中国研究生创新实践系列大赛

“华为杯”第二十届中国研究生

数学建模竞赛

学　　校	杭州电子科技大学
参赛队号	23103360062
队员姓名	1.纪桂宝 2.曹潘涛 3.邱靖颖

中国研究生创新实践系列大赛

“华为杯”第二十届中国研究生

数学建模竞赛

题　目：　DFT 类矩阵的整数分解逼近

摘　要：

离散傅里叶变换（Discrete Fourier Transform,DFT）是一种数学工具，用于将一个离散的信号从时域转换为其频域表示，被广泛应用于工程、科学以及数学领域。但由于其直接进行 DFT 计算代价高，在实际产品中通常使用快速傅里叶变换（Fast Fourier Transform,FFT）算法来快速实现 DFT。随着 5G 时代的到来 FFT 所需的芯片硬件复杂度太高，不满足实际使用要求。本文使用了**稀疏整数矩阵连乘**的方式近似逼近 DFT 矩阵，在误差不大的条件下大幅降低了计算 DFT 矩阵的硬件复杂度。

问题 1 要求使用多个每行至多两个非零元素的稀疏矩阵连乘逼近 DFT 矩阵，以此来通过减少复数乘法次数的方式来降低计算 DFT 矩阵的硬件复杂度。首先对 DFT 矩阵的性质和 N 点旋转因子的性质进行了分析，得到了 DFT 矩阵和旋转因子的性质。为使用稀疏矩阵连乘的方式逼近 DFT 矩阵，将分解后的矩阵每行至多两个非零元素作为约束，以连乘后得到的矩阵和 DFT 矩阵在 F 范数下的误差最小作为目标函数，建立优化矩阵个数、矩阵元素和矩阵缩放因子的多元混合整数规划模型。求解该模型是基于 **Cooley-Tukey 算法的 Radix-2 DIT 方法思想**，采用**递归和“位翻转”**操作将一个 N 维的 DFT 矩阵$\boldsymbol{F_N}$分解为了 $\log_2 N+1$ 个稀疏矩阵，形如$\boldsymbol{F_N}=(\prod_{i=1}^{\log_2 N-1}\boldsymbol{A_i})(\mathbf{I}_{N/2}\otimes\boldsymbol{F_2})\mathbf{P}_N$。采用这种矩阵连乘的方案，保证了分解后矩阵的稀疏性，使得计算 DFT 矩阵的硬件复杂度有效降低；同时使得误差几乎为 0，可以称为**精确的稀疏矩阵分解结果**。通过计算并统计分析 $N=2^t=2,4,8,16,...,1024$ 的分解结果的硬件复杂度等数据发现，**对应的硬件复杂度分别为 0,0,64, 576,3392, 16704,75072, 320832, 1332544,5444928**。其硬件复杂度会随着 t 的增大近指数级增长，且由于矩阵元素位宽较大，这样计算的硬件复杂度并不是一个理想的结果。

问题 2 要求在使用矩阵连乘拟合 DFT 矩阵时，通过仅使用位宽为 p=3 的整数乘法器来降低硬件复杂度，即限制连乘中使用到的矩阵中的元素实部和虚部仅在$\mathcal{P}=\{0,\pm1,\pm2,\pm4\}$内取值，使用这样的矩阵连乘拟合 DFT 矩阵使得误差最小并使硬件复杂度尽可能低。为了计算得到满足的矩阵连乘方案，将连乘中使用到的矩阵中的元素实部和虚部仅在$\mathcal{P}$内取值作为约束条件，将矩阵连乘的矩阵和DFT矩阵在F范数意义上的误差最小作为目标函数，建立优化矩阵个数、矩阵元素和矩阵缩放因子的**多元混合整数规划模型**。通过对 DFT 矩阵的可分解性进行分析，发现一个 N 维 DFT 矩阵可以分为 16 块矩阵，其他 15 块均可以通过左上角的一块矩阵表示，而这个 N/4 维的矩阵由于其范德蒙形式可以使用一个 N/4 维的向量表示，即可以通过一个 N/4 维的向量表示 N 维的 DFT 矩阵，这个向量称为 **DFT 矩阵**

的参数化向量。使用仅优化参数化向量就可以使得矩阵元素范围满足要求，将决策变量变为参数化向量和矩阵缩放因子改进模型，通过使用 Gurobi 求解器计算出了分别使用一个整数矩阵逼近 N=2,4,8,16,32 维 DFT 矩阵的方案，**误差分别为 0、0、0.146、0.49、0.83，硬件复杂度分别为 0、0、144、624、0**。由于其使用的输入数据位宽仅为 3，其硬件复杂度虽有下降，但其仍不理想且结果存在一定误差。

问题 3 要求在使用矩阵连乘拟合 DFT 矩阵时，通过同时减少乘法器个数和输入数据的位宽整来降低硬件复杂度，这样就需要使得连乘中使用到的矩阵既满足稀疏性，又满足元素范围限定，使用这样的矩阵连乘拟合 DFT 矩阵使得误差最小并尽可能降低硬件复杂度。为求解理想的方案，需要将连乘中的矩阵每个行至多两个非零元素和矩阵元素实部和虚部仅在$\mathcal{P}=\{0,\pm1,\pm2,\pm4\}$内取值作为约束条件，将连乘得到的矩阵和 DFT 矩阵误差最小作为目标，建立了**多元混合整数模型**。为求解该模型，先使用第一问的 DFT 矩阵分解方法，确定矩阵个数及矩阵形式，再使用**遗传算法**优化其中实部或虚部不在$\mathcal{P}$内的元素。通过这样求解出了 N=2,4,8,16,32 维的 DFT 矩阵的矩阵连乘拟合方案，**误差分别为 0、0、0.16、0.26、0.33，硬件复杂度均为 0**。这样的方案有效的大幅降低了计算 DFT 矩阵的硬件复杂度且拟合后的误差较为合理。

问题 4 要求应用 DFT 矩阵分解思路，分解$\boldsymbol{D}$矩阵，$\boldsymbol{D}=\mathbf{F}_{N1}\otimes\mathbf{F}_{N2}$且$\mathbf{F}_{N1}$和$\mathbf{F}_{N2}$分别是$N_1,N_2$维的 DFT 矩阵，在满足矩阵稀疏性和矩阵元素范围限定的条件下，设计合理的矩阵使得误差最小并降低硬件复杂度。首先根据 DFT 矩阵可分解的性质将$\mathbf{F}_{N1},\mathbf{F}_{N1}$进行分解并利用**混合积性质**，可以得到$\mathbf{D}=(\prod_{i=1}^{\log_2 \mathrm{N2}-1}(\boldsymbol{B_i}\otimes\boldsymbol{A_{N2,i}}))(\boldsymbol{B}_{\log_2 \mathrm{N2}}\otimes(\mathbf{I}_{\mathrm{N2}/2}\otimes\boldsymbol{F}_2))(\boldsymbol{B}_{\log_2 \mathrm{N2}+1}\otimes\mathbf{P}_{\mathrm{N2}})$。当$N_1=4,N_2=8$可以将 D 矩阵分解为 4 个矩阵$(\boldsymbol{B_1}\otimes\boldsymbol{A_{8,1}})(\boldsymbol{B_2}\otimes\boldsymbol{A_{8,2}})(\boldsymbol{B_3}\otimes(\mathbf{I_4}\otimes\boldsymbol{F}_2))(\mathbf{P_4}\otimes\mathbf{P_8})$。通过限定$\boldsymbol{B_1}$、$\boldsymbol{B_2}$、$\boldsymbol{B_3}$为每行仅有一个元素的稀疏矩阵可以使得$(\boldsymbol{B_1}\otimes\boldsymbol{A_{8,1}})$、$(\boldsymbol{B_2}\otimes\boldsymbol{A_{8,2}})$、$(\boldsymbol{B_3}\otimes(\mathbf{I_4}\otimes\boldsymbol{F}_2))$、$(\mathbf{P_4}\otimes\mathbf{P_8})$四个矩阵为每行至多有两个元素的稀疏矩阵。将这一稀疏性条件和非零元素在限定范围内作为约束条件，将 4 个矩阵连乘的结果与 **D** 矩阵的误差最小作为目标函数，建立优化模型。使用 Gurobi 求解器，计算出了稀疏整数矩阵连乘的拟合方案，**误差为 0.872195，硬件复杂度为 0**。

问题 5 要求在问题 3 的基础上增加精度的限制来设计整数矩阵连乘拟合方案。为计算出满足要求的方案，将连乘中的矩阵每个行至多两个非零元素、矩阵元素实部和虚部仅在$\mathcal{P}=\{0,\pm1,\pm2,\ldots,\pm2^{q-1}\}$内取值和误差 RMSE 小于等于 0.1 作为约束条件，将计算 DFT 矩阵的硬件复杂度作为目标函数，建立了优化连乘矩阵、矩阵缩放因子β和矩阵元素范围$\boldsymbol{\mathcal{P}}$的优化模型。为求解该模型，设计了两种求解思路。求解思路一为基于**折半查找**的方法，根据 q 越大、误差越小这一性质，首先求解出一个满足误差要求的较大 q 值q_n，在 2 到q_n中折半查找满足最小误差满足误差要求的最小 q 值。求解思路二为基于**动态规划**的方法，根据矩阵缩放因子和旋转因子的性质，可以将直接遍历 q 优化 $\log_2$N+1 个连乘矩阵的问题分解为逐个优化每个矩阵的子问题。由于思路一需要较高的算力来求解，在矩阵维度较高时难以求解，因此采用思路二的方式求解该模型。求解得到了 N=2,4,8,...,1024 维的 DFT 矩阵分解方案，均满足误差小于 0.1 的要求，**其复杂度分别为 0, 0, 48, 288, 2240, 9600, 39680, 193536, 780288, 3133440**。

本文使用矩阵连乘的形式拟合 DFT 矩阵，可以得到计算硬件复杂度和误差均较为合理的方案，比传统 FTT 硬件复杂度低。并在问题 5 中采用动态规划的思想计算出了误差可控的矩阵连乘拟合方案。

关键词： 矩阵连乘拟合；多元混合整数优化；近似 FFT 方法；Cooley-Tukey 算法

1. 问题重述

1.1. 问题背景

离散傅里叶变换（Discrete Fourier Transform,DFT）是一种数学工具，用于将一个离散的信号从时域转换为其频域表示，从而能够分析信号的频谱成分。离散傅里叶变换通过一个复杂的数学公式来实现，该公式将一个离散信号序列变换为其频域表示并将信号分解为一系列频率分量，每个频率分量都有特定的幅度和相位信息，从而帮助我们了解到信号中包含的不同频率分量的贡献。离散傅里叶变换被广泛应用于工程、科学以及数学领域。DFT 是频谱分析的关键工具，通过计算信号的 DFT 可以确定信号中存在的频率成分、幅度和相位，这对于音频处理、信号处理、通信系统、振动分析等领域非常重要。在信号处理中，DFT 可用于设计数字滤波器，通过分析信号的频域表示可以选择性的去除或增强特定频率分量，以实现滤波效果。在数据压缩中，DFT 可以用于将信号转换为频域表示，然后通过丢弃或压缩频域数据来减少数据量，同时保留关键信息，这在音频和图像压缩中广泛使用。DFT 在数字信号处理中将模拟信号转换为数字信号，进行处理与分析。离散傅立叶变换的意义在于它提供了一种分析信号的方式，使我们能够理解信号的频域特性。这有助于我们在各种应用中处理和理解信号，从音频处理到通信系统设计，都需要了解信号的频谱信息。DFT 还为许多信号处理和工程问题提供了一种有效的数学工具，以便进行分析、处理和优化。

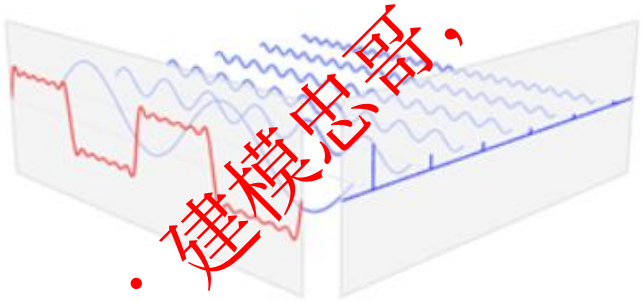

图 1-1 DFT 示意图

随着无线通信技术的进步，天线阵面越来越大、通道数越来越多、通信带宽越来越大，DFT 由于其数学上的复杂程度对硬件的消耗和算力的消耗造成的巨大的增加。为能便捷使用 DFT 并降低其对硬件的消耗，需要对 DFT 的计算过程进行优化，降低算法复杂度和对硬件的消耗。目前在实际产品中，通常使用快速傅里叶变换（Fast Fourier Transform,FFT）算法来快速实现 DFT，FFT 采用蝶形运算的思想，将 DFT 矩阵分解为合适的矩阵，大幅降低了复数乘法运算的次数，从而降低了计算 DFT 新品的硬件复杂度。但随着 5G 时代的到来，对 FFT 的需求越来越大，从而导致专用芯片上实现 FFT 的硬件开销也越来越大。为进一步降低芯片资源开销，需要提出新的算法实现 DFT 使得硬件开销大幅降低、算法复杂度降低。一种可行的思路为将 DFT 矩阵分解为整数矩阵连乘的形式，这种矩阵连乘的形式虽然会使得 DFT 矩阵的精度降低，但是这种方法使得所需的硬件复杂度大幅降低。分解为整数矩阵连乘的方法可以通过整数的约束使得硬件中数据的位数降低，通过矩阵分解的方法将矩阵分解为多个稀疏矩阵从而降低了矩阵乘法的复杂度、减少了乘法运算的数量，从而使得硬件复杂度大大降低。

综上所述，由于 DFT 的算法复杂度和硬件消耗大，需要对 DFT 矩阵使用矩阵连乘的形式进行拟合，使得 DFT 的硬件消耗降低。使用矩阵连乘的方法需要综合考虑矩阵的个数、矩阵的稀疏程度、矩阵中元素的值、矩阵的形式等因素。合理的设计矩阵连乘的形式会使

得 DFT 的算法复杂度降低并降低硬件消耗，使得 DFT 在使用中更加便捷，从而使得通信中硬件代价降低。因此，如何设计矩阵连乘的方法来拟合 DFT 矩阵是一个具有挑战性和实际意义的问题。

1.2. 问题提出

想要合理设计矩阵连乘拟合 DFT 矩阵的方式需要对整数分解逼近的过程进行分析，得到需要求解的问题。为使用矩阵连乘拟合 DFT 矩阵的方法，首要的目标是要求拟合的矩阵和原 DFT 矩阵相差不大，这里使用 Frobenius 范数意义下尽可能接近来定义拟合误差；其次的目标为使得矩阵连乘所需的硬件复杂度低，即矩阵连乘复杂度低、复数乘法次数小。为实现这两个目标，需要对矩阵的形式进行确定：首先，为降低矩阵连乘复杂度需要使得矩阵尽可能稀疏，元素数量的减少会较少矩阵乘法的复杂性；其次，矩阵元素的有效位数也影响着矩阵乘法的复杂性，矩阵元素所需的有效位数越小，意味着乘法复杂度的降低，对缓存的要求也减少，从而同时降低了硬件复杂度；最后，使用几个矩阵进行连乘拟合也影响着矩阵拟合的精度和矩阵乘法的复杂度。

因此，想要合理设计矩阵连乘拟合 DFT 矩阵的方式必须依据 DFT 矩阵已有的性质和其中旋转因子$w_N = e^{-\frac{j2\pi}{N}}$的性质。如何利用这些性质将 DFT 矩阵合理分解为多个矩阵连乘的拟合形式是一个多目标的混合整数规划问题，需要使用合理的方式进行有规律的进行矩阵分解和确定矩阵形式。综上所述，为了使得加速和简化 DFT 计算过程需要使用矩阵连乘的形式对 DFT 矩阵进行拟合，需要根据不同的约束如矩阵乘法复杂度、矩阵元素范围来确定不同的矩阵设计方案。

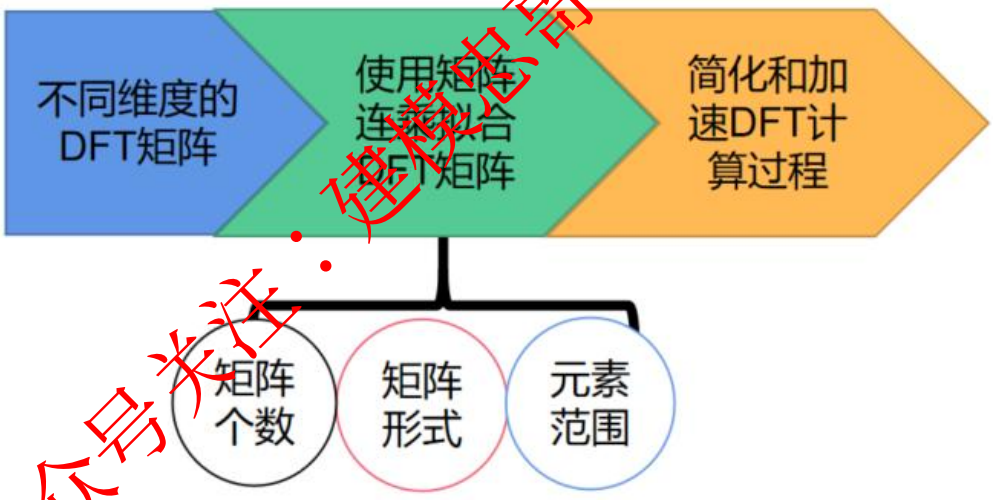

图 1-2　DFT 类矩阵的整数分解逼近问题

本论文中解决 DFT 类矩阵的整数分解逼近问题，具体问题描述为：首先，将 DFT 矩阵使用矩阵连乘的形式进行拟合，使用矩阵连乘的形式可以简化和加速 DFT 计算过程；其次，对拟合使用到的矩阵进行优化使得矩阵连乘使用到的硬件复杂度最小。为此，需要依次解决以下几个问题：

①在使用矩阵连乘拟合 DFT 矩阵时，通过减少乘法器的个数来降低硬件复杂度，即规定拟合中使用到的矩阵为稀疏矩阵，使用这样的矩阵连乘拟合 DFT 矩阵使得误差最小并使硬件复杂度尽可能低；

②在使用矩阵连乘拟合 DFT 矩阵时，通过仅使用位宽为 p 的整数乘法器来降低硬件复杂度，即限制拟合中使用到的矩阵中的元素实部和虚部仅在一定范围内的整数取值，使用这样的矩阵连乘拟合 DFT 矩阵使得误差最小并使硬件复杂度尽可能低；

③在使用矩阵连乘拟合 DFT 矩阵时，通过同时减少乘法器个数和规定元素仅在一定范围内的整数取值来降低硬件复杂度，这样就需要使得连乘中使用到的矩阵既满足稀疏性，又满足元素范围限定，使用这样的矩阵连乘拟合 DFT 矩阵使得误差最小并尽可能降低硬件复杂度；

④当$N_1=4,N_2=8$时，使用 DFT 矩阵分解思路，分解$\mathbf{F}_N=\mathbf{F}_{N1}\otimes\mathbf{F}_{N2}$且$\mathbf{F}_{N1}$和$\mathbf{F}_{N2}$分别是$N_1,N_2$维的 DFT 矩阵，在满足矩阵稀疏性和矩阵元素范围限定的条件下，设计合理的矩阵使得误差最小并降低硬件复杂度；

⑤根据问题③，设计的矩阵需要满足稀疏性和元素范围限定两个条件，本问题需要通过设计合理的矩阵和矩阵元素范围使得矩阵满足这两个条件的基础上也同时使得拟合结果误差小于 0.1，计算此时的硬件复杂度。

2. 问题分析

2.1. 总体问题分析

DFT 是对信号向量进行线性正交变换的一个过程，对一个 N 维的信号直接进行 N 点 DFT 需要进行N^2次复数乘法和 N(N − 1)次加法，对于硬件复杂度和算力要求较高。所以需要设计快速算法将 DFT 计算成本降低，通常使用的方式为使用快速傅里叶变换算法（FFT）来实现 DFT 过程但 FFT 的硬件开销不满足现有需求需要设计方案降低 FFT 的硬件开销。一种合理的方式为使用矩阵连乘来拟合 DFT 矩阵，通过设计合理的矩阵可以降低 FFT 中的硬件开销并满足精度要求。

FFT 中的硬件复杂度主要由矩阵连乘中复数乘法的次数和乘法器的复杂度决定，减少复数乘法的次数并降低每个乘法器的复杂度可以有效的使得 FFT 中的硬件开销变小。连乘中矩阵尽可能稀疏会使得乘法的次数有效减少，而乘法器的复杂度可以通过减少乘法元素的有效位数有效的降低乘法器的复杂度。因此为降低 FFT 中的硬件开销，需要对连乘中矩阵的稀疏性和元素有效位数进行合理设计，在精度尽可能大的条件下硬件复杂度降低。

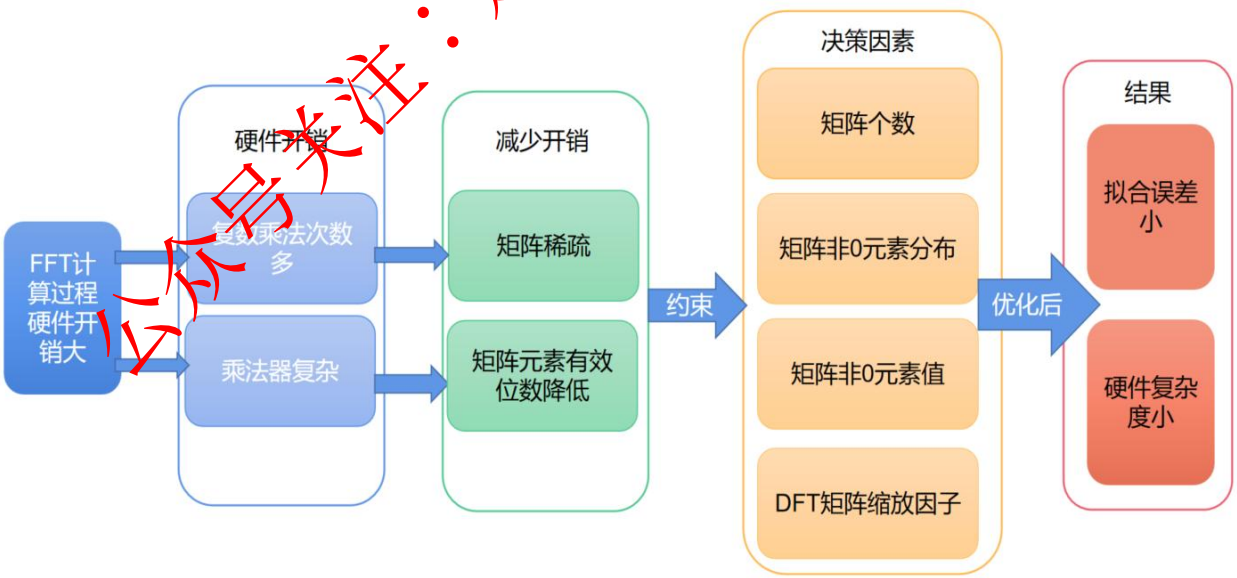

图 2-1 问题总体分析流程图

在设计好的矩阵稀疏性和元素有效位数约束下，根据不同维度的 DFT 矩阵来优化需要连乘的矩阵个数、每个矩阵元素的值和矩阵缩放因子来得到最优的矩阵连乘形式。但在规

定矩阵元素虚部和实部均为整数的条件下，直接对每个矩阵元素中的值进行优化为一个庞大的多元整数优化问题，不是一个明智的思路。因此需要分析 DFT 矩阵的性质和旋转因子w_N的性质并利用这些性质设计合理的优化方案，从而在可能的范围内求解出最合适的矩阵连乘形式。

3. 模型假设与符号说明

3.1. 模型假设

① 假设 DFT 输入的信号为复数信号；

② 假设使用矩阵连乘中的矩阵复杂度仅由复数乘法的复杂度决定，其中与 0、± 1、$\pm j$或$(\pm 1 \pm j)$相乘时不计入复数乘法次数；

③ 将乘法器的复杂度简化为仅与输入数据的取值范围相关。对于复数 g$\in \{x+jy|x,y\in\mathcal{P}\},\mathcal{P}=\{0,\pm 1,\pm 2,\ldots,\pm 2^{q-1}\}$,其与任意复数z相乘的复杂度为q。

④ 假设仅考虑 N 维的 DFT 矩阵拟合，其中$N=2^t, t=1,2,3,\ldots$。

3.2. 符号说明

表 3-1 符号定义表

符号	说明
N	DFT 矩阵维度
RMSE	拟合结果误差
j	复数虚部单位
C	矩阵连乘硬件复杂度
L	矩阵连乘中复数乘法次数
β	实值矩阵缩放因子
π	圆周率
$\boldsymbol{X_N}$	$\boldsymbol{X_N}$维度为 N
x^k	x的 k 次方
$[\boldsymbol{X}]_i$	$\boldsymbol{X}$的第 i 个元素
$\det(\boldsymbol{X})$	$\boldsymbol{X}$的行列式
$(\boldsymbol{X})^H$	$\boldsymbol{X}$的共轭转置
$\mathcal{R}$	实数域
$\mathbf{I_N}$	N 维单位矩阵
$\mathrm{diag}(\boldsymbol{X})$	$\boldsymbol{X}$对角阵
q	乘法器位宽
depth	递归深度

4. 问题 1：DFT 矩阵的稀疏矩阵分解逼近

4.1. DFT 矩阵的稀疏矩阵分解逼近分析

为了降低矩阵连乘中硬件复杂度的开销，可以用过规定矩阵的稀疏性来使得矩阵连乘中使用到的复数乘法次数大幅降低。当每个矩阵的每行至多只有 2 个非零元素时，该矩阵的稀疏性就较为满足矩阵连乘中降低复数乘法次数的要求。因此，需要将该稀疏性作为约束，对矩阵连乘的方案进行设计使得拟合结果误差最小并尽可能减少方案的硬件复杂度。

通常使用的 DFT 矩阵分解思想为 Cooley-Tukey 算法，它的基本思想是分治策略，可以将高维度的 DFT 矩阵分块为低维度的 DFT 矩阵，从而利用已知维度的 DFT 矩阵计算出维度较高的 DFT 矩阵，使用 Cooley-Tukey 算法的中的 Radix-2 DIT 方法可以将 N 维的 DFT 矩阵分解为$log_2 N$+1 个每行至多只有 2 个非零元素的稀疏矩阵连乘。这样得到的结果误差几乎为 0，且使用到的硬件复杂度较低。

4.2. 离散傅里叶变换 DFT

（1）离散傅里叶变换过程

离散傅里叶变换是广泛使用的一种数学工具，其主要功能是对 N 维输入向量做一种线性正交变换。在信号变换中，给定N点的时域一维复数信号$x_0, x_1, ..., x_{N-1}$，DFT 后得到的复数信号X_k（$k = 0,1, ..., N-1$）由下式给出（其中 j 为虚数单位）：

$$\mathrm{X}_k = \sum_{n=0}^{N-1} x_n * e^{-\frac{j2\pi nk}{N}}, k = 0,1,2, ..., N-1 \tag{4.1}$$

其矩阵形式为：

$$\begin{bmatrix} \mathrm{X}_0 \\ \mathrm{X}_1 \\ \mathrm{X}_2 \\ \vdots \\ \mathrm{X}_N \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & w & w^2 & \cdots & w^{N-1} \\ 1 & w^2 & w^4 & \cdots & w^{2(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & w^{N-1} & w^{2(N-1)} & \cdots & w^{(N-1)(N-1)} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_N \end{bmatrix}, w = \mathrm{e}^{-\frac{j2\pi}{N}} \tag{4.2}$$

其中 $\mathrm{x} = [\mathrm{x}_0\ \mathrm{x}_1 \cdots \mathrm{x}_{N-1}]^{\mathrm{T}}$为输入，一般为时域信号向量；$\mathrm{X} = [\mathrm{X}_0\ \mathrm{X}_1 \cdots \mathrm{X}_{N-1}]^{\mathrm{T}}$为输出向量，是输入的信号的频域表示。将时域信号向量变为频域信号向量的矩阵称为 DFT 矩阵。

（2）DFT 矩阵

DFT 矩阵使用旋转因子w表示出来为：

$$\mathbf{F}_N = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & w & w^2 & \cdots & w^{N-1} \\ 1 & w^2 & w^4 & \cdots & w^{2(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & w^{N-1} & w^{2(N-1)} & \cdots & w^{(N-1)(N-1)} \end{bmatrix}, w = \mathrm{e}^{-\frac{j2\pi}{N}} \tag{4.3}$$

该 DFT 矩阵为一个复数元素的范德蒙形式的矩阵，具有以下性质：

① 对称性：$[\mathbf{F}_N]_{\mathrm{k,n}} = [\mathbf{F}_N]_{\mathbf{n},\mathrm{k}}$;

② 正交性：$\mathbf{F}_N\mathbf{F}_N^{\mathrm{H}} = \mathbf{I}$，其中 $\mathbf{I}$ 为 N 维单位矩阵，$\mathbf{F}_N^{\mathrm{H}}$为$\mathbf{F}_N$的共轭转置矩阵;

③ 可逆性：$\det(\mathbf{F}_N) \neq 0$，其中 $\det(\mathbf{F}_N)$表示F_N矩阵的行列式;

④ 稠密性：矩阵元素均为非 0 值且多数为复数。

矩阵的稠密性使得直接计算 DFT 矩阵计算复杂度较为高，在 DFT 计算过程中将会产生大

量的复数乘法，造成了 DFT 计算的困难。

（3）旋转因子w

旋转因子w也称为单位根，一个 N 点的旋转因子为：

$$w_N = \mathrm{e}^{-\frac{j2\pi}{N}} \tag{4.4}$$

旋转因子可以使用欧拉公式将其写为：

$$w_N = \cos(-\frac{2\pi}{\mathrm{N}}) + \sin(-\frac{2\pi}{\mathrm{N}})\,\mathrm{j} \tag{4.5}$$

根据欧拉公式可以得到复数w_N的实部为 $\cos(-2\pi/N)$，虚部为 $\sin(-2\pi/N)$，这个复数具有以下两个性质：

① 单位性：该复数在复数平面上的点与原点之间的距离恒为 1, 如下图所示：

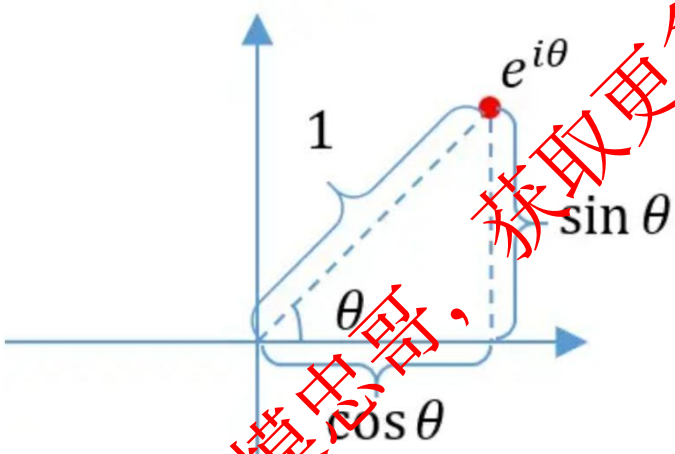

图 4-1 旋转因子在复平面示意图

② 周期性：不论 N 为何值，该复数都在复平面的单位圆上。以 N=16 为例：

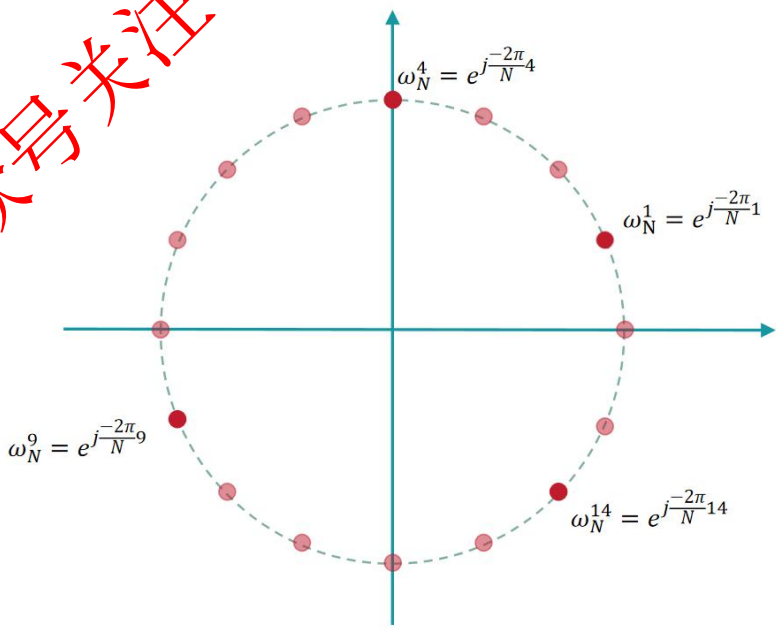

图 4-2 旋转因子周期性示意图

根据旋转因子的形式可以得到以下几个性质：

性质一：折半引理

$$w_{2N}^{2k} = w_N^k \tag{4.6}$$

性质二：消去引理

$$w_N^{k+N} = w_N^k \tag{4.7}$$

$$w_N^{k+N/2} = -w_N^k \tag{4.8}$$

$$w_N^{k+N/4} = (-j)w_N^k \tag{4.9}$$

性质三：周期性

$$w_N^{Nk} = 1 \tag{4.10}$$

4.3. DFT 矩阵的稀疏矩阵分解逼近模型

本问题使用规定矩阵连乘拟合 DFT 矩阵中连乘的矩阵必须为每行至多有两个非零元素来减少矩阵乘法次数，从而达到降低硬件复杂度的目的。所以在设计连乘矩阵时，需要将这一稀疏性要求作为约束条件，使得矩阵连乘拟合的结果误差尽可能小，为设计出误差最小的连乘矩阵需要建立 DFT 矩阵的稀疏矩阵分解逼近模型。

（1）决策变量

矩阵连乘拟合 DFT 矩阵需要设计的变量有：

① 连乘的矩阵个数 K：

$$K \in \{1,2,3,4,\dots\} \tag{4.11}$$

② 连乘中第 i 个矩阵$\mathbf{A_i}$的第 k 行，第 n 列的元素的值$[\mathbf{B_i}]_{k,n}$：

$$[\mathbf{B_i}]_{k,n} \in \{x + jy | x, y \in \boldsymbol{R}\},, i = 1,2,\dots,K; k,n = 0,1,\dots,N-1 \tag{4.12}$$

③ 连乘中第 i 个矩阵$\mathbf{A_i}$的第 k 行，第 n 列的元素的值是否为 0，$X_{i,k,n} = 0$ 表示$[\mathbf{A_i}]_{k,n} = 0$，$X_{i,k,n} = 1$ 表示$[\mathbf{A_i}]_{k,n} \neq 0$：

$$X_{i,k,n} \in \{0,1\}, i = 1,2,\dots,K; k,n = 0,1,\dots,N-1 \tag{4.13}$$

④ 实值矩阵缩放因子β：

$$\beta \in \mathcal{R} \tag{4.14}$$

其中$\mathcal{R}$ 表示实数域。

（2）目标函数

本问题使用 K 个稀疏矩阵连乘拟合近似 DFT 矩阵，所以本问题的目标为 K 个矩阵连乘尽可能和原 DFT 矩阵误差最小。使用 Frobenius 范数意义下矩阵$\mathbf{F}_N$和$\beta\mathbf{A}_1\mathbf{A}_2\cdots\mathbf{A}_K$的误差作为矩阵连乘拟合 DFT 矩阵的误差，即：

$$\min \text{RMSE} = \frac{1}{N}\sqrt{\|\mathbf{F}_N - \beta\mathbf{A}_1\mathbf{A}_2\cdots\mathbf{A}_K\|_F^2} \tag{4.15}$$

其中$[\mathbf{A_i}]_{k,n} = X_{i,k,n}[\mathbf{B_i}]_{k,n}, i = 1,2,\dots,K; k,n = 0,1,\dots,N-1$。

（3）约束条件

为减少计算 DFT 矩阵的硬件复杂度并使矩阵连乘可以正常拟合 DFT 矩阵，作出如下

约束：

① 矩阵$\mathbf{A_i}$为 N 行 N 列的矩阵；

② 矩阵$\mathbf{A_i}$的第 k 行至多有两个非 0 元素：

$$\sum_{n=0}^{N-1} \mathrm{X_{i,k,n}} \le 2, i = 1,2,\ldots,K; k = 0,1,\ldots,N-1 \tag{4.16}$$

