Multifrontal 算法原理及实现

许中兴
PLCT 实验室
智能软件研究中心
中国科学院软件研究所
xuzhongxing@iscas.ac.cn

1 本文背景介绍

稀疏矩阵方程求解是工程计算中的一个重要的需求，大量的工程问题最终都转化为一个大型线性方程组。这个线性方程组的系数矩阵往往是稀疏的。

我在开发机械臂离线仿真软件过程中，需要对多体动力学系统进行求解。针对多体动力学的建模将在其他文章中介绍。本文只介绍建模形成的对称矩阵的求解，采用的是多波前 (multifrontal) 方法。由于关于数值线性代数以及专门介绍稀疏矩阵方法的教材和文章很多，所以本文不会追求对各种方法的全面覆盖，而是着重介绍我在解决手头的应用需求中所使用的具体方法以及代码实现经验。

我的专业背景是软件工程师，不是数值方法的专家，所以本文关注的角度更多的是算法的具体实现，可能对于一些偏数学的分析理解不够深入。但因为所介绍的算法已经实现在了实际运行的软件系统当中，所以整体的正确性是有保障的，对算法的理解也会随着开发和测试的进行不断深入。

由于算法有很多种选择，甚至算法的每个环节都有很多种做法，我们在应用的时候往往会陷入选择困难。我的经验是不必过多的去进行选择，可以先实现一个基本的完整的流程，然后针对每个环节实现多种方法，在应用的过程中对各种方法进行实际测试，从而获得每种方法在不同问题上的实际效果的认知。

本文对前置知识的要求是学过线性代数和 C++ 语言编程。

2 线性方程组求解的基本方法

2.1 高斯消去法

我们拿到一个线性方程组，最基本的求解方法就是高斯消去法。高斯消去法的步骤是依次用对角线上的元素将它下面的元素消成 0，最后矩阵变成一个上三角矩阵，就可以用代入法从最后一个变量开始向上代入，解出所有的变量。对角线元素如果为 0 或者太小，则可以通过 pivoting 操作来选择合适的元素作为新的对角线元素。这里我们为了不干扰主线叙述，不讨论 pivoting 相关的技巧。

这也是我们在实现大型复杂算法时采用的方式：一开始实现的是一个尽可能简单的但又完整的 base line 版本，然后根据实际问题需求一步一步的增加优化技巧。这样的好处是便于调试，base line 版本因为简单，所以容易做对，后续哪一步引入了 bug，都可以及时回退，对比，找出 bug。

以一个 2x2 的矩阵为例来说明高斯消去法。假设方程矩阵形式是：

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

我们将系数矩阵的第 1 行乘上 $-\frac{c}{a}$，加到第 2 行上，得到新的系数矩阵：

$$\begin{bmatrix} a & b \\ & d - b \cdot \frac{c}{a} \end{bmatrix}$$

方程的右手边也做相应的变换，这里略过。

这里乘上的因子 $\frac{c}{a}$ 中分母上的项 a 称为 pivot。它的选择也有很多讲究。参考文献中有详细的讨论。

2.2 反向替换

当方程的左手边变成一个（上）三角矩阵之后，就可以通过（反向）替换法得到方程的解。

对于上面的例子，通过观察，可以知道最后一个方程中只有一个未知变量，它的解可以立刻得到，即将方程的右手边除以变量前面的因子 $d - b \cdot \frac{c}{a}$。之后将已知变量的解代入上一行的方程，上一行的方程也只剩下一个未知变量。依次替换，可以得到全部的解。

2.3 矩阵分解的观点

高斯消去法是我们在手算的时候采用的步骤。对于算法求解方程来说，更好的观点是用矩阵分解的视角来看待求解的过程。

上一节的操作我们写成矩阵相乘的形式是：

$$\begin{bmatrix} 1 & \\ -\frac{c}{a} & 1 \end{bmatrix} \begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} a & b \\ & d - b \cdot \frac{c}{a} \end{bmatrix}$$

将左手边的下三角矩阵换到右手边，得到矩阵的分解形式：

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} 1 & \\ \frac{c}{a} & 1 \end{bmatrix} \begin{bmatrix} a & b \\ & d - b \cdot \frac{c}{a} \end{bmatrix}$$

