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	武汉大学
参赛队号	23104860044
队员姓名	1. 谢学林
	2. 王何百旭
	3. 郑依凡

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：

在通信信号处理中，离散傅里叶变换（Discrete Fourier Transform，DFT）矩阵计算的硬件复杂度与其算法复杂度和数据元素的取值有关。现有的方法采用快速傅里叶变换和矩阵连乘拟合来降低 DFT 矩阵计算硬件复杂度。为进一步降低芯片的资源损耗，一种新型的可行思路是将 DFT 矩阵分解成整数矩阵连乘的形式，动态地平衡求解精度和硬件复杂度。

针对问题一，本文提出了一种新颖的蝶形-奇异值分解（Butterfly Singular Value Decomposition, BSVD）方法，对 DFT 矩阵蝶形划分并执行奇异值分解，并采用交替方向乘子优化方法对相关连乘矩阵进行优化约束，限制其行非零元素数量（稀疏约束）。为了有效验证设计矩阵的硬件复杂度，提出了基于加法矩阵复杂度统计算法。实验结果表明，在稀疏约束条件限制下，以 BSVD 方法为初置策略对矩阵进行连乘分解能够尽量在保证精度的同时，极大地降低了模型的复杂度，且时域点数越大效果越显著。当 $N=2^t, t=1,2,\cdots,7$ 时，优化的最小误差为 $(0,0.7163,1.0154,1.1039,1.2103,1.2684)$。

针对问题二，本文基于提出的快速量化算法，在满足矩阵元素实虚部取值受限的条件下（量化约束），提供了集中量化和分布量化两种策略，对连乘矩阵进行优化约束。实验证明，两种策略均能显著降低硬件复杂度，当 $N=2^t, t=1,2,\cdots,5$ 时，集中量化的最小误差为 $(0,0,0.1359,0.1617,0.1588)$，分布量化的最小误差为 $(0,0.1787,0.1656,0.2860,0.3972)$。

针对问题三，本文提出了先量化后稀疏的 $\mathcal{Q}$-$\mathcal{S}$ 算法与先稀疏后量化的 $\mathcal{S}$-$\mathcal{Q}$ 算法两种优化方案在稀疏约束与量化约束下对 DFT 矩阵进行连乘分解与近似估计。实验证明，$\mathcal{Q}$-$\mathcal{S}$ 算法具有更高的精度，近似估计的最小误差显著较小，$\mathcal{S}$-$\mathcal{Q}$ 具有更低的硬件复杂度。当 $N=2^t, t=1,2,\cdots,5$ 时，$\mathcal{Q}$-$\mathcal{S}$ 的最小误差为 $(0,0,0.8950,1.0311,1.1044)$，$\mathcal{S}$-$\mathcal{Q}$ 的最小误差为 $(0,0,0.8520,1.0742,1.1672)$。

针对问题四，本文基于 *Kronecker* 混合积的性质，结合 BSVD 算法对高维目标矩阵的连乘分解进行了简化。在稀疏约束与量化约束条件下，先对分解的子矩阵进行优化再由 *Kronecker* 的定义求取目标矩阵的连乘分解与近似估计。最终求得最小误差为 0.9742，硬件复杂度为 36。

针对问题五，基于广义逆矩阵是矛盾方程组最小二乘解的性质，本文将满足约束的近似矩阵的广义逆与估计矩阵相乘，作为新增的待优化连乘矩阵。在约束条件下，通过增加连乘矩阵的数目渐进式的降低最小误差。实验证明在不同的参数设定下，该方法能够在精度受限的情况下有效地降低最小误差。

关键词： 蝶形-奇异值分解　稀疏优化　量化优化　广义逆反演　布尔矩阵

目录

1 问题重述

1.1 背景分析

离散傅里叶变换（Discrete Fourier Transform，DFT）[1] 作为一种基本工具广泛应用于工程、科学以及数学领域。在芯片设计中，DFT 计算的硬件复杂度与其算法复杂度和数据元素取值范围相关 [2, 3]。算法复杂度越高、数据取值范围越大，其硬件复杂度就越大，如图1.1所示。在实际产品的设计中，一般采用快速傅里叶变换（Fast Fourier Transform，FFT）算法 [4, 5, 6] 来快速实现 DFT，利用 DFT 变换的各种性质，进而大幅降低 DFT 的计算复杂度。

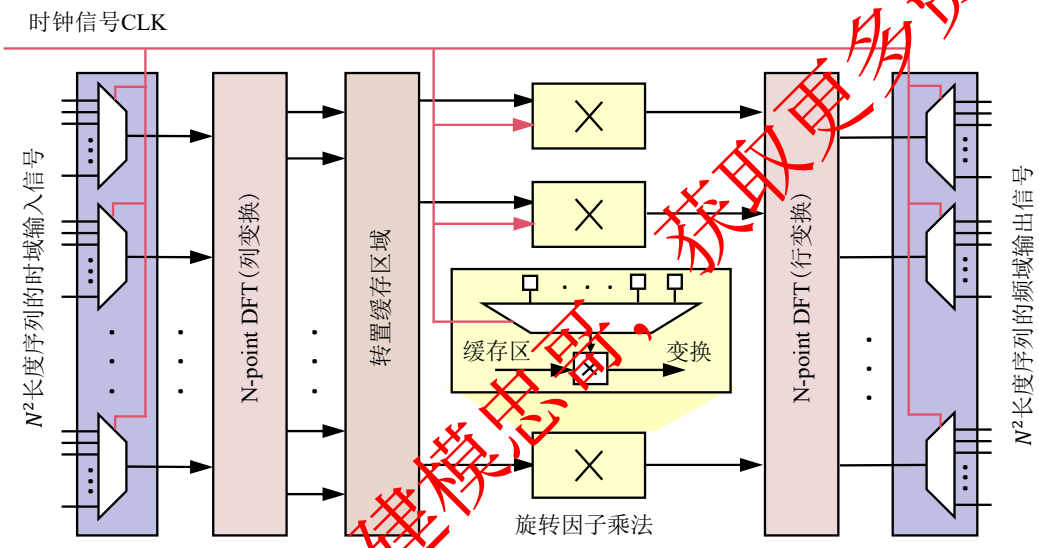


图 1.1　芯片内部的信号流图，时域信号从输入端口输入，经过预设的傅里叶变换矩阵进行行变换以及列变换，最终从输出端口输出等长的频域信号

然而，随着无线通信技术的演进，天线阵面越来越大，通道数越来越多，通信带宽越来越大，对 FFT 的需求也越来越大，从而导致专用芯片上实现 FFT 的硬件开销也越大 [7, 8, 9]。为进一步降低芯片资源开销，亟需要开发一种快速计算 DFT 矩阵的算法，或者对 DFT 矩阵进行近似估计求取近似离散傅里叶变换（Approximation DFT, ADFT）矩阵，在可忍受的误差范围内，加快信号处理的速度，提升芯片的性能。

1.2 问题提出

对于矩阵集 $\mathcal{A}$ 内的矩阵 A_k 以及实数 $\beta \in R$，在约束 1 和约束 2 的条件下求取 N 维度的 DFT 矩阵 F_N 的 ADFT 矩阵 $A = A_1 A_2 \cdots A_k$ 满足 $F_N \approx A_1 A_2 \cdots A_k$：

- **约束一：**限定 $\mathcal{A}$ 中每个矩阵 A_k 的每行至多只有 2 个非零元素（稀疏约束）。

- **约束二：**限定 $\mathcal{A}$ 中每个矩阵 A_k 满足以下要求：

$$A_k[l,m] =\in \{x+jy|x,y\in\mathcal{P}\},\quad \begin{array}{l}\mathcal{P}=\{0,\pm1,\pm2,\cdots,\pm2^{q-1}\},\\ k=1,2,\cdots,K;l,m=1,2,\cdots,N\end{array}$$

其中，$A_k[l,m]$ 表示矩阵 A_k 第 l 行第 m 列的元素（量化约束）。

特殊说明：由官方最初定义的 RMSE 损失为 $\text{RMSE}(\mathcal{A},\beta)=\frac{1}{N}\left\|\beta F_N-\hat{F}\right\|_F^2$。若把参数 β 当成可优化的参数，当将 β 取很小的值时，βF_N 矩阵将与零矩阵非常接近。极端情况下，当 $\beta\to 0$ 时，$\beta F_N\to 0$。此时要求解 $\hat{F}$ 的最佳逼近只需要将 $\mathcal{A}$ 中各个矩阵 A_k 尽可能的接近零矩阵就能达到 RMSE 趋于 0 的效果，并且严格满足约束一和约束二。这显然是没有任何意义的。因此，本文考虑将 RMSE 目标函数重新定义为：

$$\arg\min_{\mathcal{A},\beta}\text{RMSE}(\mathcal{A},\beta)=\frac{1}{N}\left\|F_N-\beta\hat{F}\right\|_F^2 \tag{1.1}$$

我们计划逐步解决以下问题：

问题一：首先通过减少乘法器个数来降低硬件复杂度。由于仅在非零元素相乘时需要使用乘法器，若 A_k 矩阵中大部分元素均为 0，则可减少乘法器的个数，因此希望 A_k 为稀疏矩阵。对于的 DFT 矩阵，请在满足约束 1 的条件下，对最优化问题中的变量 $\mathcal{A}$ 和 β 进行优化，并计算最小误差和方案的硬件复杂度 C。

问题二：讨论通过限制元素实部和虚部取值范围的方式来减少硬件复杂度的方案。对于 $N=2^t,t=1,2,3,4,5$ 的 DFT 矩阵 F_N，在满足约束 2 的条件下，对 $\mathcal{A}$ 和 β 进行优化，并计算最小误差和方案的硬件复杂度 C。

问题三：同时限制 A_k 的稀疏性和取值范围。对于 $N=2^t,t=1,2,3,4,5$ 的 DFT 矩阵 F_N，请在同时满足约束 1 和约束 2 的条件下，对 $\mathcal{A}$ 和 β 进行优化，并计算最小误差和方案的硬件复杂度 C。

问题四：进一步研究对其它矩阵的分解方案。考虑矩阵 $F_N=F_{N1}\otimes F_{N2}$，其中 F_{N1} 和 F_{N2} 分别是 N_1 和 N_2 维的 DFT 矩阵，$\otimes$ 表示 *Kronecker* 积。当 $N_1=4,N_2=8$ 时，在同时满足约束 1 和 2 的条件下，对 $\mathcal{A}$ 和 β 进行优化，并计算最小误差和方案的硬件复杂度 C。

问题五：在问题三的基础上加上精度的限制来研究矩阵分解方案。要求将精度限制在 0.1 以内，即 $\text{RMSE}\le 0.1$。对于 $N=2^t,t=1,2,3,\cdots$ 的 DFT 矩阵 F_N，请在同时满足约束 1 和 2 的条件下，对 $\mathcal{A}$ 和 $\beta,\mathcal{P}$ 进行优化，并计算方案的硬件复杂度 C。

2 模型的假设

1. 忽略生成近似 DFT 矩阵的计算时间以及复杂度对信号在芯片内部传输的影响；
2. 忽略芯片自身的精度损失对生成的近似 DFT 矩阵的影响；
3. 忽略加法的硬件复杂度，只考虑乘法器的硬件复杂度；
4. 忽略矩阵据存储的难度。

3 符号说明

符号	意义
F_N	N 维的离散傅里叶变换矩阵
$\widehat{F_N}$	满足稀疏约束的 N 维的离散傅里叶变换矩阵
$\widetilde{F_N}$	满足量化约束的 N 维的近似离散傅里叶变换矩阵
$\overset{*}{F}_N$	满足稀疏量化联合约束的 N 维的近似离散傅里叶变换矩阵
$\mathcal{A}$	内部矩阵元素乘积构成近似离散傅里叶变换矩阵的矩阵簇
$\mathcal{Q}$	对矩阵进行满足量化约束的量化操作
$\mathcal{S}$	对矩阵进行满足稀疏约束的稀疏操作
ω	N 维离散傅里叶变换矩阵的基底 $e^{-j\frac{2\pi}{N}}$
F_N^+	矩阵 F_N 的广义逆矩阵
β	求解最小误差时的缩放因子
$\mathcal{G}$	统计向量内非令元素的个数
E	误差矩阵
P	重排序矩阵，对序列进行奇偶排序
$\otimes$	*Kronecker* 积
$\|A\|_F^2$	矩阵 A 的 *Frobenius* 范数