综上所述，问题一 DFT 矩阵的稀疏矩阵分解逼近模型总结如下：

$$\min \mathrm{RMSE} = \frac{1}{N}\sqrt{\|\mathbf{F}_N - \beta\mathbf{A}_1\mathbf{A}_2\cdots\mathbf{A}_K\|_F^{\,2}}$$

$$\text{s.t.}\begin{cases} \mathrm{K} \in \{1,2,3,4,\ldots\} \\ [\mathbf{A_i}]_{\mathrm{k,n}} = \mathrm{X_{i,k,n}}[\mathbf{B_i}]_{\mathrm{k,n}}, i = 1,2,\ldots,K; k,n = 0,1,\ldots,N-1 \\ \sum_{n=0}^{N-1} \mathrm{X_{i,k,n}} \le 2, i = 1,2,\ldots,K; k = 0,1,\ldots,N-1 \\ \mathrm{X_{i,k,n}} \in \{0,1\}, i = 1,2,\ldots,K; k,n = 0,1,\ldots,N-1 \end{cases} \tag{4.17}$$

4.4. 模型求解——分治策略

4.4.1 Cooley-Tukey 算法

Cooley-Tukey 算法主要使用分治策略，递归地将 $\mathrm{N} = \mathrm{N_1N_2}$维 DFT 矩阵分解为$\mathrm{N_1}$个$\mathrm{N_2}$维 DFT 矩阵，简化 DFT 计算过程。Cooley-Tukey 算法最流行的思路为在每一步递归中将 N 维的 DFT 矩阵分解为 2 个 N/2 维的 DFT 矩阵，即 2 基底 FFT（(radix-2)）。Cooley-Tukey 算法为了实现刚才所述的拆分，需要对输入或输出进行重新排序后分组，在时域序列上重新分组称为时域抽取（DIT），在频域上重新分组称为频域抽取（DIF），DIT 应用较为广泛且较 DIF 更简单。

根据 2 基底的频域抽取的 Cooley-Tukey 算法可以将 N 维 DFT 矩阵合理分解为每行仅有两个非零元素的$\log_2\mathrm{N}$个稀疏矩阵相乘的形式，得到的分解结果拟合误差几乎为零。使用 2 基底 Cooley-Tukey 算法的思路要求 $\log_2\mathrm{N}$ 为正整数，可以求解出当$N = 2^t, t = 1,2,3,\ldots$时的 N 维 DFT 矩阵分解结果。

4.4.2 基于 Radix-2 DIT 方法的模型求解

使用 Radix-2 DIT 方法可以将一个 N 维的 DFT 矩阵分解为 2 个 N/2 维的 DFT 矩阵计算，然后通过递归分解直到分解为 2 维 DFT 矩阵就完成了分解过程。给定N点的时域一维复数信号$x_0, x_1, \ldots, x_{N-1}$，DFT 后得到的复数信号$\mathrm{X}_k$（$k = 0,1,\ldots,N-1$）的计算过程为：

$$\mathrm{X}_k = \sum_{n=0}^{N-1} x_n * w_N^{nk} \ , k = 0,1,2,\ldots,N-1 \tag{4.18}$$

其矩阵形式为：

$$\begin{bmatrix} \mathrm{X}_0 \\ \mathrm{X}_1 \\ \mathrm{X}_2 \\ \vdots \\ \mathrm{X}_\mathrm{N} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & w_N^1 & w_N^2 & \cdots & w_N^{N-1} \\ 1 & w_N^2 & w_N^4 & \cdots & w_N^{2(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & w_N^{N-1} & w_N^{2(N-1)} & \cdots & w_N^{(N-1)(N-1)} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_\mathrm{N} \end{bmatrix} \tag{4.19}$$

（1）下面介绍将 N 维 DFT 矩阵分解为 2 个 N/2 维 DFT 矩阵分别处理的过程：

步骤一：将输入的时域序列x_n按奇偶重新分组，可以将式（4.18）变为两部分：

$$\mathrm{X}_k = \sum_{n=0}^{\frac{N}{2}-1} x_{2n} * w_N^{2nk} + \sum_{n=0}^{\frac{N}{2}-1} x_{2n+1} * w_N^{(2n+1)k} \quad , k = 0,1,2,\ldots,N-1 \tag{4.20}$$

步骤二：使用旋转因子的折半引理$w_{2\mathrm{N}}^{2\mathrm{k}} = w_{\mathrm{N}}^{\mathrm{k}}$可以将上式简化：

$$\mathrm{X}_k = \sum_{n=0}^{\frac{N}{2}-1} x_{2n} * w_{\frac{N}{2}}^{nk} + w_N^k \sum_{n=0}^{\frac{N}{2}-1} x_{2n+1} * w_{\frac{N}{2}}^{nk} \quad , k = 0,1,2,\ldots,N-1 \tag{4.21}$$

步骤三：上述两部分都是$\frac{N}{2}$维的 DFT 过程，所以将这两个过程分别使用$\mathrm{G}_{1,k}$和$\mathrm{G}_{2,k}$表示：

$$\begin{aligned} \mathrm{G}_{1,k} &= \sum_{n=0}^{\frac{N}{2}-1} x_{2n} * w_{\frac{N}{2}}^{nk} \\ \mathrm{G}_{2,k} &= \sum_{n=0}^{\frac{N}{2}-1} x_{2n+1} * w_{\frac{N}{2}}^{nk} \end{aligned} \tag{4.22}$$

因此 N 维的 DFT 过程分解为两个 N/2 维 DFT 过程的关系式为：

$$\mathrm{X}_k = \mathrm{G}_{1,k} + w_N^k \mathrm{G}_{2,k} \quad , k = 0,1,2,\ldots,N-1 \tag{4.23}$$

由于使用折半引理将$\mathrm{G}_{1,k}$中旋转因子降为了$\frac{N}{2}$点的旋转因子，利用其周期性可以得到：

$$w_{\frac{N}{2}}^{\mathrm{k}+\frac{N}{2}} = w_{\frac{N}{2}}^{\mathrm{k}} \tag{4.24}$$

从而可以得到$\mathrm{G}_{,\mathrm{k}+\frac{N}{2}} = \mathrm{G}_{,\mathrm{k}}$，利用这一周期性可以将（4.23）写为矩阵形式：

$$\begin{bmatrix} \mathrm{X}_0 \\ \mathrm{X}_1 \\ \vdots \\ \mathrm{X}_{\frac{N}{2}-1} \\ \mathrm{X}_{\frac{N}{2}+0} \\ \mathrm{X}_{\frac{N}{2}+1} \\ \vdots \\ \mathrm{X}_{\frac{N}{2}+\frac{N}{2}} \end{bmatrix} = \begin{bmatrix} 1 & 0 & \cdots & 0 & w_N^0 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 & 0 & w_N^1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & 0 & 0 & \cdots & w_N^{\frac{N}{2}-1} \\ 1 & 0 & \cdots & 0 & -w_N^0 & 0 & \cdots & 0 \\ 0 & 1 & \cdots & 0 & 0 & -w_N^1 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 1 & 0 & 0 & \cdots & -w_N^{\frac{N}{2}-1} \end{bmatrix} \begin{bmatrix} \mathrm{G}_{1,0} \\ \mathrm{G}_{1,1} \\ \vdots \\ \mathrm{G}_{1,\frac{N}{2}-1} \\ \mathrm{G}_{2,0} \\ \mathrm{G}_{2,1} \\ \vdots \\ \mathrm{G}_{2,\frac{N}{2}-1} \end{bmatrix} \tag{4.25}$$

使用上述三步就可以计算出一个矩阵将一个 N 维的 DFT 过程分为两个 N/2 维的 DFT 过程。

（2）再分别对这两个 N/2 维的 DFT 过程分解为 4 个 N/4 维的 DFT 过程：

$$G_{1,k}=\sum_{n=0}^{\frac{N}{4}-1}x_{4n}*w_{\frac{N}{4}}^{nk}+w_{\frac{N}{2}}^{k}\sum_{n=0}^{\frac{N}{4}-1}x_{4n+2}*w_{\frac{N}{4}}^{nk}\ ,k=0,1,2,\ldots,\frac{N}{2}-1 \tag{4.26}$$

$$G_{2,k}=\sum_{n=0}^{\frac{N}{4}-1}x_{4n+1}*w_{\frac{N}{4}}^{nk}+w_{\frac{N}{2}}^{k}\sum_{n=0}^{\frac{N}{4}-1}x_{4n+3}*w_{\frac{N}{4}}^{nk}\ ,k=0,1,2,\ldots,\frac{N}{2}-1 \tag{4.27}$$

使用$H_{1,k}$,$H_{2,k}$,$H_{3,k}$,$H_{4,k}$分别表示分解后的 4 个 N/4 维的 DFT 过程，可得：

$$H_{1,k}=\sum_{n=0}^{\frac{N}{4}-1}x_{4n}*w_{\frac{N}{4}}^{nk} \tag{4.28}$$

$$H_{2,k}=\sum_{n=0}^{\frac{N}{4}-1}x_{4n+2}*w_{\frac{N}{4}}^{nk} \tag{4.29}$$

$$H_{3,k}=\sum_{n=0}^{\frac{N}{4}-1}x_{4n+1}*w_{\frac{N}{4}}^{nk} \tag{4.30}$$

$$H_{4,k}=\sum_{n=0}^{\frac{N}{4}-1}x_{4n+3}*w_{\frac{N}{4}}^{nk} \tag{4.31}$$

N 维的 DFT 过程就被分为 4 个 N/4 维的 DFT 过程：

$$X_k=H_{1,k}+w_{\frac{N}{2}}^{k}H_{2,k}+w_N^k(H_{3,k}+w_{\frac{N}{2}}^{k}H_{4,k})\ ,k=0,1,2,\ldots,N-1 \tag{4.32}$$

（3）递推终止条件及排序矩阵

可以使用两个矩阵就可以将 N 维的 DFT 过程分为 4 个 N/4 维的 DFT 过程，依此类推就可以推出$\log_2$N-1 个矩阵将 N 维的 DFT 过程分为 N/2 个 2 维的 DFT 过程,而 2 维的 DFT 矩阵为 2 行 2 列的实整数矩阵：

$$\boldsymbol{F_2}=\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix} \tag{4.33}$$

排序矩阵$\mathbf{P}_N$可以使用“位翻转”的方法得到，使用排序矩阵可将信号顺序调整正确。

综上所述，这样就可以使用 $\log_2$N-1 个分解矩阵、一个$\mathbf{I}_{N/2}\otimes\boldsymbol{F}_2$矩阵和一个排序矩阵$\mathbf{P}_N$连乘得到 N 维的 DFT 矩阵，即

$$\boldsymbol{F_N}=(\prod_{i=1}^{\log_2 N-1}\boldsymbol{A_i})(\mathbf{I}_{N/2}\otimes\boldsymbol{F}_2)\mathbf{P}_N \tag{4.34}$$

其中$\boldsymbol{A_i}$为第 i 个分解矩阵，$\mathbf{I}_{N/2}$为 N/2 维的单位矩阵，⊗为张量积。

表 4.1 递归分解算法表

Algorithm 1 递归计算分解矩阵
输入：维度 N
输出：分解矩阵$\boldsymbol{\mathcal{A}} = \mathbf{A}_1\mathbf{A}_2\cdots\mathbf{A}_{1+\log_2 N}$
1. 令递归深度 depth = 1，$w = \cos\frac{-2\pi}{N} + \mathbf{j}\sin\frac{-2\pi}{N}$，$\mathbf{F_2} = \begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}$
2. 如果 N == 2 则计算$\mathbf{A}_{\log_2 N} = \mathbf{I}_{\text{depth}} \otimes \mathbf{F_2}$，退出；否则执行第 3 步
3. 令$\mathbf{M_1} = \mathbf{I}_{N/2}$，$\mathbf{M_2} = \text{diag}(1 \quad w^{2^{\text{depth}-1}} \quad \cdots \quad w^{2^{\text{depth}-1}*(N/2-1)})$ $\mathbf{M_3} = \text{diag}(-1 \quad -w^{2^{\text{depth}-1}} \quad \cdots \quad -w^{2^{\text{depth}-1}*(N/2-1)})$
4. 计算$\mathbf{A}_{\text{depth}} = \begin{bmatrix}\mathbf{M_1} & \mathbf{M_2}\\ \mathbf{M_1} & \mathbf{M_3}\end{bmatrix}$，令 N = N/2，depth = depth + 1，转第 2 步
5. 用位翻转操作得到排列矩阵$\mathbf{A}_{1+\log_2 N}$

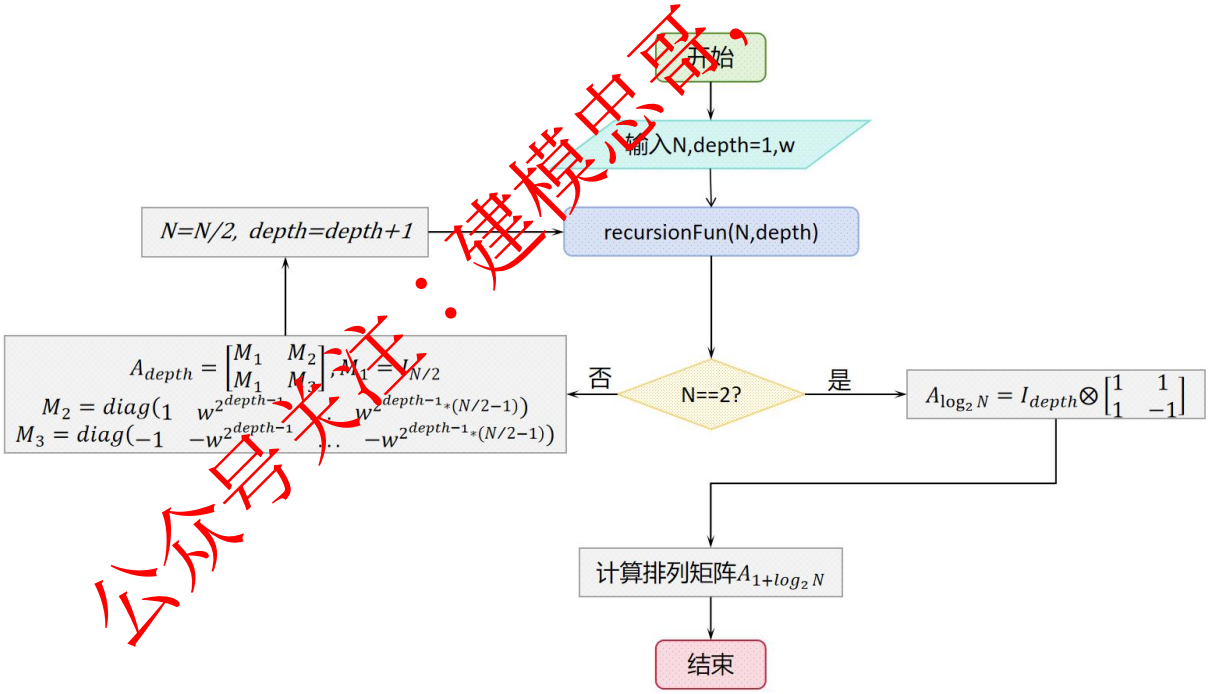

图 4-3 递归分解算法流程图

4.5. 模型求解结果

（1）t =1 时，$N = 2^t = 2$，2 维 DFT 矩阵$\boldsymbol{F}_2$不需要分解，所以矩阵数量为 K=1，矩阵为：

$$\boldsymbol{F_2} = \begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix} \tag{4.35}$$

（2）t =2 时，$N = 2^t = 4$，4 维 DFT 矩阵$\boldsymbol{F}_4$经过分解算法得到的矩阵数量为 K=3，矩阵为：

$$\boldsymbol{A_1} = \begin{bmatrix} 1 & & 1 & \\ & 1 & & -j \\ 1 & & -1 & \\ & 1 & & j \end{bmatrix} \tag{4.36}$$

$$\mathbf{I_2} \otimes \boldsymbol{F}_2 = \begin{bmatrix} 1 & 1 & & \\ 1 & -1 & & \\ & & 1 & 1 \\ & & 1 & -1 \end{bmatrix} \tag{4.37}$$

$$\boldsymbol{P_4} = [e_0 \quad e_2 \quad e_1 \quad e_3] \tag{4.38}$$

$\boldsymbol{F_4}$使用的矩阵连乘形式为：

$$\boldsymbol{F_4} = \boldsymbol{A_1}(\mathbf{I_2} \otimes \boldsymbol{F}_2)\mathbf{P_4} \tag{4.39}$$

（3）t =3 时，$N = 2^t = 8$，8 维 DFT 矩阵$\boldsymbol{F_8}$经过分解算法得到的矩阵数量为 4，矩阵为：

$$\boldsymbol{A_1} = \begin{bmatrix} 1 & & & & w_8^0 & & & \\ & 1 & & & & w_8^1 & & \\ & & 1 & & & & w_8^2 & \\ & & & 1 & & & & w_8^3 \\ 1 & & & & -w_8^0 & & & \\ & 1 & & & & -w_8^1 & & \\ & & 1 & & & & -w_8^2 & \\ & & & 1 & & & & -w_8^3 \end{bmatrix} \tag{4.40}$$

$$\boldsymbol{A_2} = \begin{bmatrix} 1 & & 1 & & & & & \\ & 1 & & w_8^2 & & & & \\ 1 & & -1 & & & & & \\ & 1 & & -w_8^2 & & & & \\ & & & & 1 & & 1 & \\ & & & & & 1 & & w_8^2 \\ & & & & 1 & & -1 & \\ & & & & & 1 & & -w_8^2 \end{bmatrix} \tag{4.41}$$

$$\boldsymbol{A_3} = \boldsymbol{I_4} \otimes \mathbf{F_2} \tag{4.42}$$

$$\boldsymbol{P_8} = [e_0 \quad e_4 \quad e_2 \quad e_6 \quad e_1 \quad e_5 \quad e_3 \quad e_7] \tag{4.43}$$

其中$w_8 = 0.707107 - 0.707107j$。

$\boldsymbol{F_8}$使用的矩阵连乘形式为：

$$\boldsymbol{F_8} = \boldsymbol{A_1}\boldsymbol{A_2}(\mathbf{I_4} \otimes \boldsymbol{F}_2)\mathbf{P_8} \tag{4.44}$$

（4）t =4 时，$N = 2^t = 16$，16 维 DFT 矩阵$\boldsymbol{F_{16}}$经过分解算法得到的矩阵数量为 K=5，矩阵为：

$$\boldsymbol{A_1} = \begin{bmatrix} \boldsymbol{I_8} & \boldsymbol{A_{1_{0,1}}} \\ \boldsymbol{I_8} & \boldsymbol{A_{1_{1,1}}} \end{bmatrix}$$

$$\boldsymbol{A_{1_{0,1}}} = \begin{bmatrix} w_{16}^0 & & & & & & & \\ & w_{16}^1 & & & & & & \\ & & w_{16}^2 & & & & & \\ & & & w_{16}^3 & & & & \\ & & & & w_{16}^4 & & & \\ & & & & & w_{16}^5 & & \\ & & & & & & w_{16}^6 & \\ & & & & & & & w_{16}^7 \end{bmatrix} \tag{4.45}$$

$$\boldsymbol{A_{1_{1,1}}} = -\boldsymbol{A_{1_{0,1}}}$$

$$\boldsymbol{A_2} = \begin{bmatrix} \boldsymbol{A_{2_{0,0}}} & \\ & \boldsymbol{A_{2_{1,1}}} \end{bmatrix}$$

$$\boldsymbol{A_{2_{0,0}}} = \begin{bmatrix} 1 & & & w_{16}^0 & & & & \\ & 1 & & & w_{16}^2 & & & \\ & & 1 & & & w_{16}^4 & & \\ & & & 1 & & & w_{16}^6 & \\ 1 & & & & -w_{16}^0 & & & \\ & 1 & & & & -w_{16}^2 & & \\ & & 1 & & & & -w_{16}^4 & \\ & & & 1 & & & & -w_{16}^6 \end{bmatrix} \tag{4.46}$$

$$\boldsymbol{A_{2_{1,1}}} = \boldsymbol{A_{2_{0,0}}}$$

$$\boldsymbol{A_3} = \boldsymbol{I_4} \otimes \boldsymbol{A_{3_{0,0}}}$$

$$\boldsymbol{A_{3_{0,0}}} = \begin{bmatrix} 1 & w_{16}^0 & & \\ & 1 & & -w_{16}^4 \\ 1 & -w_{16}^0 & & \\ & 1 & & w_{16}^4 \end{bmatrix} \tag{4.47}$$

其中$w_{16} = 0.92388 - 0.382683j$。

$$\boldsymbol{A_4} = \boldsymbol{I_8} \otimes \mathbf{F_2} \tag{4.48}$$

$$\boldsymbol{P_{16}} = [e_0 \quad e_8 \quad e_4 \quad e_{12} \quad e_2 \quad e_{10} \quad e_6 \quad e_{14} \quad e_1 \quad e_9 \quad e_5 \quad e_{13} \quad e_3 \quad e_{11} \quad e_7 \quad e_{15}] \tag{4.49}$$

$\boldsymbol{F_{16}}$使用的矩阵连乘形式为：

$$\boldsymbol{F_{16}} = \boldsymbol{A_1 A_2 A_3} (\mathbf{I_8} \otimes \boldsymbol{F}_2) \mathbf{P_{16}} \tag{4.50}$$

（**5**）t =5 时，$N=2^t=32$，32 维 DFT 矩阵$\boldsymbol{F_{32}}$经过分解算法得到的矩阵数量为 K=6，矩阵为：

$$\boldsymbol{A_1}=\begin{bmatrix} \boldsymbol{I_8} & & \boldsymbol{A_{1_{0,2}}} & \\ & \boldsymbol{I_8} & & \boldsymbol{A_{1_{1,3}}} \\ \boldsymbol{I_8} & & \boldsymbol{A_{1_{2,2}}} & \\ & \boldsymbol{I_8} & & \boldsymbol{A_{1_{3,3}}} \end{bmatrix}$$

$$\boldsymbol{A_{1_{0,2}}}=\begin{bmatrix} w_{32}^{0} & & & & & & & \\ & w_{32}^{1} & & & & & & \\ & & w_{32}^{2} & & & & & \\ & & & w_{32}^{3} & & & & \\ & & & & w_{32}^{4} & & & \\ & & & & & w_{32}^{5} & & \\ & & & & & & w_{32}^{6} & \\ & & & & & & & w_{32}^{7} \end{bmatrix}$$

$$\boldsymbol{A_{1_{1,3}}}=\begin{bmatrix} w_{32}^{8} & & & & & & & \\ & w_{32}^{9} & & & & & & \\ & & w_{32}^{10} & & & & & \\ & & & w_{32}^{11} & & & & \\ & & & & w_{32}^{12} & & & \\ & & & & & w_{32}^{13} & & \\ & & & & & & w_{32}^{14} & \\ & & & & & & & w_{32}^{15} \end{bmatrix}$$

(4.51)

$$\boldsymbol{A_{1_{2,2}}}=-\boldsymbol{A_{1_{0,2}}}$$

$$\boldsymbol{A_{1_{3,3}}}=-\boldsymbol{A_{1_{1,3}}}$$

$$\boldsymbol{A_2}=\begin{bmatrix}\boldsymbol{I_8} & \boldsymbol{A_{2_{0,1}}} & & \\ \boldsymbol{I_8} & \boldsymbol{A_{2_{1,1}}} & & \\ & & \boldsymbol{I_8} & \boldsymbol{A_{2_{2,3}}} \\ & & \boldsymbol{I_8} & \boldsymbol{A_{2_{3,3}}}\end{bmatrix}$$

$$\boldsymbol{A_{2_{0,1}}}=\begin{bmatrix} w_{32}^{0} & & & & & & & \\ & w_{32}^{2} & & & & & & \\ & & w_{32}^{4} & & & & & \\ & & & w_{32}^{6} & & & & \\ & & & & w_{32}^{8} & & & \\ & & & & & w_{32}^{10} & & \\ & & & & & & w_{32}^{12} & \\ & & & & & & & w_{32}^{14}\end{bmatrix} \tag{4.52}$$

$$\boldsymbol{A_{2_{1,1}}}=\boldsymbol{A_{2_{3,3}}}=-\boldsymbol{A_{2_{0,1}}}$$

$$\boldsymbol{A_{2_{2,3}}}=\boldsymbol{A_{2_{0,1}}}$$

$$\boldsymbol{A_3}=\boldsymbol{I_4}\otimes\boldsymbol{A_{3_{0,0}}}$$

$$\boldsymbol{A_{3_{0,0}}}=\begin{bmatrix} 1 & & & & w_{32}^{0} & & & \\ & 1 & & & & w_{32}^{4} & & \\ & & 1 & & & & w_{32}^{8} & \\ & & & 1 & & & & w_{32}^{12} \\ 1 & & & & -w_{32}^{0} & & & \\ & 1 & & & & -w_{32}^{4} & & \\ & & 1 & & & & -w_{32}^{8} & \\ & & & 1 & & & & -w_{32}^{12}\end{bmatrix} \tag{4.53}$$

$$\boldsymbol{A_4}=\boldsymbol{I_8}\otimes\boldsymbol{A_{4_{0,0}}}$$

$$\boldsymbol{A_{4_{0,0}}}=\begin{bmatrix} 1 & & w_{32}^{0} & \\ & 1 & & -w_{32}^{8} \\ 1 & & -w_{32}^{0} & \\ & 1 & & w_{32}^{8}\end{bmatrix} \tag{4.54}$$

$$\boldsymbol{A_5}=\boldsymbol{I_{16}}\otimes\mathbf{F_2} \tag{4.55}$$

$$\boldsymbol{P_{32}}=[e_0\ \ e_{16}\ \ e_8\ \ e_{24}\ \ e_4\ \ e_{20}\ \ e_{12}\ \ e_{28}\ \ e_2\ \ e_{18}\ \ e_{10}\ \ e_{26}\ \ e_6\ \ e_{22}\ \ e_{14}\ \ e_{30}\ \ e_1\ \ e_{17}\ \ e_9\ \ e_{25}\ \ e_5\ \ e_{21}\ \ e_{13}\ \ e_{29}\ \ e_3\ \ e_{19}\ \ e_{11}\ \ e_{27}\ \ e_7\ \ e_{23}\ \ e_{15}\ \ e_{31}] \tag{4.56}$$

（6）当$t=6,7,\ldots$时，$N=2^t=64,128,\ldots$，N维 DFT 矩阵$\boldsymbol{F_N}$经过分解算法得到的矩阵数量为 K=t+1，$\boldsymbol{F_N}$使用的矩阵连乘形式为：

$$\boldsymbol{F_N}=\left(\prod_{i=1}^{\log_2 \mathrm{N}-1}\boldsymbol{A_i}\right)(\mathbf{I}_{\mathrm{N}/2}\otimes\boldsymbol{F_2})\mathbf{P}_{\mathrm{N}} \tag{4.57}$$

4.6. 求解结果分析

这样的递归算法对 N 为 DFT 矩阵$\boldsymbol{F_N}$的分解需要递归的层数为 $\log_2 \mathrm{N}-1$，这样的算法

可以快速完成分解。对$t = 1,2,3,\ldots,10$ 时，N=2^t维 DFT 矩阵$\boldsymbol{F_N}$的分解矩阵个数，误差，硬件复杂度进行了统计，如表 4.2 所示。发现可以得到误差几乎均为 0，和原始 DFT 矩阵计算的硬件复杂度相比分解后的矩阵连乘硬件复杂度有着明显的下降，但由于矩阵某些元素所需的数据位宽较高，仍使得矩阵的硬件复杂度不太理想。

其中计算 DFT 的硬件复杂度仅考虑乘法器的硬件复杂度：

$$C = q \times L \tag{4.58}$$

其中 q 为乘法器位宽，L 为复数乘法次数。

表 4-2　N = 2^t维 DFT 矩阵分解结果统计表

t	N	β	$\boldsymbol{\mathcal{A}}$中矩阵个数	RMSE	C	耗时
1	2	1	2	1.2265e-16	0	1ms
2	4	1	3	7.9362e-16	0	0ms
3	8	1	4	1.1261e-14	64	2ms
4	16	1	5	5.5051e-14	576	5ms
5	32	1	6	2.6016e-13	3392	22ms
6	64	1	7	3.5867e-12	16704	96ms
7	128	1	8	8.6604e-12	75072	526ms
8	256	1	9	1.6496e-10	320832	2069ms
9	512	1	10	1.6853e-09	1332544	9073ms
10	1024	1	11	1.1969e-08	5444928	43068ms

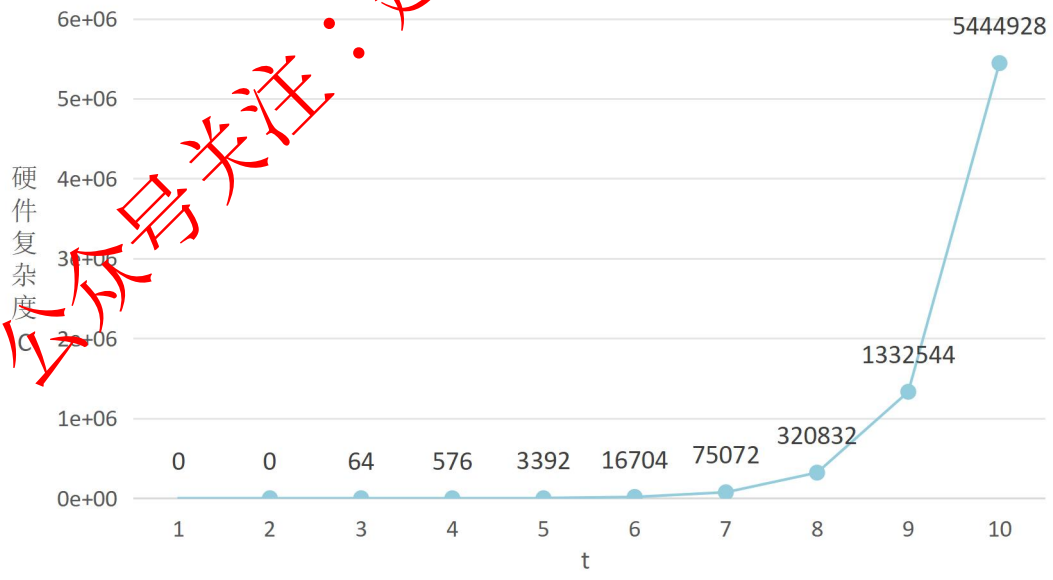

图 4-4　N = 2^t维 DFT 矩阵分解结果硬件复杂度变化折线图

5. 问题 2：DFT 矩阵的整数矩阵逼近

5.1. DFT 矩阵的整数矩阵逼近分析

减少计算 DFT 矩阵所需的硬件复杂度可以通过减少复数乘法的运算次数，即约束分解后的矩阵为稀疏矩阵的方式实现，也可以通过减少单个复数乘法器的硬件复杂度来降低总体的硬件复杂度。单个复数乘法器的硬件复杂度由输入数据的位宽决定，缩小输入数据的取值范围可以有效的减少单个乘法器所需的位宽从而降低单个复数乘法器的硬件复杂度。对于复数 g∈ $\{x+jy|x,y\in\mathcal{P}\},\mathcal{P}=\{0,\pm1,\pm2,\ldots,\pm2^{q-1}\}$,其与任意复数 z 相乘的复杂度为 q，q 越小，单个复数乘法器的硬件复杂度就越小。使用缩小输入数据的位宽来降低 DFT 矩阵计算过程的硬件复杂度，需要在设计矩阵连乘拟合 DFT 矩阵方案中限定 q 的值来限定输入数据的位宽，在这种限定下使得矩阵连乘方案的误差最小。将误差最小作为目标，将矩阵元素位宽 q=3 作为约束，连乘矩阵和矩阵缩放因子作为决策变量，可以建立出多元混合整数优化模型。

想要求解这个多元混合整数优化模型最直观的方式为穷举法，穷举矩阵连乘个数 K，穷举矩阵每个元素的整数值，这样穷举的次数为$\sum_{K=1}^{t+1}(2q+1)^{2KN^2}$，当 t=0,q=3,N=8 时，穷举次数为$7^{128}\approx1.49*10^{108}$次，这样的穷举过程明显不可取。因此，为了保证矩阵复数乘法不高且可以使用穷举法，设 K=1，使用一个整数矩阵逼近 DFT 矩阵。利用 DFT 矩阵的性质可以使用参数化向量将 DFT 矩阵表示出来，这样的参数化向量元素个数为 N/4,以这个参数化向量和矩阵缩放因子作为决策变量修改模型可以大幅降低求解的难度，且能在一定程度上满足误差要求，可以使用 Gurobi 求解。

5.2. DFT 矩阵的整数矩阵分解逼近模型

这一降低 DFT 矩阵计算硬件复杂度的方法规定矩阵连乘拟合 DFT 矩阵中连乘的矩阵的元素为 g∈ $\{x+jy|x,y\in\mathcal{P}\},\mathcal{P}=\{0,\pm1,\pm2,\ldots,\pm2^{q-1}\}$, 其中 q=3，以此来降低单个复数乘法器的硬件复杂度。所以在设计连乘矩阵时，需要将这一元素范围限定要求作为约束条件，将矩阵连乘拟合的结果误差尽可能小作为目标，矩阵个数、矩阵元素、矩阵缩放因子作为决策变量，建立 DFT 矩阵的整数矩阵分解逼近模型，该模型为一个多元混合整数优化模型。

（1）决策变量

矩阵连乘拟合 DFT 矩阵需要设计的变量有：

① 连乘的矩阵个数 K：

$$K\in\{1,2,3,4,\ldots\} \tag{5.1}$$

② 连乘中第 i 个矩阵$\mathbf{A_i}$的第 k 行，第 n 列的元素的值$[\mathbf{B_i}]_{k,n}$：

$$[\mathbf{A_i}]_{k,n}\in\{x+jy|x,y\in\mathcal{P}\},,i=1,2,\ldots,K;k,n=0,1,\ldots,N-1 \tag{5.2}$$

其中$\mathcal{P}=\{0,\pm1,\pm2,\ldots,\pm2^{q-1}\}$，这里设 q=3，$\mathcal{P}=\{0,\pm1,\pm2,\pm4\}$。

③ 实值矩阵缩放因子β：

$$\beta\in\mathcal{R} \tag{5.3}$$

其中$\mathcal{R}$表示实数域。

（2）目标函数

本问题使用 K 个整数矩阵连乘拟合近似 DFT 矩阵，所以本问题的目标为 K 个矩阵连乘尽可能和原 DFT 矩阵误差最小。使用 Frobenius 范数意义下矩阵$\mathbf{F}_N$和$\beta\mathbf{A}_1\mathbf{A}_2\cdots\mathbf{A}_K$的误差作为矩阵连乘拟合 DFT 矩阵的误差，即：

$$\min \mathrm{RMSE} = \frac{1}{N}\sqrt{\|\mathbf{F}_N - \beta\mathbf{A}_1\mathbf{A}_2\cdots\mathbf{A}_K\|_F^{\,2}} \tag{5.4}$$

（3）约束条件

为减少计算 DFT 矩阵的每个乘法器的硬件复杂度并使矩阵连乘可以正常拟合 DFT 矩阵，作出如下约束：

① 矩阵$\mathbf{A_i}$为 N 行 N 列的矩阵；

② 矩阵$\mathbf{A_i}$的每个元素在位宽为 q 可以表示的一个整数域内，即：

$$[\mathbf{A_i}]_{\mathrm{k,n}} \in \{x + jy | x, y \in \mathcal{P}\}, \mathcal{P} = \{0, \pm 1, \pm 2, \ldots, \pm 2^{q-1}\}, \mathrm{i} = 1,2,\ldots,K;\ \mathrm{k}, n = 0,2,\ldots,N-1 \tag{5.5}$$

综上所述，问题二 DFT 矩阵的整数矩阵分解逼近模型总结如下：

$$\min \mathrm{RMSE} = \frac{1}{N}\sqrt{\|\mathbf{F}_N - \beta\mathbf{A}_1\mathbf{A}_2\cdots\mathbf{A}_K\|_F^{\,2}}$$

$$\mathrm{s.t.}\begin{cases} \mathrm{K} \in \{1,2,3,4,\ldots\} \\ [\mathbf{A_i}]_{\mathrm{k,n}} \in \{x + jy | x, y \in \mathcal{P}\}, \mathrm{i} = 1,2,\ldots,K;\ \mathrm{k}, n = 0,2,\ldots,N-1 \\ \mathcal{P} = \{0, \pm 1, \pm 2, \pm 4\} \end{cases} \tag{5.6}$$

5.3. 模型求解——优化 DFT 矩阵的参数化向量

5.3.1 DFT 矩阵的参数化向量表示

N 维 DFT 矩阵中的元素可以表示为：

$$[\mathbf{F_N}]_{\mathrm{k,n}} = w_N^{nk} \tag{5.7}$$

其中的w_N^{nk}表示w_N的 nk 次方，w_N为 N 点旋转因子，满足折半性、消去性、周期性。

（1）利用消去性$w_{\mathrm{N}}^{\mathrm{k+N/2}} = -w_{\mathrm{N}}^{\mathrm{k}}$可以发现$\mathbf{F_N}$内元素存在关系：

$$[\mathbf{F_N}]_{\mathrm{k}+\frac{\mathrm{N}}{2},\mathrm{n}} = (w_N^{\frac{\mathrm{N}}{2}})^{\mathrm{n}}[\mathbf{F_N}]_{\mathrm{k,n}} = (-1)^{\mathrm{n}}[\mathbf{F_N}]_{\mathrm{k,n}} \tag{5.8}$$