写成矩阵符号：

$$A = LU$$

其中，L 是下三角矩阵，U 是上三角矩阵。有了 L 和 U 这 2 个三角矩阵，就可以通过求解 2 个三角矩阵方程得到原始方程的解了：

$$L \underbrace{Ux}_{y} = b$$

$$Ly = b$$

$$Ux = y$$

3 对称矩阵的分解

在工程应用中，得到的方程系数矩阵往往是对称的。所以我们考察对称矩阵的分解。仍以一个 2x2 的对称矩阵为例。这里多说一句，当我们在学习数学的时候，用手计算一个简单的具体的例子非常有助于我们理解问题和算法。如果不动手计算例子，只是跟着书上的抽象符号走，走不了多远就迷路了。

设对称矩阵为：

$$\begin{bmatrix} a & b \\ b & c \end{bmatrix}$$

进行一步高斯消去：

$$\begin{bmatrix} 1 & \\ -\frac{b}{a} & 1 \end{bmatrix} \begin{bmatrix} a & b \\ b & c \end{bmatrix} = \begin{bmatrix} a & b \\ & c - b \cdot \frac{b}{a} \end{bmatrix}$$

通过观察可以发现，如果我们把用于消去的下三角矩阵的转置放到右边再乘一次，就可以把原矩阵消成对角矩阵：

$$\begin{bmatrix} a & b \\ & c - b \cdot \frac{b}{a} \end{bmatrix} \begin{bmatrix} 1 & -\frac{b}{a} \\ & 1 \end{bmatrix} = \begin{bmatrix} a & \\ & c - b \cdot \frac{b}{a} \end{bmatrix}$$

写乘矩阵分解的样子就是：

$$\begin{bmatrix} a & b \\ b & c \end{bmatrix} = \begin{bmatrix} 1 & \\ \frac{b}{a} & 1 \end{bmatrix} \begin{bmatrix} a & \\ & c - b \cdot \frac{b}{a} \end{bmatrix} \begin{bmatrix} 1 & \frac{b}{a} \\ & 1 \end{bmatrix}$$

用矩阵符号表示：

$$A = LDL^\top$$

这就是矩阵的 LDL 分解。如果把 D 矩阵的对角线元素开平方之后分解到 L 中去，并且原矩阵是正定的，就得到 Cholesky 分解。我们在实际求解中使用 LDL 分解，因为可以避免求平方根的操作，并能节省一次除法。

有了矩阵的 LDL 分解，就可以通过求解 2 个三角矩阵方程和 1 个对角矩阵方程得到原方程的解：

$$L \underbrace{D \overbrace{L^\top x}^{y}}_{z} = b$$

$$Lz = b$$

$$Dy = z$$

$$L^\top x = y$$

4 稀疏矩阵概述

从工程问题中得到的方程系数矩阵有很多情况下是稀疏的，意即矩阵元素中只有很少的一部分不是 0。在这种情况下，如果还按照前面说的方式去分解矩阵，会做很多无用的计算。所以稀疏矩阵的计算需要增加额外的处理，包括以下几个方面：

- 稀疏数据结构及相关操作
- 变量排序
- 专门的分解算法，包括直接法和迭代法

对稀疏矩阵求解的全面讨论见参考文献。这里我们采用的是 multifrontal 算法。Multifrontal 算法是一种针对对称矩阵的 LDL 分解算法。Multifrontal 算法通过构建 elimination tree 数据结构，在矩阵的分解操作的每一步都只考虑这一步会涉及的元素，从而转化成对一个稠密矩阵的操作。每一步分解操作计算出最终分解矩阵的一列。最终得到完整的 L 和 D 矩阵。

Multifrontal 算法的另一个好处是它很适合并行化。根据构建出来的 elimination tree，可以很自然的对 multifrontal 算法进行并行执行。

Multifrontal 算法的主要步骤如下：

1. 构建 elimination tree
2. 分解矩阵得到 L 和 D
3. 求解 $LDL^\top x = b$

5 变量排序

变量重排序 (variable reordering) 的作用主要有以下 2 个。

- 减少矩阵分解结果中的填充 (fill-in)，带来的好处是降低存储需求以及提高计算速度
- 改变 elimination tree 的形状，提高并行度
- 让 pivot 的数值特性更好（不要除以一个很小的数）