4 问题总览

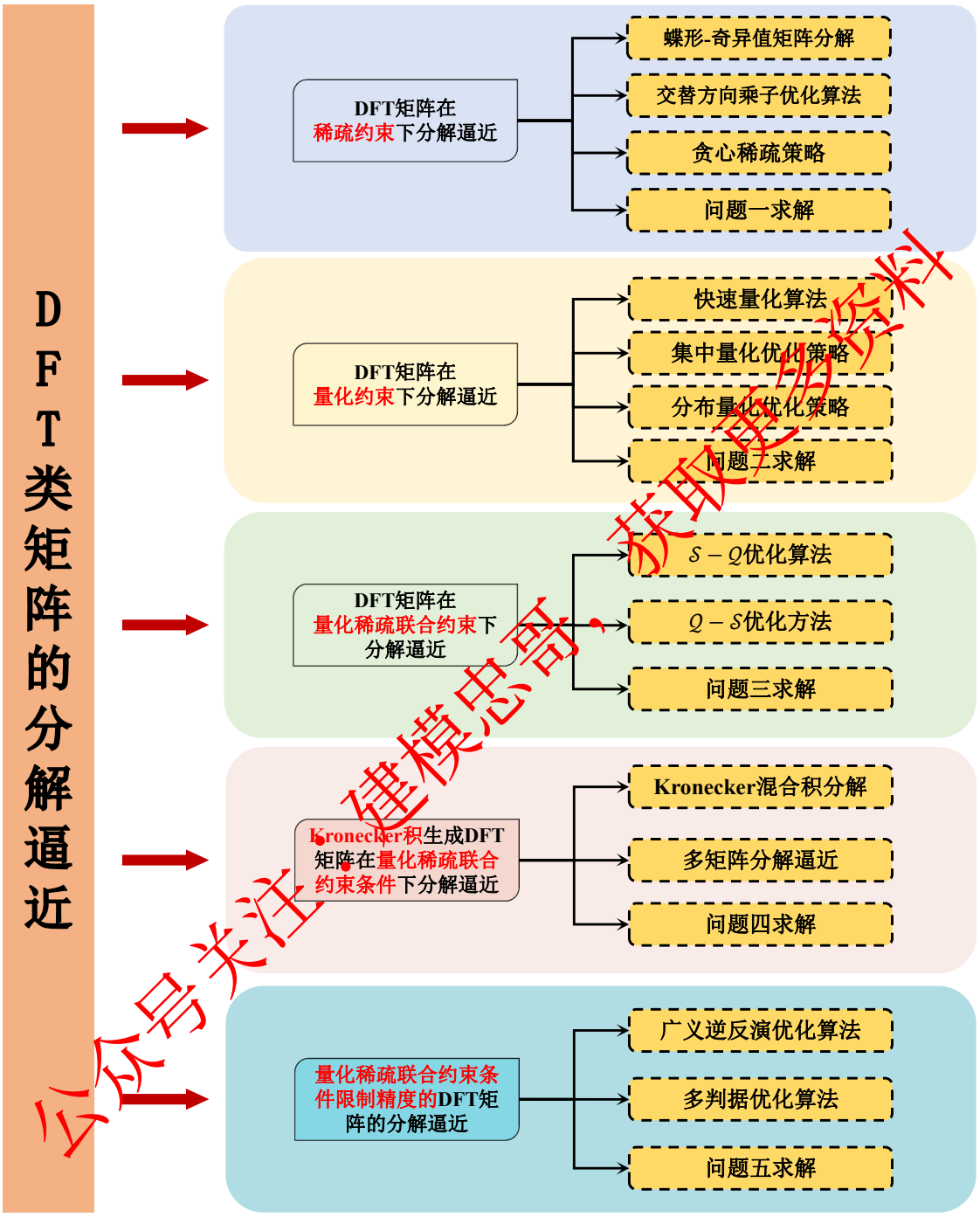


图 4.2　问题总览

DFT 类矩阵的整数分解逼近问题实质上是使用一些稀疏矩阵对原始矩阵进行分解逼近，通过减少矩阵乘积的形式大幅降低复数乘法运算的次数。**对于问题一**，在稀疏约束条件下分解逼近 DFT 矩阵，我们提出了**蝶形-奇异值矩阵分解**算法，并使用**交替方向乘子优化**和**贪心策略**对分解后的矩阵进行稀疏化处理，同时提出了计算硬件复杂度的**布尔矩阵算**

法，并对模型的精度和复杂度进行了计算。**对于问题二**，要求 DFT 矩阵在量化约束条件下进行分解逼近，我们提出一种以最小化 RMSE 为驱动的**快速量化算法**，并考虑了集中量化和分布式量化两种策略，所提出的模型在量化约束的条件下能够保证相对较高的精度及较低的复杂度。**对于问题三**，在同时满足量化和稀疏的约束条件下，对 DFT 矩阵进行分解逼近，我们提出了两种稀疏与量化约束的方案：$\mathcal{S}-\mathcal{Q}$ 和 $\mathcal{Q}-\mathcal{S}$ 算法。经过实验证明，**先量化后稀疏**的 $\mathcal{Q}-\mathcal{S}$ 算法在精度上具有更大的优势，而**先稀疏后量化**的 $\mathcal{S}-\mathcal{Q}$ 算法具有更低的复杂度。**对于问题四**，利用 *Kronecker* 混合积的性质，对 BSVD 算法的矩阵进行了简化，简化后的矩阵在计算 *Kronecker* 积时，具有更高的精度，同时其复杂度也保存在较低的水平。**对于问题五**，结合问题三中的算法与结论，同时满足约束一和约束二的条件，在尽可能使 RMSE 较低的情况下，对矩阵 A、β 和 P 进行了优化。提出了**广义逆反演算法**以获取新增的分解矩阵 A_{add}，有效地提升了 BSVD 模型的精度。

5 问题一：稀疏约束条件下 DFT 矩阵的近似估计

5.1 问题一的分析

问题一要求在满足约束一的条件下，通过减少乘法器个数来降低硬件复杂度，即对最优化问题中的变量 A_k 和 β 进行优化。由于仅在非零元素相乘时需要使用乘法器，若 A_k 矩阵中大部分元素均为 0，则可减少乘法器的个数，因此希望 A_k 为稀疏矩阵。此外，β 决定了目标函数 $RMSE(\mathcal{A},\beta)$ 的损失，因此β也是我们需要优化的对象。最终，我们希望能够以最小误差和硬件复杂度来求取合适的变量 A_k 和 β。

5.2 问题一的建模

本题在无额外约束条件下，研究 DFT 的低复杂度计算方案。对于给定的 N 维离散变量 $X_N=\{x_1,x_2,\cdots,x_n\},n\in\{1,2,\cdots,N\}$，进行离散傅里叶变换可以表述为 [10]：

$$X_K=DFT\left[X_N\right]=\sum_{n=0}^{N-1}x_n*e^{-\frac{j2\pi}{N}nk},k=0,1,2,\cdots,N-1, \tag{5.2}$$

设其进行离散傅里叶变换所需要的 DFT 矩阵为 $F_N\in\mathcal{C}^{N\times N}$, 满足：

$$F_N=\frac{1}{\sqrt{N}}\begin{bmatrix}1 & 1 & 1 & \cdots & 1\\ 1 & \omega^1 & \omega^2 & \cdots & \omega^{N-1}\\ 1 & \omega^2 & \omega^4 & \cdots & \omega^{2(N-1)}\\ \vdots & \vdots & \vdots & \ddots & \vdots\\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \cdots & \omega^{(N-1)(N-1)}\end{bmatrix}, \tag{5.3}$$

式中 ω 为该 DFT 矩阵的旋转因子，定义为 [11, 12]：

$$\omega=e^{-\frac{j2\pi}{N}}. \tag{5.4}$$

对于给定已知的 N 维 DFT 矩阵 F_N，需要设计 K 个矩阵 $\mathcal{A}=\{A_1,A_2,\cdots,A_K\}$，使得矩阵 F_N 和 $\beta A_1,A_2,\cdots,A_K$ 在 *Frobenius* 范数意义下尽可能接近，即：

$$\min_{\mathcal{A},\beta}\text{RMSE}(\boldsymbol{A},\beta)=\frac{1}{N}\sqrt{\|\mathbf{F}_N-\beta\mathbf{A}_1\mathbf{A}_2\cdots\mathbf{A}_K\|_F^2}, \tag{5.5}$$

因此，我们的目标函数设定为：

$$d(A,\beta)=\text{RMSE}(\boldsymbol{A},\beta), \tag{5.6}$$

此外，我们需要约束 $\mathcal{A}$ 中的每个矩阵 A_k 每行的元素至多有两个非零值，可以表述为下式：

$$\text{Condition 1}:\quad G(A_k^l)=\sum_{j=0}^{N-1}(A_k[l,m]!=0)\le 2, \tag{5.7}$$

式中 $G(A_k[l])$ 表示计算矩阵 A_k 第 l 行的非零元素数量，$A_k[l,m]$ 表示矩阵 A_k 第 l 行第 m 列的元素，优化任务总体表述为：

$$\begin{aligned}&\arg\min_{\mathcal{A},\beta}d(\mathcal{A},\beta)\\&s.t.\quad G(A_k[l])\le 2,i=0,1,\cdots,N-1,\end{aligned} \tag{5.8}$$

上述问题在代数领域是非常难处理的，由于不存在其他的限制，$\mathcal{A}$ 集合内的矩阵搜索空间近乎无穷。因此，我们提出了一种新颖的蝶形奇异值分解近似方法（Butterfly Singular Value Decomposition, BSVD）来以更低的硬件复杂度和更高的计算效率逼近理想的 ADFT 矩阵。

首先根据旋转因子的如下性质：

$$\omega_N^{k+\frac{N}{2}}=-\omega_N^k, \tag{5.9}$$

$$\omega_N^{Nk}=1, \tag{5.10}$$

可以从 DFT 矩阵的角度对 Cooley-Tukey 提出的蝶形加速算法 [5, 13] 进行模拟，一个 2 维度的 $N\times N$ 大小的 DFT 矩阵经过奇序和偶序的重排可以得到如下的等式：

$$F_NP^{-1}=\beta\begin{bmatrix}I_{N/2}&D_{N/2}\\I_{N/2}&-D_{N/2}\end{bmatrix}\begin{bmatrix}F_{N/2}&O\\O&F_{N/2}\end{bmatrix}, \tag{5.11}$$

式中 $I_{N/2}$ 是矩阵大小为 $\frac{N}{2}\times\frac{N}{2}$ 的单位矩阵，$D_{N/2}$ 是具有 $N/2$ 个对角元素的对角矩阵，表述为：

$$\begin{aligned}D_{N/2}&=diag(1,\omega^1,\omega^2,\omega^3,\cdots,\omega_{\frac{N}{2}-1}),\omega^i=e^{-\frac{j2\pi}{N}i}\\&=\begin{bmatrix}1&0&0&\cdots&0\\0&\omega^1&0&\cdots&0\\0&0&\omega^2&\cdots&0\\\vdots&\vdots&\vdots&\ddots&\vdots\\0&0&0&\cdots&\omega_{\frac{N}{2}-1}\end{bmatrix},\end{aligned} \tag{5.12}$$

可以推导出当 $N=2$ 和 $N=4$ 时，改变换与 FFT 矩阵相同。P^{-1} 为排序矩阵，用于对原始序列进行偶序重分和奇序重分，为了方便对 DFT 矩阵 F_N 进行近似求解，我们将单位阵 $I_{N/2}$ 与对角阵 $D_{N/2}$ 构成的矩阵记作 W，重排后的序列构成的矩阵记作 R，式 (5.11) 可以简记为：

$$F_N = WRP, \tag{5.13}$$

旋转因子 ω 的对称性质可被用于进行蝶形加速运算，从而可以大大节省生成 DFT 矩阵的计算量，但是，这同样会导致变换前后的序列顺序被打乱，因此需要矩阵 P 完成变换后的矩阵进行重排列。显然，P 矩阵不需要对 WR 矩阵的数值进行规范化处理，而仅需对 WR 矩阵进行列变换改变次序，所以 P 应该为足够稀疏的矩阵，每行每列的非零值元素仅有 1 一个，且数值为 1。根据蝶形算法二为基进行划分的性质，排序矩阵 P 的元素赋值应当满足以下关系：

$$\begin{aligned} P[i,2i] &= 1, i \le \frac{N}{2}-1, i \in Z^+, \\ P[\frac{N}{2}-1+i, 2i-1] &= 1, i \le \frac{N}{2}-1, i \in Z^+ \end{aligned} \tag{5.14}$$

可以看到旋转因子矩阵 W 和排序矩阵 P 天然的具备约束一条件下的稀疏特性，以及约束二条件下的定值特性，因此问题转换为了对矩阵 R 的分解。考虑将矩阵 R 进一步稀疏化为 $\widehat{R}$，由于 SVD 分解 [14, 15, 16] 可以将矩阵划分为正交矩阵及对角矩阵的乘积形式，而对角矩阵显然满足约束条件一。因此，对矩阵 R 进行 SVD 分解可以显著降低问题的复杂度：

$$R = USV^T, \tag{5.15}$$

在执行完 SVD 分解后，实际上仅需考虑对 SVD 分解后的正交矩阵实施稀疏化。记 SVD 分解后并稀疏的矩阵满足

$$\widehat{R} = \widehat{U}S\widehat{V}^T, \tag{5.16}$$

其中 U, V 是正交矩阵，S 是对角矩阵。更近一步的，当前实现是对矩阵 U, V 稀疏化的总体任务目标可以表述为：

$$\begin{aligned} \widehat{U}, \widehat{V} \arg\min_{U,V,\beta} d(U,V,\beta) &= \frac{1}{N}\left\|\mathbf{F}_N - \beta WUSV^T P\right\|_F^2 \\ s.t. \quad G(U[l]) &\le 2, l = 0, 1, \cdots, N-1, \\ G(V^T[l]) &\le 2, l = 0, 1, \cdots, N-1, \end{aligned} \tag{5.17}$$