其中 $\mathrm{k} = 0,1,\ldots,N/2-1$ 并且 $\mathrm{n} = 0,1,\ldots,N$;

$$[\mathbf{F_N}]_{\mathrm{k},\mathrm{n}+\frac{\mathrm{N}}{2}} = (w_N^{\frac{\mathrm{N}}{2}})^{\mathrm{k}}[\mathbf{F_N}]_{\mathrm{k,n}} = (-1)^{\mathrm{k}}[\mathbf{F_N}]_{\mathrm{k,n}} \tag{5.9}$$

其中 $\mathrm{n} = 0,1,\ldots,N/2-1$ 并且 $\mathrm{k} = 0,1,\ldots,N$。

利用（5.8）和（5.9）可以将 N 维的 DFT 矩阵分为 4 块：

$$\boldsymbol{F_N} = \begin{bmatrix} \boldsymbol{G_{0,0}} & \boldsymbol{G_{0,1}} \\ \boldsymbol{G_{1,0}} & \boldsymbol{G_{1,1}} \end{bmatrix} \tag{5.10}$$

其中其他三块可以使用$\boldsymbol{G_{0,0}}$表示出来：

$$
\begin{aligned}
[\boldsymbol{G_{0,1}}]_{\mathrm{k,n}} &= (-1)^{\mathrm{k}}[\boldsymbol{G_{0,0}}]_{\mathrm{k,n}} \\
[\boldsymbol{G_{1,0}}]_{\mathrm{k,n}} &= (-1)^{\mathrm{n}}[\boldsymbol{G_{0,0}}]_{\mathrm{k,n}} \\
[\boldsymbol{G_{1,0}}]_{\mathrm{k,n}} &= (-1)^{\mathrm{k+n}+\frac{\mathrm{N}}{2}}[\boldsymbol{G_{0,0}}]_{\mathrm{k,n}}
\end{aligned}
\tag{5.11}
$$

（2）利用消去性$\boldsymbol{w}_{\mathbf{N}}^{\mathbf{k}+\mathbf{N}/4} = (-\mathbf{j})\boldsymbol{w}_{\mathbf{N}}^{\mathbf{k}}$可以发现$\boldsymbol{G_{0,0}}$内元素存在关系：

$$
[\boldsymbol{G_{0,0}}]_{\mathrm{k}+\frac{\mathrm{N}}{4},\mathrm{n}} = (w_N^{\frac{\mathrm{N}}{4}})^{\mathrm{n}}[\boldsymbol{G_{0,0}}]_{\mathrm{k,n}} = (-\mathrm{j})^{\mathrm{n}}[\boldsymbol{G_{0,0}}]_{\mathrm{k,n}} \tag{5.12}
$$

其中 $\mathrm{k} = 0,1,\ldots,N/4-1$ 并且 $\mathrm{n} = 0,1,\ldots,N$;

$$
[\boldsymbol{G_{0,0}}]_{\mathrm{k,n}+\frac{\mathrm{N}}{4}} = (w_N^{\frac{\mathrm{N}}{4}})^{\mathrm{k}}[\boldsymbol{G_{0,0}}]_{\mathrm{k,n}} = (-\mathrm{j})^{\mathrm{k}}[\boldsymbol{G_{0,0}}]_{\mathrm{k,n}} \tag{5.13}
$$

其中 $\mathrm{n} = 0,1,\ldots,N/4-1$ 并且 $\mathrm{k} = 0,1,\ldots,N$。

利用（5.12）和（5.13）可以将$\boldsymbol{G_{0,0}}$矩阵分为 4 块：

$$
\boldsymbol{G_{0,0}} = \begin{bmatrix} \boldsymbol{H_{0,0}} & \boldsymbol{H_{0,1}} \\ \boldsymbol{H_{1,0}} & \boldsymbol{H_{1,1}} \end{bmatrix} \tag{5.14}
$$

其中其他三块可以使用$\boldsymbol{H_{0,0}}$表示出来：

$$
\begin{aligned}
[\boldsymbol{H_{0,1}}]_{\mathrm{k,n}} &= (-\mathrm{j})^{\mathrm{k}}[\boldsymbol{H_{0,0}}]_{\mathrm{k,n}} \\
[\boldsymbol{H_{1,0}}]_{\mathrm{k,n}} &= (-\mathrm{j})^{\mathrm{n}}[\boldsymbol{H_{0,0}}]_{\mathrm{k,n}} \\
[\boldsymbol{H_{1,1}}]_{\mathrm{k,n}} &= (-\mathrm{j})^{\mathrm{k+n}+\frac{\mathrm{N}}{4}}[\boldsymbol{H_{0,0}}]_{\mathrm{k,n}}
\end{aligned}
\tag{5.15}
$$

（3）利用 DFT 矩阵为范德蒙形式可以使用一个向量$\boldsymbol{a_{N/4}}$表示$\boldsymbol{H_{0,0}}$：

一个向量$\boldsymbol{a_{N/4}}$=$\{a_1, a_2,\ldots,a_{N/4}\}$，当$\boldsymbol{a_{N/4}}$=$\{w_N^0, w_N^1,\ldots,w_N^{\frac{\mathrm{N}}{4}-1}\}$时，可以用$\boldsymbol{a_{N/4}}$表示：

$$
[\boldsymbol{H_{0,0}}]_{\mathrm{k,n}} = ([\boldsymbol{a_{N/4}}]_{\mathbf{n}})^{\mathrm{k}} \tag{5.16}
$$

其中 $\mathrm{n} = 0,1,\ldots,N/4-1$ 并且 $\mathrm{k} = 0,1,\ldots,N/4-1$。

综上所述，利用上述 2 个分解过程可以将 N 为 DFT 矩阵分为 16 块，其他块均可以通过$\boldsymbol{H_{0,0}}$表示，$\boldsymbol{H_{0,0}}$可以使用$\boldsymbol{a_{N/4}}$=$\{w_N^0, w_N^1,\ldots,w_N^{\frac{\mathrm{N}}{4}-1}\}$表示，即 N 维 DFT 矩阵可以使用$\boldsymbol{a_{N/4}}$向量表示，$\boldsymbol{a_{N/4}}$称为 N 维 DFT 矩阵的参数化向量：

$$
\begin{aligned}
f\colon \mathbb{C}^{N/4-1} &\to \mathbb{C}^N \times \mathbb{C}^N \\
\boldsymbol{a_{N/4}} &\mapsto \boldsymbol{F_N}
\end{aligned}
\tag{5.17}
$$

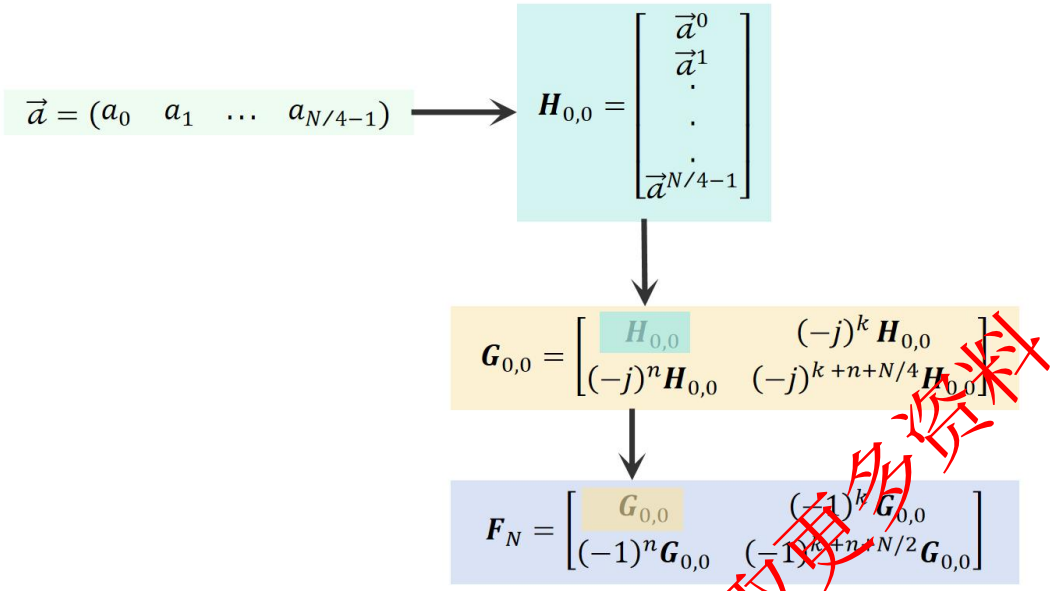

图 5-1　参数化表示 DFT 矩阵示意图

5.3.2 模型改进——优化参数化向量

由于参数化向量$\boldsymbol{a_{N/4}}$决定了 DFT 矩阵，所以将决策变量 DFT 矩阵的元素值变为参数化向量$\boldsymbol{a_{N/4}}$的元素值。这样的做法大幅降低了决策变量的个数，使得求解可以在合理的时间内完成。为使优化参数化向量得到的近似估计 DFT 矩阵$\widehat{\boldsymbol{F}}_{\boldsymbol{N}}$尽可能接近 DFT 矩阵$\boldsymbol{F_N}$，约束近似估计的 DFT 矩阵$\widehat{\boldsymbol{F}}_{\boldsymbol{N}}$尽可能满足对称性和正交性。由此修改后的模型为：

（1）决策变量

矩阵连乘拟合 DFT 矩阵需要设计的变量为：

① N 维 DFT 矩阵的参数化向量$\boldsymbol{a_{N/4}}$：

$$[\boldsymbol{a_{N/4}}]_{\boldsymbol{n}} \in \{x + jy | x, y \in \mathcal{R}\}, n = 0,1, \ldots, N/4 - 1 \tag{5.18}$$

其中$\mathcal{R}$表示实数域；

② 实值矩阵缩放因子β：

$$\beta \in \mathcal{R} \tag{5.19}$$

其中$\mathcal{R}$表示实数域。

（2）目标函数

本问题使用 K 个整数矩阵连乘拟合近似 DFT 矩阵，所以本问题的目标为 K 个矩阵连乘尽可能和原 DFT 矩阵误差最小。使用 Frobenius 范数意义下矩阵$\mathbf{F}_N$和$\beta\widehat{\boldsymbol{F}}_{\boldsymbol{N}}$的误差作为矩阵连乘拟合 DFT 矩阵的误差，即：

$$\min \text{RMSE} = \frac{1}{N}\sqrt{\left\|\mathbf{F}_N - \beta\widehat{\boldsymbol{F}}_{\boldsymbol{N}}\right\|_F^2} \tag{5.20}$$

（3）约束条件

为减少计算 DFT 矩阵的每个乘法器的硬件复杂度并使矩阵连乘可以正常拟合 DFT 矩阵，作出如下约束：

① 近似估计矩阵$\widehat{\boldsymbol{F}}_{\boldsymbol{N}}$为使用参数化向量$\boldsymbol{a}_{\boldsymbol{N}/\boldsymbol{4}}$构建的矩阵：

$$
\begin{aligned}
[\boldsymbol{H}_{\boldsymbol{0,0}}]_{\mathrm{k,n}} &= ([\boldsymbol{a}_{\boldsymbol{N}/\boldsymbol{4}}]_{\mathbf{n}})^{\mathrm{k}} \\
[\boldsymbol{H}_{\boldsymbol{0,1}}]_{\mathrm{k,n}} &= (-\mathrm{j})^{\mathrm{k}}[\boldsymbol{H}_{\boldsymbol{0,0}}]_{\mathrm{k,n}} \\
[\boldsymbol{H}_{\boldsymbol{1,0}}]_{\mathrm{k,n}} &= (-\mathrm{j})^{\mathrm{n}}[\boldsymbol{H}_{\boldsymbol{0,0}}]_{\mathrm{k,n}} \\
[\boldsymbol{H}_{\boldsymbol{1,1}}]_{\mathrm{k,n}} &= (-\mathrm{j})^{\mathrm{k+n}+\frac{\mathrm{N}}{4}}[\boldsymbol{H}_{\boldsymbol{0,0}}]_{\mathrm{k,n}} \\
\boldsymbol{G}_{\boldsymbol{0,0}} &= \begin{bmatrix} \boldsymbol{H}_{\boldsymbol{0,0}} & \boldsymbol{H}_{\boldsymbol{0,1}} \\ \boldsymbol{H}_{\boldsymbol{1,0}} & \boldsymbol{H}_{\boldsymbol{1,1}} \end{bmatrix} \\
[\boldsymbol{G}_{\boldsymbol{0,1}}]_{\mathrm{k,n}} &= (-1)^{\mathrm{k}}[\boldsymbol{G}_{\boldsymbol{0,0}}]_{\mathrm{k,n}} \\
[\boldsymbol{G}_{\boldsymbol{1,0}}]_{\mathrm{k,n}} &= (-1)^{\mathrm{n}}[\boldsymbol{G}_{\boldsymbol{0,0}}]_{\mathrm{k,n}} \\
[\boldsymbol{G}_{\boldsymbol{1,0}}]_{\mathrm{k,n}} &= (-1)^{\mathrm{k+n}+\frac{\mathrm{N}}{2}}[\boldsymbol{G}_{\boldsymbol{0,0}}]_{\mathrm{k,n}} \\
\widehat{\boldsymbol{F}}_{\boldsymbol{N}} &= \begin{bmatrix} \boldsymbol{G}_{\boldsymbol{0,0}} & \boldsymbol{G}_{\boldsymbol{0,1}} \\ \boldsymbol{G}_{\boldsymbol{1,0}} & \boldsymbol{G}_{\boldsymbol{1,1}} \end{bmatrix}
\end{aligned}
\tag{5.21}
$$

② 近似估计矩阵$\widehat{\boldsymbol{F}}_{\boldsymbol{N}}$满足正交性或近似正交性：

$$
\mathrm{ORTHO} = 1 - \frac{\left\| diag(\widehat{\boldsymbol{F}}_{\boldsymbol{N}}\widehat{\boldsymbol{F}}_{\boldsymbol{N}}{}^{\boldsymbol{H}}) \right\|_F}{\left\| \widehat{\boldsymbol{F}}_{\boldsymbol{N}}\widehat{\boldsymbol{F}}_{\boldsymbol{N}}{}^{\boldsymbol{H}} \right\|_F} \leq 0.2 \tag{5.22}
$$

③ 近似估计矩阵$\widehat{\boldsymbol{F}}_{\boldsymbol{N}}$满足可逆性：

$$
\mathrm{INVE} = \det(\widehat{\boldsymbol{F}}_{\boldsymbol{N}}) \neq 0 \tag{5.23}
$$

④ 根据$\mathbf{F}_N$的第一行和第一列全为 1 可以得到参数向量的第一个元素为 1：

$$
[\boldsymbol{a}_{\boldsymbol{N}/\boldsymbol{4}}]_{\mathbf{0}} = 1 \tag{5.24}
$$

综上所述，问题二 DFT 矩阵的整数矩阵分解逼近的改进模型总结如下：

$$
\begin{aligned}
&\min \mathrm{RMSE} = \frac{1}{N}\sqrt{\left\|\mathbf{F}_N - \beta\widehat{\boldsymbol{F}}_{\boldsymbol{N}}\right\|_F^{\,2}} \\
&\text{s.t.}\begin{cases}
[\boldsymbol{H_{0,0}}]_{\mathrm{k,n}} = ([\boldsymbol{a_{N/4}}]_{\mathbf{n}})^{\mathrm{k}} \\
[\boldsymbol{H_{0,1}}]_{\mathrm{k,n}} = (-\mathrm{j})^{\mathrm{k}}[\boldsymbol{H_{0,0}}]_{\mathrm{k,n}} \\
[\boldsymbol{H_{1,0}}]_{\mathrm{k,n}} = (-\mathrm{j})^{\mathrm{n}}[\boldsymbol{H_{0,0}}]_{\mathrm{k,n}} \\
[\boldsymbol{H_{1,1}}]_{\mathrm{k,n}} = (-\mathrm{j})^{\mathrm{k+n+\frac{N}{4}}}[\boldsymbol{H_{0,0}}]_{\mathrm{k,n}} \\
\boldsymbol{G_{0,0}} = \begin{bmatrix}\boldsymbol{H_{0,0}} & \boldsymbol{H_{0,1}} \\ \boldsymbol{H_{1,0}} & \boldsymbol{H_{1,1}}\end{bmatrix} \\
[\boldsymbol{G_{0,1}}]_{\mathrm{k,n}} = (-1)^{\mathrm{k}}[\boldsymbol{G_{0,0}}]_{\mathrm{k,n}} \\
[\boldsymbol{G_{1,0}}]_{\mathrm{k,n}} = (-1)^{\mathrm{n}}[\boldsymbol{G_{0,0}}]_{\mathrm{k,n}} \\
[\boldsymbol{G_{1,0}}]_{\mathrm{k,n}} = (-1)^{\mathrm{k+n+\frac{N}{2}}}[\boldsymbol{G_{0,0}}]_{\mathrm{k,n}} \\
\widehat{\boldsymbol{F}}_{\boldsymbol{N}} = \begin{bmatrix}\boldsymbol{G_{0,0}} & \boldsymbol{G_{0,1}} \\ \boldsymbol{G_{1,0}} & \boldsymbol{G_{1,1}}\end{bmatrix} \\
[\boldsymbol{a_{N/4}}]_{\mathbf{n}} \in \{x + jy | x, y \in \mathcal{R}\}, n = 0,1,\ldots,N/4-1 \\
\mathrm{ORTHO} = 1 - \dfrac{\left\|diag(\widehat{\boldsymbol{F}}_{\boldsymbol{N}}\widehat{\boldsymbol{F}}_{\boldsymbol{N}}{}^{\boldsymbol{H}})\right\|_F}{\left\|\widehat{\boldsymbol{F}}_{\boldsymbol{N}}\widehat{\boldsymbol{F}}_{\boldsymbol{N}}{}^{\boldsymbol{H}}\right\|_F} \le 0.2 \\
\mathrm{INVE} = \det(\widehat{\boldsymbol{F}}_{\boldsymbol{N}}) \neq 0 \\
[\boldsymbol{a_{N/4}}]_{\boldsymbol{0}} = 1
\end{cases}
\end{aligned}
\tag{5.25}
$$

5.4. 模型求解结果

（1）t =1 时，$N = 2^t = 2$，2 维 DFT 矩阵$\boldsymbol{F}_2$元素均在规定的整数范围内，近似矩阵为原矩阵：

$$\widehat{\boldsymbol{F}}_{\boldsymbol{2}} = \boldsymbol{F_2} = \begin{bmatrix}1 & 1 \\ 1 & -1\end{bmatrix} \tag{5.26}$$

（2）t =2 时，$N = 2^t = 4$，4 维 DFT 矩阵$\boldsymbol{F_4}$元素均在规定的整数范围内，近似矩阵为原矩阵：

$$\widehat{\boldsymbol{F}}_{\boldsymbol{4}} = \boldsymbol{F_4} = \begin{bmatrix}1 & & 1 & \\ & 1 & & -j \\ 1 & & -1 & \\ & 1 & & j\end{bmatrix} \tag{5.27}$$

（3）t =3 时，$N = 2^t = 8$，8 维 DFT 矩阵$\boldsymbol{F_8}$经过优化算法求解后得到的缩放因子β=0.5，参数向量和近似估计矩阵$\widehat{\boldsymbol{F}}_{\boldsymbol{8}}$为：

$$\boldsymbol{a_2}=\{1,(1-j)/2\}$$

$$\widehat{\boldsymbol{F}}_{\boldsymbol{8}}=\begin{bmatrix}2 & 2 & 2 & 2 & 2 & 2 & 2 & 2\\ 2 & 1-j & -2j & -1-j & -2 & -1+j & 2j & 1+j\\ 2 & -2j & -2 & 2j & 2 & -2j & -2 & 2j\\ 2 & -1-j & 2 & 1-j & -2 & 1+j & -2j & -1+j\\ 2 & -2 & 2 & -2 & 2 & -2 & 2 & -2\\ 2 & -1+j & -2j & 1+j & -2 & 1-j & 2j & -1-j\\ 2 & 2j & -2 & -2j & 2 & 2j & -2 & -2j\\ 2 & 1+j & 2j & -1+j & -2 & -1-j & -2j & 1-j\end{bmatrix} \tag{5.28}$$

（**4**）t =4 时，$N=2^t$ =16，16 维 DFT 矩阵$\boldsymbol{F}_{\boldsymbol{16}}$经过优化算法求解后得到的缩放因子$\beta$=0.25，参数向量和近似估计矩阵$\widehat{\boldsymbol{F}}_{\boldsymbol{16}}$为：

$$\boldsymbol{a}_{\boldsymbol{16}}=\{1,(1-j)/2,(1-j)/2,(1-j)/2\}$$

$$\widehat{\boldsymbol{F}}_{\boldsymbol{16}}=\begin{bmatrix}A_{1_{0,0}} & A_{1_{0,1}}\\ A_{1_{1,0}} & A_{1_{1,1}}\end{bmatrix} \tag{5.29}$$

$$A_{1_{0,0}}=\begin{bmatrix}4 & 4 & 4 & 4 & 4 & 4 & 4 & 4\\ 4 & 2-2j & 2-2j & 2-2j & -4j & -2-2j & -2-2j & -2-2j\\ 4 & -2j & -2j & -2j & -4 & 2j & 2j & 2j\\ 4 & -1-j & -1-j & -1-j & 4j & 1-j & 1-j & 1-j\\ 4 & -4j & -4 & 4j & 4 & -4j & -4 & 4j\\ 4 & -2-2j & -2+2j & 2+2j & -4j & -2+2j & 2+2j & 2-2j\\ 4 & -2 & 2j & 2 & -4 & 2 & -2j & -2\\ 4 & -1+j & 1+j & 1-j & 4j & -1-j & -1+j & 1+j\end{bmatrix}$$

$$A_{1_{0,1}}=\begin{bmatrix}4 & 4 & 4 & 4 & 4 & 4 & 4 & 4\\ -4 & -2+2j & -2+2j & -2+2j & 4j & 2+2j & 2+2j & 2+2j\\ 4 & -2j & -2j & -2j & -4 & 2j & 2j & 2j\\ -4 & 1+j & 1+j & 1+j & -4j & -1+j & -1+j & -1+j\\ 4 & -4j & -4 & 4j & 4 & -4j & -4 & 4j\\ -4 & 2+2j & 2-2j & -2-2j & 4j & 2-2j & -2-2j & -2+2j\\ 4 & 2 & 4j & 2 & -4 & 2 & -2j & -2\\ -4 & 1-j & -1-j & -1+j & -4j & 1+j & 1-j & -1-j\end{bmatrix}$$

$$A_{1_{1,0}}=\begin{bmatrix}4 & -4 & 4 & -4 & 4 & -4 & 4 & -4\\ 4 & -2+2j & 2-2j & -2+2j & -4j & 2+2j & -2-j & 2+2j\\ 4 & 2j & -2j & 2j & -4 & -2j & 2j & -2j\\ 4 & 1+j & -1-j & 1+j & 4j & -1+j & 1-j & -1+j\\ 4 & 4j & -4 & -4j & 4 & 4j & -4 & -4j\\ 4 & 2+2j & -2+2j & -2-2j & -4j & 2-2j & 2+2j & -2+2j\\ 4 & 2 & 2j & -2 & -4 & -2 & -2j & 2\\ 4 & 1-j & 1+j & -1+j & 4j & 1+j & -1+j & -1-j\end{bmatrix}$$

$$A_{1_{1,1}}=\begin{bmatrix} 4 & -4 & 4 & -4 & 4 & -4 & 4 & -4 \\ -4 & 2-2j & -2+2j & 2-2j & 4j & -2-2j & 2+2j & -2-2j \\ 4 & 2j & -2j & 2j & -4 & -2j & 2j & -2j \\ -4 & -1-j & 1+j & -1-j & -4j & 1-j & -1+j & 1-j \\ 4 & 4j & -4 & -4j & 4 & 4j & -4 & -4j \\ -4 & -2-2j & 2-2j & 2+2j & 4j & -2+2j & -2-2j & 2-2j \\ 4 & 2 & 2j & -2 & -4 & -2 & -2j & 2 \\ -4 & -1+j & -1-j & 1-j & -4j & -1-j & 1-j & 1+j \end{bmatrix}$$

（**5**）t =5 时，$N=2^t$ =32，32 维 DFT 矩阵$\boldsymbol{F_{32}}$经过优化算法求解后得到的缩放因子β=1，参数向量和近似估计矩阵$\widehat{\boldsymbol{F}}_{\boldsymbol{32}}$为：

$$\boldsymbol{a_{N/4}}=\{1,1,0,0,0,0,0,-j\}$$

$$\widehat{\boldsymbol{F}}_{\boldsymbol{32}}=\begin{bmatrix} A_{1_{0,0}} & A_{1_{0,1}} & A_{1_{0,2}} & A_{1_{0,3}} \\ A_{1_{1,0}} & A_{1_{1,1}} & A_{1_{1,2}} & A_{1_{1,3}} \\ A_{1_{2,0}} & A_{1_{2,1}} & A_{1_{2,2}} & A_{1_{2,3}} \\ A_{1_{3,0}} & A_{1_{3,1}} & A_{1_{3,2}} & A_{1_{3,3}} \end{bmatrix} \tag{5.30}$$

$$\boldsymbol{A_{1_{0,0}}}=\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & -j \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & -1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & j \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & -j \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & -1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & j \end{bmatrix} \quad \boldsymbol{A_{1_{0,1}}}=\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ -j & -j & 0 & 0 & 0 & 0 & 0 & -1 \\ -1 & -1 & 0 & 0 & 0 & 0 & 0 & 1 \\ j & j & 0 & 0 & 0 & 0 & 0 & -1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ -j & -j & 0 & 0 & 0 & 0 & 0 & -1 \\ -1 & -1 & 0 & 0 & 0 & 0 & 0 & 1 \\ j & j & 0 & 0 & 0 & 0 & 0 & -1 \end{bmatrix}$$

$$\boldsymbol{A_{1_{0,2}}}=\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ -1 & -1 & 0 & 0 & 0 & 0 & 0 & j \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & -1 \\ -1 & -1 & 0 & 0 & 0 & 0 & 0 & -j \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ -1 & -1 & 0 & 0 & 0 & 0 & 0 & j \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & -1 \\ -1 & -1 & 0 & 0 & 0 & 0 & 0 & -j \end{bmatrix} \quad \boldsymbol{A_{1_{0,3}}}=\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ j & j & 0 & 0 & 0 & 0 & 0 & 1 \\ -1 & -1 & 0 & 0 & 0 & 0 & 0 & 1 \\ -j & -j & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\ j & j & 0 & 0 & 0 & 0 & 0 & 1 \\ -1 & -1 & 0 & 0 & 0 & 0 & 0 & 1 \\ -j & -j & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\boldsymbol{A_{1_{1,0}}}=\begin{bmatrix} 1 & -j & -1 & j & 1 & -j & -1 & j \\ 1 & -j & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & -j & 0 & 0 & 0 & 0 & 0 & -j \\ 1 & -j & 0 & 0 & 0 & 0 & 0 & -1 \\ 1 & -j & 0 & 0 & 0 & 0 & 0 & j \\ 1 & -j & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & -j & 0 & 0 & 0 & 0 & 0 & -j \\ 1 & -j & 0 & 0 & 0 & 0 & 0 & -1 \end{bmatrix} \quad \boldsymbol{A_{1_{1,1}}}=\begin{bmatrix} 1 & -j & -1 & j & 1 & -j & -1 & j \\ -j & -1 & 0 & 0 & 0 & 0 & 0 & -j \\ -1 & j & 0 & 0 & 0 & 0 & 0 & j \\ j & 1 & 0 & 0 & 0 & 0 & 0 & -j \\ 1 & -j & 0 & 0 & 0 & 0 & 0 & j \\ -j & -1 & 0 & 0 & 0 & 0 & 0 & -j \\ -1 & j & 0 & 0 & 0 & 0 & 0 & j \\ j & 1 & 0 & 0 & 0 & 0 & 0 & -j \end{bmatrix}$$

$$A_{1_{1,2}}=\begin{bmatrix} 1 & -j & -1 & j & 1 & -j & -1 & j \\ -1 & j & 0 & 0 & 0 & 0 & 0 & -1 \\ 1 & -j & 0 & 0 & 0 & 0 & 0 & -j \\ -1 & j & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & -j & 0 & 0 & 0 & 0 & 0 & j \\ -1 & j & 0 & 0 & 0 & 0 & 0 & -1 \\ 1 & -j & 0 & 0 & 0 & 0 & 0 & -j \\ -1 & j & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} A_{1_{1,3}}=\begin{bmatrix} 1 & -j & -1 & j & 1 & -j & -1 & j \\ j & 1 & 0 & 0 & 0 & 0 & 0 & j \\ -1 & j & 0 & 0 & 0 & 0 & 0 & j \\ -j & -1 & 0 & 0 & 0 & 0 & 0 & j \\ 1 & -j & 0 & 0 & 0 & 0 & 0 & j \\ j & 1 & 0 & 0 & 0 & 0 & 0 & j \\ -1 & j & 0 & 0 & 0 & 0 & 0 & j \\ -j & -1 & 0 & 0 & 0 & 0 & 0 & j \end{bmatrix}$$

$$A_{1_{2,0}}=\begin{bmatrix} 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & j \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & -j \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & -1 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & j \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & -j \end{bmatrix} A_{1_{2,1}}=\begin{bmatrix} 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ -j & j & 0 & 0 & 0 & 0 & 0 & 1 \\ -1 & 1 & 0 & 0 & 0 & 0 & 0 & -1 \\ j & -j & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & -1 \\ -j & j & 0 & 0 & 0 & 0 & 0 & 1 \\ -1 & 1 & 0 & 0 & 0 & 0 & 0 & -1 \\ j & -j & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$A_{1_{2,2}}=\begin{bmatrix} 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ -1 & 1 & 0 & 0 & 0 & 0 & 0 & -j \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 1 \\ -1 & 1 & 0 & 0 & 0 & 0 & 0 & j \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & -1 \\ -1 & 1 & 0 & 0 & 0 & 0 & 0 & -j \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 1 \\ -1 & 1 & 0 & 0 & 0 & 0 & 0 & j \end{bmatrix} A_{1_{2,3}}=\begin{bmatrix} 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ j & -j & 0 & 0 & 0 & 0 & 0 & -1 \\ -1 & 1 & 0 & 0 & 0 & 0 & 0 & -1 \\ -j & j & 0 & 0 & 0 & 0 & 0 & -1 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & -1 \\ j & -j & 0 & 0 & 0 & 0 & 0 & -1 \\ -1 & 1 & 0 & 0 & 0 & 0 & 0 & -1 \\ -j & j & 0 & 0 & 0 & 0 & 0 & -1 \end{bmatrix}$$

$$A_{1_{3,0}}=\begin{bmatrix} 1 & j & -1 & -j & 1 & j & -1 & -j \\ 1 & j & 0 & 0 & 0 & 0 & 0 & -1 \\ 1 & j & 0 & 0 & 0 & 0 & 0 & j \\ 1 & j & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & j & 0 & 0 & 0 & 0 & 0 & -j \\ 1 & j & 0 & 0 & 0 & 0 & 0 & -1 \\ 1 & j & 0 & 0 & 0 & 0 & 0 & j \\ 1 & j & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} A_{1_{3,1}}=\begin{bmatrix} 1 & j & -1 & -j & 1 & j & -1 & -j \\ -j & 1 & 0 & 0 & 0 & 0 & 0 & j \\ -1 & -j & 0 & 0 & 0 & 0 & 0 & -j \\ j & -1 & 0 & 0 & 0 & 0 & 0 & j \\ 1 & j & 0 & 0 & 0 & 0 & 0 & -j \\ -j & 1 & 0 & 0 & 0 & 0 & 0 & j \\ -1 & -j & 0 & 0 & 0 & 0 & 0 & -j \\ j & -1 & 0 & 0 & 0 & 0 & 0 & j \end{bmatrix}$$

$$A_{1_{3,2}}=\begin{bmatrix} 1 & j & -1 & -j & 1 & j & -1 & -j \\ 1 & -j & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & j & 0 & 0 & 0 & 0 & 0 & j \\ -1 & -j & 0 & 0 & 0 & 0 & 0 & -1 \\ 1 & j & 0 & 0 & 0 & 0 & 0 & -j \\ -1 & -j & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & j & 0 & 0 & 0 & 0 & 0 & j \\ -1 & -j & 0 & 0 & 0 & 0 & 0 & -1 \end{bmatrix} A_{1_{3,3}}=\begin{bmatrix} 1 & j & -1 & -j & 1 & j & -1 & -j \\ j & -1 & 0 & 0 & 0 & 0 & 0 & -j \\ -1 & -j & 0 & 0 & 0 & 0 & 0 & -j \\ -j & 1 & 0 & 0 & 0 & 0 & 0 & -j \\ 1 & j & 0 & 0 & 0 & 0 & 0 & -j \\ j & -1 & 0 & 0 & 0 & 0 & 0 & -j \\ -1 & -j & 0 & 0 & 0 & 0 & 0 & -j \\ -j & 1 & 0 & 0 & 0 & 0 & 0 & -j \end{bmatrix}$$

5.5. 求解结果分析

这样的改进的 DFT 整数矩阵逼近模型在求解 N 维 DFT 近似矩阵时的决策变量仅为 N/4 个参数化向量元素和矩阵缩放因子β，这样的优化模型可以使用 gb 在较少的时间内得到误差尽可能小的整数逼近矩阵，该矩阵的元素全部在限定的范围内。对$t=1,2,3,4,5$时，2^t维 DFT 矩阵$\boldsymbol{F}_{2^t}$的矩阵缩放因子，误差，硬件复杂度进行了统计，如表 5.1 所示。

可以发现，和原始 DFT 矩阵计算的硬件复杂度相比，分解后的矩阵连乘硬件复杂度有着明显的下降。但由于矩阵的稀疏性不太理想，仍使得计算矩阵的硬件复杂度不太理想，并且由于 q=3 可表示的整数范围有限，当 DFT 矩阵维度大时，近似矩阵和 DFT 矩阵的误差并不理想。

其中计算 DFT 的硬件复杂度仅考虑乘法器的硬件复杂度：

$$C = q \times L \tag{5.31}$$

其中 q 为乘法器位宽，L 为复数乘法次数。

表 5-1　N=2^t维 DFT 矩阵近似矩阵分析统计表

t	N	β	$\mathcal{A}$中矩阵个数	RMSE	C
1	2	1	1	1.2265e-16	0
2	4	1	1	7.9362e-16	0
3	8	0.5	1	0.146447	144
4	16	0.25	1	0.49593	624
5	32	1	1	0.835666	0

6. 问题 3：DFT 矩阵的稀疏整数矩阵分解逼近

6.1. DFT 矩阵的稀疏整数矩阵分解逼近分析

为了使得矩阵连乘中硬件复杂度最小，同时降低复数乘法次数和每个复数乘法器的复杂度，即同时规定分解后矩阵的稀疏性和元素的取值范围。具体限定为：每个矩阵的每行至多有 2 个非零元素；每个矩阵的元素必须在属于$\{x+jy|x,y\in\mathcal{P}\},\mathcal{P}=\{0,\pm1,\pm2,\ldots,\pm2^{q-1}\}$。将这两个限定作为约束条件，将矩阵连乘结果与 DFT 矩阵的误差最小作为目标函数，建立优化连乘矩阵元素和矩阵缩放因子的多元混合整数优化模型。

直接求解该模型决策变量较为庞大，为一种不可取的思虑。所以为求解该多元混合整数优化模型，可以先使用问题一的分治策略将矩阵进行精确分解，这样得到的连乘矩阵均为每行至多 2 个非零元素的矩阵。但分解结果中存在元素不在约定的范围$\{x+jy|x,y\in\mathcal{P}\},\mathcal{P}=\{0,\pm1,\pm2,\ldots,\pm2^{q-1}\}$，此时仅需要考虑将矩阵元素全部限定到给定范围内。以矩阵元素全部在限定的整数范围内为约束条件，以矩阵连乘后的结果误差最小为目标，建立优化分解后取值非范围内的元素和矩阵缩放因子的多元混合整数优化模型。该模型需要优化的决策变量较少且为一个合理模型，对该模型可以使用穷举法和 gurobi 求解器进行求解，可以得到误差较小且硬件复杂度较小的 DFT 矩阵分解逼近方案。

6.2. DFT 矩阵的稀疏整数矩阵分解逼近模型

本问题同时降低复数乘法次数和每个复数乘法器的复杂度，即同时规定分解后矩阵的稀疏性和元素的取值范围，从而达到最大程度上的降低硬件复杂度，所以在设计连乘矩阵时，需要将稀疏性和整数范围这两个要求作为约束条件。为使得满足约束的条件下，矩阵连乘结果与 DFT 矩阵的误差最小，需要建立 DFT 矩阵的稀疏整数矩阵分解逼近模型。