我要求解的动力学问题所生成的矩阵用变量原始排序本身的性能也不差，如果用 minimum degree 这样的排序方法得到的排序会用一个很小的数作为 pivot，还需要引入别的技巧来修复这个问题，所以我还没有实现 reordering 功能，对它的细节和效果还没有实践认知。在这里先跳过它的讲解。

在未来实现算法并行化的时候，会考虑采用 nested bisection 方法对变量进行排序。

6 单步 LDL 分解

假设我们有一个对称矩阵要做一步 LDL 分解：

$$\begin{bmatrix} a & v^\top \\ v & C \end{bmatrix}$$

其中，a 是一个元素，v 是列向量，C 是右下角的子矩阵。分解公式为：

$$\begin{bmatrix} a & v^\top \\ v & C \end{bmatrix} = \begin{bmatrix} 1 & \\ \underbrace{\frac{1}{a}v}_{L_1} & I \end{bmatrix} \begin{bmatrix} \overbrace{a}^{D_1} & \\ & C - \frac{1}{a}vv^\top \end{bmatrix} \begin{bmatrix} 1 & \frac{1}{a}v^\top \\ & I \end{bmatrix}$$

这里面包含了 L 的第一列：

$$\begin{bmatrix} 1 \\ \frac{1}{a}v \end{bmatrix}$$

以及 D 的第一个元素 a。

我们先定义 2 个术语。当前要分解的矩阵称为 a 的 frontal 矩阵。对 frontal 矩阵进行一步分解之后形成的右下角矩阵 $C - \frac{1}{a}vv^\top$ 称为元素 a 的 update 矩阵，因为这个矩阵之后要被用来形成新的 frontal 矩阵。a 的 update 矩阵本质上是 a 在做高斯消去时形成的对它右下角的元素的副作用。

如果矩阵是稠密的，那么我们只需要依次取对角线元素和它对应的行和列，合并上它之前的对角线元素的 update 矩阵，然后进行一步分解，得到 L 的列和 D 的对角线元素，以及它的 update 矩阵。

但是如果矩阵是稀疏的，对于一个对角线元素 a 来说，它之前的对角线元素不一定对它的分解有影响。前面哪些对角线元素的 update 矩阵需要用来形成它的 frontal 矩阵呢？这就需要构造 elimination tree 来确定了。

7 构建 elimination tree

下面的矩阵如果没有特殊说明，都假设为对称矩阵。并且假设对角线元素均不为 0。在实际工程问题中，这样的条件并不难满足。为了表示的清晰和简洁，我们只写出矩阵的下三角部分。

考虑一个稀疏对称矩阵：

$$\begin{bmatrix} a & & & \\ & b & & \\ e & & c & \\ & f & g & d \end{bmatrix}$$

我们仔细观察在用 a 和 b 做高斯消去的时候发生了什么。在用 a 做高斯消去的时候，我们消去的是 e，影响的是 e 所在的行。在用 b 做高斯消去的时候，我们消去的是 f，影响的是 f 所在的行。第二句话的前提是，用 a 做高斯消去的时候没有在 b 这一列引入新的非 0 元素，称为 fill-in。如果有 fill-in，则需要以当时的矩阵的样子为准来计算。

也就是说我们用 a 做高斯消去的时候，如果矩阵变成：

$$\begin{bmatrix} a & \\ e & c \end{bmatrix}$$

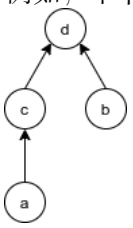
的样子对计算是没有影响的。

也就是说，对于元素 c 来说，只有 a 元素的 update 矩阵对 c 的 frontal 矩阵才有影响。因为 a 所在的列当中 c 对应的行处有一个不为 0 的元素 e，a 需要去把这个元素消为 0。这里可以看出，a 的 update 矩阵本质上是 a 在做高斯消去时形成的对矩阵剩下的元素的副作用。

Elimination tree 的含义是：子节点的 update 矩阵会对父节点的 frontal 矩阵产生影响。下面给出 elimination tree 的定义。

Elimination tree 以矩阵的对角线元素为节点。对于每个对角线元素 a，在高斯消去得到的当前矩阵中，它所在的列的对角线以下部分中第一个不为 0 的元素所在的行所对应的对角线元素就是 a 的父节点。