此时，U 矩阵和 V 矩阵内任何元素的扰动都会目标函数产生影响。由于 U 矩阵和 V^T 矩阵为正交矩阵，两者间的元素相互不作用。因此，对于上述问题，我们采用交替方向乘子优化算法 (Alternating Direction Method of Multipliers, ADMM) 算法 [17, 18]，分别对 U 矩阵

和 V 矩阵进行交替稀疏优化，引入两个辅助变量 Y_U 和 Y_V，式 (5.17) 进一步表述为：

$$\begin{aligned}\arg\min_{U,Y_U,V,Y_V,\beta} & \frac{1}{2N}\left\|\mathbf{F}_N-\beta WUSY_V^TP\right\|_F^2+\frac{1}{2N}\left\|\mathbf{F}_N-\beta WY_USV^TP\right\|_F^2 \\ & +\frac{\lambda_u}{2}\|U-Y_U\|_F^2+\frac{\lambda_v}{2}\|V-Y_V\|_F^2 \\ s.t. \quad & G(U_k[l])\le 2, i=0,1,\cdots,N-1, \\ & G(V_k[l])\le 2, i=0,1,\cdots,N-1,\end{aligned} \tag{5.18}$$

上述问题可以进一步分解为下列三个子问题：

- U 的优化子问题

$$\arg\min_U \frac{1}{2N}\|\mathbf{F}_N-\beta WUSY_V^TP\|_F^2+\frac{\lambda_u}{2}\|U-Y_U\|_F^2, \tag{5.19}$$

- V 的优化子问题

$$\arg\min_V \frac{1}{2N}\|\mathbf{F}_N-\beta WY_USV^TP\|_F^2+\frac{\lambda_v}{2}\|V-Y_V\|_F^2, \tag{5.20}$$

- β 的优化

$$\arg\min_\beta \frac{1}{2N}\|\mathbf{F}_N-\beta WUSY_V^TP\|_F^2+\frac{1}{2N}\|\mathbf{F}_N-\beta WY_USV^TP\|_F^2, \tag{5.21}$$

由于给定的判据条件难以进行优化，采用贪心稀疏 [19, 20] 的策略，交替的对矩阵 U 和 V 的元素置零值，以逐渐达到稀疏的目的，从而满足约束 1。对于 U 的第 k 次优化，首先逐个将其中的元素置为 0，每个位置的元素的变动必定会带来误差的提升，故可以建立误差矩阵 E_U^k 来表示第 k 次优化时，U 被稀疏产生的误差，满足：

$$E_U^k[l,m]=d(WU^kSV^{Tk}P,\beta)\Big|_{U^k[l,m]=0}, \tag{5.22}$$

式中 $[l,m]$ 表示取第 l 行，第 m 列的元素值。由于元素的稀疏必然导致误差的上升，因此在约束条件内保留更多的元素对降低误差是尤为必要的。故在满足矩阵 U^k 符合约束一的条件下，选择 E_U^k 中合适的非零极小值的行列坐标 l_{sparse}^U 和 m_{sparse}^U，并对 U^k 矩阵内该坐标下的元素置零，表述为：

$$U^{k+1}=U^k\Big|_{U^k[l_{\text{sparse}}^U,m_{\text{sparse}}^U]=0}, \tag{5.23}$$

随后对矩阵 V^k 进行第 k 次优化，同样建立误差矩阵 E_V^k 满足:

$$E_V^k[l,m]=d(WU^{k+1}SV^{Tk}P,\beta)\Big|_{V^k[l,m]=0}, \tag{5.24}$$

则同样寻找合适的下标 l_{sparse}^V 和 m_{sparse}^V，并对 V^k 矩阵内该坐标下的元素置零，可得：

$$V^{k+1}=V^k\Big|_{V^k[l_{\text{sparse}}^V,m_{\text{sparse}}^V]=0}, \tag{5.25}$$

最后，对于 β 在第 k 次的优化，我们采用范数比来更新 β 的数值：

$$\beta^{k+1} = \frac{\|\mathbf{F}_N\|_F}{\|WU^{(k+1)}SV^{(k+1)T}P\|_F}, \tag{5.26}$$

上述贪心稀疏算法的迭代过程如图5.3所示，从矩阵的第一行开始，先判断矩阵的行稀疏度是否小于等于 2，若不满足条件则计算该行矩阵所有非零元素删除后的误差，再根据误差删除元素直到该行满足约束条件，若满足条件，则对其它行继续执行这样的操作，一直迭代到最后一行和最后一列。

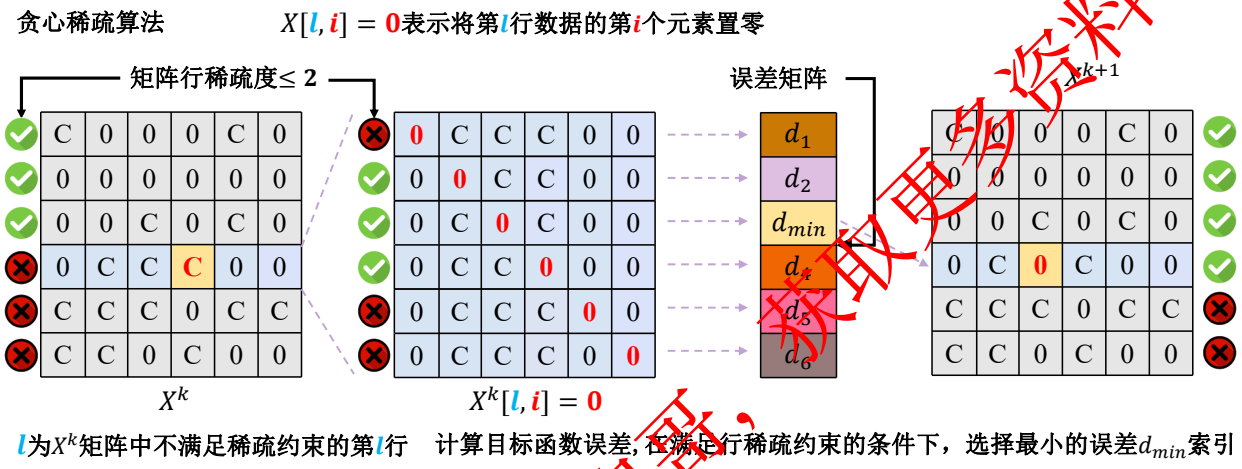


图 5.3　贪心稀疏算法示意图

最终迭代优化得到的矩阵分别记为 $\hat{U},\hat{V},\hat{\beta}$。如此一来，我们设计了 $K=5$ 个矩阵，使得矩阵 $W\hat{U}S\hat{V}^TP$ 在 *Frobenius* 范数意义下尽可能接近 DFT 矩阵，将 DFT 矩阵分解成多个矩阵的连乘形式，表述为：

$$\hat{F}_N = W\hat{U}S\hat{V}^TP, \tag{5.27}$$

根据上述分析可知，S 为奇异值矩阵，仅包含对角的奇异值元素，与其他矩阵相乘仅仅改变向量的模长。P 为排序矩阵，仅仅影响矩阵的行列变换，不改变矩阵的稀疏性。因此，我们额外考虑 $\mathcal{A}$ 中矩阵的个数 K 的取值的限制条件，要求 K 越小，硬件复杂度越低。故可将上述矩阵进行合并得到下述表达式：

$$\hat{F}_N = W(\hat{U}S)(\hat{V}^TP) = W\hat{U}(S\hat{V}^TP) = A_1A_2A_3, \tag{5.28}$$

如此，我们近似分解矩阵最大的 K 值为 3，在某些极端 N 的条件下，上式可进一步简化。上述提出的蝶形-奇异值分解（BSVD）算法求取近似 DFT 矩阵的算法流程图如算法1所示。

为了方便衡量 BSVD 算法分解得到的矩阵给硬件计算带来的复杂度 C，我们额外的提出了一种快速且鲁棒的复杂度统计算法，算法流程如图2所示。

算法 1 蝶形-奇异值分解稀疏求解近似 DFT 矩阵

输入: 时域点数 N。

输出: 稀疏矩阵 $\widehat{U}$，$\widehat{V}^T$，尺度系数 $\widehat{\beta}$，最终 RMSE 误差 $\widehat{E}$。

初始化 DFT 矩阵 F_N，尺度系数 $\beta = 1$，误差矩阵 $E_U^0 = E_V^0 = \text{Inf}$;

对 DFT 矩阵按照式 (5.13)(5.15) 进行分解 $F_N = WUSV^TP$;

采用交替乘子优化方法 ADMM 对矩阵 U 和 V^T 交替逐步进行稀疏化;

for k=1: 最大迭代次数 **do**

$$E_U^k[l,m] = d(WU^kSV^{Tk}P, \beta^k)\Big|_{U^k[l,m]=0}, \quad l,m \in \{0,1,\cdots,N-1\}$$

$$l_{\text{sparse}}^U, m_{\text{sparse}}^U = \arg\min_{l,m} E_U^k[l,m], \quad s.t.\text{Condition 1}$$

$$U^k[l_{\text{sparse}}^U, m_{\text{sparse}}^U] = 0 \rightarrow U^{k+1} = U^k$$

$$E_V^k[l,m] = d(WU^{k+1}SV^{Tk}P, \beta^k)\Big|_{V^k[l,m]=0}, \quad l,m \in \{0,1,\cdots,N-1\}$$

$$l_{\text{sparse}}^V, m_{\text{sparse}}^V = \arg\min_{l,m} E_V^k[l,m], \quad s.t.\text{Condition 1}$$

$$V^k[l_{\text{sparse}}^V, m_{\text{sparse}}^V] = 0 \rightarrow V^{k+1} = V^k$$

$$\beta^{k+1} = \frac{\|\mathbf{F}_N\|_F}{\|WU^{(k+1)}SV^{(k+1)T}P\|_F}$$

end

输出最终的稀疏矩阵 $\widehat{U}$，$\widehat{V}$，保留迭代后的参数 $\widehat{\beta}$;

整合输出矩阵输出估计矩阵 $F_N = W\widehat{U}S\widehat{V}^TP) = W\widehat{U}(S\widehat{V}^TP)$;

存储矩阵 $A_1 = W, A_2 = \widehat{U}, A_3 = S\widehat{V}^TP$;

计算最终的 RMSE 误差：$\widehat{E} = \sqrt{\|F_N - \beta\widehat{F}\|_F^2/N}$;

返回：程序结束。

算法 2 硬件复杂度 C 统计算法

输入: 矩阵 A_1，$\hat{A}_2$，$\hat{A}_3$ 的值;

输出: 复数乘法的次数 L，硬件复杂度 C;

根据所给复数乘法定义得到条件标记矩阵 Mask $= A_k \in \{\mathbf{R}, \pm j, \pm 1 \pm j\}$;

初始化布尔矩阵。$B_k = true(size(A_k))$;

根据条件标记得到 A_1, A_2, A_3 的布尔掩模，记作 B_1, B_2, B_3, 令 $B_k(\text{Mask}) = false$;

计算两次矩阵乘法的复乘次数 $M_1 = nnz(B_1B_2)$ 和 $M_2 = nnz(B_1B_2B_3)$, 其中 $nnz(\cdot)$ 为计算并返回输入矩阵非零元素个数;

计算硬件复杂度 $C = q \times (M_1 + M_2)$;

返回：程序结束。

5.3 问题一的求解

根据蝴蝶-奇异值分解算法（BSVD）我们可以针对不同的维度大小 N 快速的计算相应的稀疏矩阵以及重构误差。对于 $N=2^t, t\in Z^+$，我们给出了不同 t 条件下矩阵 A_1, A_2 和 A_3 的值。首先，当 $t=1$ 时，$N=2$，根据算法1可将 F_2 分解为：

$$F_2 = W_2 U_2 S_2 V_2^T P_2,$$

$$W_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, U_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, S_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, V_2^T = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, P_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \tag{5.29}$$

所求的 $\beta=1$ 依据式 (5.28) 兼并上述矩阵可得合并后的矩阵 A_1, A_2, A_3 分别为:

$$A_1 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, A_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, A_3 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \tag{5.30}$$

上述分解的矩阵严格符合约束 1，且满足：

$$F_2 = \beta\widehat{F_2} = \beta A_1 A_2 A_3 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \tag{5.31}$$

