集，也都能正确快速地完成处理。

为比较对偶单纯形和单纯形两阶段法在运行效率和精度上的不同，使用修改约束后的对偶单纯形数据用单纯形进行测试，结果如下：

```
Name      #Vars     #Cstrs    Optimal             Result              Error     Time

adlittle  97       56       886684.636819       886684.636819       0.000000  82
afiro     32       27       0.0                 0.000000            0.000000  100
agg3      302      516      Infeasible                              NaN       3052
bandm     472      305      515.893222                                        1534
emptyest  11       6        0.0                 0.000000            0.000000  54
example   10       5        0.0                 0.000000            0.000000  35
maros     5        3        134.444444          134.444444          0.000000  36
nazareth  3        2        4.0                 4.000000            0.000000  31
nexample  4        3        2.0                 2.000000            0.000000  29
sc50a     48       50       74.666667           74.666667           0.000000  41
sc50b     48       50       70.0                70.000000           0.000000  72
```

```
stocfor1  111       117       32711.883383       32711.883383       0.000000  143
testprob  3         3         68.0               68.000000          0.000000  99
```

发现双方在一定规模的数据上都能达到较高的正确率和精度，运行时间差别不大，但在 bandm 数据集上单纯形法出现了误判无解的情况，其原因可能是较多次数的迭代后浮点数 rounding 带来的舍入误差累积，体现在约束条件的判定上，导致误判，类似的情况也出现在其他一些更大的测试数据集上测试单纯形和对偶单纯形时。

六、分工说明

成员	分工
沈心逸	总体设计、松弛和标准化、测试、报告撰写
林柯舟	对偶单纯形法、报告撰写
陈谋祥	大M法和两阶段法、报告撰写
王彦皓	数据生成、测试、报告撰写