例如，本节中的 4x4 矩阵的 elimination tree 为：



注意我们在构建 elimination tree 的时候不是使用的原始矩阵，而是以高斯消去过程得到的矩阵为准，高斯消去过程会引入新的非 0 元素。在构建 elimination tree 的算法中需要注意这个问题。

例如对于下面的矩阵：

$$\begin{bmatrix} a & & \\ d & b & \\ e & & c \end{bmatrix}$$

在用 a 消去 e 时，会在 $\times$ 处引入 fill-in，从而使得 b 的父节点变为 c。

$$\begin{bmatrix} a & & \\ & b & \\ & \times & c \end{bmatrix}$$

如果使用朴素的方法构建 elimination tree，需要执行完整的高斯消去过程，尽管不需要计算具体的值，只需要记录引入的非 0 元素，其复杂度也较高。下面给出一种低复杂度的构建方法，它的核心是引入了一个 root 数组，用于记录每个元素所在的子树当前的根节点。一个核心的观察是：在原始矩阵中，对于某个对角线元素 u，它所在行里的排在它前面的非 0 元素 c 所在的列对应的对角线元素 v 必定是它的子孙节点（不一定是子节点），并且，v 所在的子树的根节点 w 也必然是 u 的子孙节点。理由是：当使用 v 消去 c 时，必然会在 w 的列对应于 u 的行引入一个 fill-in。这段推理可能有一些绕，看了下面的矩阵就应该清楚了。

$$\begin{bmatrix} v & & & & \\ & \ddots & & & \\ b & & w & & \\ & & & \ddots & \\ c & & \times & & u \end{bmatrix}$$

这里我们只需要关注子树的根节点，为它设置由 fill-in 引入的父节点，而无需考虑子树中其它的节点，因为那些节点已经有父节点了。

下面给出完整的算法。

```
树节点（依次对应于对角线元素）编号为1至n
数组parent保存每个树节点的父节点
数组root保存每个树节点所在子树当前的根节点

for i from 1 to n {
    // 初始化节点i的状态为无效状态
```

```
    parent[i] = MAX
    root[i] = MAX

    // 遍历节点i所在行的位于i之前的非0元素
    for e in row(i) {
        // e的列编号j就是对应的树节点
        令j = column_index(e),

        // 只处理i之前的节点
        if (j >= i)
            continue
        // 递归找j当前的根节点及其根节点的根节点
        // 设置根节点的根节点为i
        while (root[j] < i) {
            unsigned next = root[j]
            root[j] = i
            j = next
        }
        // 更新最终的根节点状态
        root[j] = i
        parent[j] = i
    }
}
```

8 稀疏矩阵的 Extend-Add 操作

Extend-add 是 2 个稀疏矩阵之间的相加操作。稀疏矩阵有可能不包含全部的行和列。它们之间的相加操作规则为：如果有对应位置的元素，则将对应位置元素相加；如果一个位置上只有一个矩阵有元素，则相加的结果矩阵中对应的位置取该矩阵中的元素。

举例如下：

矩阵 A 包含第 1，3 行和第 1，3 列相交处的元素：

$$\begin{bmatrix} a & & b \\ & & \\ c & & d \end{bmatrix}$$

矩阵 B 包含 (2,2) 位置上的元素：

$$\begin{bmatrix} & & \\ & e & \\ & & \end{bmatrix}$$

Extend-add(A, B) 得到：

$$\begin{bmatrix} a & & b \\ & e & \\ c & & d \end{bmatrix}$$

9 Multifrontal 算法

有了 elimination tree, 就可以进行 multifrontal 算法的求解了。

Multifrontal 算法的每一步将 LDL 分解往前推进一步：用当前对角线元素做高斯消去，计算出 L 和 D 的新的一列。那么现在的问题就是当前的 frontal 矩阵包含哪些内容。

首先应该包含的是当前对角线元素对应的列和行，仅包括在行和列里位于它之后的非 0 元素。

但是不需要包含当前对角线元素之后的对角线元素对应的行和列中的元素（即它右下角的元素），因为这些元素不归当前对角线元素管（消去）。它们由它们对应的对角线元素去消去。