根据式 (5.5) 可以计算得到当前的 ADFT 矩阵的损失为 $\mathrm{RMSE}(F_2, \beta\widehat{F_2}) = 0$。当 $t=2$ 时，$N=4$，重复执行算法1，对 F_4 进行分解得到：

$$F_4 = W_4 U_4 S_4 V_4^T P_4,$$

$$W_4 = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}, U_4 = \begin{bmatrix} -\frac{\sqrt{2}}{2} & 0 & -\frac{\sqrt{2}}{2} & 0 \\ -\frac{\sqrt{2}}{2} & 0 & \frac{\sqrt{2}}{2} & 0 \\ 0 & -\frac{\sqrt{2}}{2} & 0 & -\frac{\sqrt{2}}{2} \\ 0 & -\frac{\sqrt{2}}{2} & 0 & \frac{\sqrt{2}}{2} \end{bmatrix},$$

$$S_4 = \begin{bmatrix} \sqrt{2} & 0 & 0 & 0 \\ 0 & \sqrt{2} & 0 & 0 \\ 0 & 0 & \sqrt{2} & 0 \\ 0 & 0 & 0 & \sqrt{2} \end{bmatrix}, V_4^T = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}, P_4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{5.32}$$

所求的 $\beta=1$ 依据式 (5.28) 兼并上述矩阵可得合并后的矩阵 A_1, A_2, A_3 分别为:

$$A_1 = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}, A_2 = \begin{bmatrix} -\frac{\sqrt{2}}{2} & 0 & -\frac{\sqrt{2}}{2} & 0 \\ -\frac{\sqrt{2}}{2} & 0 & \frac{\sqrt{2}}{2} & 0 \\ 0 & -\frac{\sqrt{2}}{2} & 0 & -\frac{\sqrt{2}}{2} \\ 0 & -\frac{\sqrt{2}}{2} & 0 & \frac{\sqrt{2}}{2} \end{bmatrix}, A_3 = \begin{bmatrix} -\sqrt{2} & 0 & 0 & 0 \\ 0j & -\sqrt{2} & 0 & 0 \\ 0 & 0.0 & -\sqrt{2} & 0 \\ 0 & 0 & 0 & -\sqrt{2} \end{bmatrix}, \tag{5.33}$$

上述分解的矩阵严格符合约束 1，且满足：

$$F_4 = \beta\widehat{F_4} = \beta A_1 A_2 A_3 \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & -1 & j \\ 1 & -1 & 1 & -1 \\ 1 & j & -1 & -j \end{bmatrix}, \tag{5.34}$$

根据式 (5.5) 可以计算得到当前的 ADFT 矩阵的损失为 $\mathrm{RMSE}(F_4, \beta\widehat{F_4}) = 0$。当 $t = 3$ 时，$N = 8$，对 F_8 进行分解可得:

$$F_8 = W_8 U_8 S_8 V_8^T P_8,$$

$$W_8 = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & \frac{\sqrt{2}}{2}(1-j) & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & \frac{\sqrt{2}}{2}(1+j) \\ 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & \frac{\sqrt{2}}{2}(j-1) & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & j & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & \frac{\sqrt{2}}{2}(1+j) \end{bmatrix},$$

$$V_8^T = \begin{bmatrix} -1.000 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1.000 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1.000 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1.000 & 0 & 0 \\ 0 & 0 & -0.196j & 0.981 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0.971+0.139j & 0.028-0.194j & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -0.196j & 0.981 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0.971+0.139j & 0.028-0.194j \end{bmatrix},$$

$$U_8 = \begin{bmatrix} -0.5 & 0.5 & 0 & 0 & 0 & 0 & 0 & 0 \\ -0.5 & 0 & 0 & 0 & 0 & -0.582+0.083j & 0 & 0 \\ -0.5 & 0 & 0 & 0 & -0.490+0.098j & 0 & 0 & 0 \\ -0.5 & 0 & 0 & 0 & -0.588j & 0 & 0 & 0 \\ 0 & 0 & -0.5 & 0.5 & 0 & 0 & 0 & 0 \\ 0 & 0 & -0.5 & 0 & 0 & 0 & 0 & -0.582+0.083j \\ 0 & 0 & -0.5 & 0 & 0 & 0 & -0.490+0.098j & 0 \\ 0 & 0 & -0.5 & 0 & 0 & 0 & 0.588j & 0 \end{bmatrix},$$

$$S_8 = \begin{bmatrix} 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2, \end{bmatrix}, P_8 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}, \tag{5.35}$$

所求的 $\beta = 1.3451$ 依据式 (5.28) 兼并上述矩阵可得合并后的矩阵 A_1, A_2, A_3 分别为:

$$A_1 = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & \frac{\sqrt{2}}{2}(1-j) & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & -\frac{\sqrt{2}}{2}(1+j) \\ 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & \frac{\sqrt{2}}{2}(j-1) & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & j & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & \frac{\sqrt{2}}{2}(1+j) \end{bmatrix},$$

$$A_2 = \begin{bmatrix} -0.5 & 0.5 & 0 & 0 & 0 & 0 & 0 & 0 \\ -0.5 & 0 & 0 & 0 & 0 & -0.582+0.083j & 0 & 0 \\ -0.5 & 0 & 0 & 0 & -0.490+0.098j & 0 & 0 & 0 \\ -0.5 & 0 & 0 & 0 & -0.588j & 0 & 0 & 0 \\ 0 & 0 & -0.5 & 0.5 & 0j & 0 & 0 & 0 \\ 0 & 0 & -0.5 & 0 & 0 & 0 & 0 & -0.582+0.083j \\ 0 & 0 & -0.5 & 0 & 0 & 0 & -0.490+0.098j & 0 \\ 0 & 0 & -0.5 & 0 & 0 & 0.000 & 0.588j & 0 \end{bmatrix}, \tag{5.36}$$

$$A_3 = \begin{bmatrix} -2.000 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2.000 & 0 & 0 & 0 & 0 & 0 \\ 0 & -2.000 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2.000 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -0.392j & 0 & 1.961 & 0 \\ 0 & 0 & 0 & 0 & 1.941+0.278j & 0 & 0.056-0.388j & 0 \\ 0 & 0 & 0 & 0 & 0 & -0.392j & 0 & 1.961 \\ 0 & 0 & 0 & 0 & 0 & 1.941+0.278j & 0 & 0.056-0.388j \end{bmatrix},$$

上述分解的矩阵严格符合约束 1，可以求得估计的 ADFT 矩阵 $\widehat{F_8}$ 为：

$$\widehat{F_8} = \begin{bmatrix} 1.000 & 1.000 & 1.000 & 1.000 & 0 & 0 & 0 & 0 \\ 1.000 & 0.707(1-j) & 0 & 0 & -1.154 & 0.816(j-1) & 0.231j & 0.163(1+j) \\ 1.000 & -1.000j & 0 & 0 & 0.038+0.192j & 0.192-0.038j & -0.962+0.192j & 0.192+0.962j \\ 1.000 & -0.707(1+j) & 0 & 0 & -0.231 & 0.163(1+j) & -1.154j & 0.816(j-1) \\ 1.000 & -1.000 & 1.000 & -1.000 & 0 & 0 & 0 & 0 \\ 1.000 & 0.707(j-1) & 0 & 0 & -1.154 & 0.816(1-j) & 0.231j & -0.163(1+j) \\ 1.000 & 1.000j & 0 & 0 & 0.038+0.192j & -0.192+0.038j & -0.962+0.192j & -0.192-0.962j \\ 1.000 & 0.707(1+j) & 0 & 0 & -0.231 & -0.163(1+j) & -1.154j & 0.816(1-j) \end{bmatrix}, \tag{5.37}$$

根据式 (5.5) 可以计算得到当前的 ADFT 矩阵的损失为 RMSE$(F_8, \beta\widehat{F_8}) = 1.0154$。

上述例子有效地证明了我们提出的 BSVD 方法的有效性，为了进一步的进行数据分析，我们将 t 的数值逐步推广至更多的实值整数，并统计了稀疏约束条件下，$t = 1, 2, \cdots, 7$ 时对应的 β 数值、最小误差 $d(\mathcal{A}, \beta)$，基于蝶形-奇异值分解 BSVD 稀疏算法的硬件复杂度 C_{BSVD}，以及普通蝶形分解稀疏算法的硬件复杂度 C_B，如表5.1所示。

表 5.1　稀疏约束条件下指数 t 与相关数据的统计（$q = 16$）

t	1	2	3	4	5	6	7
β	1.0000	1.0000	1.3451	2.1957	3.4449	6.0387	9.9772
$d(\mathcal{A}, \beta)$	0	0	0.7163	1.0154	1.1039	1.2103	1.2684
C_{BSVD}	0	0	192	1312	3168	6784	14272
C_{B}	0	0	0	512	4096	22528	106496

此外，为了更加清晰的提供数据分析，我们还提供了数据的可视化结果，其中最小误差 $d(\mathcal{A}, \beta)$ 随指数 t 的变化如图5.4所示。硬件复杂度 C 随时域点数 $N = 2^t$ 的变化如图5.5所示，为了彰显差异，我们在原始坐标系和对数坐标系内分别进行了绘制，对应图5.5a和图5.5b。

5.4 结果分析

由图5.4所示，可以观察到，我们所提 BSVD 算法的最小误差 RMSE（即 $d(\mathcal{A}, \beta)$）随时域点数 $N = 2^t$ 的增加呈现出先递增的趋势。当时域点为 $N = 2^1$ 和 2^2 时，我们的算法与 FFT 算法等价，此时没有 RMSE 误差和硬件复杂度。由表5.1可知，当对应的时域点为 $N = 2^3$ 时，出现的最小非零 RMSE 误差为 0.7163，而当时域点为 $N = 2^7$ 时，算法所测试点中 RMSE 最大，其值为 1.2684。因此，我们可以进行如下推测；矩阵的稀疏化是一种对

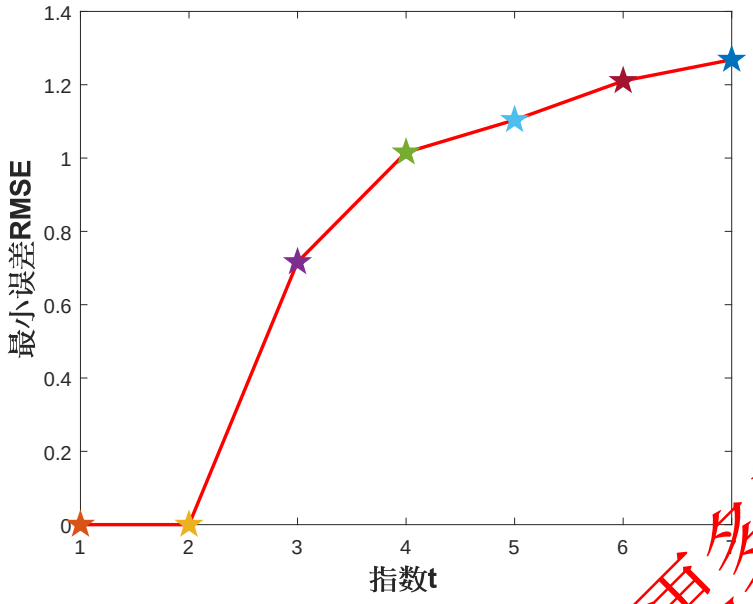
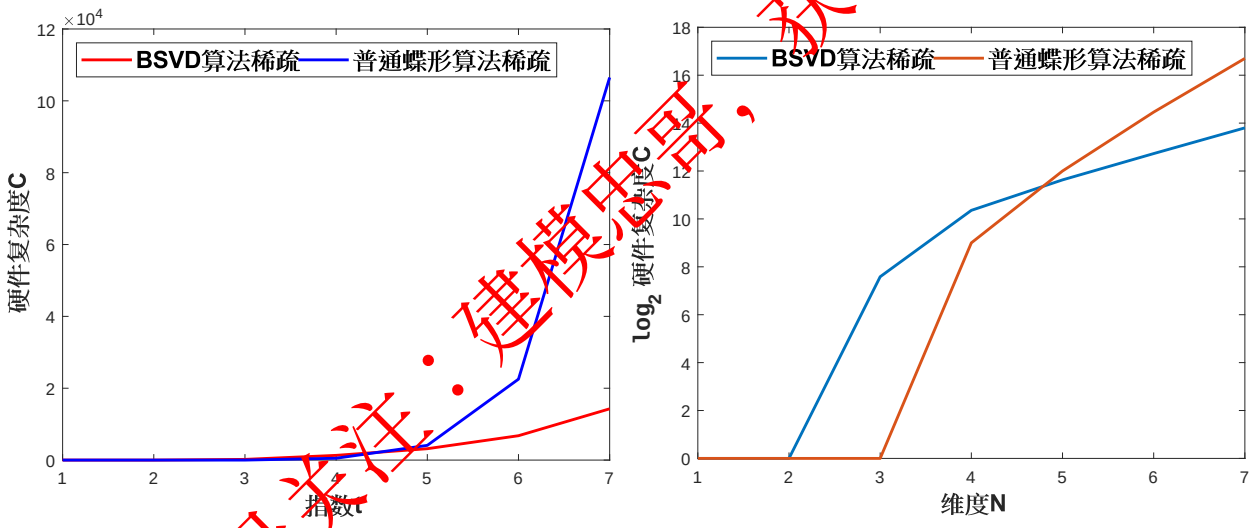