（1）决策变量

矩阵连乘拟合 DFT 矩阵需要设计的变量有：

① 连乘的矩阵个数 K：

$$\mathrm{K}\in\{1,2,3,4,\ldots\} \tag{6.1}$$

② 连乘中第 i 个矩阵$\mathbf{A_i}$的第 k 行，第 n 列的元素的值$[\mathbf{B_i}]_{k,n}$：

$$[\mathbf{B_i}]_{k,n} \in \{x+jy|x,y \in \mathcal{P}\}, \mathcal{P}=\{0,\pm 1,\pm 2,...,\pm 2^{q-1}\}, i=1,2,...,K; k,n=0,1,...,N-1 \tag{6.2}$$

本问题中 q=3，所以$\mathcal{P}=\{0,\pm 1,\pm 2,\pm 4\}$。

③ 连乘中第 i 个矩阵$\mathbf{A_i}$的第 k 行，第 n 列的元素的值是否为 0，$X_{i,k,n}=0$ 表示$[\mathbf{A_i}]_{k,n}=0$，$X_{i,k,n}=1$ 表示$[\mathbf{A_i}]_{k,n}\neq 0$：

$$X_{i,k,n} \in \{0,1\}, i=1,2,...,K; k,n=0,1,...,N-1 \tag{6.3}$$

④ 实值矩阵缩放因子β：

$$\beta \in \mathcal{R} \tag{6.4}$$

其中$\mathcal{R}$表示实数域。

（2）目标函数

本问题使用 K 个稀疏矩阵连乘拟合近似 DFT 矩阵，所以本问题的目标仍为 K 个矩阵连乘尽可能和原 DFT 矩阵误差最小。使用 Frobenius 范数意义下矩阵$\mathbf{F}_N$和$\beta\mathbf{A}_1\mathbf{A}_2\cdots\mathbf{A}_K$的误差作为矩阵连乘拟合 DFT 矩阵的误差，即：

$$\min \text{RMSE}=\frac{1}{N}\sqrt{\|\mathbf{F}_N-\beta\mathbf{A}_1\mathbf{A}_2\cdots\mathbf{A}_K\|_F^2} \tag{6.5}$$

其中$[\mathbf{A_i}]_{k,n}=X_{i,k,n}[\mathbf{B_i}]_{k,n}, i=1,2,...,K; k,n=0,1,...,N-1$。

（3）约束条件

为减少计算 DFT 矩阵的硬件复杂度并使矩阵连乘可以正常拟合 DFT 矩阵，作出如下约束：

① 矩阵$\mathbf{A_i}$为 N 行 N 列的矩阵；

② 矩阵$\mathbf{A_i}$的第 k 行至多有两个非 0 元素：

$$\sum_{n=0}^{N-1} X_{i,k,n} \leq 2, i=1,2,...,K; k=0,1,...,N-1 \tag{6.6}$$

③ 连乘中第 i 个矩阵$\mathbf{A_i}$的第 k 行，第 n 列的元素的值$[\mathbf{B_i}]_{k,n}$：

$$[\mathbf{B_i}]_{k,n} \in \{x+jy|x,y \in \mathcal{P}\}, \mathcal{P}=\{0,\pm 1,\pm 2,\pm 4\}, i=1,2,...,K; k,n=0,1,...,N-1 \tag{6.7}$$

综上所述，问题三 DFT 矩阵的稀疏整数矩阵分解逼近模型总结如下：

$$\min \text{RMSE}=\frac{1}{N}\sqrt{\|\mathbf{F}_N-\beta\mathbf{A}_1\mathbf{A}_2\cdots\mathbf{A}_K\|_F^2}$$

$$\text{s.t.}\begin{cases} \text{K} \in \{1,2,3,4,\dots\} \\ [\mathbf{A_i}]_{k,n}=X_{i,k,n}[\mathbf{B_i}]_{k,n}, i=1,2,...,K; k,n=0,1,...,N-1 \\ \sum_{n=0}^{N-1} X_{i,k,n} \leq 2, i=1,2,...,K; k=0,1,...,N-1 \\ [\mathbf{B_i}]_{k,n} \in \{x+jy|x,y \in \mathcal{P}\}, i=1,2,...,K; k,n=0,1,...,N-1 \\ \mathcal{P}=\{0,\pm 1,\pm 2,\pm 4\} \\ X_{i,k,n} \in \{0,1\}, i=1,2,...,K; k,n=0,1,...,N-1 \end{cases} \tag{6.8}$$

6.3. 模型求解——先分解再优化

6.3.1. 精确分解 N 维 DFT 矩阵

先使用问题一基于分治思想的分解方法将需要求解的 N 维 DFT 矩阵进行精确分解，可以得到一组元素任意的稀疏矩阵连乘结果：

$$\boldsymbol{F}_N = (\prod_{i=1}^{\log_2 \mathrm{N}-1} \boldsymbol{A_i})(\mathbf{I}_{\mathrm{N}/2} \otimes \boldsymbol{F}_2)\mathbf{P}_{\mathrm{N}} \tag{6.9}$$

分析连乘的矩阵可以发现，值非范围的元素在$\mathrm{A}_1, \mathrm{A}_2 \ldots, \mathrm{A}_{\log_2 \mathrm{N}-2}$的矩阵中，对这些矩阵中的值非范围的元素进行统计。使用一个三元组集合 **V** 表示分解后的矩阵中值非范围元素的矩阵编号、行号、列号构成的三元组的集合。

6.3.2. 整数矩阵连乘逼近 N 维 DFT 矩阵

将这些元素作为新的决策变量，将这些元素的值限定到范围内作为约束条件，建立优化这些元素的值和矩阵缩放因子以实现拟合误差最小的多元混合整数规划模型。

（1）决策变量

矩阵连乘拟合 DFT 矩阵需要设计的变量有：

① 分解后的矩阵中值非范围元素的值$[\mathbf{A_i}]_{\mathrm{k,n}}$：

$$[\mathbf{A_i}]_{\mathrm{k,n}} \in \{x + jy | x, y \in \mathcal{P}\}, \mathcal{P} = \{0, \pm 1, \pm 2, \ldots, \pm 2^{q-1}\}, (i,k,n) \in \mathrm{V} \tag{6.10}$$

$(i,k,n) \in \mathrm{V}$ 表示，三元组(i,k,n)为 V 中的一个元素。本问题中 q=3，所以$\mathcal{P} = \{0, \pm 1, \pm 2, \pm 4\}$。

② 实值矩阵缩放因子β：

$$\beta \in \mathcal{R} \tag{6.11}$$

其中$\mathcal{R}$表示实数域。

（2）目标函数

本问题使用 K 个稀疏矩阵连乘拟合近似 DFT 矩阵，所以本问题的目标仍为 K 个矩阵连乘尽可能和原 DFT 矩阵误差最小。使用 Frobenius 范数意义下矩阵$\mathbf{F}_N$和$\beta\mathbf{A}_1\mathbf{A}_2\cdots\mathbf{A}_K$的误差作为矩阵连乘拟合 DFT 矩阵的误差，即：

$$\min \mathrm{RMSE} = \frac{1}{N}\sqrt{\left\|\mathbf{F}_N - \beta(\prod_{i=1}^{\log_2 \mathrm{N}-1} \boldsymbol{A_i})(\mathbf{I}_{\mathrm{N}/2} \otimes \boldsymbol{F}_2)\mathbf{P}_{\mathrm{N}}\right\|_F^2} \tag{6.12}$$

（3）约束条件

为减少计算 DFT 矩阵的硬件复杂度并使矩阵连乘可以正常拟合 DFT 矩阵，作出如下约束：

① 矩阵$\mathbf{A_i}$为 N 行 N 列的矩阵；

② 分解后的矩阵中值非范围元素的值$[\mathbf{A_i}]_{\mathrm{k,n}}$：

$$[\mathbf{A_i}]_{\mathrm{k,n}} \in \{x + jy | x, y \in \mathcal{P}\}, \mathcal{P} = \{0, \pm 1, \pm 2, \ldots, \pm 2^{q-1}\}, (i,k,n) \in \mathbf{V} \tag{6.13}$$

V 表示分解后的矩阵中值非范围元素的矩阵编号、行号、列号构成的三元组的集合。

综上所述，问题三经过分解后的 DFT 矩阵的稀疏整数矩阵分解逼近模型总结如下：

$$\min \mathrm{RMSE} = \frac{1}{N}\sqrt{\left\|\mathbf{F}_N - \beta\left(\prod_{i=1}^{\log_2 \mathrm{N}-1} \boldsymbol{A_i}\right)(\mathbf{I}_{\mathrm{N}/2} \otimes \boldsymbol{F}_2)\mathbf{P}_{\mathrm{N}}\right\|_F^2} \tag{6.14}$$

$$\text{s.t.}\begin{cases}[\mathbf{A_i}]_{\mathrm{k,n}} \in \{x + jy | x, y \in \mathcal{P}\}, (i,k,n) \in \mathbf{V} \\ \mathcal{P} = \{0, \pm 1, \pm 2, \ldots, \pm 2^{q-1}\}\end{cases}$$

6.3.3. 模型求解——遗传算法

本模型求解的流程为：

1. 决策变量：
   $\beta, \mathrm{c}_1, \mathrm{c}_2, \ldots, \mathrm{c}_{\mathrm{N}/4-1}, \mathrm{d}_1, \mathrm{d}_2, \ldots, \mathrm{d}_{\mathrm{N}/4-1}, \mathrm{e}_1, \mathrm{e}_2, \ldots, \mathrm{e}_{\mathrm{N}/4-1}, \mathrm{f}_1, \mathrm{f}_2, \ldots, \mathrm{f}_{\mathrm{N}/4-1}$
2. 初始化种群：随机生成一些满足的初始解
3. 计算适应度：计算误差 RMSE，如果达到最大进化次数，转第 8 步
4. 选择：通过轮盘赌算法，将 RMSE 较低的选择出来，将其遗传给下一代
5. 交叉：将从当前代中选择的双亲样本的部分染色体交叉互换，得到下一代的两个新个体
6. 变异：将从当前代中选择的双亲样本的部分染色体进行突变，得到随机的下一代个体
7. 将得到的下一代个体作为输入，转第 3 步
8. 输出最小误差 RMSE 以及决策变量，其中$w_i = a_i + \mathrm{j}b_i, a_i = c_i * 2^d, b_i = e_i * 2^f$

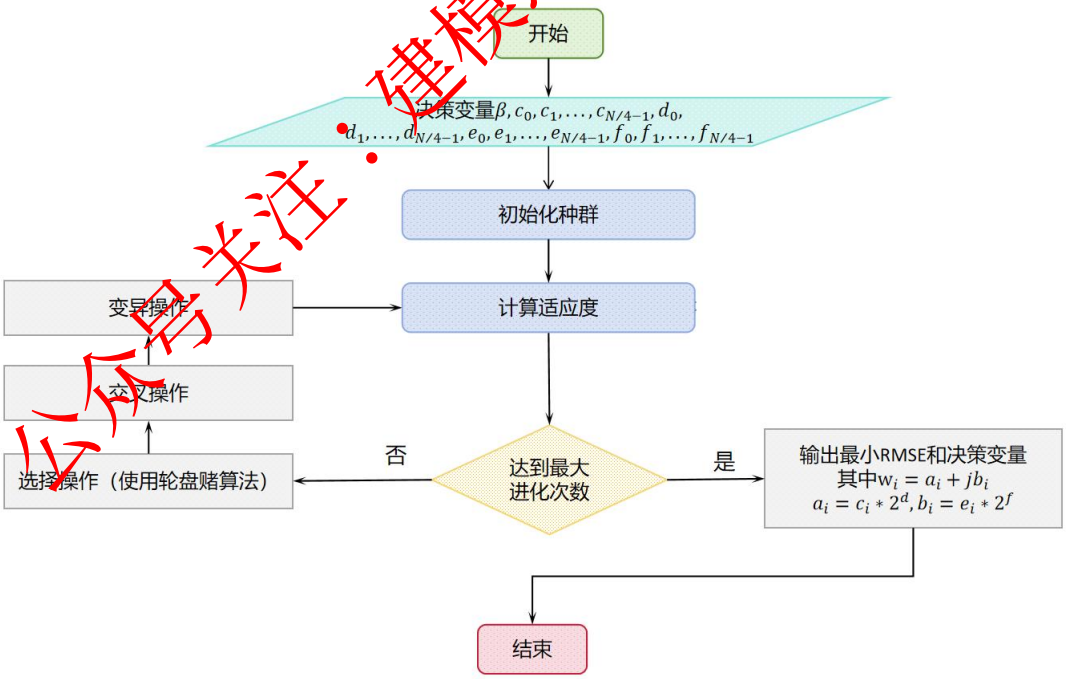

图 6-2　遗传算法流程图

6.4. 模型求解结果

（1）t =1 时，$N = 2^t = 2$，2 维 DFT 矩阵$\boldsymbol{F}_2$不需要分解，所以矩阵数量为 K=1，矩阵为：

$$\boldsymbol{F_2} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{6.15}$$

（2）t =2 时，$N = 2^t = 4$，4 维 DFT 矩阵$\boldsymbol{F_4}$经过分解算法得到的矩阵数量为 K=3，矩阵为：

$$\boldsymbol{A_1} = \begin{bmatrix} 1 & & 1 & \\ & 1 & & -j \\ 1 & & -1 & \\ & 1 & & j \end{bmatrix} \tag{6.16}$$

$$\mathbf{I_2} \otimes \boldsymbol{F}_2 = \begin{bmatrix} 1 & 1 & & \\ 1 & -1 & & \\ & & 1 & 1 \\ & & 1 & -1 \end{bmatrix} \tag{6.17}$$

$$\boldsymbol{P_4} = [e_0 \quad e_2 \quad e_1 \quad e_3] \tag{6.18}$$

$\boldsymbol{F_4}$使用的矩阵连乘形式为：

$$\boldsymbol{F_4} = \boldsymbol{A_1}(\mathbf{I_2} \otimes \boldsymbol{F}_2)\mathbf{P_4} \tag{6.19}$$

（3）t =3 时，$N = 2^t = 8$，8 维 DFT 矩阵$\boldsymbol{F_8}$经过分解算法得到的矩阵数量为 K=4，矩阵为：

$$\boldsymbol{A_1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1-j & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & -j & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 \\ 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & -1+j & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & j & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.20}$$

$$\boldsymbol{A_2} = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -j & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & j & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & -j \\ 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & j \end{bmatrix} \tag{6.21}$$

$$\boldsymbol{A_3} = \boldsymbol{I_4} \otimes \mathbf{F_2} \tag{6.22}$$

$$\boldsymbol{P_8} = [e_0 \quad e_4 \quad e_2 \quad e_6 \quad e_1 \quad e_5 \quad e_3 \quad e_7] \tag{6.23}$$

$\boldsymbol{F_8}$使用的矩阵连乘形式为：

$$\boldsymbol{F_8} = \boldsymbol{A_1}\boldsymbol{A_2}(\mathbf{I_4} \otimes \boldsymbol{F}_2)\mathbf{P_8} \tag{6.24}$$

（4）t =4 时，$N = 2^t = 16$，16 维 DFT 矩阵$\boldsymbol{F_{16}}$经过分解算法得到的矩阵数量为 K=5，矩阵为：

$$\boldsymbol{A_1} = \begin{bmatrix} \boldsymbol{I_8} & \boldsymbol{A_{1_{0,1}}} \\ \boldsymbol{I_8} & \boldsymbol{A_{1_{1,1}}} \end{bmatrix}$$

$$\boldsymbol{A_{1_{0,1}}} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1-j & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -j & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -j & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -j & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1-j & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{bmatrix}$$

$$\boldsymbol{A_{1_{1,1}}} = \begin{bmatrix} -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1+j & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & j & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & j & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & j & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1+j & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \tag{6.25}$$

$$\boldsymbol{A_2} = \begin{bmatrix} \boldsymbol{A_{2_{0,0}}} \\ \boldsymbol{A_{2_{1,1}}} \end{bmatrix}$$

$$\boldsymbol{A_{2_{0,0}}} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1-j & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & -j & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1-j \\ 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & -1+j & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & j & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1+j \end{bmatrix}$$

$$\boldsymbol{A_{2_{1,1}}} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1-j & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & -j & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1-j \\ 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & -1+j & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & j & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1+j \end{bmatrix} \tag{6.26}$$

$$A_3 = \begin{bmatrix} A_{3_{0,0}} & \\ & A_{3_{1,1}} \end{bmatrix}$$

$$A_{3_{0,0}} = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -j & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & j & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & -j \\ 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & j \end{bmatrix}$$

$$A_{3_{1,1}} = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -j & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & j & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & -j \\ 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & j \end{bmatrix} \quad (6.27)$$

$$A_4 = I_8 \otimes F_2$$

$$P_{16} = [e_0 \quad e_8 \quad e_4 \quad e_{12} \quad e_2 \quad e_{10} \quad e_6 \quad e_{14} \quad e_1 \quad e_9 \quad e_5 \quad e_{13} \quad e_3 \quad e_{11} \quad e_7 \quad e_{15}]$$

F_{16}使用的矩阵连乘形式为：

$$F_{16} = A_1 A_2 A_3 (I_8 \otimes F_2) P_{16} \quad (6.28)$$

（**5**）t =5 时，$N=2^t=32$，32 维 DFT 矩阵$\boldsymbol{F_{32}}$经过分解算法得到的矩阵数量为 K=6，矩阵为：

$$\boldsymbol{A_1}=\begin{bmatrix}\boldsymbol{I_8} & & \boldsymbol{A_{1_{0,2}}} & \\ & \boldsymbol{I_8} & & \boldsymbol{A_{1_{1,3}}} \\ \boldsymbol{I_8} & & \boldsymbol{A_{1_{2,2}}} & \\ & \boldsymbol{I_8} & & \boldsymbol{A_{1_{3,3}}}\end{bmatrix}$$

$$\boldsymbol{A_{1_{0,2}}}=\begin{bmatrix}1 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 1-j & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & -j & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 0 & -j & 0\\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -j\end{bmatrix} \tag{6.29}$$

$$\boldsymbol{A_{1_{1,3}}}=\begin{bmatrix}-j & 0 & 0 & 0 & 0 & 0 & 0 & 0\\ 0 & -j & 0 & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & -j & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & -j & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & -1-j & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0\\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1\end{bmatrix}$$

$$\boldsymbol{A_{1_{2,2}}}=-\boldsymbol{A_{1_{0,2}}}$$

$$\boldsymbol{A_{1_{3,3}}}=-\boldsymbol{A_{1_{1,3}}}$$

$$\boldsymbol{A_2}=\begin{bmatrix}\boldsymbol{I_8} & \boldsymbol{A_{2_{0,1}}} & & \\ \boldsymbol{I_8} & \boldsymbol{A_{2_{1,1}}} & & \\ & & \boldsymbol{I_8} & \boldsymbol{A_{2_{2,3}}} \\ & & \boldsymbol{I_8} & \boldsymbol{A_{2_{3,3}}}\end{bmatrix}$$

$$\boldsymbol{A_{2_{0,1}}}=\begin{bmatrix}1 & 0 & 0 & 0 & 0 & 0 & 0 & 0\\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 1-j & 0 & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & -j & 0 & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & -j & 0 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & -j & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 0 & -1-j & 0\\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1\end{bmatrix} \tag{6.30}$$

$$\boldsymbol{A_{2_{1,1}}}=\boldsymbol{A_{2_{3,3}}}=-\boldsymbol{A_{2_{0,1}}}$$

$$\boldsymbol{A_{2_{2,3}}}=\boldsymbol{A_{2_{0,1}}}$$

$$\boldsymbol{A_3} = \boldsymbol{I_4}\otimes\boldsymbol{A_{3_{0,0}}}$$

$$\boldsymbol{A_{3_{0,0}}} = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1-j & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & -j & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1-j \\ 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & -1+j & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & j & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1+j \end{bmatrix} \tag{6.31}$$

$$\boldsymbol{A_4} = \boldsymbol{I_8}\otimes\boldsymbol{A_{4_{0,0}}} \tag{6.32}$$

$$\boldsymbol{A_{4_{0,0}}} = \begin{bmatrix} 1 & & 1 & \\ & 1 & & -j \\ 1 & & -1 & \\ & 1 & & j \end{bmatrix}$$

$$\boldsymbol{A_5} = \boldsymbol{I_{16}}\otimes\mathbf{F_2} \tag{6.33}$$

$$\boldsymbol{P_{32}} = [e_0 \quad e_{16} \quad e_8 \quad e_{24} \quad e_4 \quad e_{20} \quad e_{12} \quad e_{28} \quad e_2 \quad e_{18} \quad e_{10} \quad e_{26} \quad e_6 \quad e_{22} \quad e_{14} \quad e_{30}$$
$$e_1 \quad e_{17} \quad e_9 \quad e_{25} \quad e_5 \quad e_{21} \quad e_{13} \quad e_{29} \quad e_3 \quad e_{19} \quad e_{11} \quad e_{27} \quad e_7 \quad e_{23} \quad e_{15} \quad e_{31}]$$

$\boldsymbol{F_{32}}$使用的矩阵连乘形式为：

$$\boldsymbol{F_{32}} = \boldsymbol{A_1}\boldsymbol{A_2}\boldsymbol{A_3}\boldsymbol{A_4}(\mathbf{I_{16}} \otimes \boldsymbol{F}_2)\mathbf{P_{32}} \tag{6.34}$$

（6）当$t = 6,7,\ldots$时，$N = 2^t = 64,128,\ldots$，N维 DFT 矩阵$\boldsymbol{F}_N$经过分解算法得到的矩阵数量为 K=t+1，$\boldsymbol{F}_N$使用的矩阵连乘形式为：

$$\boldsymbol{F_N} = \left(\prod_{i=1}^{\log_2 N-1} \boldsymbol{A_i}\right)(\mathbf{I}_{N/2} \otimes \boldsymbol{F}_2)\mathbf{P}_N \tag{6.35}$$

6.5. 求解结果分析

这样的递归算法对 N 为 DFT 矩阵$\boldsymbol{F_N}$的分解需要递归的层数为 $\log_2 N - 1$，这样的算法可以快速完成分解。对$t = 1,2,3,4,5$ 时，2^t维 DFT 矩阵$\boldsymbol{F}_{2^t}$的分解矩阵个数，误差，硬件复杂度进行了统计，如表 6.1 所示。发现可以得到误差几乎均为 0，和原始 DFT 矩阵计算的硬件复杂度相比分解后的矩阵连乘硬件复杂度有着明显的下降，仍使得矩阵的硬件复杂度不太理想。

其中计算 DFT 的硬件复杂度仅考虑乘法器的硬件复杂度：

$$C = q \times L \tag{6.36}$$

其中 q 为乘法器位宽，L 为复数乘法次数。

表 6-1　N = 2^t维 DFT 矩阵分解结果统计表

t	N	β	$\mathcal{A}$中矩阵个数	RMSE	C
1	2	1	2	1.2265e-16	0
2	4	1	3	7.9362e-16	0
3	8	0.882843	4	0.160424	0
4	16	0.823534	5	0.259739	0
5	32	0.78603	6	0.33444	0

7. 问题 4：2 个 DFT 矩阵张量积矩阵的分解逼近

7.1. 2 个 DFT 矩阵张量积矩阵分解逼近分析

当一非 DFT 矩阵是由两个 DFT 矩阵通过张量积的形式得到时，形如$\mathbf{D} = \mathbf{F}_{N1} \otimes \mathbf{F}_{N2}$，其中$\mathbf{F}_{N1}$和$\mathbf{F}_{N2}$分别是$N_1$和$N_2$维的 DFT 矩阵，使用 DFT 矩阵近似分解的思想同样可以将矩阵$\mathbf{D}$分解为多个矩阵连乘近似的形式，通过限定矩阵的稀疏性和元素范围，可以在一定精度下达到降低计算$\mathbf{F}_N$矩阵硬件复杂度的目的。

由前三问可知一个 DFT 矩阵可以被分解为多个稀疏矩阵相乘的形式，即$\boldsymbol{F_{N1}} = (\prod_{i=1}^{\log_2 N1-1} \boldsymbol{A_i})(\mathbf{I}_{N1/2} \otimes \boldsymbol{F}_2)\mathbf{P}_{N1}$。当$N_1$ = 4 时，$\mathbf{F_4}$可以被分解为$\boldsymbol{A_{4,1}}$、$(\mathbf{I_2} \otimes \boldsymbol{F}_2)$、$\mathbf{P_4}$这三个矩阵；当$N_2$ = 8 时，$\mathbf{F_8}$可以被分解为$\boldsymbol{A_{8,1}}$、$\boldsymbol{A_{8,2}}$、$(\mathbf{I_4} \otimes \boldsymbol{F}_2)$、$\mathbf{P_8}$这三个矩阵。当$\mathbf{D} = \mathbf{F_4} \otimes \mathbf{F_8}$时，$\mathbf{D}$可以被分解为$(\boldsymbol{A_{4,1}}(\mathbf{I_2} \otimes \boldsymbol{F}_2)\mathbf{P_4}) \otimes (\boldsymbol{A_{8,1}}\boldsymbol{A_{8,2}}(\mathbf{I_4} \otimes \boldsymbol{F}_2)\mathbf{P_8})$。当$\boldsymbol{A_{4,1}}(\mathbf{I_2} \otimes \boldsymbol{F}_2)$这个矩阵可以被分解为$\boldsymbol{B_{4,1}}\boldsymbol{B_{4,2}}\boldsymbol{B_{4,3}}$时，D 就被分解为了$(\boldsymbol{B_{4,1}}\boldsymbol{B_{4,2}}\boldsymbol{B_{4,3}}\mathbf{P_4}) \otimes (\boldsymbol{A_{8,1}}\boldsymbol{A_{8,2}}(\mathbf{I_4} \otimes \boldsymbol{F}_2)\mathbf{P_8})$就可以变为$(\boldsymbol{B_{4,1}} \otimes \boldsymbol{A_{8,1}})(\boldsymbol{B_{4,2}} \otimes \boldsymbol{A_{8,2}})(\boldsymbol{B_{4,3}} \otimes (\mathbf{I_4} \otimes \boldsymbol{F}_2))(\mathbf{P_4} \otimes \mathbf{P_8})$，即将 D 分解为了 4 个 32 维的矩阵。

通过限制这 4 个矩阵的稀疏性和元素取值范围可以使得计算 D 矩阵所需的硬件复杂度有效降低，且由于 DFT 矩阵分解误差不高，最后 D 矩阵的分解误差同样可以满足要求。所以将限定分解后的 4 个矩阵每行至多有两个非零元素和每个元素的取值范围为$\{x + jy | x, y \in \mathcal{P}\}, \mathcal{P} = \{0, \pm 1, \pm 2, ..., \pm 2^{q-1}\}$作为两个约束条件，将矩阵分解误差最小设置为目标函数，建立优化矩阵中非零元素和矩阵缩放因子的多元混合整数模型。通过求解这个模型就可以在误差较小的条件下，有效的降低计算 D 矩阵的硬件复杂度。

7.2. 2 个 DFT 矩阵张量积矩阵的分解

（1）混合乘积运算性质：

混合乘积运算拥有一个类似分配率的性质：

$$\mathbf{AC} \otimes \boldsymbol{BD} = (\mathbf{A} \otimes \boldsymbol{B})(\boldsymbol{C} \otimes \boldsymbol{D}) \tag{7.1}$$

将上述 4 个矩阵的混合乘积运算性质扩展到 2N 个矩阵，即为：

$$\prod_{i=1}^{N} \boldsymbol{A_i} \otimes \prod_{i=1}^{N} \boldsymbol{B_i} = \prod_{i=1}^{N} (\boldsymbol{A_i} \otimes \boldsymbol{B_i}) \tag{7.2}$$

其中$\prod_{i=1}^{N} \boldsymbol{A_i}$表示$\boldsymbol{A_1 A_2 ... A_N}$。

（2）矩阵 D 的分解

设一个矩阵 D 由N_1和N_2维的 DFT 矩阵$\mathbf{F}_{N1}$和$\mathbf{F}_{N2}$张量积后构成且$N1<N2$，即：

$$\mathbf{D} = \mathbf{F}_{N1} \otimes \mathbf{F}_{N2} \tag{7.3}$$

根据第一问可知 DFT 矩阵$\boldsymbol{F_N} = (\prod_{i=1}^{\log_2 N-1} \boldsymbol{A_{N,i}})(\mathbf{I}_{N/2} \otimes \boldsymbol{F}_2)\mathbf{P}_N$，将其带入式 7.3 得：

$$\mathbf{D}=(\prod_{i=1}^{\log_2 N1-1}\boldsymbol{A_{N1,i}})(\mathbf{I}_{N1/2}\otimes\boldsymbol{F}_2)\mathbf{P}_{N1}\otimes(\prod_{i=1}^{\log_2 N2-1}\boldsymbol{A_{N2,i}})(\mathbf{I}_{N2/2}\otimes\boldsymbol{F}_2)\mathbf{P}_{N2} \tag{7.4}$$

若可以将$\mathbf{F}_{N1}$分解为 $\log_2 N2+1$ 个矩阵，即：

$$(\prod_{i=1}^{\log_2 N1-1}\boldsymbol{A_{N1,i}})(\mathbf{I}_{N1/2}\otimes\boldsymbol{F}_2)\mathbf{P}_{N1}=\prod_{i=1}^{\log_2 N2+1}\boldsymbol{B_i} \tag{7.5}$$

式 7.4 就可以变为：

$$\mathbf{D}=\prod_{i=1}^{\log_2 N2+1}\boldsymbol{B_i}\otimes((\prod_{i=1}^{\log_2 N2-1}\boldsymbol{A_{N2,i}})(\mathbf{I}_{N2/2}\otimes\boldsymbol{F}_2)\mathbf{P}_{N2}) \tag{7.6}$$

利用混合乘积运算的性质可得：

$$\mathbf{D}=(\prod_{i=1}^{\log_2 N2-1}(\boldsymbol{B_i}\otimes\boldsymbol{A_{N2,i}}))(\boldsymbol{B}_{\log_2 N2}\otimes(\mathbf{I}_{N2/2}\otimes\boldsymbol{F}_2))(\boldsymbol{B}_{\log_2 N2+1}\otimes\mathbf{P}_{N2}) \tag{7.7}$$

这样就得到了**D**矩阵的矩阵连乘形式：

$$\mathbf{D}=\prod_{i=1}^{\log_2 N2+1}\boldsymbol{D_i} \tag{7.8}$$

$$\boldsymbol{D_i}=\begin{cases}(\boldsymbol{B_i}\otimes\boldsymbol{A_{N2,i}}) & i=1,\ldots,\log_2 N2-1\\ (\boldsymbol{B}_{\log_2 N2}\otimes(\mathbf{I}_{N2/2}\otimes\boldsymbol{F}_2)) & i=\log_2 N2\\ (\boldsymbol{B}_{\log_2 N2+1}\otimes\mathbf{P}_{N2}) & i=\log_2 N2+1\end{cases} \tag{7.9}$$

（3）当$N_1=4,\ N_2=8$时矩阵 D 的分解

当$N_1=4,\ N_2=8$时，矩阵$\mathbf{D}=\mathbf{F_4}\otimes\mathbf{F_8}$为 32 维矩阵，$\mathbf{F_4}$和$\mathbf{F_8}$可以精确分解为：

$$\begin{aligned}\mathbf{F_4}&=\boldsymbol{A_{4,1}}(\mathbf{I_2}\otimes\boldsymbol{F}_2)\mathbf{P_4}\\ \mathbf{F_8}&=\boldsymbol{A_{8,1}}\boldsymbol{A_{8,2}}(\mathbf{I_4}\otimes\boldsymbol{F}_2)\mathbf{P_8}\end{aligned} \tag{7.10}$$

由于排列矩阵有着优秀的稀疏性和元素均为 1 的特点，在将$\mathbf{F_4}$矩阵进一步分解时保留。若将$\boldsymbol{A_{4,1}}(\mathbf{I_2}\otimes\boldsymbol{F}_2)$分解为 3 个矩阵$\boldsymbol{B_1}$、$\boldsymbol{B_2}$、$\boldsymbol{B_3}$就可以使用混合乘积性质将$\mathbf{D}=\mathbf{F_4}\otimes\mathbf{F_8}$分解为：

$$\mathbf{D}=(\boldsymbol{B_1}\otimes\boldsymbol{A_{8,1}})(\boldsymbol{B_2}\otimes\boldsymbol{A_{8,2}})(\boldsymbol{B_3}\otimes(\mathbf{I_4}\otimes\boldsymbol{F}_2))(\mathbf{P_4}\otimes\mathbf{P_8}) \tag{7.11}$$

就可以将 D 矩阵的矩阵分解形式表示出来。

7.3. D 矩阵的稀疏整数矩阵连乘逼近模型

根据问题三的思想，为了减少计算 **D** 矩阵所需的硬件复杂度，需要将 **D** 矩阵使用矩阵连乘的形式进行拟合，当规定连乘中的矩阵为稀疏矩阵且元素为限定范围内的整数时，既降低了复数乘法次数又降低了复数乘法器的复杂度，使得计算 D 矩阵所需的硬件复杂度大幅降低。为设计当$N_1=4,\ N_2=8$ 时矩阵 **D** 的矩阵连乘形式，需要所以根据公式 7.11 将 D 矩阵分解为 4 个矩阵连乘的形式。

7.3.1 限定分解后矩阵为稀疏矩阵

根据问题三的求解思路，首先需要根据分解形式规定矩阵的稀疏性，将分解后的矩阵每行至多两个非零元素作为约束，限定分解后的矩阵满足该稀疏性就要求$(\boldsymbol{B_1}\otimes\boldsymbol{A_{8,1}})$、$(\boldsymbol{B_2}\otimes\boldsymbol{A_{8,2}})$、$(\boldsymbol{B_3}\otimes(\mathbf{I_4}\otimes\boldsymbol{F}_2))$、$(\mathbf{P_4}\otimes\mathbf{P_8})$均为每行至多两个非零元素的稀疏矩阵，通过

问题一的分解结果可知，$\boldsymbol{A}_{8,1}$、$\boldsymbol{A}_{8,2}$、$(\mathbf{I}_4 \otimes \boldsymbol{F}_2)$均为每行两个非零元素的稀疏矩阵。根据张量积的定义可知，只有当$\boldsymbol{B}_1$、$\boldsymbol{B}_2$、$\boldsymbol{B}_3$为每行仅有一个非零元素的稀疏矩阵时，才会使得 D 矩阵分解后的矩阵为每行至多两个非零元素的稀疏矩阵。

7.3.2 D 矩阵的整数分解逼近模型

为满足约束 1 的稀疏性要求，已将矩阵分解个数，非零元素位置做出的限定。为满足约束 2 使得分解后的矩阵 $\boldsymbol{D}_i[l,m] \in \{x+jy|x,y \in \mathcal{P}\}, \mathcal{P}=\{0,\pm 1,\pm 2,\pm 4\}, \mathrm{i}=1,2,\ldots,4$；$l,m=0,1,\ldots,31$，需要将已知的矩阵分解个数、非零元素位置限定、非零元素值范围作为约束条件，将矩阵连乘拟合结果误差最小作为目标，建立优化非零元素值和矩阵缩放因子的多元混合整数模型。

（1）决策变量

矩阵连乘拟合 **D** 矩阵需要设计的变量有：

① 公式 7.11 中矩阵$\boldsymbol{B}_i$的第 k 行，第 n 列的元素的值$[\mathbf{B_i}]_{\mathrm{k,n}}$：

$$[\mathbf{B_i}]_{\mathrm{k,n}} \in \{x+jy|x,y \in \mathcal{P}\}, \mathcal{P}=\{0,\pm 1,\pm 2,\pm 4\}, i=1,2,3; k,n=0,1,2,3 \tag{7.13}$$

② 分别统计 7.11 中矩阵$\boldsymbol{A}_{8,1}$,$\boldsymbol{A}_{8,2}$中的值非范围元素，将其行号、列号作为二元组分别保存到二元组集合$\boldsymbol{V}_1$,$\boldsymbol{V}_2$，值非范围元素的值$[\boldsymbol{A}_{8,1}]_{\mathrm{k,n}}$, $[\boldsymbol{A}_{8,2}]_{\mathrm{k,n}}$

$$\begin{aligned}[\boldsymbol{A}_{8,1}]_{\mathrm{k,n}} &\in \{x+jy|x,y \in \mathcal{P}\}, \mathcal{P}=\{0,\pm 1,\pm 2,\pm 4\}, (k,n) \in \boldsymbol{V}_1\\ [\boldsymbol{A}_{8,2}]_{\mathrm{k,n}} &\in \{x+jy|x,y \in \mathcal{P}\}, \mathcal{P}=\{0,\pm 1,\pm 2,\pm 4\}, (k,n) \in \boldsymbol{V}_2\end{aligned} \tag{7.14}$$

③ 实值矩阵缩放因子β：

$$\beta \in \mathcal{R} \tag{7.15}$$

其中$\mathcal{R}$表示实数域。

（2）目标函数

本问题使用公式 7.11 所示的 4 个稀疏矩阵连乘拟合近似 **D** 矩阵，所以本问题的目标为 4 个矩阵连乘尽可能和原 **D** 矩阵误差最小。使用 Frobenius 范数意义下矩阵**D**和$\beta\mathbf{A}_1\mathbf{A}_2\cdots\mathbf{A}_K$的误差作为矩阵连乘拟合 DFT 矩阵的误差，即：

$$\min \mathrm{RMSE}=\frac{1}{N}\sqrt{\left\|\boldsymbol{D}-\beta(\boldsymbol{B}_1\otimes\boldsymbol{A}_{8,1})(\boldsymbol{B}_2\otimes\boldsymbol{A}_{8,2})(\boldsymbol{B}_3\otimes(\mathbf{I}_4\otimes\boldsymbol{F}_2))(\mathbf{P}_4\otimes\mathbf{P}_8)\right\|_F^2} \tag{7.16}$$