同时，需要将当前对角线元素在 elimination tree 中的子节点对应的 update 矩阵通过 extend-add 操作纳入进 frontal 矩阵。因为这些 update 矩阵包含的信息是子节点在做高斯消去时带给父节点及后续节点的相关元素的副作用。这些副作用信息需要通过当前对角线元素的 frontal 矩阵传递下去。它们不仅仅包含对当前对角线元素的行和列的影响，也包含对后续对角线元素的行和列的影响。

有了完整的 frontal 矩阵，进行一步 LDL 分解，得到对应于当前对角线元素的 L 中的列和 D 中的元素，以及当前对角线元素对应的 update 矩阵，供当前对角线元素的父节点使用。

下面给出 multifrontal 算法的完整描述。

for i from 1 to n {
令a为第i个对角线元素
令v为a所在列位于a之下的非0元素形成的列
令P为a和v形成的矩阵：
$$P = \begin{bmatrix} a & v^\top \\ v & \end{bmatrix}$$

令c_1, ..., c_s为i在elimination tree中的子节点
令U_{c_1}, ..., U_{c_s}为对应的update矩阵
将U_{c_1}, ..., U_{c_s} extend−add在一起，形成U

将P和U extend−add在一起形成最终的frontal矩阵F

对F进行一步LDL分解：
$$F = \begin{bmatrix} 1 & \\ l & I \end{bmatrix} \begin{bmatrix} d & \\ & C \end{bmatrix} \begin{bmatrix} 1 & l^\top \\ & I \end{bmatrix}$$
将得到的列l加入L,
得到的左上角元素d加入D,
得到的右下角子矩阵C作为第i个对角线元素的update矩阵
}

10 稀疏矩阵的数据结构

在实践中，稀疏矩阵的数据结构设计是需要根据需求仔细考虑的。这里直接给出我在项目中使用的数据结构实现。使用的语言是 C++，还使用了 Eigen 数学库。

```
// 基本计算单位，一个矩阵块
class EBlock {
    Eigen::MatrixXd m;
};
```

```
// 稀疏矩阵的一列
class EVector {
public:
    // 表示对应的block在完整向量中的位置，-1表示该block已经被删除
    std::vector<int> index;

    // 块矩阵
    std::vector<EBlock*> block;
};

class EMatrix {
public:
    // 表示对应的列在完整矩阵中的位置，-1表示该列已经被删除
    std::vector<int> index;

    // 列向量
    std::vector<EVector*> column;
};
```

这里矩阵的基本元素不是一个数，而是一个块矩阵，这是由应用的特点决定的，因为动力学模拟中的质量矩阵和 Jacobian 矩阵都是块矩阵。块矩阵的实现使用了 Eigen 数学库，是一个动态方阵。块矩阵的基本计算操作使用 Eigen 的实现。

稀疏矩阵的一个列由 C++ 的 vector 表示。index 表示对应的 block 在列向量中的位置。如果要增加块，则增加在向量的后面。如果要删除块，因为 vector 的删除操作很费时，所以我们用将该块对应的 index 设置为-1，表示该位置上的块已经被删除了。

稀疏矩阵由列构成。同样的，index 向量表示对应的列在矩阵中的位置。如果要增加列，则增加在向量的后面。如果要删除列，将该列对应的 index 设置为-1，表示该列已经被删除了。

有了这样一套数据结构，它们的操作的实现就很直接了，这里不再赘述。

注意在 vector 中存储的都是数据对象的指针，配合上 C++ 11 的 move 语义（Eigen 已经支持），可以实现高效率的数据操作。

11 参考文献

线性代数的教材推荐 [1]。矩阵计算的经典教材是 [2]。稀疏矩阵的直接方法参考 [3]。Multifrontal 算法的一篇综述性教程是 [4]。

参考文献

[1] Gilbert Strang. 线性代数. 清华大学出版社, 5 edition, 2019.

[2] Gene H. Golub and Charles F. Van Loan. 矩阵计算. 人民邮电出版社, 4 edition, 2020.

[3] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Oxford University Press, 2 edition, 2017.

[4] Joseph W. H. Liu. The multifrontal method for sparse matrix solution: Theory and practice. *SIAM Review*, 34(1):82–109, March 1992.