图 5.4　$d(\mathcal{A},\beta)$ 随时域点数 $N=2^t$ 的变化（BSVD 算法稀疏）



(a) 原始坐标系　　(b) 对数坐标系

图 5.5　BSVD 算法稀疏与普通蝶形算法稀疏的硬件复杂度 C 随时域点数 $N=2^t$ 变化

非保留元素置零的批量化操作，等价于对矩阵内的信息进行特定丢失。当矩阵维度或数据维度较小时，矩阵和数据包含的信息较少，因此丢失的平均信息较少；反观矩阵维度或数据维度不断增大时，矩阵和数据包含的信息也逐渐增多，因此丢失的信息数量和大小也逐渐递增。因此，稀疏约束下最小误差 RMSE 随数据维度的增加而增加这一现象是正常的，并且该现象在矩阵和数据的稀疏化中经常出现。我们方法的优势在于使用 SVD 分解将 R 的稀疏优化转化为了正交矩阵 U 和 V^T 的优化，其中 R 矩阵所对应的奇异值矩阵 S 内对角元素的值反应了当前 U 和 V 矩阵列向量的重要程度，在次基础上进行有选择的稀疏，能够最大程度的把控稀疏约束条件下，矩阵信息的损失。从图5.4也可以看出 RMSE 的变化

趋势逐渐减缓，且趋于稳定，侧面说明，我们的算法在处理高维问题时可能具有更好的效果。

另一方面，对于算法的复杂度而言，由复杂度定义 $C = qL$（q 为分解后矩阵实部和虚部的范围，L 为复数乘法的次数）可知，此时对矩阵进行稀疏，本质是使矩阵产生更多的零元素。由于分解后的矩阵元素实部和虚部并非都为整数，故此时我们复杂度的计算采用了默认值 $q = 16$。由图5.5，可以观察到，硬件复杂度随时域点数 $N = 2^t$ 呈明显的增长关系。当时域点超过 $N = 2^5$ 时，我们的算法比普通的 FFT 算法复杂度更低，特别是我们的复杂度增加趋势远低于普通的 FFT 算法（FFT 算法在时域点较多时增长显著）。例如，由表5.1，当时域点 $N = 2^7$ 时，我们的复杂度为 14272，而 FFT 算法的复杂度为 106496，此时复杂度少了一个数量级。因此，随着时域点的增加，我们的算法在复杂度上具有明显的优势。总的来说，我们的算法在精度和复杂度上具有一定的保证。特别地，当 N 越来越大时，我们的算法在降低硬件复杂度上具有较为明显的优势。

6 问题二：量化约束条件下 DFT 矩阵的近似估计

6.1 问题二的分析

在通信系统中，时常需要对数据进行离散处理转换为数字信号。因此，通信系统内传输的信号其实部和虚部通常会有固定的取值范围，现讨论通过限制 ADFT 矩阵元素实部和虚部取值范围的方式来减少硬件复杂度的方案。本问题限制在 $q = 3$，即对于矩阵集合 $\mathcal{A}$ 内的矩阵 A_k，其内部元素的值满足 $A_k[l, m] \in \{x + jy | x, y \in \{0, 1, 2, 4\}\}$。在这一约束条件下，对于 $N = 2^t, t = 1, 2, 3, 4, 5$ 的 DFT 矩阵 F_N，需要对 $\mathcal{A}$ 和 β 进行优化，并计算最小误差和方案的硬件复杂度 C。

6.2 问题二的建模

根据问题二的分析，上述问题可以视作是一种线性编码问题，旨在对矩阵内的元素进行量化，以尽可能的减小最下误差。考虑以下普适的量化方式，假定序列 $X_N = \{x_1, x_2, \cdots, x_n\}, , n \in \{1, 2, \cdots, N\}$ 是一系列的复信号，满足:

$$\begin{aligned} & x_i = a_i + b_i j, \\ s.t. \quad & 0 \leq \|a_i\| \leq \max(\|a_i\|), \\ & 0 \leq \|b_i\| \leq \max(\|b_i\|), \end{aligned} \tag{6.38}$$

其中实部 a 和虚部 b 的量化是相互独立的，因此可以对式 (6.38) 内的实部、虚部进行规范化，简化为：

$$\begin{aligned} \bar{x_i} = & \frac{a_i}{\max(\|a_i\|)} + \frac{b_i}{\max(\|b_i\|)} j = \bar{a_i} + \bar{b_i} j, \\ s.t. \quad & 0 \leq \|\bar{a_i}\| \leq 1, 0 \leq \|\bar{b_i}\| \leq 1, \end{aligned} \tag{6.39}$$

假定矩阵 $A_{n\times n}$，矩阵 $B_{n\times n}$ 和矩阵 $C_{n\times n}$ 内第 i 行第 j 列的元素分别为 $a_{i,j}$，$b_{i,j}$ 及 $c_{i,j}$，并且每个元素都是经过规范化的。其中矩阵 C 由矩阵 A, B 相乘得到，其矩阵元素满足：

$$c_{i,j} = \sum_{k=0}^{N-1} a_{i,k} b_{k,j}, \tag{6.40}$$

当矩阵 A 中的 i 行元素 $a_{i,k}, k = 0, 1, \cdots, N-1$ 被量化为 $\widetilde{a_{i,k}}, k = 0, 1, \cdots, N-1$，那么此时 C 矩阵中的第 i 行元素也发生变化为 $\widetilde{c_{i,k}}, k = 0, 1, \cdots, N-1$ 满足：

$$\widetilde{c_{i,k}} = \sum_{k=0}^{N-1} \widetilde{a_{i,k}} b_{k,j}, \tag{6.41}$$

则误差可以定义为：

$$err_i = d(\widetilde{c_{i,k}}, c_{i,k}) = \frac{1}{N}\sqrt{\sum_{k=0}^{N-1}\left(\widetilde{c_{i,k}} - c_{i,k}\right)^2} = \frac{1}{N}\sqrt{\sum_{k=0}^{N-1}\sum_{k=0}^{N-1} b_{k,k}^2\left(\widetilde{a_{i,k}} - a_{i,k}\right)^2}, \tag{6.42}$$

由此可以推导出，总体误差正比于量化误差：

$$err = \sum_{i=0}^{N-1} err_i \propto \sum_{i=0}^{N-1}\left(\widetilde{a_{i,k}} - a_{i,k}\right)^2, \tag{6.43}$$

考虑在对 DFT 矩阵 F_N 进行近似的求取 $\widetilde{F_N}$ 时，优化目标为:

$$\begin{aligned} \arg\min_{\beta, \widetilde{F_N}} \quad & \frac{1}{N}\|\mathbf{F}_N - \beta\widetilde{F_N}\|_F^2 \\ s.t. \quad & \widetilde{F_N}[l,m] \in \{x + jy | x, y \in \{2^0, 2^1, \cdots, 2^{q-1}\}\} \\ & l, m \in \{0, 1, \cdots, N-1\}, \end{aligned} \tag{6.44}$$

其中上述优化目标的约束条件为量化约束，定义为 Condition 2。DFT 矩阵 F_N 与 ADFT 矩阵 $\widetilde{F_N}$ 的元素是一一对应的，根据上述理论，若能有效的降低量化误差，则必定能减小最小误差。这种优化策略下，对 $\mathcal{A}$ 的优化转换为对 F_N 的优化，记作：

$$\mathcal{A} = A_1 = \widetilde{F_N}, \tag{6.45}$$

于是，我们称该算法为集中量化。由于 DFT 矩阵 F_N 为非稀疏矩阵，且存在大量的复数元素，因此可以考虑对 F_N 采用式 (5.13) 和式 (5.15) 分解的方式进行分解得到：

$$F_N = \beta WUSV^T P, \tag{6.46}$$

其中 S 为对角元素相同的对角矩阵，可以通过调节 β 的值以符合约束条件 2，P 所含有的元素值仅为 1。因此，在这种策略下，需要对 W, U, V 三个矩阵进行量化，使之皆符合约束条件 2。于是，我们将称该算法为分布优化。

由于 q 值限定，对于规范化序列中的单个元素 $\bar{x}_i = \bar{a}_i + \bar{b}_i j$ 的优化而言，需要自发的让实部 $\bar{a}_i$ 和虚部 $\bar{b}_i$ 趋近最临近的量化点。如果不进行尺度的缩放，那么以 $\omega = e^{-j\frac{2\pi}{N}}$ 为基本元素的 DFT 矩阵 F_N 内的元素的实虚部将不会被量化为非 $0, \pm 1, \pm j$ 以外的数值，从而产生严重的信息失真。因此，需要对待量化的矩阵进行缩放，即定义缩放因子 σ 为：

$$\sigma_{\bar{a}_i} = \sigma_{\bar{b}_i} = 2^{q-1}, \tag{6.47}$$

值得注意的是，这种尺度缩放是全局的，可以由 β 进行调控。在这种情况下，我们认为对最小误差 $d(\mathcal{A}, \beta)$ 等价于要求元素内量化期望最低：

$$\begin{aligned} \widetilde{a}_i, \widetilde{b}_i = & \arg\min_{\widetilde{a}_i, \widetilde{b}_i} E\left[(a_i - \widetilde{a}_i)^2 + (b_i - \widetilde{b}_i)^2\right] \\ s.t. \quad & \widetilde{a}_i \in \{x + yj | x, y \in \{1, 2, \cdots, 2^{q-1}\}\} \\ & \widetilde{a}_i \in \{x + yj | x, y \in \{1, 2, \cdots, 2^{q-1}\}\} \\ & a_i \in \sigma_{\bar{a}_i}\mathcal{A}, b_i \in \sigma_{\bar{b}_i}\mathcal{A} \end{aligned} \tag{6.48}$$

在量化约束条件下的总体算法流程图如算法3所示：

算法 3 快速量化算法

输入: 数据维度 N。

输出: 量化的矩阵簇 $\widetilde{\mathcal{A}}$，尺度系数 $\widetilde{\beta}$，最终 RMSE 误差 $\widetilde{E}$。

1: 初始化 DFT 矩阵 F_N，尺度系数 $\beta = 1$，误差矩阵 $E^{A_i} = \text{Inf}$;
2: 根据量化策略生成待量化的矩阵簇 $\mathcal{A}$ 与缩放因子 σ;
3: 采用交替乘子优化方法 ADMM 对矩阵簇内的矩阵 $A_k \in \mathcal{A}$ 交替逐步进行量化;
4: **for** t=1: 最大迭代次数 **do**
5: **for** k=1: 矩阵簇内的矩阵数量 **do**
6: $E^t_{A_k}[l, m] = \text{RMSE}(\beta\sigma A^t_1 \sigma A^t_2 \cdots \sigma A^t_k, \sigma^k, F_N)\Big|_{A^t_k[l,m]=0}, l, m \in \{0, 1, \cdots, N-1\}$;
7: $l^{A_k}_{\text{quant}}, m^{A_k}_{\text{quant}} = \arg\min_{l,m} E^t_{A_k}[l, m], \quad s.t.\text{Condition 2}$;
8: $A^t_k[l^{A_k}_{\text{quant}}, m^{A_k}_{\text{quant}}] = 0 \rightarrow A^{t+1}_k = A^t_k$
9: $\beta^{k+1} = \frac{\|\mathbf{F}_N\|_F}{\|A_1 A_2 \cdots A_k\|_F}$
10: **end**
11: **end**
12: 输出最终的量化的矩阵簇 $\widetilde{\sigma A}$ 保留迭代后的参数 $\widetilde{\beta}$;
13: 整合输出矩阵输出估计矩阵 $\widetilde{F_N} = \beta \prod_1^K \sigma A_k = \beta\sigma^k \prod_1^K A_k$;
14: 存储矩阵簇 $\mathcal{A}$ 以及尺度系数 $\widetilde{\beta} = \beta\sigma^k$;
15: 计算最终的 RMSE 误差：$\text{RMSE} = \sqrt{\|F_N - \widetilde{F_N}\|_F^2}/\text{N}$;
16: 返回：程序结束。