（3）约束条件

为减少计算 DFT 矩阵的硬件复杂度并使矩阵连乘可以正常拟合 **D** 矩阵，作出如下约束：

① 矩阵$\mathbf{B_i}$中每行只有一个元素为非零元素：

$$\sum_{n=0}^{3} f([\mathbf{B_i}]_{\mathrm{k,n}})=1, i=1,2,3; k=0,1,2,3 \tag{7.17}$$

$$f(\mathrm{x})=\begin{cases}0, \mathrm{x}=0\\ 1, \mathrm{x}\neq 0\end{cases}$$

② 公式 7.11 中矩阵$\boldsymbol{B}_i$的第 k 行，第 n 列的元素的值$[\mathbf{B_i}]_{\mathrm{k,n}}$：

$$[\mathbf{B_i}]_{\mathrm{k,n}} \in \{x+jy|x,y \in \mathcal{P}\}, \mathcal{P}=\{0,\pm 1,\pm 2,\pm 4\}, i=1,2,3; k,n=0,1,2,3 \tag{7.18}$$

③ 分别统计 7.11 中矩阵$\boldsymbol{A}_{8,1}$,$\boldsymbol{A}_{8,2}$中的值非范围元素，将其行号、列号作为二元组分别保

存到二元组集合$\boldsymbol{V_1}$,$\boldsymbol{V_2}$，值非范围元素的值$[\boldsymbol{A_{8,1}}]_{\mathrm{k,n}}$, $[\boldsymbol{A_{8,2}}]_{\mathrm{k,n}}$：

$$\begin{aligned}&[\boldsymbol{A_{8,1}}]_{\mathrm{k,n}} \in \{x+jy|x,y\in\mathcal{P}\},\mathcal{P}=\{0,\pm1,\pm2,\pm4\},(k,n)\in\boldsymbol{V_1}\\&[\boldsymbol{A_{8,2}}]_{\mathrm{k,n}} \in \{x+jy|x,y\in\mathcal{P}\},\mathcal{P}=\{0,\pm1,\pm2,\pm4\},(k,n)\in\boldsymbol{V_2}\end{aligned} \tag{7.19}$$

综上所述，问题四 **D** 矩阵的稀疏整数矩阵分解逼近模型总结如下：

$$\min \mathrm{RMSE}=\frac{1}{N}\sqrt{\left\|\mathbf{D}-\beta(\boldsymbol{B_1}\otimes\boldsymbol{A_{8,1}})(\boldsymbol{B_2}\otimes\boldsymbol{A_{8,2}})(\boldsymbol{B_3}\otimes(\mathbf{I_4}\otimes\boldsymbol{F}_2))(\mathbf{P_4}\otimes\mathbf{P_8})\right\|_F^2}$$

$$\text{s.t.}\begin{cases}\sum_{n=0}^{3} f([\mathbf{B_i}]_{\mathrm{k,n}})=1, i=1,2,3; k=0,1,2,3\\ [\mathbf{B_i}]_{\mathrm{k,n}}\in\{x+jy|x,y\in\mathcal{P}\}, i=1,2,3; k,n=0,1,2,3\\ [\boldsymbol{A_{8,1}}]_{\mathrm{k,n}}\in\{x+jy|x,y\in\mathcal{P}\},(k,n)\in\boldsymbol{V_1}\\ [\boldsymbol{A_{8,2}}]_{\mathrm{k,n}}\in\{x+jy|x,y\in\mathcal{P}\},(k,n)\in\boldsymbol{V_2}\\ \mathcal{P}=\{0,\pm1,\pm2,\pm4\}\end{cases} \tag{7.20}$$

7.4. 模型求解

根据模型建立的过程，首先需要将$\boldsymbol{F_8}$精确分解得到$\boldsymbol{A_{8,1}}$、$\boldsymbol{A_{8,2}}$、$(\mathbf{I_4}\otimes\boldsymbol{F}_2)$、$\mathbf{P_8}$结果如问题一所示，将$\boldsymbol{A_{8,1}}$、$\boldsymbol{A_{8,2}}$中值非范围的元素行号，列号作为二元组分别存入二元组集合$\boldsymbol{V_1}$,$\boldsymbol{V_2}$。然后使用穷举法求解 **D** 矩阵的稀疏整数矩阵分解逼近这一多元混合整数规划问题。Gurobi 采用最新的优化技术，充分利用多核处理器的优势，经过 Gurobi 的处理，得到的结果确定而非随机，对于分析求解数学规划优化问题有着巨大优势。将 **D** 矩阵的稀疏整数矩阵分解逼近模型使用 Gurobi 对该问题进行求解，在 C++环境下，得到了和穷举法相同的结果。

7.5. 模型求解结果

当$\boldsymbol{N_1}=\mathbf{4}$, $\boldsymbol{N_2}=\mathbf{8}$时，矩阵$\mathbf{D}=\mathbf{F_4}\otimes\mathbf{F_8}$为 32 维矩阵，根据上述模型求解得到的近似分解结果为：

$$\mathbf{D}=\beta(\boldsymbol{B_1}\otimes\boldsymbol{A_{8,1}})(\boldsymbol{B_2}\otimes\boldsymbol{A_{8,2}})(\boldsymbol{B_3}\otimes(\mathbf{I_4}\otimes\boldsymbol{F}_2))(\mathbf{P_4}\otimes\mathbf{P_8}) \tag{7.21}$$

$$\boldsymbol{D_1}=\begin{bmatrix}\boldsymbol{D_{1_{0,0}}} & & & \\ & \boldsymbol{D_{1_{1,1}}} & & \\ & & \boldsymbol{D_{1_{2,2}}} & \\ & & & \boldsymbol{D_{1_{3,3}}}\end{bmatrix}$$

$$\boldsymbol{D_{1_{0,0}}}=\begin{bmatrix}1&0&0&0&1&0&0&0\\0&1&0&0&0&0&0&0\\0&0&1&0&0&0&0&0\\0&0&0&1&0&0&0&0\\1&0&0&0&-1&0&0&0\\0&1&0&0&0&0&0&0\\0&0&1&0&0&0&0&0\\0&0&0&1&0&0&0&0\end{bmatrix}$$

$$D_{1_{1,1}} = \begin{bmatrix} -j & 0 & 0 & 0 & -j & 0 & 0 & 0 \\ 0 & -j & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -j & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -j & 0 & 0 & 0 & 0 \\ -j & 0 & 0 & 0 & j & 0 & 0 & 0 \\ 0 & -j & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -j & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -j & 0 & 0 & 0 & 0 \end{bmatrix}$$

$$D_{1_{2,2}} = D_{1_{3,3}} = D_{1_{0,0}}$$

$$D_2 = \begin{bmatrix} D_{2_{0,0}} & & & \\ & D_{2_{1,2}} & & \\ & & D_{2_{2,1}} & \\ & & & D_{2_{3,3}} \end{bmatrix}$$

$$D_{2_{0,0}} = \begin{bmatrix} -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \end{bmatrix}$$

$$D_{2_{1,2}} = D_{2_{2,1}} = -D_{2_{0,0}}$$

$$D_{2_{3,3}} = D_{2_{0,0}}$$

$$D_3 = \begin{bmatrix} & D_{3_{0,1}} & & \\ & & & D_{3_{1,3}} \\ & & D_{3_{2,2}} & \\ D_{3_{3,0}} & & & \end{bmatrix}$$

$$D_{3_{0,1}} = \begin{bmatrix} -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 \end{bmatrix}$$

$$D_{3_{1,3}} = D_{3_{0,1}}$$

$$D_{3_{2,2}} = -D_{3_{0,1}}$$

$$D_{3_{3,0}} = \begin{bmatrix} j & j & 0 & 0 & 0 & 0 & 0 & 0 \\ j & -j & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & j & j & 0 & 0 & 0 & 0 \\ 0 & 0 & j & -j & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & j & j & 0 & 0 \\ 0 & 0 & 0 & 0 & j & -j & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & j & j \\ 0 & 0 & 0 & 0 & 0 & 0 & j & -j \end{bmatrix}$$

$$D_4 = \begin{bmatrix} D_{4_{0,0}} & & & \\ & D_{4_{1,2}} & & \\ & & D_{4_{2,1}} & \\ & & & D_{4_{3,3}} \end{bmatrix}$$

$$D_{4_{0,0}} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$D_{4_{1,2}} = D_{4_{2,1}} = D_{4_{3,3}} = D_{4_{0,0}}$$

7.6. 求解结果分析

这样的模型求解可以在合理的时间内这完成近似分解。当$N_1 = 4,\ N_2 = 8$时，矩阵$\mathbf{D} = \mathbf{F_4} \otimes \mathbf{F_8}$的分解矩阵个数为4，误差为，硬件复杂度为。发现可以得到了在误差较小的情况下和原始 DFT 矩阵计算的硬件复杂度相比分解后的矩阵连乘硬件复杂度有着明显的下降。但由于元素整数范围不足使得近似矩阵的硬件误差不太理想。

8. 问题 5：DFT 矩阵的高精度整数分解逼近

8.1. DFT 矩阵的高精度整数分解逼近分析

为了使得设计出的矩阵连乘方案可以运用到实际问题当中，需要对矩阵连乘得到的近似矩阵和原 DFT 矩阵之间的误差进行约束，本问题中约束误差 RMSE ≤ 0.1。为了在降低矩阵连乘所需硬件复杂度的同时满足精度要求，通常需要设计合适的单个乘法器位宽。为同时满足稀疏性要求，使用第三问同样的建模思路，对第一问精确分解后的矩阵进行矩阵元素整数化，求出满足误差要求的最小矩阵元素范围。由于使用分治策略的分解结果矩阵满足稀疏性要求，仅需要将满足误差要求和矩阵元素范围要求作为约束条件，以最小化硬件复杂度为目标，建立出优化矩阵元素和矩阵缩放因子的多元混合整数优化模型。

有规律的遍历 q 的值，使得矩阵元素范围为$\mathbf{A}_k[l,m] \in \{x + jy | x, y \in \mathcal{P}\}, \mathcal{P} = \{0, \pm 1, \pm$

$2, \ldots, \pm 2^{q-1}\}$，使用这一元素范围作为约束修改问题 3 的模型，求解其最小误差，就可得到该 q 值是否能使得误差满足要求。由于 q 值越大，矩阵元素的范围就越大，矩阵误差就会越小，所以可以先将找到一个能使误差满足要求的较大 q 值，再采用**折半查找**的思路就可以寻找到满足误差要求的最小 q 值。

另一种求解思路采用**动态规划**的思想将直接优化分解后的全部矩阵递归分解求解单个矩阵的子问题，单个矩阵的最优解包含在其上一级。同样先将矩阵精确分解结果求出，将所有需要元素整数化的矩阵$\boldsymbol{A}_i$乘以不同的$\beta_i = 2^n, n \in \{1,2,3,\ldots,\}$倍数将其放大，再做整数优化就可以使矩阵$\boldsymbol{B_i} = \beta_i \boldsymbol{A}_i$中的元素在误差允许的范围内满足范围限定，此时就将求解满足误差要求的最小 q 值转换为求解最小的β_i。当$\boldsymbol{F_N} = \prod_{i=0}^{\boldsymbol{logN}} \boldsymbol{A}_i$时，放大所需的$\beta_1 \geq \beta_2 \geq \ldots \geq \beta_{logN}$,所以可以先令除$\boldsymbol{A}_i$以外的矩阵全为精确值，优化$\beta_i$得到最小的$\beta_i$使得误差满足要求，使用由优化的得到的矩阵替换原矩阵在用同样的方法优化β_{i-1}，直到求解出得到使误差满足要求的全部的β_i，q 就为$log_2\beta_1$+1。

8.2. DFT 矩阵的精确整数分解逼近模型

本问题在问题三的基础上增加误差小于等于 0.1 这一约束，同样将矩阵元素范围表示参数 q 作为决策变量，并将优化目标修改为最小化 q。为仍满足矩阵连乘拟合 DFT 矩阵中连乘的矩阵必须为每行至多有两个非零元素，所以和问题三同样，首先需要对 DFT 矩阵使用分治策略进行精确分解。再将分解结果中非范围的矩阵元素作为决策变量，将误差小于 0.1 作为约束条件，将矩阵元素表示范围 q 最小化作为目标，建立优化矩阵元素、矩阵元素范围表示参数 q、矩阵缩放因子β的多元混合整数优化模型。

首先根据问题一将 DFT 矩阵分解为：

$$\boldsymbol{F_N} = \left(\prod_{i=1}^{\log_2 \mathrm{N}-1} \boldsymbol{A_i}\right)(\boldsymbol{I}_{\mathrm{N}/2} \otimes \boldsymbol{F}_2)\mathbf{P}_{\mathrm{N}} \tag{8.1}$$

（1）决策变量

矩阵连乘拟合 DFT 矩阵需要设计的变量有：

① 分解后的矩阵中值非范围元素的值$[\mathbf{A_i}]_{\mathrm{k,n}}$：

$$[\mathbf{A_i}]_{\mathrm{k,n}} \in \{x + jy | x, y \in \mathcal{P}\}, \mathcal{P} = \{0, \pm 1, \pm 2, \ldots, \pm 2^{q-1}\}, (i,k,n) \in \mathrm{V} \tag{8.2}$$

$(i,k,n) \in \mathrm{V}$ 表示一三元组(i,k,n)为 V 中的一个元素；

② 实值矩阵缩放因子β：

$$\beta \in \mathcal{R} \tag{8.3}$$

其中$\mathcal{R}$表示实数域。

③ 矩阵元素范围表示参数 q：

$$\mathrm{q} \in \{1,2,3,\ldots\} \tag{8.4}$$

（2）目标函数

本问题使用 K 个稀疏矩阵连乘拟合近似 DFT 矩阵，需要使拟合 DFT 矩阵的矩阵连乘过所需的硬件复杂度最小，即：

$$min\ C = q \times L \tag{8.5}$$

其中 q 为分解后矩阵元素的取值范围参数，L 为矩阵连乘中复数乘法的次数。

（3）约束条件

为减少计算 DFT 矩阵的硬件复杂度并使矩阵连乘可以精确拟合 DFT 矩阵，作出如下

约束：

① 分解后的矩阵中值非范围元素的值$[\mathbf{A_i}]_{\mathrm{k,n}}$：

$$[\mathbf{A_i}]_{\mathrm{k,n}} \in \{x+jy|x,y \in \mathcal{P}\}, \mathcal{P}=\{0,\pm 1,\pm 2,\ldots,\pm 2^{q-1}\}, (i,k,n) \in \mathbf{V} \tag{8.6}$$

$\mathbf{V}$ 表示分解后的矩阵中值非范围元素的矩阵编号、行号、列号构成的三元组的集合。

② 矩阵连乘结果和原 DFT 矩阵误差小于 0.1。使用 Frobenius 范数意义下矩阵$\mathbf{F}_N$和$\beta\mathbf{A}_1\mathbf{A}_2\cdots\mathbf{A}_K$的误差作为矩阵连乘拟合 DFT 矩阵的误差，即：

$$\mathrm{RMSE}=\frac{1}{N}\sqrt{\left\|\mathbf{F}_N-\beta\left(\prod_{i=1}^{\log_2 \mathrm{N}-1}\boldsymbol{A_i}\right)(\mathbf{I}_{\mathrm{N}/2}\otimes\boldsymbol{F}_2)\mathbf{P}_{\mathrm{N}}\right\|_F^2} \leq 0.1 \tag{8.7}$$

综上所述，问题五 DFT 矩阵的精确整数矩阵分解逼近模型总结如下：

$$\text{s.t.}\begin{cases} min\ C=q\times L \\ \mathrm{q}\in\{1,2,3,\ldots\} \\ [\mathbf{A_i}]_{\mathrm{k,n}}\in\{x+jy|x,y\in\mathcal{P}\},(i,k,n)\in\mathbf{V} \\ \mathcal{P}=\{0,\pm 1,\pm 2,\ldots,\pm 2^{q-1}\} \\ \mathrm{RMSE}=\dfrac{1}{N}\sqrt{\left\|\mathbf{F}_N-\beta\left(\prod_{i=1}^{\log_2 \mathrm{N}-1}\boldsymbol{A_i}\right)(\mathbf{I}_{\mathrm{N}/2}\otimes\boldsymbol{F}_2)\mathbf{P}_{\mathrm{N}}\right\|_F^2}\leq 0.1 \end{cases} \tag{8.8}$$

8.3. 模型求解思路一——基于二分查找 q 的求解方法

由于 q 的大小决定了分解后的矩阵元素的取值范围，而更大的矩阵元素取值范围会使得矩阵连乘拟合的结果误差更小，所以 q 值越大，矩阵连乘拟合的结果误差越小。所以可以首先找到一个满足误差小于 0.1 的较大 q 值。寻找过程可以使用每次将 q 扩大 2 倍这样的方式寻找合适的 q 值。每次给定一个 q 值就可以使用问题三的求解过程得到该 q 值下最小的误差值，根据该误差是否小于 0.1，确定该 q 值是否为满足要求的 q 值。

当找到一个满足误差小于 0.1 的较大 q 值时，设该值为q_n,停止寻找更大的 q 值。将 q=2 作为左值，q=q_n作为右值，二分查找的思想在（2,q_n）范围内寻找满足误差小于 0.1 的最小 q 值。具体算法流程为：

（1）设 left=2,right=q_n；

（2）判断 left 是否小于等于 right,小于等于时到（3）步，否则到（5）步；

（3）mid=(left+right)/2 向上整；

（4）将问题三模型中的 q 值设为 mid，使用其求解过程得到此时的最小误差，判断其是否小于 0.1，若此时的最小误差小于 0.1 则令 right=mid 后到（2）步；反之，令 left=mid 后到（2）步；

（5）满足误差小于 0.1 的最小 q 为 left。

8.4. 模型求解思路二——基于动态规划的求解方法

（1）将求解满足误差要求的最小 q 转换为求解满足误差要求的最小β_i

由第一问可知 N 维 DFT 矩阵可以精确分解为 $\log_2$N+1 个矩阵：

$$\boldsymbol{F_N} = (\prod_{i=1}^{\log_2 N-1} \boldsymbol{A_i})(\mathbf{I}_{N/2} \otimes \boldsymbol{F}_2)\mathbf{P}_N \tag{8.8}$$

其中$(\mathbf{I}_{N/2} \otimes \boldsymbol{F}_2)$、$\mathbf{P}_N$为整数矩阵不考虑其整数优化。

由第三问整数矩阵逼近法，存在整数矩阵可以近似逼近原 DFT 矩阵：

$$\widehat{\boldsymbol{F_N}} = \frac{1}{\beta}(\prod_{i=1}^{\log_2 N-1} \boldsymbol{B_i})(\mathbf{I}_{N/2} \otimes \boldsymbol{F}_2)\mathbf{P}_N \approx \boldsymbol{F_N} \tag{8.9}$$

其中$\boldsymbol{B_i}$**为整数矩阵，**$1/\beta$为矩阵缩放因子，设$\beta=\beta_1\beta_2...\beta_{\log_2 N-1}$可得：

$$\boldsymbol{B_i} \approx \beta_i \boldsymbol{A_i}, i = 1,2,..,\log_2 N - 1 \tag{8.10}$$

由于旋转因子的实部和虚部均为小于等于 1 的实数，所以对于整数逼近后的矩阵$\boldsymbol{B_i}$其实部和虚部均为小于等于β_i的实数，即矩阵$\boldsymbol{B_i}$中元素的实部和虚部的最大值为β_i，求解 q 的最小值就是求解满足精度要求的矩阵元素的实部和虚部的最大值最小值，即求解β_i的最小值。

（2）$\boldsymbol{\beta_i}$的性质

由于旋转因子的实部和虚部满足当 m ≥ n 时，$|a|^m \le |a|^n$。由于$\beta_1|a|^m=\beta_2|a|^n$,所以$\beta_1 \ge \beta_2$，依次类推，可以得到$\beta_1 \ge \beta_2 \ge \beta_3 ... \ge \beta_{\log_2 N-1}$。因此求解满足误差的所有最小$\beta_i$就是求解满足误差要求的$\beta_1$的最小值。

（3）求解满足误差要求的$\boldsymbol{\beta_1}$的最小值

由于要求分解后的矩阵元素为 2 的整数次幂的数，所以必须使得$\beta_i=2^{in_i}$，可以得到β_i由n_i决定。设$a_i = n_i - n_{i-1}, i = 1,2,3,..\log_2 N - 2$, b=$n_{\log_2 N-1}$,就可以得到：

$$\beta_i = 2^{b+\sum_{j=i}^{\log_2 N-2} a_i}, i = 1,2,..,\log_2 N - 1 \tag{8.11}$$

因此可以将该问题转换为求解满足误差要求的a_i和 b 的值。

由β_i的性质可知，β_i的优化空间包含关系为 $sp(\beta_1) \subseteq sp(\beta_2) \subseteq ...sp(\beta_{\log_2 N-1})$，其中$sp(\beta_i)$表示$\beta_i$的取值空间，满足动态规划的性质，可以通过问题的最优解包含子问题最优解，可以通过先找到最优的$\beta_{\log_2 N-1}$来依次找到最优的β_i。

所以先设$a_i = 0$， $i = 1,2,..,\log_2 N - 2$，b=0 利用第一问分解后的结果先对$\boldsymbol{A}_{\log_2 N-1}$矩阵中的元素使用问题三的模型将其整数化近似，这时仅优化矩阵$\boldsymbol{A}_{\log_2 N-1}$中的元素，穷举 b 找到最下的满足误差的 b 值为b_n，然后使用此时优化出的矩阵$\boldsymbol{B}_{\log_2 N-1}$替换精确分解中的矩阵$\boldsymbol{A}_{\log_2 N-1}$。

再穷举$a_{\log_2 N-2}$求解得到最优的$a_{\log_2 N-2}$值，依次类推，求解出所有的a_i。也就求解出了满足误差的元素范围最小的矩阵分解。

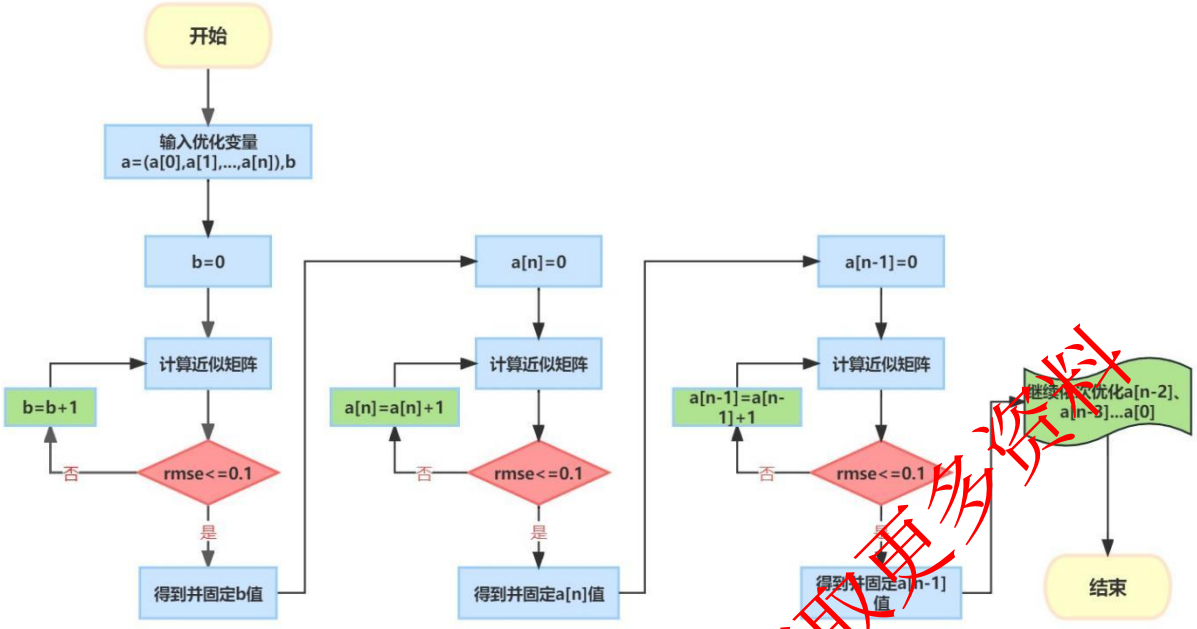

图 8-2 基于动态规划的求解方法流程图

8.5. 模型求解结果

使用求解思想一求解时，发现需要对 q 的确定需要做的混合整数优化次数太多且每次同事优化全部分解后的矩阵，计算消耗太大，不易计算。所以使用求解思想二进行了求解，思想二每次仅求解一个矩阵的混合整数优化模型，可以在有限时间内求解得到满足误差小于等于 0.1 的最小 q 值，但此时的分解后的矩阵元素绝对值范围为 0 到2^{q-1}的全部整数，即：

$$\mathbf{A}_k[l,m] \in \{x+jy | x,y \in \mathcal{P}\}, \mathcal{P} = \{0, \pm 1, \pm 2, \pm 3 ..., \pm 2^{q-1}\}, k = 1,2,...,K;\ \ l,m = 1,2,...,N \tag{8.12}$$

此时单个乘法器所需的位宽为 q+1,使用 q+1 位的 2 进制数就可以表示该范围内所有整数。

使用动态规划的思想路二求解结果如下所示：

（1）t =1 时，$N = 2^t = 2$，2 维 DFT 矩阵$\boldsymbol{F}_2$不需要分解，所以矩阵数量为 K=1，矩阵为：

$$\boldsymbol{F_2} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \tag{8.13}$$

（2）t =2 时，$N = 2^t = 4$，4 维 DFT 矩阵$\boldsymbol{F_4}$经过分解算法得到的矩阵数量为 K=3，矩阵为：

$$\boldsymbol{B_1} = \begin{bmatrix} 1 & & 1 & \\ & 1 & & -j \\ 1 & & -1 & \\ & 1 & & j \end{bmatrix} \tag{8.14}$$

$$\mathbf{I_2} \otimes \boldsymbol{F}_2 = \begin{bmatrix} 1 & 1 & & \\ 1 & -1 & & \\ & & 1 & 1 \\ & & 1 & -1 \end{bmatrix} \tag{8.15}$$

$$\mathbf{P_4} = \begin{bmatrix} 1 & 1 & & \\ 1 & -1 & & \\ & & 1 & 1 \\ & & 1 & -1 \end{bmatrix} \tag{8.16}$$

$\boldsymbol{F_4}$使用的矩阵连乘形式为：

$$\boldsymbol{F_4} = \boldsymbol{B_1}(\mathbf{I_2} \otimes \boldsymbol{F_2})\mathbf{P_4} \tag{8.17}$$

（3）t =3 时，$N = 2^t = 8$，8 维 DFT 矩阵$\boldsymbol{F_8}$经过分解算法得到的矩阵数量为 K=4，矩阵为：

$$\boldsymbol{B_1} = \begin{bmatrix} 4 & 0 & 0 & 0 & 4 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 & 3-3j & 0 & 0 \\ 0 & 0 & 4 & 0 & 0 & 0 & -4j & 0 \\ 0 & 0 & 0 & 4 & 0 & 0 & 0 & -3+3j \\ 4 & 0 & 0 & 0 & -4 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 & -3+3j & 0 & 0 \\ 0 & 0 & 4 & 0 & 0 & 0 & 4j & 0 \\ 0 & 0 & 0 & 4 & 0 & 0 & 0 & 3+3j \end{bmatrix}$$

$$\boldsymbol{B_2} = \begin{bmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -j & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & j & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & -j \\ 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & j \end{bmatrix}$$

$$\boldsymbol{B_3} = \boldsymbol{I_4} \otimes \boldsymbol{F_2}$$

$$\boldsymbol{P_8} = [e_0 \quad e_4 \quad e_2 \quad e_6 \quad e_1 \quad e_5 \quad e_3 \quad e_7]$$

$\boldsymbol{F_8}$使用的矩阵连乘形式为：

$$\boldsymbol{F_8} = \boldsymbol{B_1}\boldsymbol{B_2}(\mathbf{I_4} \otimes \boldsymbol{F_2})\mathbf{P_8} \tag{8.18}$$

（4）t =4 时，$N = 2^t = 16$，16 维 DFT 矩阵$\boldsymbol{F_{16}}$经过分解算法得到的矩阵数量为 K=5，矩阵为：

$$\boldsymbol{B_1} = \begin{bmatrix} \mathbf{4} * \boldsymbol{I_8} & \boldsymbol{B_{1_{0,1}}} \\ \mathbf{4} * \boldsymbol{I_8} & \boldsymbol{B_{1_{1,1}}} \end{bmatrix}$$

$$\boldsymbol{B_{1_{0,1}}} = \begin{bmatrix} 4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 4-2j & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3-3j & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2-4j & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -4j & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -2-4j & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -3-3j & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -4-2j \end{bmatrix}$$

$$\boldsymbol{B}_{1_{1,1}} = -\boldsymbol{B}_{1_{0,1}}$$

$$\boldsymbol{B}_2 = \begin{bmatrix} \boldsymbol{B}_{2_{0,0}} & \\ & \boldsymbol{B}_{2_{1,1}} \end{bmatrix}$$

$$\boldsymbol{B}_{2_{0,0}} = \begin{bmatrix} 4 & 0 & 0 & 0 & 4 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 & 3-3j & 0 & 0 \\ 0 & 0 & 4 & 0 & 0 & 0 & -4j & 0 \\ 0 & 0 & 0 & 4 & 0 & 0 & 0 & -3-3j \\ 4 & 0 & 0 & 0 & -4 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 & -3+3j & 0 & 0 \\ 0 & 0 & 4 & 0 & 0 & 0 & 4j & 0 \\ 0 & 0 & 0 & 4 & 0 & 0 & 0 & 3+3j \end{bmatrix}$$

$$\boldsymbol{B}_{2_{1,1}} = \boldsymbol{B}_{2_{0,0}}$$

$$\boldsymbol{B}_3 = \boldsymbol{I}_4 \otimes \boldsymbol{B}_{3_{0,0}}$$

$$\boldsymbol{B}_{3_{0,0}} = \begin{bmatrix} 1 & 1 & & \\ 1 & & & -j \\ & & -1 & \\ & 1 & & j \end{bmatrix}$$

$$\boldsymbol{B}_4 = \boldsymbol{I}_8 \otimes \boldsymbol{F}_2$$

$$\boldsymbol{P}_{16} = [e_0 \quad e_8 \quad e_4 \quad e_{12} \quad e_2 \quad e_{10} \quad e_6 \quad e_{14} \quad e_1 \quad e_9 \quad e_5 \quad e_{13} \quad e_3 \quad e_{11} \quad e_7 \quad e_{15}]$$

$\boldsymbol{F}_{16}$使用的矩阵连乘形式为：

$$\boldsymbol{F}_{16} = \boldsymbol{B}_1\boldsymbol{B}_2\boldsymbol{B}_3(\mathbf{I}_8 \otimes \boldsymbol{F}_2)\mathbf{P}_{16} \tag{8.19}$$

（5）t =5 时，$N = 2^t$ =32，32 维 DFT 矩阵$\boldsymbol{F}_{32}$经过分解算法得到的矩阵数量为 K=6，矩阵为：

$$\boldsymbol{B}_1 = \begin{bmatrix} \mathbf{16} * \boldsymbol{I}_8 & & \boldsymbol{B}_{1_{0,2}} & \\ & \mathbf{16} * \boldsymbol{I}_8 & & \boldsymbol{B}_{1_{1,3}} \\ \mathbf{16} * \boldsymbol{I}_8 & & \boldsymbol{B}_{1_{2,2}} & \\ & \mathbf{16} * \boldsymbol{I}_8 & & \boldsymbol{B}_{1_{3,3}} \end{bmatrix}$$

$$B_{1_{0,2}} = \begin{bmatrix} 16 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 16-3j & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 15-6j & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 13-9j & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 11-11j & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 9-13j & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 6-15j & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 3-16j \end{bmatrix}$$

$$B_{1_{1,3}} = \begin{bmatrix} -16j & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -3-16j & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -6-15j & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -9-13j & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -11-11j & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -13-9j & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -15-6j & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -16-3j \end{bmatrix}$$

$$B_{1_{2,2}} = -B_{1_{0,2}}$$

$$B_{1_{3,3}} = -B_{1_{1,3}}$$

$$B_2 = \begin{bmatrix} 4 * I_8 & B_{2_{0,1}} & & \\ 4 * I_8 & B_{2_{1,1}} & & \\ & & 4 * I_8 & B_{2_{2,3}} \\ & & 4 * I_8 & B_{2_{3,3}} \end{bmatrix}$$

$$B_{2_{0,1}} = \begin{bmatrix} 4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 4-2j & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 3-3j & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2-4j & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -4j & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -2-4j & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -3-3j & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -4-2j \end{bmatrix}$$

$$B_{2_{1,1}} = B_{2_{3,3}} = -B_{2_{0,1}}$$

$$B_{2_{2,3}} = B_{2_{0,1}}$$

$$B_3 = I_4 \otimes B_{3_{0,0}}$$

$$B_{3_{0,0}} = \begin{bmatrix} 4 & 0 & 0 & 0 & 4 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 & 3-3j & 0 & 0 \\ 0 & 0 & 4 & 0 & 0 & 0 & -4j & 0 \\ 0 & 0 & 0 & 4 & 0 & 0 & 0 & -3-3j \\ 4 & 0 & 0 & 0 & -4 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 & -3+3j & 0 & 0 \\ 0 & 0 & 4 & 0 & 0 & 0 & 4j & 0 \\ 0 & 0 & 0 & 4 & 0 & 0 & 0 & 3+3j \end{bmatrix}$$

$$B_4 = I_8 \otimes B_{4_{0,0}}$$

$$B_{4_{0,0}} = \begin{bmatrix} 1 & & 1 & \\ & 1 & & -j \\ 1 & & -1 & \\ & 1 & & j \end{bmatrix}$$

$$B_5 = I_{16} \otimes F_2$$

$$P_{32} = [e_0 \quad e_{16} \quad e_8 \quad e_{24} \quad e_4 \quad e_{20} \quad e_{12} \quad e_{28} \quad e_2 \quad e_{18} \quad e_{10} \quad e_{26} \quad e_6 \quad e_{22} \quad e_{14} \quad e_{30} \quad e_1 \quad e_{17} \quad e_9 \quad e_{25} \quad e_5 \quad e_{21} \quad e_{13} \quad e_{29} \quad e_3 \quad e_{19} \quad e_{11} \quad e_{27} \quad e_7 \quad e_{23} \quad e_{15} \quad e_{31}]$$

F_{32}使用的矩阵连乘形式为：

$$F_{32} = B_1 B_2 B_3 B_4 (I_{16} \otimes F_2) P_{32} \tag{8.20}$$

（6）当$t = 6,7,\ldots$时，$N = 2^t = 32,64,128,\ldots$，$N$维 DFT 矩阵$F_N$经过分解算法得到的矩阵数量为 K=t+1，$F_N$使用的矩阵连乘形式为：

$$F_N = \left(\prod_{i=1}^{\log_2 N - 1} B_i \right) (I_{N/2} \otimes F_2) P_N \tag{8.21}$$

8.6. 求解结果分析

这样的递归算法对 N 为 DFT 矩阵F_N的分解需要递归的层数为 $\log_2 N - 1$，这样的算法可以快速完成分解。对$t = 1,2,3,\ldots,10$ 时，2^t维 DFT 矩阵F_{2^t}的分解矩阵个数，误差，硬件复杂度进行了统计，如表 4.2 所示。发现可以得到误差几乎均为 0，和原始 DFT 矩阵计算的硬件复杂度相比分解后的矩阵连乘硬件复杂度有着明显的下降，但由于矩阵某些元素所需的数据位宽较高，仍使得矩阵的硬件复杂度不太理想。

表 8-1　2^t维 DFT 矩阵分解结果统计表

t	N	β	$\mathcal{A}$中矩阵个数	RMSE	C	耗时
1	2	1	2	6.12323e-17	0	0ms
2	4	1	3	2.14313e-16	0	0ms
3	8	1/4	4	0.0303301	48	1ms
4	16	1/16	5	0.091805	288	4ms
5	32	1/256	6	0.092596	2240	33ms
6	64	1/4096	7	0.0934097	9600	300ms
7	128	1/65536	8	0.0977792	39680	2729ms
8	256	1/2.09715e+06	9	0.0984291	193536	28064ms
9	512	1/6.71089e+07	10	0.0986981	780288	265784ms
10	1024	1/2.14748e+09	11	0.0989267	3133440	2785954ms

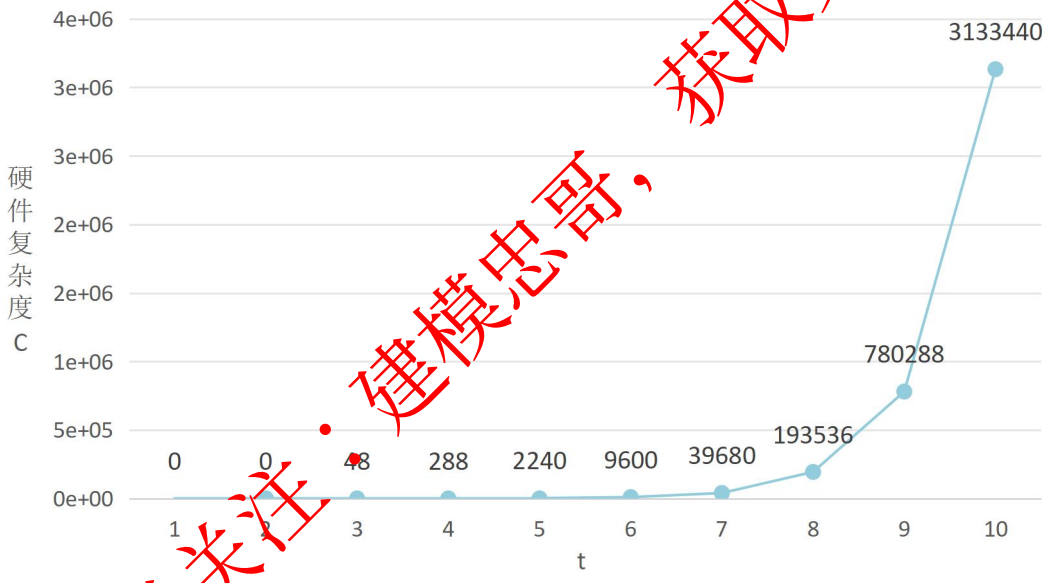

图 8-1　N=2^t维 DFT 矩阵分解结果硬件复杂度变化折线图

9. 模型评价与算法分析

9.1 算法的有效性和复杂度分析

问题一中使用库-图算法递归求解每个分解矩阵，不管是多大规模的矩阵都能得到精确解。

问题二中约束只要求矩阵元素为整数，我们使用 DFS+剪枝法来优化矩阵的元素，此时有 N * N 个变量，每个变量分为实部和虚部两个部分，$\mathcal{P}=\{0,\pm1,\pm2,\pm4\}$，其时间复杂度为(N * N * 2)^7。由于通过对矩阵的分解，使得 N 维矩阵的变量个数变为$\left(\frac{N}{4}-1\right)*2$，此时有$\left[\left(\frac{N}{4}-1\right)*2\right]^7$种情况，DFS 深度为变量的个数，每一层可分为 7 枝，当复数的实部和虚部同时为 0 时，矩阵第一行全为 0，此时我们剪掉这一枝，故最终求解$\left[\left(\frac{N}{4}-1\right)*2\right]^7-$

$\sum_{i=1}^{\left(\frac{N}{4}-1\right)*2-2} i^7$种情况，N 最大为 32，可解。

问题三中，由于第一题的分解矩阵只不满足约束 2 且精度为 0，因此我们在此基础上优化第一题已经分解的矩阵，固定矩阵中每个元素的位置，只将其优化为$\mathcal{P}$中的值，矩阵中的元素为 0，1，-1 和w_n的幂，而只有w_n的幂不满足约束 2，所以我们只需要优化含有w_n的项，对于 N 维矩阵需要优化$\left(\frac{N}{2}-1\right)*2$ 个变量，对于每个变量，我们将其拆分成系数 k（k=1,-1,0）和指数 a（a=0,1,2）两部分，然后用遗传算法求近似解。

问题四中需要同时优化多个矩阵，且优化变量很多，无法使用穷举法求解，因此我们使用 Gurobi 求解。

问题五采用动态规划的思想将问题分解为一系列子问题，对于每个子问题的最优解即为最终的全局最优解，求解复杂度为 O(2^n)，在矩阵规模为 256 维及以下时都能快速地得到解，但是在 512 维及以上时求解用时大幅增加。

9.2 优点分析

（1）本文利用离散傅里叶变换矩阵的库-图分解得到了优化范围很小的精确解，然后在该精确解的基础上进行整数逼近，可以高效地得到高精度的整数逼近结果。

（2）本文利用了离散傅里叶变换矩阵的性质对矩阵元素结构进行了简化，大幅减少了优化变量的个数。

（3）本文利用了傅里叶变换矩阵分解后的矩阵的性质，结合分解后各个矩阵之间的元素范围在一定精度下的关系，将矩阵元素范围的求解分解为一系列嵌套的子问题，然后基于动态规划的思想提出了一种有效的优化算法，可以在短时间内得到比较精确的结果。

9.3 缺点分析

（1）建立的模型均为混合整数规划模型，优化求解困难，只能使用 Gurobi 或相对耗时的通用方法求解，但是在矩阵规模较大时求解复杂度很大，难以得到精确结果，因此只能通过放宽约束来得到结果。

（2）除了第五问外，本文的模型在求解过程中没有将硬件复杂度作为求解目标，因此得到的硬件复杂度较高。

参考文献

[1] Coutinho, Vitor A., et al. "A low-SWaP 16-beam 2.4 GHz digital phased array receiver using DFT approximation." *Ieee Transactions on Aerospace and Electronic Systems* 56.5 (2020): 3645-3654.