上述快速量化算法的迭代过程如图6.6所示：

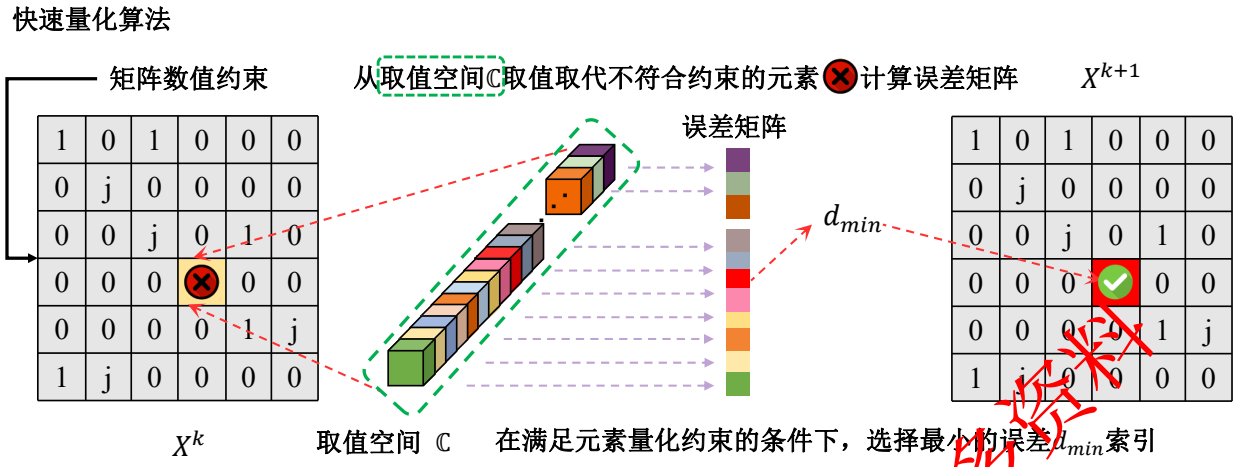


图 6.6　快速量化算法示意图

6.3 问题二的求解

6.3.1 集中量化算法

集中量化矩阵集合 $\mathcal{A}$ 仅含有一个矩阵 A_1，相当于对 DFT 矩阵 F_N 的每个元素进行量化，因此不存在硬件复杂度或者说其硬件复杂度为 0。依据算法 (3) 对 $N=2^t, t\in\{1,2,3,4,5\}$ 不同维度下的矩阵进行量化，可以得到如下结果：当 $t=1,2$ 时，分别对应 $N=2,4$，此时对应的 $\beta\big|_{t=1,2}=0.25$，矩阵可以有多种量化结果，$\widetilde{F_2}$ 和 $\widetilde{F_4}$ 如下所示：

$$\widetilde{F_2}=\begin{bmatrix}4 & 4\\ 4 & -4\end{bmatrix},\widetilde{F_4}=\begin{bmatrix}4 & 4 & 4 & 4\\ 4 & -4j & -4 & 4j\\ 4 & -4 & 4 & -4\\ 4 & 4j & -4 & -4j\end{bmatrix}, \tag{6.49}$$

当 $t=3$ 时，$N=8$，量化得到的矩阵 F_8 为：

$$\widetilde{F_8}=\begin{bmatrix}4 & 4 & 4 & 4 & 4 & 4 & 4 & 4\\ 4 & 2(1-j) & -4j & -2(1+j) & -4 & 2(j-1) & 4j & 2(j+1)\\ 4 & -4j & -4 & 4j & 4 & -4j & -4 & 4j\\ 4 & -2(1+j) & 4j & 2(1-j) & -4 & 2(j+1) & -4j & 2(j-1)\\ 4 & -4 & 4 & -4 & 4 & -4 & 4 & -4\\ 4 & 2(j-1) & -4j & 2(j+1) & -4 & 2(1-j) & 4j & -2(1+j)\\ 4 & 4j & -4 & -4j & 4 & 4j & -4 & -4j\\ 4 & 2(j+1) & 4j & 2(j-1) & -4 & -2(1+j) & -4j & 2(1-j)\end{bmatrix}, \tag{6.50}$$

显然，$\widetilde{F_8}$ 严格遵守量化约束条件，此时 $\beta = 0.26726$，所求的最小误差为 $d(\mathcal{A}, \beta) = 0.1359$。对于 t 为其他数值的情况，我们给出了相应的定性和定量结果，分别如图6.7以及表6.2所示。

表 6.2　指数 t 与相关数据的统计（集中量化算法目标函数）

t	1	2	3	4	5
β	0.25	0.25	0.26726	0.2582	0.24903
$d(\mathcal{A}, \beta)$	0	0	0.1359	0.1617	0.1588

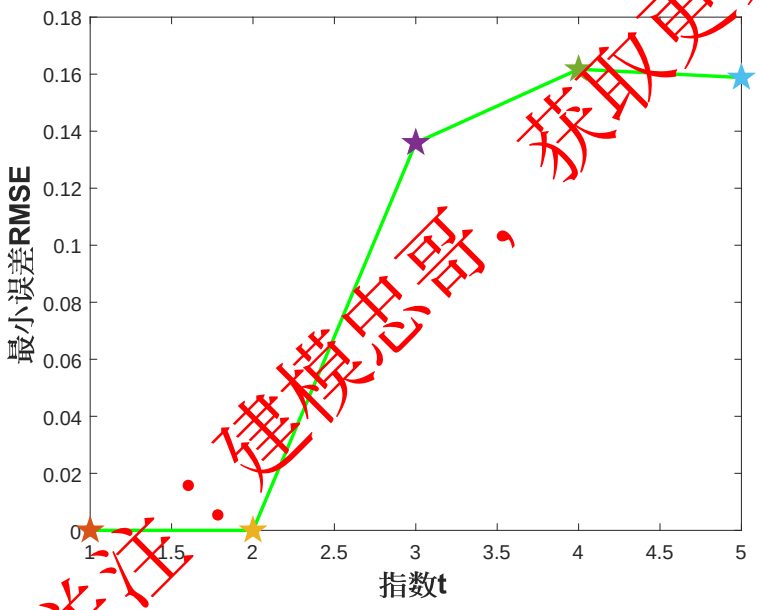


图 6.7　集中量化算法目标函数 $d(\widetilde{\mathrm{F_N}}, \mathrm{F_N})$ 随时域点数 $N = 2^t$ 的变化

6.3.2 分布优化算法

分布量化矩阵集合 $\mathcal{A}$ 包含 A_1, A_2, A_3，是将 F_N 先分解再对分解后的矩阵中的每个元素进行量化，量化后的 $\mathcal{A}$ 相乘实现对矩阵的量化优化。相比于集中优化算法，分布优化算法虽然会带来一定的算法复杂度，但是由于矩阵分解后各矩阵元素的权重相较原矩阵元素权重会变小，量化带来的误差会得到有效的收敛。依据算法3对 $N = 2^t, t \in \{1, 2, 3, 4, 5\}$ 不同维度下的矩阵进行量化，

基于 BSVD 奇异值分解算法可将 F_2 分解为：

$$F_2 = W_2 U_2 S_2 V_2^T P_2,$$

再利用算法3得到：

$$\widetilde{F_2} = \widetilde{W_2}\widetilde{U_2}\widetilde{S_2}\widetilde{V_2^T}\widetilde{P_2},$$

$$\widetilde{W_2} = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \widetilde{U_2} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \widetilde{S_2} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \widetilde{V_2^T} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \widetilde{P_2} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \tag{6.51}$$

所求的 β 依据式 (5.28) 兼并上述矩阵可得合并后的矩阵 A_1, A_2, A_3 分别为:

$$A_1 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, A_2 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, A_3 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \tag{6.52}$$

上述分解的矩阵严格符合约束 2，且满足：

$$F_2 = \beta\widetilde{F_2} = \beta A_1 A_2 A_3 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \tag{6.53}$$

根据式 (5.5) 可以计算得到当前的 ADFT 矩阵的损失为 $\mathrm{RMSE}(F_2, \beta\widetilde{F_2}) = 0$。当 $t = 2$ 时，$N = 4$，重复执行基于 BSVD 的奇异值分解算法，可将 F_4 分解为：

$$F_4 = W_4 U_4 S_4 V_4^T P_4,$$

其量化后的结果可由算法3得到：

$$\widetilde{F_4} = \widetilde{W_4}\widetilde{U_4}\widetilde{S_4}\widetilde{V_4^T}\widetilde{P_4},$$

$$\widetilde{W_4} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}, \widetilde{U_4} = \begin{bmatrix} -2 & 0 & -4 & 0 \\ -2 & 0 & 2 & 0 \\ 0 & -2 & 0 & -4 \\ 0 & -2 & 0 & 2 \end{bmatrix},$$

$$\widetilde{S_4} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \widetilde{V_4^T} = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}, \widetilde{P_4} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{6.54}$$

所求的 $\beta = 0.017728$ 依据式 (5.28) 兼并上述矩阵可得合并后的矩阵 A_1, A_2, A_3 分别为:

$$A_1 = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}, A_2 = \begin{bmatrix} -2 & 0 & -4 & 0 \\ -2 & 0 & 2 & 0 \\ 0 & -2 & 0 & -4 \\ 0 & -2 & 0 & 2 \end{bmatrix}, A_3 = \begin{bmatrix} -4 & 0 & 0 & 0 \\ 1 & -4 & 0 & 0 \\ 0 & 0 & -4 & 0 \\ 0 & 0 & 1 & -4 \end{bmatrix}, \tag{6.55}$$

上述分解的矩阵严格符合约束 2，且满足：

$$F_4 = \beta\widetilde{F_4} = \beta A_1 A_2 A_3 = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & -1 & j \\ 1 & -1 & 1 & -1 \\ 1 & j & -1 & -j \end{bmatrix}, \tag{6.56}$$

根据式 (5.5) 可以计算得到当前的 ADFT 矩阵的损失为 RMSE$(F_4, \beta\widetilde{F_4}) = 0.17877$。当 $t = 3$ 时，$N = 8$，对 F_8 进行分解可得:

$$F_8 = W_8 U_8 S_8 V_8^T P_8,$$

再利用算法3得到量化的结果为：

$$\widetilde{F_8} = \widetilde{W_8}\widetilde{U_8}\widetilde{S_8}\widetilde{V_8^T}\widetilde{P_8},$$

$$\widetilde{W_8} = \begin{bmatrix} 4 & 0 & 0 & 0 & 4 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 & 2(1-j) & 0 & 0 \\ 0 & 0 & 4 & 0 & 0 & 0 & -4j & 0 \\ 0 & 0 & 0 & 4 & 0 & 0 & 0 & -2(1+j) \\ 4 & 0 & 0 & 0 & -4 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 & 2(j-1) & 0 & 0 \\ 0 & 0 & 4 & 0 & 0 & 0 & 4j & 0 \\ 0 & 0 & 0 & 4 & 0 & 0 & 0 & 2(1+j) \end{bmatrix},$$

$$\widetilde{V_8^T} = \begin{bmatrix} -4+j & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 4+j & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -4+j & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 4+j & 0 & 0 \\ 0 & 0 & -1j & 4+j & 0 & 0 & 0 & 0 \\ 0 & 0 & 4+1j & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1j & 4+j \\ 0 & 0 & 0 & 0 & 0 & 0 & 4+j & 0 \end{bmatrix},$$

$$\widetilde{U_8} = \begin{bmatrix} -2 & 2 & 0 & 0 & 2 & 2 & 0 & 0 \\ -2 & -2j & 0 & 0 & 2j & -2 & 0 & 0 \\ -2 & -2 & 0 & 0 & -2 & 2-1j & 0 & 0 \\ -2 & 2j & 0 & 0 & -2j & -1 & 0 & 0 \\ 0 & 0 & -2 & 2 & 0 & 0 & 2 & 2 \\ 0 & 0 & -2 & -2j & 0 & 0 & 2j & -4 \\ 0 & 0 & -2 & -2 & 0 & 0 & -2 & 2-1j \\ 0 & 0 & -2 & 2j & 0 & 0 & -2j & -2 \end{bmatrix},$$

$$\widetilde{S_8} = \begin{bmatrix} 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2, \end{bmatrix}, \widetilde{P_8} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}, \tag{6.57}$$

所求的 $\dot{\beta} = 0.016328$ 依据式 (5.28) 兼并上述矩阵可得合并后的矩阵 A_1, A_2, A_3 分别为:

$$A_1 = \begin{bmatrix} 4 & 0 & 0 & 0 & 4 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 & 2(1-j) & 0 & 0 \\ 0 & 0 & 4 & 0 & 0 & 0 & -4j & 0 \\ 0 & 0 & 0 & 4 & 0 & 0 & 0 & -2(1+j) \\ 4 & 0 & 0 & 0 & -4 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 & 2(j-1) & 0 & 0 \\ 0 & 0 & 4 & 0 & 0 & 0 & 4j & 0 \\ 0 & 0 & 0 & 4 & 0 & 0 & 0 & 2(1+j) \end{bmatrix},$$

$$A_2 = \begin{bmatrix} -2 & 2 & 0 & 0 & 2 & 2 & 0 & 0 \\ -2 & -2j & 0 & 0 & 2j & -2 & 0 & 0 \\ -2 & -2 & 0 & 0 & -2 & 2-1j & 0 & 0 \\ -2 & 2j & 0 & 0 & -2j & -1 & 0 & 0 \\ 0 & 0 & -2 & 2 & 0 & 0 & 2 & 2 \\ 0 & 0 & -2 & -2j & 0 & 0 & 2j & -4 \\ 0 & 0 & -2 & -2 & 0 & 0 & -2 & 2-1j \\ 0 & 0 & -2 & 2j & 0 & 0 & -2j & -2 \end{bmatrix}, A_3 = \begin{bmatrix} -8 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 8 & 0 & 0 & 0 & 0 & 0 \\ 0 & -8 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 8 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -2j & 0 & 8 & 0 \\ 0 & 0 & 0 & 0 & 8+2j & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -2j & 0 & 8 \\ 0 & 0 & 0 & 0 & 0 & 8 & 0 & 0 \end{bmatrix},$$

上述分解的矩阵严格符合约束 2，可以求得量化后估计得到的 ADFT 矩阵 $\widetilde{F_8}$ 为：

$$\widetilde{F_8} = \begin{bmatrix} 64 & 64 & 64 & 64 & 64 & 64-16j & 64 & 64 \\ 64 & 32-32j & -64j & -32-32j & -48-16j & -56+56j & 64j & 32+32j \\ 64 & -64j & -64 & 64j & 72 & -16-64j & -64 & 64j \\ 64 & -32-32j & 64j & 32-32j & -48-8j & 40+40j & -64j & -32+32j \\ 64 & -64 & 64 & -64 & 64 & -64+16j & 64 & -64 \\ 64 & -32+32j & -64j & 32+32j & -48-16j & 56-56j & 64j & -32-32j \\ 64 & 64j & -64 & -64j & 72 & 16+64j & -64 & -64j \\ 64 & 32+32j & 64j & -32+32j & -48-8j & -40-40j & -64j & 32-32j \end{bmatrix}, \tag{6.58}$$

且满足：

$$\beta\widetilde{F_8} \approx F_8 = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \frac{\sqrt{2}}{2}(1-j) & -j & -\frac{\sqrt{2}}{2}(1+j) & -1 & \frac{\sqrt{2}}{2}(j-1) & j & \frac{\sqrt{2}}{2}(j+1) \\ 1 & -j & -1 & j & 1 & -j & -1 & j \\ 1 & -\frac{\sqrt{2}}{2}(1+j) & j & \frac{\sqrt{2}}{2}(1-j) & -1 & \frac{\sqrt{2}}{2}(j+1) & -j & \frac{\sqrt{2}}{2}(j-1) \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & \frac{\sqrt{2}}{2}(j-1) & -j & \frac{\sqrt{2}}{2}(j+1) & -1 & \frac{\sqrt{2}}{2}(1-j) & j & -\frac{\sqrt{2}}{2}(1+j) \\ 1 & j & -1 & -j & 1 & j & -1 & -j \\ 1 & \frac{\sqrt{2}}{2}(j+1) & j & \frac{\sqrt{2}}{2}(j-1) & -1 & -\frac{\sqrt{2}}{2}(1+j) & -j & \frac{\sqrt{2}}{2}(1-j) \end{bmatrix}, \tag{6.59}$$

根据式 (5.5) 可以计算得到当前的 ADFT 矩阵的损失为 $\mathrm{RMSE}(F_8, \beta\ddot{F}_8) = 0.16569$。对于 t 为其他数值的情况，我们给出了相应的定性和定量结果，分别如图6.8以及表6.3所示。

表 6.3 指数 t 与相关数据的统计（分布量化算法目标函数）

t	1	2	3	4	5
β	0.015625	0.017728	0.016328	0.016364	0.014759
$d(\mathcal{A}, \beta)$	0	0.17877	0.16569	0.28608	0.39725
$C_{\text{量化后}}$	0	0	36	90	426
$C_{\text{未量化}}$	0	0	90	504	2700

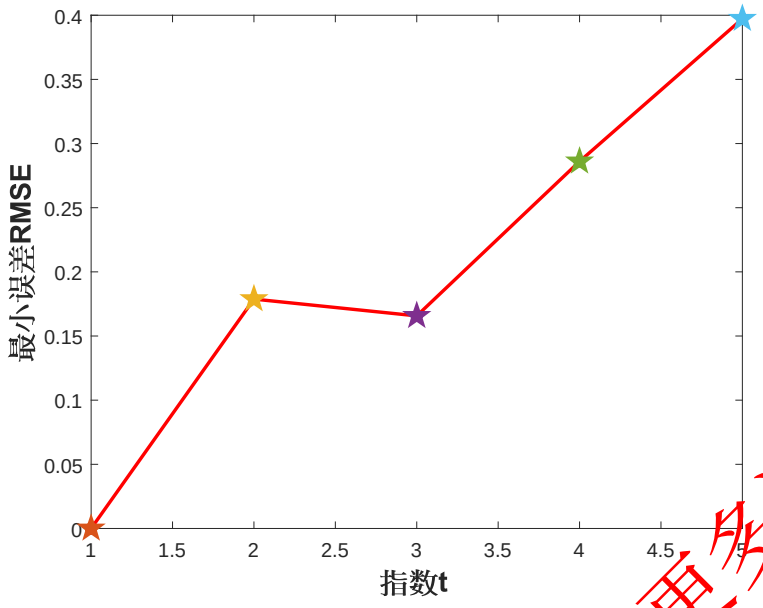


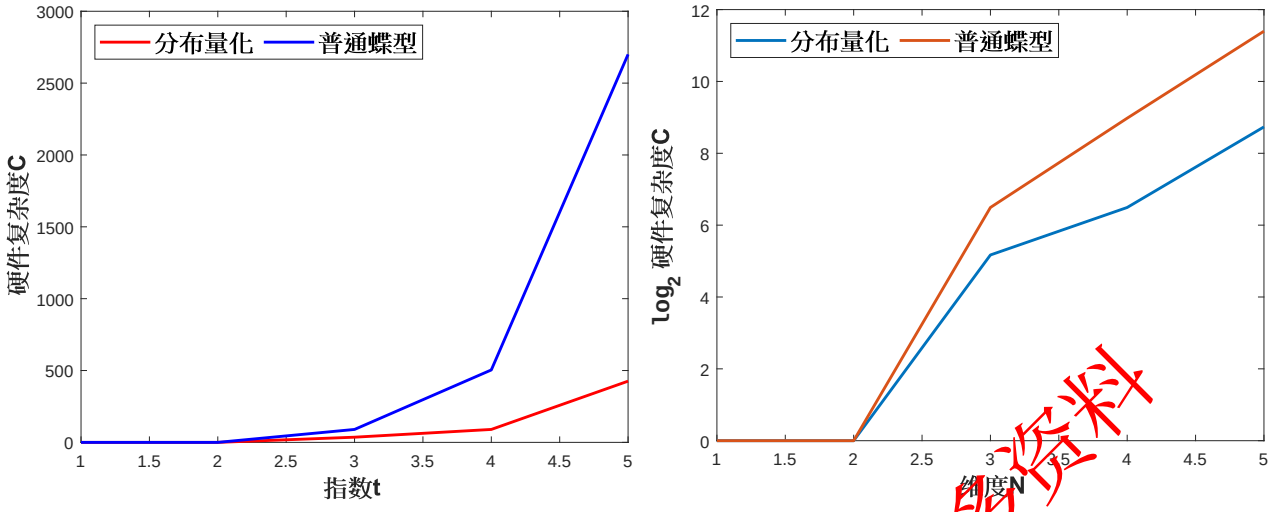
图 6.8　分布量化算法目标函数 $d(\mathcal{A}, \beta)$ 随时域点数 $N = 2^t$ 的变化

6.4 结果分析

由图6.7，图6.8和表6.2所示，对于集中量化和分布量化算法，它们的最小误差 RMSE 都随时域点数 $N = 2^t$ 的增加而增加。进一步我们可以发现，集中量化算法的 RMSE 整体小于分布式量化算法。出现这种现象的原因是集中式优化算法并未对矩阵进行分解，而分布式量化算法对矩阵进行了 SVD 分解。对 SVD 分解后的子矩阵进行量化会让更多的值发生变化，但同时，由于量化会让矩阵元素产生更多的 0、1 和 $1+i$，$1-i$,... 等特殊的形式，因此，对于算法的硬件复杂度而言，量化后的矩阵在进行乘法器运算时，其复杂度将呈现下降的趋势。

例如，由图6.9和表6.2，可以观察到，不管是否实施量化，硬件复杂度随时域点数 $N = 2^t$ 都呈明显的增长关系。尽管集中量化算法的 RMSE 整体小于分布式量化算法，但可喜的是，我们的分解算法执行量化后的复杂度有了明显的改善，特别是复杂度增加趋势远低于量化前的值。例如，由表6.2，当时域点 $N = 2^5$ 时，算法分解量化后的复杂度为 426，而量化前的复杂度为 2700，此时复杂度少了一个数量级。因此，实施量化确实能降低模型的硬件复杂度，但同时也会对模型的精度造成影响。

总的来说，量化算法在一定程度上能够有效的减少运算的硬件复杂度，但在降低复杂度的同时，也会损失模型的精度。我们的 BSVD 算法在实施量化后的精度损失并不明显，但却极大的降低了硬件复杂度，因此，是一种有效的方案。



(a) 原始坐标系　　(b) 对数坐标系

图 6.9　集中和分布量化算法硬件复杂度 C 随时域点数 $N = 2^t$ 的变化

7 问题三：稀疏约束与量化约束下 DFT 矩阵近似估计

7.1 问题三的分析

问题三要求同时满足稀疏约束与量化约束，即限制 A_k 矩阵内元素的取值范围以及行元素的个数。对于 $N = 2^t, t = 1, 2, 3, ...$ 的 DFT 矩阵 F_N，在同时满足约束 1 和约束 2 的条件下，对 $\mathcal{A}$ 和 β 进行优化，并计算最小误差和方案的硬件复杂度 C。由于同时涉及到稀疏和量化操作，因此在执行的过程中可能需要考虑稀疏和量化执行的顺序。

7.2 问题三的建模

为了简化，我们定义矩阵的稀疏操作为 $\mathcal{S}(\cdot)$ 以及矩阵的量化操作为 $\mathcal{Q}(\cdot)$。针对问题三，考虑两种合适的基本方案：方案 (I) 为先对分解后的矩阵执行稀疏化 $\mathcal{S}$ 以满足约束条件 1，然后再执行量化 $\mathcal{Q}$ 满足约束条件 2，记为 $\mathcal{S}-\mathcal{Q}$ 算法；方案 (II) 为先对分解后的矩阵执行量化 $\mathcal{Q}$ 操作，然后再对量化矩阵进行稀疏 $\mathcal{S}$，简称为 $\mathcal{Q}-\mathcal{S}$ 算法。

7.2.1 $\mathcal{S}-\mathcal{Q}$ 算法

假定 N 维 DFT 矩阵为 F_N，可以被分解为多个矩阵的乘积，记为：

$$F_N = W_N R_N P_N = W_N U_N S_N V_N^T P_N \tag{7.60}$$

其中，$W_N = \begin{bmatrix} I_{N/2} & D_{N/2} \\ I_{N/2} & -D_{N/2} \end{bmatrix}$，$I_{N/2}$ 为单位矩阵，$D_{N/2} = diag(1, \omega^1, ..., \omega^{\frac{N}{2}-1})$，$P_N$ 为排序矩阵。U_N，S_N 和 ${V_N}^T$ 是对矩阵 R_N 进行 SVD 分解后的矩阵。

需注意的是，基于我们提出的 BSVD 算法分解的矩阵 W_N、S_N 和 P_N 本身都是稀疏

的。因此，在进行稀疏化的过程时，这三个矩阵不用进行额外的稀疏操作。由于傅里叶基的特殊性质，其奇异值矩阵 S_N 的奇异值为相同值 λ，故可将该矩阵简化为 $S_N = \lambda I_N$, 其中 I_N 为 $N \times N$ 的单位矩阵，满足 $q = 3$ 的量化约束。因此，在进行量化的过程时，仅需要对 W_N, U_N, V_N 矩阵进行量化操作。类似的，我们先对矩阵 U_N 和矩阵 V_N 交替进行稀疏优化，再对稀疏化后的矩阵 $\widehat{U}_N$ 和 $\widehat{V}_N$ 以及 $\widehat{W_N}$ 进行量化优化，$\mathcal{S}-\mathcal{Q}$ 算法具体操作如算法4所示：

算法 4 $\mathcal{S}-\mathcal{Q}$ 算法

输入: 数据维度 N，数据位数 q。

输出: 量化且稀疏的矩阵簇 $\overset{*}{\mathcal{A}}$，尺度系数 $\overset{*}{\beta}$，最终 RMSE 误差 $\overset{*}{E}$。

1: 初始化 DFT 矩阵 F_N，尺度系数 $\beta = 1$，误差矩阵 $E = \text{Inf}$;
2: 对矩阵 F_N 进行蝶形-奇异值分解为，$F_N = WUSV^TP = \lambda WUV^TP$;
3: 根据算法1，对矩阵 U, V 进行稀疏化得到 $\widehat{U} = \mathcal{S}(U), \widehat{V} = \mathcal{S}(V)$ 以及尺度系数 β;
4: 根据算法3，对矩阵 $W, \widehat{U}, \widehat{V}$ 进行量化得到 $\widetilde{W} = \mathcal{Q}(W), \widetilde{U} = \mathcal{Q}(U), \widetilde{V} = \mathcal{Q}(V)$
5: 整合输出矩阵得到估计矩阵 $\overset{*}{F_N} = 2^{3(1-q)}\lambda\beta\widetilde{W}\widetilde{U}\widetilde{W}P$;
6: 存储矩阵簇 $\overset{*}{A}_1 = \widetilde{W}, \overset{*}{A}_2 = \widetilde{U}, \overset{*}{A}_3 = \widetilde{V}P$ 以及尺度系数 $\overset{*}{\beta} = 2^{3(1-q)}\lambda\beta$;
7: 计算最终的 RMSE 误差：$\text{RMSE} = \sqrt{\|F_N - \overset{*}{F_N}\|_F^2}/\text{N}$;
8: 返回：程序结束。