[2] Madanayake, Arjuna, et al. "Towards a low-SWaP 1024-beam digital array: A 32-beam subsystem at 5.8 GHz." *IEEE Transactions on Antennas and Propagation* 68.2 (2019): 900-912.

[3] Coutinho, V. A., et al. "An 8-beam 2.4 GHz digital array receiver based on a fast multiplierless spatial DFT approximation." *2018 IEEE/MTT-S International Microwave Symposium-IMS*. IEEE, 2018.

[4] Ariyarathna, Viduneth, et al. "Analog approximate-FFT 8/16-beam algorithms, architectures and CMOS circuits for 5G beamforming MIMO transceivers." *IEEE Journal on Emerging and Selected Topics in Circuits and Systems* 8.3 (2018): 466-479.

[5] Suarez, Dora, et al. "Multi - beam RF aperture using multiplierless FFT approximation." *Electronics Letters* 50.24 (2014): 1788-1790.

[6] Madanayake, Arjuna, et al. "Fast radix-32 approximate DFTs for 1024-beam digital RF beamforming." *IEEE Access* 8 (2020): 96613-96627.

[7] Ersoy, Okan K., and Neng-Chung Hu. "Fast algorithms for the real discrete Fourier transform." (1988).

[8] Ariyarathna, Viduneth, et al. "Multibeam digital array receiver using a 16-point multiplierless DFT approximation." *IEEE Transactions on Antennas and Propagation* 67.2 (2018): 925-933.

[9] Johnson, Steven G., and Matteo Frigo. "A modified split-radix FFT with fewer arithmetic operations." *IEEE Transactions on Signal Processing* 55.1 (2006): 111-119.

[10] Potluri, U. S., et al. "Multiplier-free DCT approximations for RF multi-beam digital aperture-array space imaging and directional sensing." *Measurement Science and Technology* 23.11 (2012): 114003.

[11] Portella, Luan, et al. "Radix-$ N $ Algorithm for Computing $ N^{2^{n}} $-Point DFT Approximations." *IEEE Signal Processing Letters* 29 (2022): 1838-1842.

[12] Cintra, Renato J. "An integer approximation method for discrete sinusoidal transforms." *Circuits, Systems, and Signal Processing* 30 (2011): 1481-1501.

[13] K. R. Rao, D. N. Kim, and J. J. Hwang, Fast Fourier Transform: Algorithms and Applications, Springer, 2010. （中译本：快速傅里叶变换：算法与应用，万帅，杨付正译，机械工业出版社，2012.）

[14] (美)布莱赫特(Blahut， R.E.)著；肖先赐等译. 数字信号处理的快速算法[M]. 北京：科学出版社, 1992

附录：C++代码

#1 遗传算法头文件 GeneticAlgorithm.h

```
#pragma once
#include<random>
#include<vector>
#include<iostream>
#include <fstream>
#include<cmath>
#include<ctime>
#include <string>
#include <cstdlib>
#include<iomanip>
#include <functional>
#define PI 3.141592653589793

namespace GENE
{
    //自变量取值范围类，适用于多个变量
    class X_Range
    {
    public:
        X_Range(double m_Lower, double m_Upper);//构造函数
        double GetUpper()const;//获取变量上限
        double GetLower()const;//获取变量下限
    private:
        double Upper;//变量的上界取值
        double Lower;//变量的下界取值
    };

    //个体类
    class Individual
    {
    public:
        Individual() {}//默认构造函数
        Individual(std::vector<double>& m_Variable, std::vector<X_Range>& Ranges);//构造函数
        std::vector<double> GetVariable();//获取变量值
        void ChaFitness(const double m_fitness);//修改适应值
```

```
        void ChaReFitness(const double m_ReFitness);//修改适应值概率
        void ChaSumFitness(const double m_SumFitness);//修改累加概率
        double GetFitness()const;//获取适应值
        double GetReFitness()const;//获取适应值概率
        double GetSumFitness()const;//获取累加概率
    private:
        std::vector<double> Variable;//存放变量 x1,x2,x3.....
        double Fitness = 0;//适应值
        double ReFitness = 0;//适应值概率
        double SumFitness = 0;//累加概率，为轮盘转做准备
    };

    //遗传算法变量
#define GENE_MAXIMIZE 0 //优化目标：最大值
#define GENE_MINIMIZE 1 //优化目标：最小值
#define GENE_MAX 1e08 //max var
#define GENE_MIN -1e08 //min var
#define GENE_PSIZE 0 //种群规模参数标签
#define GENE_EVALGEBRA 1 //进化代数参数标签
#define GENE_OVPROBABILITY 2 //交叉概率参数标签
#define GENE_VAPROBABILITY 3 //变异概率参数标签
#define GENE_OPTSIZE 4 //目标数据维数参数标签

    //遗传算法决策变量
    typedef struct GENEVar
    {
        int _idx = -1;
        double _var;
        GENEVar(int id, double x) :_idx(id), _var(x) {};
    };

    //遗传算法类
    class Genetic
    {
        friend class X_Range;
        friend class Individual;

    public:
```

```
        Genetic() {};
        ~Genetic() {};

        //添加决策变量（最小值，最大值，精确到小数点后几位）
        GENEVar addVar(double _min, double _max, int _pNum);

        //获取当前最优变量值（第_pid 组数据）
        void getVar(GENEVar& _var, int _pid = 0);

        //获取当前最优目标值（第_pid 组数据）
        double getObjVar(int _pid = 0);

        //设置优化目标
        void setObjective(std::function<double(std::vector<GENEVar>&)>& obj,
std::vector<GENEVar>& vars, int sense = 1);

        //设置参数：种群规模，进化代数，交叉概率，变异概率，目标数据组数
        void setPara(int tag, double a);

        //优化目标
        void optimize();

    private:
        void Initialize();//随机初始化种群，得到第一代个体
        void CaculaFitness();//计算个体的适应值
        void CaculaReFitness();//计算个体的适应值概率
        void CalculaSumFitness();//计算累加个体概率
        void select();//种群选择
        double Scand();//随机产生 0 到 1 的小数
        void crossing();//杂交
        void variating();//变异

        std::vector<unsigned short int> BitSet(int x, int len); //编码
        int DeBitSet(std::vector<unsigned short int>& arr); //解码

        int Po_Size = 500;//种群规模
        int Ev_Algebra = 500;//进化代数
        double Ov_Probability = 0.900; //交叉概率,交叉概率用于判断两两个体是否需要交叉
```

```
        double Va_Probability = 0.100;//变异概率,变异概率用于判断任一个体是否需要变异
        int De_Variable = 0;//函数自变量的个数,如果要进行多元函数的最值求解，直接修改自变量数个
De_Variable 即可
        std::vector<int> Lens; //变量编码长度数组
        std::vector<int> PNums; //变量精确度（精确到小数点后 PNum[i]位）数组
        std::vector<X_Range> Ranges;//自变量的取值范围
        std::vector<double> Offsets;//自变量的偏移量（编码时要求全为正数）
        std::vector<Individual> nowpopulation;//P(t)种群
        std::vector<Individual> midpopulation;//中间种群，存放轮盘选择后的优秀个体
        std::vector<Individual> nextpopulation;//P(t+1)种群

        std::function<double(std::vector<GENEVar>&)> _objFunc; //目标函数
        std::vector<GENEVar> _objVars; //目标函数变量
        int _objSense = 0; //优化目标:  0:最大值，1:最小值
        std::vector<std::vector<double>> optVars; //最优变量
        std::vector<double> optObjValue; //最优值
        int optSize = 1; //最优数据组数

        void PrintTopInfo();
    };
}
```

#2 遗传算法源文件 GeneticAlgorithm.cpp

```
#include"GeneticAlgorithm.h"
#include <conio.h>
#include <tbb/tbb.h>

namespace GENE
{
    //X_Range 类实现
    X_Range::X_Range(double m_Lower, double m_Upper) :Lower(m_Lower), Upper(m_Upper)
{}//X_Range 类构造函数实现
    double X_Range::GetUpper()const//获取变量上限
    {
        return Upper;
    }
    double X_Range::GetLower()const//获取变量下限
```

```
    {
        return Lower;
    }

    //Individual 类实现
    Individual::Individual(std::vector<double>& m_Variable, std::vector<X_Range>& Ranges)//构造函数
    {
        Variable.resize(m_Variable.size());
        for (int i = 0; i < m_Variable.size(); i++)//用 for 循环自变量逐个赋值
        {
            if (m_Variable[i] >= Ranges[i].GetLower() && m_Variable[i] <= Ranges[i].GetUpper())//这里要进行自变量取值范围判断
                Variable[i] = m_Variable[i];//自变量赋值
            else//不满足要求则发出出错警告并返回
            {
                std::cerr << "自变量取值不满足要求\n";
                exit(1);//停止程序，我会以随机函数的方式生成自变量的值(基因值)，这里说明基因值不在规定范围内
            }
        }
        //初始化时默认适应值等值为 0
        this->Fitness = 0;
        this->ReFitness = 0;
        this->SumFitness = 0;
    }
    std::vector<double> Individual::GetVariable()//获取基因值
    {
        return Variable;
    }
    double Individual::GetFitness()const//获取适应值
    {
        return Fitness;
    }
    double Individual::GetReFitness()const //获取适应值概率
    {
        return ReFitness;
    }
```

```
double Individual::GetSumFitness()const//获取累加概率
{
    return SumFitness;
}
void Individual::ChaFitness(const double m_fitness)//修改适应值
{
    this->Fitness = m_fitness;
}
void Individual::ChaReFitness(const double m_ReFitness)//修改适应值概率
{
    this->ReFitness = m_ReFitness;
}
void Individual::ChaSumFitness(const double m_SumFitness)//修改累加概率
{
    this->SumFitness = m_SumFitness;
}

//编码
std::vector<unsigned short int> Genetic::BitSet(int x, int len)
{
    std::vector<unsigned short int> arr;
    if (len <= 0)return arr;
    while (x)
    {
        arr.push_back(x % 2);
        x /= 2;
    }
    for (int i = arr.size(); i < len; i++)
        arr.push_back(0);
    std::reverse(arr.begin(), arr.end());
    return arr;
}

//解码
int Genetic::DeBitSet(std::vector<unsigned short int>& arr)
{
    if (arr.empty())return 0;
    int x = 0, n = arr.size();
```

```
        for (int i = 0; i < n; i++)
            x += arr[i] * pow(2, n - i - 1);
        return x;
    }

    void Genetic::Initialize()//随机初始化种群，得到第一代种群
    {
        //产生指定范围的随机变量（基因）
        std::vector<std::vector<double>> X(Po_Size, std::vector<double>(De_Variable));//为了使程序可以满足多元函数最值的计算，用矩阵保存产生的随机数变量值
        for (int j = 0; j < De_Variable; j++)
        {
            std::default_random_engine e(time(0));//引擎，生成随机序列
            std::uniform_real_distribution<double> u(Ranges[j].GetLower(), Ranges[j].GetUpper());//分布
            for (int i = 0; i < Po_Size; i++)//先按列存储随机数
                X[i][j] = u(e);//循环结束时，所有随机值就保存在 X 矩阵中
        }
        //生成对象（染色体）并加入到初始种群中
        for (int i = 0; i < Po_Size; i++)
        {
            std::vector<double> variable(De_Variable);
            for (int j = 0; j < De_Variable; j++)
                variable[j] = X[i][j];//按行保存
            Individual Indivi(variable, Ranges);//生成一个对象（染色体）
            nowpopulation.push_back(Indivi);//加入到种群 population 中
        }
        //设置结果变量数组大小
        optVars.resize(optSize, std::vector<double>(De_Variable));
        optObjValue.resize(optSize, GENE_MIN);
    }

    //计算个体的适应值
    void Genetic::CaculaFitness()
    {
        auto _map = [&](const tbb::blocked_range<uint32_t>& range) {
            for (uint32_t i = range.begin(); i < range.end(); ++i)
            {
```

```
            std::vector<double> x(De_Variable);//临时存储自变量（基因）
            for (int j = 0; j < De_Variable; j++)
                x[j] = nowpopulation.at(i).GetVariable()[j];
            for (int j = 0; j < _objVars.size(); j++)
                _objVars[j]._var = x[_objVars[j]._idx];
            // 适应度计算
            double fitness = _objFunc(_objVars);
            if (fitness > GENE_MAX)fitness = GENE_MAX;
            if (fitness < GENE_MIN)fitness = GENE_MIN;
            if (_objSense == 1)
                fitness = -fitness;
            nowpopulation.at(i).ChaFitness(fitness);//修改当前染色体的适应值
        }};
    tbb::parallel_for(tbb::blocked_range<uint32_t>(0, Po_Size), map);
}

//计算适应值概率
void Genetic::CaculaReFitness()
{
    long double sum = 0;//适应值累加器
    double temp = 0;
    for (int i = 0; i < Po_Size; i++)//计算出适应值之和
        sum += nowpopulation.at(i).GetFitness();
    for (int i = 0; i < Po_Size; i++)
    {
        temp = nowpopulation.at(i).GetFitness() / sum;//计算概率
        nowpopulation.at(i).ChaReFitness(temp);//修改个体的适应度概率
    }
}

//计算累加个体概率
void Genetic::CalculaSumFitness()
{
    double summation = 0;//累加器
    for (int i = 0; i < Po_Size; i++)
    {
        summation += nowpopulation.at(i).GetReFitness();
        nowpopulation.at(i).ChaSumFitness(summation);//当前累加结果赋值
```

```
        }
    }

    //种群选择
    void Genetic::seclect()
    {
        //随机生生成 0 到 1 的小数
        std::vector<double> array(Po_Size);//随机数保存变量
        std::default_random_engine e(time(0));//引擎，生成随机序列
        std::uniform_real_distribution<double> u(0.0, 1.0); //分布
        for (int i = 0; i < Po_Size; i++)
            array[i] = u(e);
        //轮盘进行选择
        for (int j = 0; j < Po_Size; j++)
        {
            for (int i = 1; i < Po_Size; i++)
            {
                if (array[j] < nowpopulation[i - 1].GetSumFitness())
                    midpopulation.push_back(nowpopulation.at(i - 1));//加入到中间种群
                if (array[j] >= nowpopulation.at(i - 1).GetSumFitness() && array[j] <=
nowpopulation.at(i).GetSumFitness())
                    midpopulation.push_back(nowpopulation.at(i));//加入到中间种群
            }
        }
        nowpopulation.clear();//清空 nowpopulation
    }

    //随机产生 0 到 1 的小数
    double Genetic::Scand()
    {
        int N = rand() % 1000;
        return double(N) / 999.0;;//随机产生 0 到 1 的小数
    }

    //杂交
    void Genetic::crossing()
    {
        int num = 0;//记录次数
```

```
        double corss = 0.0;//保存随机产生的概率值
        srand((unsigned)time(NULL));//根据系统时间设置随机数种子,设置一次随机种子就行
        std::vector<double> array1(De_Variable), array2(De_Variable);//临时存储父亲和母亲的变量值
        while (num < Po_Size - 1)//个体 1 与个体 2 杂交，个体 3 与个体 4 杂交.....个体 i 和个体 i+1 杂交
        {
            //判断双亲是否需要杂交，随机生成一个 0 到 1 的小数，如果这个数大于杂交概率，则放弃杂交，直接遗传给下一代，否则，对父母体进行杂交
            corss = Scand();
            if (corss <= Ov_Probability)//如果 corss 小于等于杂交概率 Ov_Probability 就进行单点杂交
            {
                //首先寻找对应下标的个体并且保存
                for (int i = 0; i < De_Variable; i++)
                {
                    array1[i] = midpopulation.at(num).GetVariable()[i];//父亲的自变量
                    array2[i] = midpopulation.at(num + 1).GetVariable()[i];//母亲自变量
                }
                std::vector<double> newx1(De_Variable), newx2(De_Variable);//分别用来保存基因交换后所对应自变量值
                bool fg = true;
                //然后对双亲变量进行编码并且进行单点杂交
                for (int i = 0; i < De_Variable && fg; i++)//array1 的 x1 编码之后和 array2 的 x1 编码后进行单点杂交，以此类推
                {
                    //对变量进行编码并且杂交
                    auto array1b = BitSet((array1[i] + Offsets[i]) * pow(10, PNums[i]), Lens[i]);//进行母体 1 的 x1 编码
                    auto array2b = BitSet((array2[i] + Offsets[i]) * pow(10, PNums[i]), Lens[i]);//进行母体 1 的 x1 编码
                    //现在随机生成 0 到 Len-1 的数，确定交叉点的位置
                    int localx = rand() % Lens[i];
                    //现在进行单点交叉，交换双亲 localx 后面的基因
                    for (int j = 0; j < localx; j++)
                        std::swap(array1b[j], array2b[j]);
                    //新值保存在 newx1 数组中，x1 基因完成单点杂交操作
                    newx1[i] = DeBitSet(array1b) / pow(10, PNums[i]) - Offsets[i];
                    newx2[i] = DeBitSet(array2b) / pow(10, PNums[i]) - Offsets[i];
```

```
                //对新产生的值进行判断，判断是否超出范围，如果超出范围则不杂交
                if (newx1[i]< Ranges[i].GetLower() || newx1[i]>Ranges[i].GetUpper() ||
                    newx2[i]<Ranges[i].GetLower() || newx2[i]>Ranges[i].GetUpper())
                    fg = false;
            }
            if (fg)
            {
                Individual newchiled1(newx1, Ranges);
                Individual newchiled2(newx2, Ranges);
                nextpopulation.push_back(newchiled1);
                nextpopulation.push_back(newchiled2);
            }
            else//将原来的个体遗传给下一代
            {
                nextpopulation.push_back(midpopulation.at(num));
                nextpopulation.push_back(midpopulation.at(num + 1));
            }
        }
        else//否则直接遗传给下一代 nextpopulation
        {
            nextpopulation.push_back(midpopulation.at(num));//生成一个新的个体并且加入到
nextpopulation 中
            nextpopulation.push_back(midpopulation.at(num + 1));
        }
        num += 2;
    }
    midpopulation.clear();//清空 midpopulation
}

//变异
void Genetic::variating()
{
    int num = 0;
    while (num < Po_Size)
    {
        double variation = Scand();//随机产生一个 0 到 1 的小数，用于判断是否进行变异
        if (variation <= Va_Probability)//如果 variation 小于变异系数，则需要进行变异
        {
```

```
            std::vector<double> x(De_Variable);
            bool fg = true;
            for (int i = 0; i < De_Variable; i++)
            {
                x[i] = nextpopulation.at(num).GetVariable()[i];
                auto arrayb = BitSet((x[i] + Offsets[i]) * pow(10, PNums[i]), Lens[i]);//编码
                int xlocal = rand() % Lens[i];
                arrayb[xlocal] = (arrayb[xlocal] + 1) % 2;//arrayb 该位取反
                x[i] = DeBitSet(arrayb) / pow(10, PNums[i]) - Offsets[i];//解码
                //判断是否符合条件
                if (x[i]< Ranges[i].GetLower() || x[i]>Ranges[i].GetUpper())
                    fg = false;
            }
            if (!fg)
                nowpopulation.push_back(nextpopulation.at(num));
            else
            {
                Individual newchiled(x, Ranges);
                nowpopulation.push_back(newchiled);
            }
        }
        else
            nowpopulation.push_back(nextpopulation.at(num));
        num++;
    }
    nextpopulation.clear();//清空 nextpopulation
}

//添加决策变量（最小值，最大值，精确到小数点后几位）
GENEVar Genetic::addVar(double _min, double _max, int _pNum)
{
    Ranges.push_back(X_Range(_min, _max));
    PNums.push_back(_pNum);
    if (_min < 0)
    {
        _max -= _min;
        Offsets.push_back(-_min);
    }
```

```
        else
            Offsets.push_back(0.0);
        _max *= pow(10, _pNum);
        Lens.push_back(int(0.5 + log(_max + 1) / log(2)));
        return GENEVar(De_Variable++, 0);
    }

    //获取当前最优变量值（第_pid 组数据）
    void Genetic::getVar(GENEVar& _var, int _pid)
    {
        _var._var = optVars[_pid][_var._idx];
    }

    //获取当前最优目标值（第_pid 组数据）
    double Genetic::getObjVar(int _pid)
    {
        return optObjValue[_pid];
    }

    //设置优化目标
    void Genetic::setObjective(std::function<double(std::vector<GENEVar>&)>& obj,
std::vector<GENEVar>& vars, int sense)
    {
        this->_objFunc = obj;
        this->_objVars = vars;
        this->_objSense = sense;
    }

    //设置参数：种群规模，进化代数，交叉概率，变异概率
    void Genetic::setPara(int tag, double a)
    {
        switch (tag)
        {
        case GENE_PSIZE:
            this->Po_Size = a;
            break;
        case GENE_EVALGEBRA:
            this->Ev_Algebra = a;
```

```
            break;
        case GENE_OVPROBABILITY:
            this->Ov_Probability = a;
            break;
        case GENE_VAPROBABILITY:
            this->Va_Probability = a;
            break;
        case GENE_OPTSIZE:
            this->optSize = int(a);
            break;
        default:
            break;
        }
    }

    //优化目标
    void Genetic::optimize()
    {
        PrintTopInfo();
        Initialize();//初始化种群,随机生成第一代个体
        //进化 Ev_Algebra 代
        std::vector<std::pair<int, double>> fs(Po_Size);//临时存放种群适应度（从大到小排序）
        auto t_optVars = optVars;
        auto t_optObjValue = optObjValue;
        int iter = 0;
        HANDLE h = GetStdHandle(STD_OUTPUT_HANDLE);
        SetConsoleTextAttribute(h, 0xf3);
        for (int k = 0; k <= Ev_Algebra; k++)
        {
            iter++;
            CaculaFitness();//适应度计算
            //更新最优值
            for (int i = 0; i < Po_Size; i++)
                fs[i] = std::make_pair(i, nowpopulation.at(i).GetFitness());
            std::sort(fs.begin(), fs.end(), [](std::pair<int, double> a, std::pair<int, double> b) {return
a.second > b.second; });
            bool fg = false;
            for (int kk = 0, i = 0, j = 0; kk < optSize; kk++)
```

```
{
    if (fs[i].second > optObjValue[j])
    {
        for (int ii = 0; ii < De_Variable; ii++)
            t_optVars[kk][ii] = nowpopulation.at(fs[i].first).GetVariable()[ii];
        t_optObjValue[kk] = fs[i].second;
        i++;
        fg = true;
    }
    else
    {
        for (int ii = 0; ii < De_Variable; ii++)
            t_optVars[kk][ii] = optVars[j][ii];
        t_optObjValue[kk] = optObjValue[j];
        j++;
        if (fs[i].second == optObjValue[j])
            i++;
    }
}
SetConsoleTextAttribute(h, 0xf2);
std::cout << "\r";
std::cout << "Evolution iter ";
SetConsoleTextAttribute(h, 0xf2);
std::cout << k + 1 << "/" << Ev_Algebra;
if (!fg)
{
    //打印进度条
    SetConsoleTextAttribute(h, 0x3f);
    int pk = 50.0 * (k + 1) / Ev_Algebra;
    for (int i = 0; i < pk; i++)
        std::wcout << "■";
    SetConsoleTextAttribute(h, 0x3f);
    for (int i = pk; i < 50; i++)
        std::wcout << " ";
    SetConsoleTextAttribute(h, 0xf1);
    std::cout << std::setprecision(2) << 100.0 * (k + 1) / Ev_Algebra << "%";
}
else
```

```
        {
            //打印新的最优值
            optVars = t_optVars;
            optObjValue = t_optObjValue;
            std::cout.setf(std::ios::fixed, std::ios::floatfield);
            SetConsoleTextAttribute(h, 0xf3);
            std::cout << " get a better
result:                                                    \n";
            for (int i = 0; i < optSize; i++)
            {
                SetConsoleTextAttribute(h, 0xf3);
                std::cout << "group";
                SetConsoleTextAttribute(h, 0xf5);
                std::cout << std::setfill('0') << std::setw(2) << i << ":  ";
                for (int j = 0; j < De_Variable; j++)
                {
                    SetConsoleTextAttribute(h, 0xf3);
                    std::cout << "x" << j << " = ";
                    SetConsoleTextAttribute(h, 0xf4);
                    std::cout << std::setprecision(6) << optVars[i][j] << "        ";
                }
                SetConsoleTextAttribute(h, 0xf3);
                if (_objSense == 0)
                {
                    std::cout << "y_max = ";
                    SetConsoleTextAttribute(h, 0xf4);
                    std::cout << std::setprecision(6) << optObjValue[i] << "\n";
                }
                else
                {
                    std::cout << "y_min = ";
                    SetConsoleTextAttribute(h, 0xf4);
                    std::cout << std::setprecision(6) << -optObjValue[i] << "\n";
                }
            }
            std::cout << "\n";
        }
        if (k == Ev_Algebra)break;
```

```
        CaculaReFitness();//适应度概率计算
        CalculaSumFitness();//计算累加个体概率
        seclect();//选择
        crossing();//杂交
        variating();//变异
        if (_kbhit())
        {
            switch (tolower(_getch()))
            {
            case 'p':
                k = Ev_Algebra;
                break;
            default:
                break;
            }
        }
    }
    //进化 Ev_Algebra 代之后输出
    /*SetConsoleTextAttribute(h, 0xf3);
    for (int i = 0; i < De_Variable; i++)
        std::cout << "x" << i << std::setw(10);
    std::cout << std::setw(5) << "Fitness\n";
    SetConsoleTextAttribute(h, 0xf5);
    for (int i = 0; i < Po_Size; i++)
    {
        for (int j = 0; j < De_Variable; j++)
            std::cout << nowpopulation.at(i).GetVariable()[j] << std::setw(10);
        std::cout << "\n";
    }*/
    SetConsoleTextAttribute(h, 0xf3);
    std::cout << "\ntotall iteration: " << iter << "\n";
    std::cout << "the optimal solution：\n";
    for (int i = 0; i < optSize; i++)
    {
        SetConsoleTextAttribute(h, 0xf3);
        std::cout << "group";
        SetConsoleTextAttribute(h, 0xf5);
        std::cout << std::setfill('0') << std::setw(2) << i << ":    ";
```

```
        for (int j = 0; j < De_Variable; j++)
        {
            SetConsoleTextAttribute(h, 0xf3);
            std::cout << "x" << j << " = ";
            SetConsoleTextAttribute(h, 0xf4);
            std::cout << std::setprecision(6) << optVars[i][j] << "        ";
        }
        SetConsoleTextAttribute(h, 0xf3);
        if (_objSense == 0)
        {
            std::cout << "y_max = ";
            SetConsoleTextAttribute(h, 0xf4);
            std::cout << std::setprecision(6) << optObjValue[i] << "\n";
        }
        else
        {
            optObjValue[i] *= -1;
            std::cout << "y_min = ";
            SetConsoleTextAttribute(h, 0xf4);
            std::cout << std::setprecision(6) << optObjValue[i] << "\n";
        }
    }
    std::cout << "\n";
}

void Genetic::PrintTopInfo()
{
    HANDLE h = GetStdHandle(STD_OUTPUT_HANDLE);
    SetConsoleTextAttribute(h, 0xf1);
    std::cout << "----------------------------Genetic Algorithm created by Qiu----------------------------\n";
    SetConsoleTextAttribute(h, 0xf3);
    std::cout << "Num of variables: ";
    SetConsoleTextAttribute(h, 0xf5);
    std::cout << De_Variable << "\n";
    SetConsoleTextAttribute(h, 0xf3);
    std::cout << "Population size: ";
    SetConsoleTextAttribute(h, 0xf5);
    std::cout << Po_Size << "\n";
```

```
        SetConsoleTextAttribute(h, 0xf3);
        std::cout << "Algebra of evolution: ";
        SetConsoleTextAttribute(h, 0xf5);
        std::cout << Ev_Algebra << "\n";
        SetConsoleTextAttribute(h, 0xf3);
        std::cout << "Crossover probability: ";
        SetConsoleTextAttribute(h, 0xf5);
        std::cout << Ov_Probability << "\n";
        SetConsoleTextAttribute(h, 0xf3);
        std::cout << "Mutation probability: ";
        SetConsoleTextAttribute(h, 0xf5);
        std::cout << Va_Probability << "\n";
        SetConsoleTextAttribute(h, 0xf3);
        std::cout << "Opt_obj_group_size: ";
        SetConsoleTextAttribute(h, 0xf5);
        std::cout << optSize << "\n";
        SetConsoleTextAttribute(h, 0xf3);
        std::cout << "Optimize type: ";
        SetConsoleTextAttribute(h, 0xf5);
        if (_objSense == 0)std::cout << "maximum\n";
        else std::cout << "minimum\n";
        SetConsoleTextAttribute(h, 0xf2);
        std::cout << "do optimizing\n";
        std::cout << "--------------------------------------------------------------------------------\n";
        std::cout << "--------------------------------------------------------------------------------\n";
    }
}
```


#3 模型求解主函数 main.cpp

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <vector>
#include <string>
#include <algorithm>
#include <functional>
#include <math.h>
```

```
#include<time.h>
#include <unordered_map>
#include<iomanip>
#include <complex>
#include <Eigen/Dense>
#include <gurobi_c++.h>
#include"GeneticAlgorithm.h"

#define PN 1000 //种群规模
#define EN 2000 //进化代数
int G_TIME = 3 * 60; //Gurobi time

//误差计算
double calcRMSE(Eigen::MatrixXcd& F, std::vector<Eigen::MatrixXcd>& A, double beta)
{
    Eigen::MatrixXcd B = Eigen::MatrixXcd::Identity(F.rows(), F.cols());
    for (size_t i = 0; i < A.size(); i++)
        B *= A[i];
    return (F - beta * B).norm() / F.rows();
}

//复杂度计算
int calcComplexMultipleNumber(std::complex<double> beta, std::vector<Eigen::MatrixXcd>
matrixes)
{
    auto a = beta.real();
    auto b = beta.imag();
    std::complex<double> betaInverse(a / (a * a - b * b), b / (a * a - b * b));

    auto isComplexMultiple = [](std::complex<double> c1, std::complex<double> c2) {
        auto flag = [](std::complex<double> c) {
            auto absr = std::abs(c.real());
            auto absi = std::abs(c.imag());
            if ((absr == 0 && absi == 0) || (absr == 1 && absi == 0) || (absr == 0 && absi == 1) || (absr
== 1 && absi == 1))
                return true;
            return false;
        };
```

```
        if (flag(c1) || flag(c2))
            return 0;
        return 1;
    };

    int ret = 0;
    for (int i = 0; i < matrixes[0].rows(); i++)
    {
        for (int j = 0; j < matrixes[0].cols(); j++)
        {
            ret += isComplexMultiple(betaInverse, matrixes[0](i, j));
            matrixes[0](i, j) *= betaInverse;
        }
    }
    auto midMatrix = matrixes[0];
    for (int k = 1; k < matrixes.size(); k++)
    {
        for (int i = 0; i < matrixes[0].rows(); i++)
        {
            for (int j = 0; j < matrixes[k].cols(); j++)
            {
                std::complex<double> comSum(0, 0);
                for (int l = 0; l < matrixes[0].rows(); l++)
                {
                    ret += isComplexMultiple(matrixes[0](i, l), matrixes[k](l, j));
                    comSum += (matrixes[0](i, l) * matrixes[k](l, j));
                }
                midMatrix(i, j) = comSum;
            }
        }
        matrixes[0] = midMatrix;
    }

    return ret;
}

Eigen::MatrixXcd calcKroneckerProduct(int depth, Eigen::MatrixXcd mat2)
```

```
{
    int N = std::pow(2, depth - 1);
    Eigen::MatrixXi diaMat(N, N);
    diaMat.setZero();
    for (int i = 0; i < N; i++)
    {
        diaMat(i, i) = 1;
    }

    // 计算张量积
    Eigen::MatrixXcd ret(diaMat.rows() * mat2.rows(), diaMat.cols() * mat2.cols());

    for (int i = 0; i < diaMat.rows(); ++i) {
        for (int j = 0; j < diaMat.cols(); ++j) {
            ret.block(i * mat2.rows(), j * mat2.cols(), mat2.rows(), mat2.cols()) = diaMat(i, j) * mat2;
        }
    }

    return ret;
}

void recursionCalcMatrix(int N, int& depth, std::vector<Eigen::MatrixXcd>& retMats,
std::complex<double> w)
{
    if (N == 2)
    {
        Eigen::MatrixXcd mat22(2, 2);
        mat22 << std::complex<double>(1, 0), std::complex<double>(1, 0),
            std::complex<double>(1, 0), std::complex<double>(-1, 0);
        auto finalMat = calcKroneckerProduct(depth, mat22);
        retMats.push_back(finalMat);
        return;
    }

    Eigen::MatrixXcd mat1(N / 2, N / 2);
    mat1.setZero();
    Eigen::MatrixXcd mat2(N / 2, N / 2);
    mat2.setZero();
```

```
    Eigen::MatrixXcd mat3(N / 2, N / 2);
    mat3.setZero();
    for (int i = 0; i < N / 2; i++)
    {
        mat1(i, i) = std::complex<double>(1, 0);
        mat2(i, i) = std::pow(w, std::pow(2, depth - 1) * i);
        mat3(i, i) = (-1.0) * std::pow(w, std::pow(2, depth - 1) * i);
    }

    Eigen::MatrixXcd totalMat(N, N);
    totalMat << mat1, mat2,
        mat1, mat3;
    auto resultMat = calcKroneckerProduct(depth, totalMat);

    retMats.push_back(resultMat);

    recursionCalcMatrix(N / 2, ++depth, retMats, w);
}

// 计算张量积
Eigen::MatrixXcd calcuKn(Eigen::MatrixXcd& A, Eigen::MatrixXcd& B)
{
    Eigen::MatrixXcd ret(A.rows() * B.rows(), A.cols() * B.cols());
    for (int i = 0; i < A.rows(); ++i) {
        for (int j = 0; j < A.cols(); ++j) {
            ret.block(i * B.rows(), j * B.cols(), B.rows(), B.cols()) = A(i, j) * B;
        }
    }
    return ret;
}

void Solve1(int t)
{
    int N = std::pow(2, t);//矩阵的维度
    std::complex<double> w(std::cos(-2 * PI / N), std::sin(-2 * PI / N));
    std::complex<double> beta(1, 0);
    std::vector<Eigen::MatrixXcd> retMats;
    int depth = 1;
```

```
recursionCalcMatrix(N, depth, retMats, w);

Eigen::MatrixXcd rMat(N, N);
rMat.setZero();
for (int i = 0; i < N; i++)
{
    std::string binaryStr = "";
    auto t = i;
    while (t > 0) {
        binaryStr = std::to_string(t % 2) + binaryStr;
        t /= 2;
    }
    std::string ts(std::log2(N) - binaryStr.size(), '0');
    binaryStr = ts + binaryStr;
    std::reverse(binaryStr.begin(), binaryStr.end());
    int ret = 0;
    for (int j = binaryStr.size() - 1; j >= 0; j--)
    {
        ret += (binaryStr[j] - '0') * std::pow(2, binaryStr.size() - 1 - j);
    }
    rMat(i, ret) = std::complex<double>(1, 0);
}
retMats.push_back(rMat);

std::string name = "mats.txt";
std::ofstream ofs(name, std::ios::out);

for (int i = 0; i < retMats.size(); i++)
{
    //std::cout << retMats[i] << std::endl;
    ofs << retMats[i] << std::endl;
    ofs << std::endl;
    ofs << std::endl;
    ofs << std::endl;
}
ofs.close();

std::cout << "cmp : " << calcComplexMultipleNumber(beta, retMats) << std::endl;;
```

```
    for (int i = 1; i < retMats.size(); i++)
    {
        retMats[0] = retMats[0] * retMats[i];
    }

    //输出误差
    Eigen::MatrixXcd FN(N, N);
    for (size_t i = 0; i < N; i++)
        for (size_t j = 0; j < N; j++)
            FN(i, j) = pow(w, i * j);
    std::cout << "RMSE : " << (retMats[0] - FN).norm() << std::endl;

    //std::cout << FN << std::endl;
}

void Solve2()
{
    int q = 3;
    int q_a = 2;
    int t = 4;
    int N = pow(2, t);//矩阵维度
    double beta = 0, rmse = 0, cmp = 0, td = 0;//缩放因子，误差，复杂度，正交性误差
    std::vector<std::pair<double, double>> an(N / 4);
    Eigen::MatrixXcd F(N, N); //F 矩阵
    Eigen::MatrixXcd A(N, N); //F 的近似有理矩阵
    //初始化 F 矩阵
    std::complex<double> w(cos(2 * PI / N), -sin(2 * PI / N));
    for (size_t i = 0; i < N; i++)
        for (size_t j = 0; j < N; j++)
            F(i, j) = pow(w, i * j);
    //F /= sqrt(N);

    // Gurobi 求解
    GRBEnv* env = 0;
    try
    {
```

```
        env = new GRBEnv();
        GRBModel model = GRBModel(*env);

        //create variables
        GRBVar g_beta = model.addVar(1, pow(2, q - 1), 1, GRB_CONTINUOUS); //beta
        GRBVar g_beta_X = model.addVar(0, q - 1, 1, GRB_INTEGER); //beta  指数
        std::vector<std::pair<GRBVar, GRBVar>> g_a(N / 4); //  参数向量
        std::vector<std::pair<GRBVar, GRBVar>> g_b(N / 4); // g_a 的指数
        std::vector<std::pair<GRBVar, GRBVar>> g_c(N / 4); // g_a 的系数
        std::vector<std::vector<std::pair<GRBVar, GRBVar>>>g_A(N,
std::vector<std::pair<GRBVar, GRBVar>>(N));//F 的近似有理矩阵
        for (size_t i = 0; i < g_a.size(); i++)
        {
            g_a[i].first = model.addVar(-pow(2, q_a - 1), pow(2, q_a - 1), 1, GRB_CONTINUOUS);
            g_a[i].second = model.addVar(-pow(2, q_a - 1), pow(2, q_a - 1), 1,
GRB_CONTINUOUS);
            g_b[i].first = model.addVar(-1, q_a - 1, 1, GRB_INTEGER);
            g_b[i].second = model.addVar(-1, q_a - 1, 1, GRB_INTEGER);
            g_c[i].first = model.addVar(-1, 1, 1, GRB_INTEGER);
            g_c[i].second = model.addVar(-1, 1, 1, GRB_INTEGER);
        }
        for (size_t i = 0; i < N; i++)
        {
            for (size_t j = 0; j < N; j++)
            {
                g_A[i][j].first = model.addVar(-pow(2, q - 1), pow(2, q - 1), 1, GRB_INTEGER);
                g_A[i][j].second = model.addVar(-pow(2, q - 1), pow(2, q - 1), 1, GRB_INTEGER);
            }
        }

        /**************build problem**************/
        // a0 = 1
        model.addConstr(g_a[0].first == 1);
        model.addConstr(g_a[0].second == 0);
        model.addGenConstrExpA(g_beta_X, g_beta, 2);
        // a = 0 or +/-2^n, 0<=n<=q-1
        for (size_t i = 1; i < N / 4; i++)
        {
```

```
            std::pair<GRBVar, GRBVar> a;
            a.first = model.addVar(-GRB_INFINITY, GRB_INFINITY, 1, GRB_CONTINUOUS);
            a.second = model.addVar(-GRB_INFINITY, GRB_INFINITY, 1, GRB_CONTINUOUS);
            model.addGenConstrExpA(g_b[i].first, a.first, 2);
            model.addGenConstrExpA(g_b[i].second, a.second, 2);
            model.addQConstr(g_a[i].first == g_c[i].first * a.first);
            model.addQConstr(g_a[i].second == g_c[i].second * a.second);
        }
        // build A/2
        for (size_t i = 0; i < N / 4; i++)
        {
            for (size_t j = 0; j < N / 4; j++)
            {
                // A/2(0 0)
                GRBQuadExpr a = 1, b = 0;
                for (size_t k = 0; k < i; k++)
                {
                    GRBVar e1 = model.addVar(-GRB_INFINITY, GRB_INFINITY, 1,
GRB_CONTINUOUS);
                    GRBVar e2 = model.addVar(-GRB_INFINITY, GRB_INFINITY, 1,
GRB_CONTINUOUS);
                    model.addQConstr(e1 == a);
                    model.addQConstr(e2 == b);
                    a = e1 * g_a[j].first - e2 * g_a[j].second;
                    b = e1 * g_a[j].second + e2 * g_a[j].first;
                }
                GRBVar va = model.addVar(-GRB_INFINITY, GRB_INFINITY, 1,
GRB_CONTINUOUS);
                GRBVar vb = model.addVar(-GRB_INFINITY, GRB_INFINITY, 1,
GRB_CONTINUOUS);
                model.addQConstr(va == a);
                model.addQConstr(vb == b);
                model.addQConstr(g_A[i][j].first == g_beta * va);
                model.addQConstr(g_A[i][j].second == g_beta * vb);
                //A / 2(1 0)
                GRBLinExpr e1, e2;
                if (j % 2 == 0)
                {
```

```
            e1 = pow(-1, j / 2) * g_A[i][j].first;
            e2 = pow(-1, j / 2) * g_A[i][j].second;
        }
        else
        {
            e1 = pow(-1, (j - 1) / 2) * g_A[i][j].second;
            e2 = -pow(-1, (j - 1) / 2) * g_A[i][j].first;
        }
        model.addConstr(g_A[i + N / 4][j].first == e1);
        model.addConstr(g_A[i + N / 4][j].second == e2);
        // A/2(0 1)
        if (i % 2 == 0)
        {
            e1 = pow(-1, i / 2) * g_A[i][j].first;
            e2 = pow(-1, i / 2) * g_A[i][j].second;
        }
        else
        {
            e1 = pow(-1, (i - 1) / 2) * g_A[i][j].second;
            e2 = -pow(-1, (i - 1) / 2) * g_A[i][j].first;
        }
        model.addConstr(g_A[i][j + N / 4].first == e1);
        model.addConstr(g_A[i][j + N / 4].second == e2);
        // A/2(1 1)
        int k = i + j + N / 4;
        if (k % 2 == 0)
        {
            e1 = pow(-1, k / 2) * g_A[i][j].first;
            e2 = pow(-1, k / 2) * g_A[i][j].second;
        }
        else
        {
            e1 = pow(-1, (k - 1) / 2) * g_A[i][j].second;
            e2 = -pow(-1, (k - 1) / 2) * g_A[i][j].first;
        }
        model.addConstr(g_A[i + N / 4][j + N / 4].first == e1);
        model.addConstr(g_A[i + N / 4][j + N / 4].second == e2);
    }
```

```
        }
        // build A
        for (size_t i = 0; i < N / 2; i++)
        {
            for (size_t j = 0; j < N / 2; j++)
            {
                // A(1 0)
                model.addConstr(g_A[i + N / 2][j].first == pow(-1, j) * g_A[i][j].first);
                model.addConstr(g_A[i + N / 2][j].second == pow(-1, j) * g_A[i][j].second);
                // A(0 1)
                model.addConstr(g_A[i][j + N / 2].first == pow(-1, i) * g_A[i][j].first);
                model.addConstr(g_A[i][j + N / 2].second == pow(-1, i) * g_A[i][j].second);
                // A(1 1)
                model.addConstr(g_A[i + N / 2][j + N / 2].first == pow(-1, i + j + N / 2) * g_A[i][j].first);
                model.addConstr(g_A[i + N / 2][j + N / 2].second == pow(-1, i + j + N / 2) *
g_A[i][j].second);
            }
        }

        //for (int i = 0; i < N; i++)
        //{
        //  for (int j = 0; j < N; j++)
        //  {
        //      /*if (i * j == 0)
        //      {
        //          model.addConstr(g_A[i][j].first == 1);
        //          model.addConstr(g_A[i][j].second == 0);
        //          continue;
        //      }*/
        //      int _p = j * (i / (N / 2)) + i * (j / (N / 2));
        //      int _t = j * (i / (N / 4)) + i * (j / (N / 4));
        //      int idx = j % (N / 4);
        //      int m = i % (N / 4);
        //      GRBQuadExpr a = 1, b = 1;
        //      for (size_t k = 0; k < m; k++)
        //      {
        //          GRBVar e1 = model.addVar(-GRB_INFINITY, GRB_INFINITY, 1, GRB_INTEGER);
        //          GRBVar e2 = model.addVar(-GRB_INFINITY, GRB_INFINITY, 1, GRB_INTEGER);
```

```
        //            model.addQConstr(e1 == a);
        //            model.addQConstr(e2 == b);
        //            a = e1 * g_a[idx].first - e2 * g_a[idx].second;
        //            b = e1 * g_a[idx].second + e2 * g_a[idx].first;
        //        }
        //        if (m == 0)
        //            b = 0;
        //        if (_t % 2 == 0)
        //        {
        //            model.addQConstr(g_A[i][j].first == pow(-1, _p + _t) * pow(-1, _t / 2) * a);
        //            model.addQConstr(g_A[i][j].second == pow(-1, _p + _t) * pow(-1, _t / 2) * b);
        //        }
        //        else
        //        {
        //            model.addQConstr(g_A[i][j].first == pow(-1, _p + _t) * pow(-1, (_t + 1) / 2) * b);
        //            model.addQConstr(g_A[i][j].second == pow(-1, _p + _t) * pow(-1, (_t - 1) / 2) * a);
        //        }
        //    }
        //}

        //计算 F 范数
        GRBQuadExpr Ef = 0; // rmse = sqrt(Ef)/N
        std::vector<std::vector<std::pair<GRBLinExpr, GRBLinExpr>>> FA(N,
std::vector<std::pair<GRBLinExpr, GRBLinExpr>>(N));
        for (size_t i = 0; i < N; i++)
        {
            for (size_t j = 0; j < N; j++)
            {
                GRBVar a = model.addVar(-GRB_INFINITY, GRB_INFINITY, 1,
GRB_CONTINUOUS);
                GRBVar b = model.addVar(-GRB_INFINITY, GRB_INFINITY, 1,
GRB_CONTINUOUS);
                model.addQConstr(a * g_beta == g_A[i][j].first);
                model.addQConstr(b * g_beta == g_A[i][j].second);
                FA[i][j] = { F(i, j).real() - a,F(i,j).imag() - b };
                //FA[i][j] = { F(i, j).real() * g_beta - g_A[i][j].first,F(i,j).imag() * g_beta - g_A[i][j].second };
            }
        }
```

```
        for (size_t i = 0; i < N; i++)
            for (size_t j = 0; j < N; j++)
                Ef += FA[i][j].first * FA[i][j].first + FA[i][j].second * FA[i][j].second;

        //set constrains
        // 近似正交性约束
        std::vector<std::vector<std::pair<GRBVar, GRBVar>>> FN(N,
std::vector<std::pair<GRBVar, GRBVar>>(N));
        std::vector<std::vector<std::pair<GRBVar, GRBVar>>>g_AT(N,
std::vector<std::pair<GRBVar, GRBVar>>(N));
        for (size_t i = 0; i < N; i++)
        {
            for (size_t j = 0; j < N; j++)
            {
                g_AT[i][j].first = model.addVar(-pow(2, q - 1), pow(2, q - 1), 1, GRB_INTEGER);
                g_AT[i][j].second = model.addVar(-pow(2, q - 1), pow(2, q - 1), 1, GRB_INTEGER);
                model.addConstr(g_AT[i][j].first == g_A[j][i].first);
                model.addConstr(g_AT[i][j].second == -g_A[j][i].second);
            }
        }
        for (size_t i = 0; i < N; i++)
        {
            for (size_t j = 0; j < N; j++)
            {
                GRBQuadExpr e1 = 0, e2 = 0;
                for (size_t k = 0; k < N; k++)
                {
                    e1 += g_A[i][k].first * g_AT[k][j].first - g_A[i][k].second * g_AT[k][j].second;
                    e2 += g_A[i][k].first * g_AT[k][j].second + g_A[i][k].second * g_AT[k][j].first;
                }
                FN[i][j].first = model.addVar(-GRB_INFINITY, GRB_INFINITY, 1, GRB_INTEGER);
                FN[i][j].second = model.addVar(-GRB_INFINITY, GRB_INFINITY, 1, GRB_INTEGER);
                model.addQConstr(FN[i][j].first == e1);
                model.addQConstr(FN[i][j].second == e2);
            }
        }
        GRBQuadExpr diagF = 0, fnF = 0;
        for (size_t i = 0; i < N; i++)
```

```
{
    diagF += FN[i][i].first * FN[i][i].first + FN[i][i].second * FN[i][i].second;
    for (size_t j = 0; j < N; j++)
        fnF += FN[i][j].first * FN[i][j].first + FN[i][j].second * FN[i][j].second;
}
GRBVar d = model.addVar(0, GRB_INFINITY, 1, GRB_CONTINUOUS);
GRBVar dfnF = model.addVar(0, GRB_INFINITY, 1, GRB_CONTINUOUS);
GRBVar ed = model.addVar(0, GRB_INFINITY, 1, GRB_CONTINUOUS);
model.addQConstr(dfnF == fnF);
model.addQConstr(d * dfnF == diagF);
model.addQConstr(ed * ed == d);
model.addConstr(1 - ed <= 0.2);

//set optimize object : min { Ef }
model.setObjective(Ef, GRB_MINIMIZE);
model.set(GRB_DoubleParam_TimeLimit, G_TIME);
model.set(GRB_DoubleParam_MIPGap, 1e-14);
model.set(GRB_IntParam_NonConvex, 2);
model.optimize();
//get results
beta = g_beta.get(GRB_DoubleAttr_X);
rmse = sqrt(Ef.getValue()) / N;
td = 1 - ed.get(GRB_DoubleAttr_X);
for (size_t i = 0; i < N*(4+1); i++)
{
    an[i].first = g_a[i].first.get(GRB_DoubleAttr_X);
    an[i].second = g_a[i].second.get(GRB_DoubleAttr_X);
}
for (size_t i = 0; i < N; i++)
{
    for (size_t j = 0; j < N; j++)
    {
        double a = g_A[i][j].first.get(GRB_DoubleAttr_X);
        double b = g_A[i][j].second.get(GRB_DoubleAttr_X);
        A(i, j) = std::complex<double>(a, b);
    }
}
cmp = q * calcComplexMultipleNumber(std::complex<double>(1, 0), { A });
```

```
}
catch (GRBException e)
{
    std::cout << " Error code = " << e.getErrorCode() << std::endl;
    std::cout << e.getMessage() << std::endl;
}
catch (...) {
    std::cout << " Exception during optimization " << std::endl;
}
delete env;

//output
std::cout << "N = " << N << "\n";
std::cout << "K = " << 1 << "\n";
std::cout << "beta = 1/" << beta << "\n";
std::cout << "C = " << cmp << "\n";
std::cout << "RMSE = " << rmse << "\n";
std::cout << "td = " << td << "\n";
std::cout << std::fixed << std::setprecision(3);
std::cout << "F = \n";
for (size_t i = 0; i < N; i++)
{
    for (size_t j = 0; j < N; j++)
    {
        double a = F(i, j).real();
        double b = F(i, j).imag();
        std::cout << std::left << "(" << a << ", " << b << ")   ";
    }
    std::cout << "\n";
}
std::cout << "\n";
std::cout << std::fixed << std::setprecision(3);
std::cout << "a = [";
for (size_t i = 0; i < N / 4; i++)
{
    std::cout << "(" << an[i].first << ", " << an[i].second << ")";
    if (i < N / 4 - 1)
        std::cout << "   ";
```

```
}
std::cout << "]\n";
std::cout << "\n";
std::cout << "A = \n";
for (size_t i = 0; i < N; i++)
{
    for (size_t j = 0; j < N; j++)
    {
        double a = A(i, j).real();
        double b = A(i, j).imag();
        std::cout << std::left << "(" << a << ", " << b << ")   ";
    }
    std::cout << "\n";
}
//write
std::string path = "result2.txt";
std::ofstream outputFile(path);
if (outputFile.is_open())
{
    outputFile << "Gurobi result2\n";
    outputFile << "N = " << N << "\n";
    outputFile << "K = " << 1 << "\n";
    outputFile << "beta = 1/" << beta << "\n";
    outputFile << "C = " << cmp << "\n";
    outputFile << "RMSE = " << rmse << "\n";
    outputFile << "td = " << td << "\n";
    outputFile << std::fixed << std::setprecision(3);
    outputFile << "F = \n";
    for (size_t i = 0; i < N; i++)
    {
        for (size_t j = 0; j < N; j++)
        {
            double a = F(i, j).real();
            double b = F(i, j).imag();
            outputFile << std::left << "(" << a << ", " << b << ")   ";
        }
        outputFile << "\n";
    }
```

```
        outputFile << "\n";
        std::cout << std::fixed << std::setprecision(3);
        outputFile << "a = [";
        for (size_t i = 0; i < N / 4; i++)
        {
            outputFile << "(" << an[i].first << ", " << an[i].second << ")";
            if (i < N / 4 - 1)
                outputFile << "   ";
        }
        outputFile << "]\n";
        outputFile << "\n";
        outputFile << "A = \n";
        for (size_t i = 0; i < N; i++)
        {
            for (size_t j = 0; j < N; j++)
            {
                double a = A(i, j).real();
                double b = A(i, j).imag();
                outputFile << std::left << "(" << a << "," << b << ")   ";
            }
            outputFile << "\n";
        }
        std::cout << "write success.\n";
    }
    else
        std::cout << "write failed.\n";
}

void Solve3()
{
    int t = 4;
    int q = 3;
    int K = t + 1;//A 矩阵个数
    int N = pow(2, t);//矩阵维度
    double beta, rmse, cmp;//缩放因子，误差，复杂度
    Eigen::MatrixXcd F(N, N); //F 矩阵
    std::vector<Eigen::MatrixXcd> A(K, Eigen::MatrixXcd(N, N)); //分解后的一系列 A 矩阵
```

```
    std::vector<std::vector<int>>B(K, std::vector<int>(N)); //表示 A 矩阵的每行除对角线外另一个非零元素的列索引（若等于行索引说明该行只有主对角线元素非零）
    //初始化 F 矩阵
    std::complex<double> w(cos(2 * PI / N), -sin(2 * PI / N));
    for (size_t i = 0; i < N; i++)
        for (size_t j = 0; j < N; j++)
            F(i, j) = pow(w, i * j);
    F /= sqrt(N);
    // 遗传算法求解
    GENE::Genetic* model = new GENE::Genetic();
    //vars: 先存 beta，再存 B，然后按行存每个 A 矩阵的元素（实部 虚部）
    std::vector<GENE::GENEVar> vars;
    vars.push_back(model->addVar(1, 1, 0)); //beta
    for (size_t i = 0; i < K; i++) //B
        for (size_t j = 0; j < N; j++)
            vars.push_back(model->addVar(0, N - 1, 0));
    for (size_t i = 0; i < K; i++) //A
        for (size_t r = 0; r < N; r++)
            for (size_t c = 0; c < N; c++)
            {
                vars.push_back(model->addVar(-pow(2, q - 1), pow(2, q - 1), 1)); //实部
                vars.push_back(model->addVar(-pow(2, q - 1), pow(2, q - 1), 1)); //虚部
            }
    //能量函数
    std::function<double(std::vector<GENE::GENEVar>&)> objF =
        [=](std::vector<GENE::GENEVar>& _vars)
    {
        std::vector<Eigen::MatrixXcd> _A(K, Eigen::MatrixXcd(N, N));
        std::vector<std::vector<int>> _B(K, std::vector<int>(N));
        int k = 0;
        double _beta = _vars[k++]._var;
        for (size_t i = 0; i < K; i++)
            for (size_t j = 0; j < N; j++)
                _B[i][j] = _vars[k++]._var;
        for (size_t i = 0; i < K; i++)
            for (size_t r = 0; r < N; r++)
                for (size_t c = 0; c < N; c++, k += 2)
                    if (_B[i][r] == c || r == c)
```

```
                    _A[i](r, c) = std::complex<double>(_vars[k]._var, _vars[k + 1]._var);
                else
                    _A[i](r, c) = 0;
    Eigen::MatrixXcd _F = F * _beta;
    for (size_t i = 1; i < K; i++)
        _A[i] = _A[i - 1] * _A[i];
    _F = _F - _A[K - 1];
    double _rmse = sqrt(pow(_F.norm(), 2)) / N;
    return _rmse;
};
//set parameters
model->setPara(GENE_PSIZE, PN); //种群规模
model->setPara(GENE_EVALGEBRA, EN); //进化代数
model->setPara(GENE_OVPROBABILITY, 0.950); //交叉概率
model->setPara(GENE_VAPROBABILITY, 0.050); //变异概率
model->setPara(GENE_OPTSIZE, 1); //目标数据组数
//set object
model->setObjective(objF, vars, GENE_MINIMIZE);
//optimize
model->optimize();
//get result
for (int i = 0; i < vars.size(); i++)
    model->getVar(vars[i]);
int k = 0;
beta = vars[k++]._var;
for (size_t i = 0; i < K; i++)
    for (size_t j = 0; j < N; j++)
        B[i][j] = vars[k++]._var;
for (size_t i = 0; i < K; i++)
    for (size_t r = 0; r < N; r++)
        for (size_t c = 0; c < N; c++, k += 2)
            if (B[i][r] == c || r == c)
                A[i](r, c) = std::complex<double>(vars[k]._var, vars[k + 1]._var);
            else
                A[i](r, c) = 0;
Eigen::MatrixXcd resA = Eigen::MatrixXcd::Identity(N, N);
for (size_t i = 0; i < K; i++)
    resA *= A[i];
```

```
rmse = model->getObjVar();
cmp = q * calcComplexMultipleNumber(std::complex<double>(beta, 0), A);
delete model;

//output
std::cout << "N = " << N << "\n";
std::cout << "K = " << K << "\n";
std::cout << "beta = 1/" << beta << "\n";
std::cout << "C = " << cmp << "\n";
std::cout << "RMSE = " << rmse << "\n";
std::cout << std::fixed << std::setprecision(3);
std::cout << "F = \n";
for (size_t i = 0; i < N; i++)
{
    for (size_t j = 0; j < N; j++)
    {
        double a = F(i, j).real();
        double b = F(i, j).imag();
        std::cout << std::left << "(" << a << "," << b << ")   ";
    }
    std::cout << "\n";
}
std::cout << "\n";
std::cout << std::fixed << std::setprecision(0);
std::cout << "F^ = \n";
for (size_t i = 0; i < N; i++)
{
    for (size_t j = 0; j < N; j++)
    {
        double a = resA(i, j).real();
        double b = resA(i, j).imag();
        std::cout << std::left << "(" << a << ", " << b << ")   ";
    }
    std::cout << "\n";
}
std::cout << "\n";
for (size_t k = 0; k < K; k++)
{
```

```
        std::cout << "A" << k << " = \n";
        for (size_t i = 0; i < N; i++)
        {
            for (size_t j = 0; j < N; j++)
            {
                double a = A[k](i, j).real();
                double b = A[k](i, j).imag();
                std::cout << std::left << "(" << a << ", " << b << ")   ";
            }
            std::cout << "\n";
        }
        std::cout << "\n";
    }
    //write
    std::string path = "result3.txt";
    std::ofstream outputFile(path);
    if (outputFile.is_open())
    {
        outputFile << "GENE result3\n";
        outputFile << "N = " << N << "\n";
        outputFile << "K = " << K << "\n";
        outputFile << "beta = 1/" << beta << "\n";
        outputFile << "C = " << cmp << "\n";
        outputFile << "RMSE = " << rmse << "\n";
        outputFile << std::fixed << std::setprecision(3);
        outputFile << "F = \n";
        for (size_t i = 0; i < N; i++)
        {
            for (size_t j = 0; j < N; j++)
            {
                double a = F(i, j).real();
                double b = F(i, j).imag();
                outputFile << std::left << "(" << a << ", " << b << ")   ";
            }
            outputFile << "\n";
        }
        outputFile << "\n";
        outputFile << std::fixed << std::setprecision(0);
```

```
        outputFile << "F^ = \n";
        for (size_t i = 0; i < N; i++)
        {
            for (size_t j = 0; j < N; j++)
            {
                double a = resA(i, j).real();
                double b = resA(i, j).imag();
                outputFile << std::left << "(" << a << ", " << b << ")   ";
            }
            outputFile << "\n";
        }
        outputFile << "\n";
        for (size_t k = 0; k < K; k++)
        {
            outputFile << "A " << k << " = \n";
            for (size_t i = 0; i < N; i++)
            {
                for (size_t j = 0; j < N; j++)
                {
                    double a = A[k](i, j).real();
                    double b = A[k](i, j).imag();
                    outputFile << std::left << "(" << a << ", " << b << ")   ";
                }
                outputFile << "\n";
            }
            outputFile << "\n";
        }
        std::cout << "write success.\n";
    }
    else
        std::cout << "write failed.\n";
}

void Solve3_2()
{
    int t = 4;
    int q = 3;
    int K = t + 1;//A 矩阵个数
```

```
int N = pow(2, t);//矩阵维度
double beta, rmse, cmp;//缩放因子，误差，复杂度
Eigen::MatrixXcd F(N, N); //F 矩阵
std::vector<Eigen::MatrixXcd> A; //分解后的一系列 A 矩阵
//初始化 F 矩阵
std::complex<double> w(cos(2 * PI / N), -sin(2 * PI / N));
for (size_t i = 0; i < N; i++)
    for (size_t j = 0; j < N; j++)
        F(i, j) = pow(w, i * j);
//F /= sqrt(N);

int dp = 1;
recursionCalcMatrix(N, dp, A, w);
Eigen::MatrixXcd rMat(N, N);
rMat.setZero();
for (int i = 0; i < N; i++)
{
    std::string binaryStr = "";
    auto t = i;
    while (t > 0) {
        binaryStr = std::to_string(t % 2) + binaryStr;
        t /= 2;
    }
    std::string ts(std::log2(N) - binaryStr.size(), '0');
    binaryStr = ts + binaryStr;
    std::reverse(binaryStr.begin(), binaryStr.end());
    int ret = 0;
    for (int j = binaryStr.size() - 1; j >= 0; j--)
    {
        ret += (binaryStr[j] - '0') * std::pow(2, binaryStr.size() - 1 - j);
    }
    rMat(i, ret) = std::complex<double>(1, 0);
}
A.push_back(rMat);

std::vector<std::vector<std::vector<int>>> all_w_idx(N);
for (int it = 0; it < N; it++)
{
```

```
        if (it == 0) continue;
        auto tmpw = std::pow(w, it);
        for (int k = 0; k < A.size(); k++)
        {
            for (int i = 0; i < N; i++)
            {
                for (int j = 0; j < N; j++)
                {
                    if (tmpw == A[k](i, j))
                    {
                        all_w_idx[it].push_back({ k,i,j,1 });
                        std::cout << it << " : " << k << " " << i << " " << j << " " << 1 << std::endl;
                    }
                    else if (-tmpw == A[k](i, j))
                    {
                        all_w_idx[it].push_back({ k,i,j ,-1 });
                        std::cout << it << " : " << k << " " << i << " " << j << " " << -1 << std::endl;
                    }
                }
            }
        }
    }

    // Gurobi 求解
    GRBEnv* env = 0;
    try
    {
        env = new GRBEnv();
        GRBModel model = GRBModel(*env);

        //create variables
        GRBVar g_b = model.addVar(1, GRB_INFINITY, 1, GRB_INTEGER); //beta
        std::vector<GRBVar> g_beta(K, model.addVar(1, pow(2, q - 1), 1, GRB_INTEGER)); // beta
for g_A
        std::vector<std::vector<std::vector<std::pair<GRBVar, GRBVar>>>>g_A(K);//分解后的一系
列 A 矩阵
        for (size_t k = 0; k < K; k++)
        {
```

```
            g_A[k].resize(N, std::vector<std::pair<GRBVar, GRBVar>>(N));
            for (size_t i = 0; i < N; i++)
                for (size_t j = 0; j < N; j++)
                {
                    g_A[k][i][j].first = model.addVar(-pow(2, q - 1), pow(2, q - 1), 1, GRB_INTEGER);
                    g_A[k][i][j].second = model.addVar(-pow(2, q - 1), pow(2, q - 1), 1,
GRB_INTEGER);
                }
        }

        /**************build problem**************/
        //计算 A 矩阵连乘
        std::vector<std::vector<std::pair<GRBLinExpr, GRBLinExpr>>>mA(N,
std::vector<std::pair<GRBLinExpr, GRBLinExpr>>(N));
        for (size_t i = 0; i < N; i++)
        {
            for (size_t j = 0; j < N; j++)
            {
                if (i == j)
                    mA[i][j].first = 1;
                else
                    mA[i][j].first = 0;
                mA[i][j].second = 0;
            }
        }
        for (size_t k = 0; k < K; k++)
        {
            std::vector<std::vector<std::pair<GRBVar, GRBVar>>>tA(N,
std::vector<std::pair<GRBVar, GRBVar>>(N));
            for (size_t i = 0; i < N; i++)
            {
                for (size_t j = 0; j < N; j++)
                {
                    GRBQuadExpr e1 = 0, e2 = 0;
                    for (size_t l = 0; l < N; l++)
                    {
                        e1 += mA[i][l].first * g_A[k][l][j].first - mA[i][l].second * g_A[k][l][j].second;
                        e2 += mA[i][l].first * g_A[k][l][j].second + mA[i][l].second * g_A[k][l][j].first;
```

```
                }
                tA[i][j].first = model.addVar(-GRB_INFINITY, GRB_INFINITY, 1, GRB_INTEGER);
                tA[i][j].second = model.addVar(-GRB_INFINITY, GRB_INFINITY, 1,
GRB_INTEGER);
                model.addQConstr(tA[i][j].first == e1);
                model.addQConstr(tA[i][j].second == e2);
            }
        }
        for (size_t i = 0; i < N; i++)
        {
            for (size_t j = 0; j < N; j++)
            {
                mA[i][j].first = tA[i][j].first;
                mA[i][j].second = tA[i][j].second;
            }
        }
        // beta
        GRBLinExpr sb = 1;
        for (size_t k = 0; k < K; k++)
        {
            GRBVar tb = model.addVar(-GRB_INFINITY, GRB_INFINITY, 1, GRB_INTEGER);
            model.addQConstr(tb == sb * g_beta[k]);
            sb = tb;
        }
        model.addConstr(g_b == sb);
        //计算 F 范数 ||F-beta*A||
        GRBQuadExpr Ef = 0; // rmse = sqrt(Ef)/N
        std::vector<std::vector<std::pair<GRBLinExpr, GRBLinExpr>>> FA(N,
std::vector<std::pair<GRBLinExpr, GRBLinExpr>>(N));
        for (size_t i = 0; i < N; i++)
        {
            for (size_t j = 0; j < N; j++)
            {
                GRBVar a = model.addVar(-GRB_INFINITY, GRB_INFINITY, 1,
GRB_CONTINUOUS);
                GRBVar b = model.addVar(-GRB_INFINITY, GRB_INFINITY, 1,
GRB_CONTINUOUS);
```

```
            model.addQConstr(a * g_b == mA[i][j].first);
            model.addQConstr(b * g_b == mA[i][j].second);
            FA[i][j] = { F(i, j).real() - a,F(i,j).imag() - b };
            //FA[i][j] = { F(i, j).real() * g_beta - mA[i][j].first,F(i,j).imag() * g_beta - mA[i][j].second };
        }
    }
    for (size_t i = 0; i < N; i++)
        for (size_t j = 0; j < N; j++)
            Ef += FA[i][j].first * FA[i][j].first + FA[i][j].second * FA[i][j].second;

    //set constrains
    // g_A[k][i][j] == b*2^a, b=0/1/-2, a=0,1,...
    for (size_t k = 0; k < K; k++)
    {
        for (size_t i = 0; i < N; i++)
        {
            for (size_t j = 0; j < N; j++)
            {
                if (A[k](i, j).real() == 0 && A[k](i, j).imag() == 0)
                {
                    model.addConstr(g_A[k][i][j].first == 0);
                    model.addConstr(g_A[k][i][j].second == 0);
                }
                else
                {
                    if (abs(A[k](i, j).real()) == 1 && A[k](i, j).imag() == 0)
                    {
                        model.addQConstr(g_A[k][i][j].first == A[k](i, j).real() * g_beta[k]);
                        model.addConstr(g_A[k][i][j].second == 0);
                    }
                    GRBVar a1 = model.addVar(0, q - 1, 1, GRB_INTEGER);
                    GRBVar a2 = model.addVar(0, q - 1, 1, GRB_INTEGER);
                    GRBVar b1 = model.addVar(-1, 1, 1, GRB_INTEGER);
                    GRBVar b2 = model.addVar(-1, 1, 1, GRB_INTEGER);
                    GRBVar c1 = model.addVar(0, GRB_INFINITY, 1, GRB_INTEGER);
                    GRBVar c2 = model.addVar(0, GRB_INFINITY, 1, GRB_INTEGER);
                    model.addGenConstrExpA(a1, c1, 2);
                    model.addGenConstrExpA(a2, c2, 2);
```

```
                        model.addQConstr(g_A[k][i][j].first == b1 * c1);
                        model.addQConstr(g_A[k][i][j].second == b2 * c2);
                    }
                }
            }
        }
        for (auto p : all_w_idx)
        {
            if (p.size() < 2)
                continue;
            for (size_t i = 1; i < p.size(); i++)
            {
                model.addConstr(g_A[p[i][0]][p[i][1]][p[i][2]].first * p[i][3] == g_A[p[i - 1][0]][p[i - 1][1]][p[i - 1][2]].first * p[i - 1][3]);
                model.addConstr(g_A[p[i][0]][p[i][1]][p[i][2]].second * p[i][3] == g_A[p[i - 1][0]][p[i - 1][1]][p[i - 1][2]].second * p[i - 1][3]);
            }
        }

        //set optimize object
        model.setObjective(Ef, GRB_MINIMIZE);
        model.set(GRB_DoubleParam_TimeLimit, G_TIME);
        model.set(GRB_DoubleParam_MIPGap, 1e-8);
        model.set(GRB_IntParam_NonConvex, 2);
        model.optimize();
        //get results
        rmse = sqrt(Ef.getValue()) / N;
        for (size_t k = 0; k < K; k++)
        {
            for (size_t i = 0; i < N; i++)
            {
                for (size_t j = 0; j < N; j++)
                {
                    int a = g_A[k][i][j].first.get(GRB_DoubleAttr_X);
                    int b = g_A[k][i][j].second.get(GRB_DoubleAttr_X);
                    A[k](i, j) = std::complex<double>(a, b);
                }
            }
```

```
        }
        beta = g_b.get(GRB_DoubleAttr_X);
        cmp = q * calcComplexMultipleNumber(std::complex<double>(1.0, 0), A);
    }
    catch (GRBException e)
    {
        std::cout << " Error code = " << e.getErrorCode() << std::endl;
        std::cout << e.getMessage() << std::endl;
    }
    catch (...) {
        std::cout << " Exception during optimization " << std::endl;
    }
    delete env;

    Eigen::MatrixXcd resA = Eigen::MatrixXcd::Identity(N, N);
    for (size_t i = 0; i < K; i++)
        resA *= A[i];

    //output
    std::cout << "N = " << N << "\n";
    std::cout << "K = " << K << "\n";
    std::cout << "beta = 1/" << beta << "\n";
    std::cout << "C = " << cmp << "\n";
    std::cout << "RMSE = " << rmse << "\n";
    std::cout << std::fixed << std::setprecision(3);
    std::cout << "F = \n";
    for (size_t i = 0; i < N; i++)
    {
        for (size_t j = 0; j < N; j++)
        {
            double a = F(i, j).real();
            double b = F(i, j).imag();
            std::cout << std::left << "(" << a << ", " << b << ")   ";
        }
        std::cout << "\n";
    }
    std::cout << "\n";
    std::cout << std::fixed << std::setprecision(0);
```

```
std::cout << "F^ = \n";
for (size_t i = 0; i < N; i++)
{
    for (size_t j = 0; j < N; j++)
    {
        double a = resA(i, j).real();
        double b = resA(i, j).imag();
        std::cout << std::left << "(" << a << ", " << b << ")   ";
    }
    std::cout << "\n";
}
std::cout << "\n";
for (size_t k = 0; k < K; k++)
{
    std::cout << "A" << k << " = \n";
    for (size_t i = 0; i < N; i++)
    {
        for (size_t j = 0; j < N; j++)
        {
            double a = A[k](i, j).real();
            double b = A[k](i, j).imag();
            std::cout << std::left << "(" << a << ", " << b << ")   ";
        }
        std::cout << "\n";
    }
    std::cout << "\n";
}
//write
std::string path = "result3.txt";
std::ofstream outputFile(path);
if (outputFile.is_open())
{
    outputFile << "Gurobi result3\n";
    outputFile << "N = " << N << "\n";
    outputFile << "K = " << K << "\n";
    outputFile << "beta = 1/" << beta << "\n";
    outputFile << "C = " << cmp << "\n";
    outputFile << "RMSE = " << rmse << "\n";
```

```
    outputFile << std::fixed << std::setprecision(3);
    outputFile << "F = \n";
    for (size_t i = 0; i < N; i++)
    {
        for (size_t j = 0; j < N; j++)
        {
            double a = F(i, j).real();
            double b = F(i, j).imag();
            outputFile << std::left << "(" << a << ", " << b << ")   ";
        }
        outputFile << "\n";
    }
    outputFile << "\n";
    outputFile << std::fixed << std::setprecision(0);
    outputFile << "F^ = \n";
    for (size_t i = 0; i < N; i++)
    {
        for (size_t j = 0; j < N; j++)
        {
            double a = resA(i, j).real();
            double b = resA(i, j).imag();
            outputFile << std::left << "(" << a << ", " << b << ")   ";
        }
        outputFile << "\n";
    }
    outputFile << "\n";
    for (size_t k = 0; k < K; k++)
    {
        outputFile << "A " << k << " = \n";
        for (size_t i = 0; i < N; i++)
        {
            for (size_t j = 0; j < N; j++)
            {
                double a = A[k](i, j).real();
                double b = A[k](i, j).imag();
                outputFile << std::left << "(" << a << ", " << b << ")   ";
            }
            outputFile << "\n";
```

```
            }
            outputFile << "\n";
        }
        std::cout << "write success.\n";
    }
    else
        std::cout << "write failed.\n";
}

void Solve4()
{
    Eigen::MatrixXcd F4(4, 4), F8(8, 8), F32(32, 32);
    std::function<void(Eigen::MatrixXcd&, int)> buildF = [&](Eigen::MatrixXcd& F, int N)
    {
        std::complex<double> w(cos(2 * PI / N), -sin(2 * PI / N));
        for (size_t i = 0; i < N; i++)
            for (size_t j = 0; j < N; j++)
                F(i, j) = pow(w, i * j);
    };
    buildF(F4, 4);
    buildF(F8, 8);
    F32 = calcuKn(F4, F8);
    //分解矩阵
    std::vector<Eigen::MatrixXcd> A4, A8; //分解后的一系列 A 矩阵
    int dp = 1;
    std::complex<double> w4(cos(2 * PI / 4), -sin(2 * PI / 4));
    A4.push_back(Eigen::MatrixXcd::Identity(4, 4));
    recursionCalcMatrix(4, dp, A4, w4);
    Eigen::MatrixXcd rMat4(4, 4);
    rMat4.setZero();
    for (int i = 0; i < 4; i++)
    {
        std::string binaryStr = "";
        auto t = i;
        while (t > 0) {
            binaryStr = std::to_string(t % 2) + binaryStr;
            t /= 2;
        }
```

```
        std::string ts(std::log2(4) - binaryStr.size(), '0');
        binaryStr = ts + binaryStr;
        std::reverse(binaryStr.begin(), binaryStr.end());
        int ret = 0;
        for (int j = binaryStr.size() - 1; j >= 0; j--)
        {
            ret += (binaryStr[j] - '0') * std::pow(2, binaryStr.size() - 1 - j);
        }
        rMat4(i, ret) = std::complex<double>(1, 0);
    }
    A4.push_back(rMat4);
    dp = 1;
    std::complex<double> w8(cos(2 * PI / 8), -sin(2 * PI / 8));
    recursionCalcMatrix(8, dp, A8, w8);
    Eigen::MatrixXcd rMat8(8, 8);
    rMat8.setZero();
    for (int i = 0; i < 8; i++)
    {
        std::string binaryStr = "";
        auto t = i;
        while (t > 0) {
            binaryStr = std::to_string(t % 2) + binaryStr;
            t /= 2;
        }
        std::string ts(std::log2(8) - binaryStr.size(), '0');
        binaryStr = ts + binaryStr;
        std::reverse(binaryStr.begin(), binaryStr.end());
        int ret = 0;
        for (int j = binaryStr.size() - 1; j >= 0; j--)
        {
            ret += (binaryStr[j] - '0') * std::pow(2, binaryStr.size() - 1 - j);
        }
        rMat8(i, ret) = std::complex<double>(1, 0);
    }
    A8.push_back(rMat8);

    //A4 优化为 4 个矩阵，A8 优化为整数
    std::vector<std::vector<std::vector<int>>> all_w_idx(8);
```

```
for (int it = 0; it < 8; it++)
{
    if (it == 0) continue;
    auto tmpw = std::pow(w8, it);
    for (int k = 0; k < A8.size(); k++)
    {
        for (int i = 0; i < 8; i++)
        {
            for (int j = 0; j < 8; j++)
            {
                if (tmpw == A8[k](i, j))
                {
                    all_w_idx[it].push_back({ k,i,j,1 });
                    std::cout << it << " : " << k << " " << i << " " << j << " " << 1 << std::endl;
                }
                else if (-tmpw == A8[k](i, j))
                {
                    all_w_idx[it].push_back({ k,i,j,-1 });
                    std::cout << it << " : " << k << " " << i << " " << j << " " << -1 << std::endl;
                }
            }
        }
    }
}

int q = 3;
double beta, rmse, cmp, td_4 = 0; // 缩放因子，误差，复杂度，A4 分解误差
std::vector<Eigen::MatrixXcd> A32(4); // F32 分解后的一系列 A 矩阵
// Gurobi 求解
GRBEnv* env = 0;
try
{
    env = new GRBEnv();
    GRBModel model = GRBModel(*env);

    //create variables
    GRBVar g_beta = model.addVar(0, pow(2, q - 1), 1, GRB_CONTINUOUS); //beta
```

```
        std::vector<std::vector<std::vector<std::pair<GRBVar, GRBVar>>>>g_A4(4); // F4 分解后的一系列 A 矩阵
        std::vector<std::vector<std::vector<std::pair<GRBVar, GRBVar>>>>g_A8(4); // F8 分解后的一系列 A 矩阵
        for (size_t k = 0; k < 4; k++)
        {
            g_A4[k].resize(4, std::vector<std::pair<GRBVar, GRBVar>>(4));
            for (size_t i = 0; i < 4; i++)
                for (size_t j = 0; j < 4; j++)
                {
                    g_A4[k][i][j].first = model.addVar(-pow(2, q - 1), pow(2, q - 1) - 1, GRB_INTEGER);
                    g_A4[k][i][j].second = model.addVar(-pow(2, q - 1), pow(2, q - 1), 1,
GRB_INTEGER);
                }
            g_A8[k].resize(8, std::vector<std::pair<GRBVar, GRBVar>>(8));
            for (size_t i = 0; i < 8; i++)
                for (size_t j = 0; j < 8; j++)
                {
                    g_A8[k][i][j].first = model.addVar(-pow(2, q - 1), pow(2, q - 1), 1, GRB_INTEGER);
                    g_A8[k][i][j].second = model.addVar(-pow(2, q - 1), pow(2, q - 1), 1,
GRB_INTEGER);
                }
        }

        /**************build problem***************/
        //计算 A4 矩阵连乘结果 mA4，前三个矩阵连乘结果 tA4
        std::vector<std::vector<std::pair<GRBLinExpr, GRBLinExpr>>>mA4(4,
std::vector<std::pair<GRBLinExpr, GRBLinExpr>>(4));
        std::vector<std::vector<std::pair<GRBLinExpr, GRBLinExpr>>>tA4(4,
std::vector<std::pair<GRBLinExpr, GRBLinExpr>>(4));
        for (size_t i = 0; i < 4; i++)
        {
            for (size_t j = 0; j < 4; j++)
            {
                if (i == j)
                    mA4[i][j].first = 1;
                else
                    mA4[i][j].first = 0;
```

```
                mA4[i][j].second = 0;
            }
        }
        for (size_t k = 0; k < 4; k++)
        {
            std::vector<std::vector<std::pair<GRBVar, GRBVar>>>tA(4,
std::vector<std::pair<GRBVar, GRBVar>>(4));
            for (size_t i = 0; i < 4; i++)
            {
                for (size_t j = 0; j < 4; j++)
                {
                    GRBQuadExpr e1 = 0, e2 = 0;
                    for (size_t l = 0; l < 4; l++)
                    {
                        e1 += mA4[i][l].first * g_A4[k][l][j].first - mA4[i][l].second * g_A4[k][l][j].second;
                        e2 += mA4[i][l].first * g_A4[k][l][j].second + mA4[i][l].second * g_A4[k][l][j].first;
                    }
                    tA[i][j].first = model.addVar(-GRB_INFINITY, GRB_INFINITY, 1, GRB_INTEGER);
                    tA[i][j].second = model.addVar(-GRB_INFINITY, GRB_INFINITY, 1,
GRB_INTEGER);
                    model.addQConstr(tA[i][j].first == e1);
                    model.addQConstr(tA[i][j].second == e2);
                }
            }
            for (size_t i = 0; i < 4; i++)
            {
                for (size_t j = 0; j < 4; j++)
                {
                    mA4[i][j].first = tA[i][j].first;
                    mA4[i][j].second = tA[i][j].second;
                    if (k == 2)
                    {
                        tA4[i][j].first = tA[i][j].first;
                        tA4[i][j].second = tA[i][j].second;
                    }
                }
            }
        }
```

```
        //计算 A8 矩阵连乘结果 mA8
        std::vector<std::vector<std::pair<GRBLinExpr, GRBLinExpr>>>mA8(8,
std::vector<std::pair<GRBLinExpr, GRBLinExpr>>(8));
        for (size_t i = 0; i < 8; i++)
        {
            for (size_t j = 0; j < 8; j++)
            {
                if (i == j)
                    mA8[i][j].first = 1;
                else
                    mA8[i][j].first = 0;
                mA8[i][j].second = 0;
            }
        }
        for (size_t k = 0; k < 4; k++)
        {
            std::vector<std::vector<std::pair<GRBVar, GRBVar>>>tA(8,
std::vector<std::pair<GRBVar, GRBVar>>(8));
            for (size_t i = 0; i < 8; i++)
            {
                for (size_t j = 0; j < 8; j++)
                {
                    GRBQuadExpr e1 = 0, e2 = 0;
                    for (size_t l = 0; l < 8; l++)
                    {
                        e1 += mA8[i][l].first * g_A8[k][l][j].first - mA8[i][l].second * g_A8[k][l][j].second;
                        e2 += mA8[i][l].first * g_A8[k][l][j].second + mA8[i][l].second * g_A8[k][l][j].first;
                    }
                    tA[i][j].first = model.addVar(-GRB_INFINITY, GRB_INFINITY, 1, GRB_INTEGER);
                    tA[i][j].second = model.addVar(-GRB_INFINITY, GRB_INFINITY, 1,
GRB_INTEGER);
                    model.addQConstr(tA[i][j].first == e1);
                    model.addQConstr(tA[i][j].second == e2);
                }
            }
            for (size_t i = 0; i < 8; i++)
            {
                for (size_t j = 0; j < 8; j++)
```

```
                {
                    mA8[i][j].first = tA[i][j].first;
                    mA8[i][j].second = tA[i][j].second;
                }
            }
        }
        //计算 mA4 与 mA8 的张量积 mA32
        std::vector<std::vector<std::pair<GRBVar, GRBVar>>>mA32(32,
std::vector<std::pair<GRBVar, GRBVar>>(32));
        for (size_t i = 0; i < 4; i++)
        {
            for (size_t j = 0; j < 4; j++)
            {
                for (size_t k = 0; k < 8; k++)
                {
                    for (size_t n = 0; n < 8; n++)
                    {
                        int r = i * 8 + k;
                        int c = j * 8 + n;
                        GRBQuadExpr e1 = mA4[i][j].first * mA8[k][n].first - mA4[i][j].second *
mA8[k][n].second;
                        GRBQuadExpr e2 = mA4[i][j].first * mA8[k][n].second + mA4[i][j].second *
mA8[k][n].first;
                        mA32[r][c].first = model.addVar(-GRB_INFINITY, GRB_INFINITY, 1,
GRB_INTEGER);
                        mA32[r][c].second = model.addVar(-GRB_INFINITY, GRB_INFINITY, 1,
GRB_INTEGER);
                        model.addQConstr(mA32[r][c].first == e1);
                        model.addQConstr(mA32[r][c].second == e2);
                    }
                }
            }
        }
        //计算 F 范数 ||F-beta*A||
        GRBQuadExpr Ef = 0; // rmse = sqrt(Ef)/N
        std::vector<std::vector<std::pair<GRBLinExpr, GRBLinExpr>>> FA(32,
std::vector<std::pair<GRBLinExpr, GRBLinExpr>>(32));
        for (size_t i = 0; i < 32; i++)
```

```
        {
            for (size_t j = 0; j < 32; j++)
            {
                GRBVar a = model.addVar(-GRB_INFINITY, GRB_INFINITY, 1,
GRB_CONTINUOUS);
                GRBVar b = model.addVar(-GRB_INFINITY, GRB_INFINITY, 1,
GRB_CONTINUOUS);
                model.addQConstr(a == mA32[i][j].first * g_beta);
                model.addQConstr(b == mA32[i][j].second * g_beta);
                FA[i][j] = { F32(i, j).real() - a,F32(i,j).imag() - b };
            }
        }
        for (size_t i = 0; i < 32; i++)
            for (size_t j = 0; j < 32; j++)
                Ef += FA[i][j].first * FA[i][j].first + FA[i][j].second * FA[i][j].second;

        //set constrains
        // A4 的第 4 个矩阵设为 F4 分解后的排列阵，前三个矩阵连乘等于 F4 分解后的第一个矩阵乘第
二个矩阵
        for (size_t i = 0; i < 4; i++)
        {
            for (size_t j = 0; j < 4; j++)
            {
                model.addConstr(g_A4[3][i][j].first == A4[3](i, j).real()); //A4 前面 push 了一个单位阵
                model.addConstr(g_A4[3][i][j].second == A4[3](i, j).imag());
            }
        }
        GRBQuadExpr Ea = 0; // ||g_A4[0]*g_A4[1]*g_A4[2] - A4[1]*A4[2]||<0.1
        Eigen::MatrixXcd tA401 = A4[1] * A4[2];
        for (size_t i = 0; i < 4; i++)
        {
            for (size_t j = 0; j < 4; j++)
            {
                GRBLinExpr e1 = tA401(i, j).real() - tA4[i][j].first;
                GRBLinExpr e2 = tA401(i, j).real() - tA4[i][j].first;
                Ea += e1 * e1 + e2 * e2;
            }
        }
```

```
        //每个矩阵每行的非零元素个数为1（第4个为排列阵，已经满足）
        double pwlx[6] = { -pow(2,q - 1),-0.75,-0.5,0.5,0.75,pow(2,q - 1) };
        double pwly[6] = { 1,1,0,0,1,1 };
        for (size_t k = 0; k < 3; k++)
        {
            for (size_t i = 0; i < 4; i++)
            {
                GRBLinExpr sn = 0;
                for (size_t j = 0; j < 4; j++)
                {
                    GRBVar x = model.addVar(0, GRB_INFINITY, 1, GRB_INTEGER);
                    GRBVar y = model.addVar(0, GRB_INFINITY, 1, GRB_INTEGER);
                    model.addQConstr(x == g_A4[k][i][j].first * g_A4[k][i][j].first + g_A4[k][i][j].second *
g_A4[k][i][j].second);
                    model.addGenConstrPWL(x, y, 6, pwlx, pwly);
                    sn += y;
                }
                model.addConstr(sn == 1);
            }
        }
        //每个矩阵每列的非零元素个数为1（第4个为排列阵，已经满足）
        for (size_t k = 0; k < 3; k++)
        {
            for (size_t j = 0; j < 4; j++)
            {
                GRBLinExpr sn = 0;
                for (size_t i = 0; i < 4; i++)
                {
                    GRBVar x = model.addVar(0, GRB_INFINITY, 1, GRB_INTEGER);
                    GRBVar y = model.addVar(0, GRB_INFINITY, 1, GRB_INTEGER);
                    model.addQConstr(x == g_A4[k][i][j].first * g_A4[k][i][j].first + g_A4[k][i][j].second *
g_A4[k][i][j].second);
                    model.addGenConstrPWL(x, y, 6, pwlx, pwly);
                    sn += y;
                }
                model.addConstr(sn == 1);
            }
        }
```

```
        // A8 优化为整数即可
        for (size_t k = 0; k < 4; k++)
        {
            for (size_t i = 0; i < 8; i++)
            {
                for (size_t j = 0; j < 8; j++)
                {
                    if (A8[k](i, j).real() == 0 && A8[k](i, j).imag() == 0)
                    {
                        model.addConstr(g_A8[k][i][j].first == 0);
                        model.addConstr(g_A8[k][i][j].second == 0);
                    }
                    else if (abs(A8[k](i, j).real()) == 1 && A8[k](i, j).imag() == 0)
                    {
                        model.addConstr(g_A8[k][i][j].first == A8[k](i, j).real());
                        model.addConstr(g_A8[k][i][j].second == 0);
                    }
                }
            }
        }
        for (auto p : all_w_idx)
        {
            if (p.size() < 2)
                continue;
            for (size_t i = 1; i < p.size(); i++)
            {
                model.addConstr(g_A8[p[i][0]][p[i][1]][p[i][2]].first * p[i][3] == g_A8[p[i - 1][0]][p[i -
1][1]][p[i - 1][2]].first * p[i - 1][3]);
                model.addConstr(g_A8[p[i][0]][p[i][1]][p[i][2]].second * p[i][3] == g_A8[p[i - 1][0]][p[i -
1][1]][p[i - 1][2]].second * p[i - 1][3]);
            }
        }
        model.addConstr(g_beta == 1);

        //set optimize object min { sqrt(Ef)/32 + sqrt(Ea)/4 }
        //GRBVar ef = model.addVar(-GRB_INFINITY, GRB_INFINITY, 1, GRB_CONTINUOUS);
        GRBVar ea = model.addVar(-GRB_INFINITY, GRB_INFINITY, 1, GRB_CONTINUOUS);
        //model.addQConstr(ef * ef == Ef);
```

```
        model.addQConstr(ea * ea == Ea);
        model.addConstr(ea <= 4);
        //model.setObjective(ef / 8 + ea, GRB_MINIMIZE);
        model.setObjective(Ef, GRB_MINIMIZE);
        model.set(GRB_DoubleParam_TimeLimit, G_TIME);
        model.set(GRB_DoubleParam_MIPGap, 1e-8);
        model.set(GRB_IntParam_NonConvex, 2);
        model.optimize();
        //get results
        rmse = sqrt(Ef.getValue()) / 32;
        td_4 = sqrt(Ea.getValue()) / 4;
        for (size_t k = 0; k < 4; k++)
        {
            for (size_t i = 0; i < 4; i++)
            {
                for (size_t j = 0; j < 4; j++)
                {
                    int a = g_A4[k][i][j].first.get(GRB_DoubleAttr_X);
                    int b = g_A4[k][i][j].second.get(GRB_DoubleAttr_X);
                    A4[k](i, j) = std::complex<double>(a, b);
                }
            }
            for (size_t i = 0; i < 8; i++)
            {
                for (size_t j = 0; j < 8; j++)
                {
                    int a = g_A8[k][i][j].first.get(GRB_DoubleAttr_X);
                    int b = g_A8[k][i][j].second.get(GRB_DoubleAttr_X);
                    A8[k](i, j) = std::complex<double>(a, b);
                }
            }
        }
        beta = g_beta.get(GRB_DoubleAttr_X);
        for (size_t k = 0; k < 4; k++)
            A32[k] = calcuKn(A4[k], A8[k]);
        cmp = q * calcComplexMultipleNumber(std::complex<double>(1, 0), A32);
    }
    catch (GRBException e)
```

```
{
    std::cout << " Error code = " << e.getErrorCode() << std::endl;
    std::cout << e.getMessage() << std::endl;
}
catch (...) {
    std::cout << " Exception during optimization " << std::endl;
}
delete env;

Eigen::MatrixXcd resA = Eigen::MatrixXcd::Identity(32, 32);
for (size_t i = 0; i < 4; i++)
    resA *= A32[i];

//output
std::cout << "N = " << 32 << "\n";
std::cout << "K = " << 4 << "\n";
std::cout << "beta = " << beta << "\n";
std::cout << "C = " << cmp << "\n";
std::cout << "RMSE = " << rmse << "\n";
std::cout << "td_4 = " << td_4 << "\n";
std::cout << std::fixed << std::setprecision(3);
std::cout << "F_32 = \n";
for (size_t i = 0; i < 32; i++)
{
    for (size_t j = 0; j < 32; j++)
    {
        double a = F32(i, j).real();
        double b = F32(i, j).imag();
        std::cout << std::left << "(" << a << ", " << b << ")   ";
    }
    std::cout << "\n";
}
std::cout << "\n";
std::cout << std::fixed << std::setprecision(0);
std::cout << "F_32^ = \n";
for (size_t i = 0; i < 32; i++)
{
    for (size_t j = 0; j < 32; j++)
```

```
        {
            double a = resA(i, j).real();
            double b = resA(i, j).imag();
            std::cout << std::left << "(" << a << ", " << b << ")   ";
        }
        std::cout << "\n";
    }
    std::cout << "\n";
    for (size_t k = 0; k < 4; k++)
    {
        std::cout << "A4_" << k << " = \n";
        for (size_t i = 0; i < 4; i++)
        {
            for (size_t j = 0; j < 4; j++)
            {
                double a = A4[k](i, j).real();
                double b = A4[k](i, j).imag();
                std::cout << std::left << "(" << a << ", " << b << ")   ";
            }
            std::cout << "\n";
        }
        std::cout << "\n";
    }
    std::cout << "\n";
    for (size_t k = 0; k < 4; k++)
    {
        std::cout << "A8_" << k << " = \n";
        for (size_t i = 0; i < 8; i++)
        {
            for (size_t j = 0; j < 8; j++)
            {
                double a = A8[k](i, j).real();
                double b = A8[k](i, j).imag();
                std::cout << std::left << "(" << a << ", " << b << ")   ";
            }
            std::cout << "\n";
        }
        std::cout << "\n";
```

```
}
std::cout << "\n";
for (size_t k = 0; k < 4; k++)
{
    std::cout << "A32_" << k << " = \n";
    for (size_t i = 0; i < 32; i++)
    {
        for (size_t j = 0; j < 32; j++)
        {
            double a = A32[k](i, j).real();
            double b = A32[k](i, j).imag();
            std::cout << std::left << "(" << a << ", " << b << ")   ";
        }
        std::cout << "\n";
    }
    std::cout << "\n";
}
//write
std::string path = "result4.txt";
std::ofstream outputFile(path);
if (outputFile.is_open())
{
    outputFile << "Gurobi result4\n";
    outputFile << "N = " << 32 << "\n";
    outputFile << "K = " << 4 << "\n";
    outputFile << "beta = " << beta << "\n";
    outputFile << "C = " << cmp << "\n";
    outputFile << "RMSE = " << rmse << "\n";
    outputFile << "td_4 = " << td_4 << "\n";
    outputFile << std::fixed << std::setprecision(3);
    outputFile << "F_32 = \n";
    for (size_t i = 0; i < 32; i++)
    {
        for (size_t j = 0; j < 32; j++)
        {
            double a = F32(i, j).real();
            double b = F32(i, j).imag();
            outputFile << std::left << "(" << a << ", " << b << ")   ";
```

```
        }
        outputFile << "\n";
    }
    outputFile << "\n";
    outputFile << std::fixed << std::setprecision(0);
    outputFile << "F_32^ = \n";
    for (size_t i = 0; i < 32; i++)
    {
        for (size_t j = 0; j < 32; j++)
        {
            double a = resA(i, j).real();
            double b = resA(i, j).imag();
            outputFile << std::left << "(" << a << ", " << b << ")   ";
        }
        outputFile << "\n";
    }
    outputFile << "\n";
    for (size_t k = 0; k < 4; k++)
    {
        outputFile << "A4_ " << k << " = \n";
        for (size_t i = 0; i < 4; i++)
        {
            for (size_t j = 0; j < 4; j++)
            {
                double a = A4[k](i, j).real();
                double b = A4[k](i, j).imag();
                outputFile << std::left << "(" << a << ", " << b << ")   ";
            }
            outputFile << "\n";
        }
        outputFile << "\n";
    }
    outputFile << "\n";
    for (size_t k = 0; k < 4; k++)
    {
        outputFile << "A8_ " << k << " = \n";
        for (size_t i = 0; i < 8; i++)
        {
```

```
                for (size_t j = 0; j < 8; j++)
                {
                    double a = A8[k](i, j).real();
                    double b = A8[k](i, j).imag();
                    outputFile << std::left << "(" << a << ", " << b << ")   ";
                }
                outputFile << "\n";
            }
            outputFile << "\n";
        }
        outputFile << "\n";
        for (size_t k = 0; k < 4; k++)
        {
            outputFile << "A32_ " << k << " = \n";
            for (size_t i = 0; i < 32; i++)
            {
                for (size_t j = 0; j < 32; j++)
                {
                    double a = A32[k](i, j).real();
                    double b = A32[k](i, j).imag();
                    outputFile << std::left << "(" << a << ", " << b << ")   ";
                }
                outputFile << "\n";
            }
            outputFile << "\n";
        }
        std::cout << "write success.\n";
    }
    else
        std::cout << "write failed.\n";
}

void Solve5(int t)
{
    int K = t + 1;//A 矩阵个数
    int N = pow(2, t);//矩阵维度
    double tole = 0.1;
    double beta = 0, rmse = 0, cmp = 0;//缩放因子，误差，复杂度
```

```
    Eigen::MatrixXcd F(N, N); //F 矩阵
    std::vector<Eigen::MatrixXcd> A; //分解后的一系列 A 矩阵
    //初始化 F 矩阵
    std::complex<double> w(cos(2 * PI / N), -sin(2 * PI / N));
    for (size_t i = 0; i < N; i++)
        for (size_t j = 0; j < N; j++)
            F(i, j) = pow(w, i * j);
    //初始化 A 矩阵
    int dp = 1;
    recursionCalcMatrix(N, dp, A, w);
    Eigen::MatrixXcd rMat(N, N);
    rMat.setZero();
    for (int i = 0; i < N; i++)
    {
        std::string binaryStr = "";
        auto t = i;
        while (t > 0) {
            binaryStr = std::to_string(t % 2) + binaryStr;
            t /= 2;
        }
        std::string ts(std::log2(N) - binaryStr.size(), '0');
        binaryStr = ts + binaryStr;
        std::reverse(binaryStr.begin(), binaryStr.end());
        int ret = 0;
        for (int j = binaryStr.size() - 1; j >= 0; j--)
        {
            ret += (binaryStr[j] - '0') * std::pow(2, binaryStr.size() - 1 - j);
        }
        rMat(i, ret) = std::complex<double>(1, 0);
    }
    A.push_back(rMat);

    /**************分块梯度下降穷举搜索**************/
    // 定义 K-2 个优化参数
    std::vector<int> arr_a(K - 2, 0);
    // 整数化
    std::function<void(std::vector<Eigen::MatrixXcd>&, std::vector<Eigen::MatrixXcd>&,
std::vector<double>&, int)> optSolve =
```

```
        [&](std::vector<Eigen::MatrixXcd>& A, std::vector<Eigen::MatrixXcd>& B,
std::vector<double>& bs, int idx)
    {
        for (size_t k = 0; k < bs.size(); k++)
        {
            B[k] = A[k] * bs[k];
            if (k < idx)
                continue;
            for (size_t i = 0; i < N; i++)
            {
                for (size_t j = 0; j < N; j++)
                {
                    double t1 = abs(B[k](i, j).real());
                    double t2 = abs(B[k](i, j).imag());
                    int f1 = 0;
                    int f2 = 0;
                    if (t1 != 0)
                    {
                        t1 = int(0.5 + t1);
                        //t1 = pow(2, int(0.5 + log2(t1)));
                        f1 = A[k](i, j).real() > 0 ? 1 : -1;
                    }
                    if (t2 != 0)
                    {
                        t2 = int(0.5 + t2);
                        //t2 = pow(2, int(0.5 + log2(t2)));
                        f2 = A[k](i, j).imag() > 0 ? 1 : -1;
                    }
                    B[k](i, j) = std::complex<double>(f1 * t1, f2 * t2);
                }
            }
        }
    };
    // 递归
    std::vector<Eigen::MatrixXcd> B = A;
    std::function<void(int)> dfs = [&](int idx)
    {
        if (idx < 0)
```

```
            return;
        std::cout << "a[" << idx << "] = " << arr_a[idx] << " ";
        beta = 1;
        std::vector<double> bs(K - 2, 1);
        for (size_t i = 0; i < K - 2; i++)
        {
            double tb = 0;
            for (size_t j = i; j < K - 2; j++)
                tb += arr_a[j];
            bs[i] = pow(2, tb);
            beta *= bs[i];
        }
        optSolve(A, B, bs, idx);
        rmse = calcRMSE(F, B, 1.0 / beta);
        std::cout << "rmse = " << rmse << "\n";
        if (rmse <= 0.1)
            idx--;
        else
            arr_a[idx]++;
        dfs(idx);
    };
    clock_t t1 = clock();
    dfs(arr_a.size() - 1);
    clock_t t2 = clock();
    //计算结果
    beta = 1;
    int q = 0;
    std::vector<double> bs(K - 2, 1);
    for (size_t i = 0; i < K - 2; i++)
    {
        double tb = 0;
        for (size_t j = i; j < K - 2; j++)
            tb += arr_a[j];
        bs[i] = pow(2, tb);
        beta *= bs[i];
        if (i == 0)
            q = tb + 1;
    }
```

```
optSolve(A, A, bs, 0);
rmse = calcRMSE(F, A, 1.0 / beta);
cmp = q * calcComplexMultipleNumber({ 1.0 / beta,0 }, A);
//output
std::cout << "N = " << N << "\n";
std::cout << "K = " << K << "\n";
std::cout << "q = " << q << "\n";
std::cout << "beta = 1/" << beta << "\n";
std::cout << "C = " << cmp << "\n";
std::cout << "RMSE = " << rmse << "\n";
std::cout << "time = " << t2 - t1 << "ms\n";
std::cout << std::fixed << std::setprecision(3);
std::cout << "F = \n";
for (size_t i = 0; i < N; i++)
{
    for (size_t j = 0; j < N; j++)
    {
        double a = F(i, j).real();
        double b = F(i, j).imag();
        std::cout << std::left << "(" << a << ", " << b << ")   ";
    }
    std::cout << "\n";
}
std::cout << "\n";
std::cout << std::fixed << std::setprecision(0);
for (size_t k = 0; k < K; k++)
{
    std::cout << "A" << k << " = \n";
    for (size_t i = 0; i < N; i++)
    {
        for (size_t j = 0; j < N; j++)
        {
            double a = A[k](i, j).real();
            double b = A[k](i, j).imag();
            std::cout << std::left << "(" << a << ", " << b << ")   ";
        }
        std::cout << "\n";
    }
```

```
        std::cout << "\n";
    }
    //write
    std::string path = "result5_t=" + std::to_string(t) + ".txt";
    std::ofstream outputFile(path);
    if (outputFile.is_open())
    {
        outputFile << "Gurobi result5\n";
        outputFile << "N = " << N << "\n";
        outputFile << "K = " << K << "\n";
        outputFile << "q = " << q << "\n";
        outputFile << "beta = 1/" << beta << "\n";
        outputFile << "C = " << cmp << "\n";
        outputFile << "RMSE = " << rmse << "\n";
        outputFile << "time = " << t2 - t1 << "ms\n";
        outputFile << std::fixed << std::setprecision(3);
        outputFile << "F = \n";
        for (size_t i = 0; i < N; i++)
        {
            for (size_t j = 0; j < N; j++)
            {
                double a = F(i, j).real();
                double b = F(i, j).imag();
                outputFile << std::left << "(" << a << ", " << b << ")   ";
            }
            outputFile << "\n";
        }
        outputFile << "\n";
        outputFile << std::fixed << std::setprecision(0);
        for (size_t k = 0; k < K; k++)
        {
            outputFile << "A " << k << " = \n";
            for (size_t i = 0; i < N; i++)
            {
                for (size_t j = 0; j < N; j++)
                {
                    double a = A[k](i, j).real();
                    double b = A[k](i, j).imag();
```

```
                    outputFile << std::left << "(" << a << ", " << b << ")   ";
                }
                outputFile << "\n";
            }
            outputFile << "\n";
        }
        std::cout << "write success.\n";
    }
    else
        std::cout << "write failed.\n";
}

int main()
{
    int n = 0;
    std::cout << "要求问题几？ \n";
    std::cin >> n;
    switch (n)
    {
    case 1:
        for (size_t i = 1; i <= 10; i++)
            Solve1(i);
        break;
    case 2:
        Solve2();
        break;
    case 3:
    {
        int m;
        std::cout << "输入 1 使用遗传算法，否则使用 Gurobi: \n";
        std::cin >> m;
        if (m == 1)
            Solve3();
        else
            Solve3_2();
    }
    break;
    case 4:
```

```
        Solve4();
        break;
    case 5:
        for (size_t i = 1; i <= 10; i++)
            Solve5(i);
        break;
    default:
        break;
    }

    system("pause");
    return 0;
}
```