算法 5 $\mathcal{Q}-\mathcal{S}$ 算法

输入: 数据维度 N，数据位数 q。

输出: 量化且稀疏的矩阵簇 $\overset{*}{\mathcal{A}}$，尺度系数 $\overset{*}{\beta}$，最终 RMSE 误差 $\overset{*}{E}$。

1: 初始化 DFT 矩阵 F_N，尺度系数 $\beta = 1$，误差矩阵 $E = \text{Inf}$;
2: 对矩阵 F_N 进行蝶形-奇异值分解为，$F_N = WUSV^TP = \lambda WUV^TP$;
3: 根据算法3，对矩阵 W, U, V 进行量化得到 $\widetilde{W} = \mathcal{Q}(W), \widetilde{U} = \mathcal{Q}(U), \widetilde{V} = \mathcal{Q}(V)$ 以及尺度系数 β;
4: 根据算法1，对矩阵 $\widetilde{U}, \widetilde{V}$ 进行稀疏化得到 $\widehat{U} = \mathcal{S}(\widetilde{U}), \widehat{V} = \mathcal{S}(\widetilde{V})$;
5: 整合输出矩阵得到估计矩阵 $\overset{*}{F_N} = 2^{3(1-q)}\lambda\beta\widetilde{W}\widetilde{U}\widetilde{W}P$;
6: 存储矩阵簇 $\overset{*}{A}_1 = \widetilde{W}, \overset{*}{A}_2 = \widetilde{U}, \overset{*}{A}_3 = \widetilde{V}P$ 以及尺度系数 $\overset{*}{\beta} = 2^{3(1-q)}\lambda\beta$;
7: 计算最终的 RMSE 误差：$\overset{*}{E} = \sqrt{\|F_N - \overset{*}{F_N}\|_F^2}/\text{N}$;
8: 返回：程序结束。

7.2.2 $\mathcal{Q}-\mathcal{S}$ 算法

与 $\mathcal{S}-\mathcal{Q}$ 算法不同的是，$\mathcal{Q}-\mathcal{S}$ 算法先执行的是量化操作，然后再执行稀疏操作。同样地，在我们提出的 BSVD 算法的分解式 (7.60) 中，矩阵 S_N 和 P_N 本身实质上都可以看

成量化后的矩阵，因为其每个元素都是非复数形式且具备公共因子。因此，这两个矩阵不用执行量化过程，对其余三个矩阵执行量化操作可以得到 $\widetilde{W_N}$、$\widetilde{U_N}$ 和 $\widetilde{V_N}$。由于矩阵 W_N、S_N 和 P_N 本身都是稀疏的，量化过程不会增加矩阵的稀疏性，因此这三个矩阵不用执行后续的稀疏过程。$\mathcal{Q}-\mathcal{S}$ 算法具体操作如算法5所示。

7.3 问题三的求解

7.3.1 $\mathcal{S}-\mathcal{Q}$ 优化

根据方案 (I) 所设计的算法4，对 N 维 DFT 矩阵进行蝶形-奇异值分解算法可将 F_2 分解为：

$$F_2 = W_2 U_2 S_2 V_2^T P_2,$$

先利用算法1对上述分解后的矩阵进行稀疏处理，再利用算法3对稀疏后的新矩阵进行量化操作，得到的矩阵为：

$$\overset{*}{F}_2 = \overset{*}{W}_2 \overset{*}{U}_2 \overset{*}{S}_2 \overset{*}{V}_2^T \overset{*}{P}_2,$$

$$\overset{*}{W}_2 = 4\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}, \overset{*}{U}_2 = 4\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}, \overset{*}{S}_2 = \begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}, \overset{*}{V}_2^T = 4\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}, \overset{*}{P}_2 = \begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}, \tag{7.61}$$

所求的 $\beta = 0.015625$ 依据式 (5.28) 兼关上述矩阵可得合并后的矩阵 A_1, A_2, A_3 分别为:

$$A_1 = 4\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}, A_2 = 4\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}, A_3 = 4\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}, \tag{7.62}$$

上述分解的矩阵严格符合约束 1 和约束 2，且满足：

$$F_2 = \beta \overset{*}{F}_2 = \beta A_1 A_2 A_3 = \begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}, \tag{7.63}$$

根据式 (5.5) 可以计算得到当前的 ADFT 矩阵的损失为 $\mathrm{RMSE}(F_2, \beta\overset{*}{F}_2) = 0$。当 $t = 2$ 时，$N = 4$，重复执行基于 BSVD 奇异值分解算法可将 F_4 分解为：

$$F_4 = W_4 U_4 S_4 V_4^T P_4,$$

先利用算法1对上述分解后的矩阵进行稀疏处理，再利用算法3对稀疏后的新矩阵进行量化操作，得到的矩阵为：

$$\overset{*}{F}_4 = \overset{*}{W}_4 \overset{*}{U}_4 \overset{*}{S}_4 \overset{*}{V}_4^T \overset{*}{P}_4,$$

$$\overset{*}{W}_4 = 4\begin{bmatrix}1 & 0 & 1 & 0\\ 0 & 1 & 0 & -j\\ 1 & 0 & -1 & 0\\ 0 & 1 & 0 & j\end{bmatrix}, \overset{*}{U}_4 = \begin{bmatrix}-2 & 0 & -2 & 0\\ -2 & 0 & 2 & 0\\ 0 & -2 & 0 & -2\\ 0 & -2 & 0 & 2\end{bmatrix},$$

$$\overset{*}{S}_4 = \begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}, \overset{*}{V}_4^T = 4\begin{bmatrix}-1 & 0 & 0 & 0\\ 0 & 0 & -1 & 0\\ 0 & -1 & 0 & 0\\ 0 & 0 & 0 & -1\end{bmatrix}, \overset{*}{P}_4 = \begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 0 & 1\end{bmatrix}, \tag{7.64}$$

所求的 $\beta = 0.015625$ 依据式 (5.28) 兼并上述矩阵可得合并后的矩阵 A_1, A_2, A_3 分别为:

$$A_1 = 4\begin{bmatrix}1 & 0 & 1 & 0\\ 0 & 1 & 0 & -j\\ 1 & 0 & -1 & 0\\ 0 & 1 & 0 & j\end{bmatrix}, A_2 = \begin{bmatrix}-2 & 0 & -2 & 0\\ -2 & 0 & 2 & 0\\ 0 & -2 & 0 & -2\\ 0 & -2 & 0 & 2\end{bmatrix}, A_3 = 4\begin{bmatrix}-1 & 0 & 0 & 0\\ 1 & -1 & 0 & 0\\ 0 & 0 & -1 & 0\\ 0 & 0 & 1 & -1\end{bmatrix} \tag{7.65}$$

上述分解的矩阵严格符合约束 1 和约束 2，且满足：

$$F_4 = \beta\overset{*}{F}_4 = \beta A_1 A_2 A_3 = \begin{bmatrix}1 & 1 & 1 & 1\\ 1 & -1 & -1 & j\\ 1 & -1 & 1 & -1\\ 1 & j & -1 & -j\end{bmatrix} \tag{7.66}$$

根据式 (5.5) 可以计算得到当前的 ADFT 矩阵的损失为 RMSE$(F_4, \beta\overset{*}{F}_4) = 0$。当 $t = 3$ 时，$N = 8$，对 F_8 进行分解可得

$$F_8 = W_8 U_8 S_8 V_8^T P_8$$

先利用算法1对上述分解后的矩阵进行稀疏处理，再利用算法3对稀疏后的新矩阵进行量化操作，得到的矩阵为：

$$\overset{*}{F}_8 = \overset{*}{W}_8\overset{*}{U}_8\overset{*}{S}_8\overset{*}{V}_8^T\overset{*}{P}_8,$$

$$\overset{*}{W}_8 = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 1-j & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 & -1j & 0 \\ 0 & 0 & 0 & 4 & 0 & 0 & 0 & -(1+j) \\ 4 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 & 2(j-1) & 0 & 0 \\ 0 & 0 & 4-j & 0 & 0 & 0 & 2j & 0 \\ 0 & 0 & 0 & 4 & 0 & 0 & 0 & 2(1+j) \end{bmatrix},$$

$$\overset{*}{V}_8^T = \begin{bmatrix} -4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -4 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1+j & 4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4 & -j & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1+j & 4 \\ 0 & 0 & 0 & 0 & 0 & 0 & 4 & 0 \end{bmatrix} \overset{*}{S}_8 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1, \end{bmatrix}, \quad (7.67)$$

$$\overset{*}{U}_8 = \begin{bmatrix} -2 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & -2 & 0 & 0 \\ -1 & 0 & 0 & 0 & -2 & 1+j & 0 & 0 \\ -1 & 0 & 0 & 0 & -2j & -1 & 0 & 0 \\ 0 & 0 & -2 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & -2 & 0 & 0 & 0 & 0 & -2 \\ 0 & 0 & -2 & 0 & 0 & 0 & -4+j & 2+j \\ 0 & 0 & -2 & 0 & 0 & 0 & -2j & -1 \end{bmatrix}, \overset{*}{P}_8 = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0. & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix} \quad (7.68)$$

所求的 $\dot{\beta} = 0.008051958$，依据式 (5.28) 兼并上述矩阵可得合并后的矩阵 A_1, A_2, A_3 分别为:

$$A_1 = \begin{bmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 2 & 0 & 0 & 0 & 1-j & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 & -1j & 0 \\ 0 & 0 & 0 & 4 & 0 & 0 & 0 & -(1+j) \\ 4 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 4 & 0 & 0 & 0 & 2(j-1) & 0 & 0 \\ 0 & 0 & 4-j & 0 & 0 & 0 & 2j & 0 \\ 0 & 0 & 0 & 4 & 0 & 0 & 0 & 2(1+j) \end{bmatrix}, A_3 = \begin{bmatrix} -4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -4 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1+j & 0 & 4 & 0 \\ 0 & 0 & 0 & 0 & 4 & 0 & -j & 0 \\ 0 & 0 & 0 & 0 & 0 & 1+j & 0 & 4 \\ 0 & 0 & 0 & 0 & 0 & 4 & 0 & 0 \end{bmatrix},$$

$$A_2 = \begin{bmatrix} -2 & 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 0 & -2 & 0 & 0 \\ -1 & 0 & 0 & 0 & -2 & 1+j & 0 & 0 \\ -1 & 0 & 0 & 0 & -2j & -1 & 0 & 0 \\ 0 & 0 & -2 & 2 & 0 & 0 & 0 & 0 \\ 0 & 0 & -2 & 0 & 0 & 0 & 0 & -2 \\ 0 & 0 & -2 & 0 & 0 & 0 & -4+j & 2+j \\ 0 & 0 & -2 & 0 & 0 & 0 & -2j & -1 \end{bmatrix}, \tag{7.69}$$

上述分解的矩阵严格符合约束 1 和约束 2，可以求得估计的 ADFT 矩阵 $\overset{*}{F}_8$ 为：

$$\overset{*}{F}_8 = \begin{bmatrix} 8 & 8 & 2 & 2 & 0 & 0 & 0 & 0 \\ 8 & 8-8j & 0 & 0 & -16 & -8+8j & 4j & 0 \\ 8 & -8j & 0 & 0 & 4+4j & 1-3j & -14-2j & 4+16j \\ 16 & -8-8j & 0 & 0 & -8-8j & 4j & -28j & -8+8j \\ 32 & -8 & 8 & -2 & 0 & 0 & 0 & 0 \\ 16 & -16+16j & 0 & 0 & -32 & 16-16j & 8j & 0 \\ 16-4j & 16j & 0 & 0 & 10+6j & -12+6j & -29+3j & -8-32j \\ 16 & 16+16j & 0 & 0 & -8-8j & -8j & -28j & 16-16j \end{bmatrix} \tag{7.70}$$

且满足：

$$F_8 \approx \beta \overset{*}{F}_8 = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \frac{\sqrt{2}}{2}(1-j) & -j & -\frac{\sqrt{2}}{2}(1+j) & -1 & \frac{\sqrt{2}}{2}(j-1) & j & \frac{\sqrt{2}}{2}(j+1) \\ 1 & -j & -1 & j & 1 & -j & -1 & j \\ 1 & -\frac{\sqrt{2}}{2}(1+j) & j & \frac{\sqrt{2}}{2}(1-j) & -1 & \frac{\sqrt{2}}{2}(j+1) & -j & \frac{\sqrt{2}}{2}(j-1) \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & \frac{\sqrt{2}}{2}(j-1) & -j & \frac{\sqrt{2}}{2}(j+1) & -1 & \frac{\sqrt{2}}{2}(1-j) & j & -\frac{\sqrt{2}}{2}(1+j) \\ 1 & j & -1 & -j & 1 & j & -1 & -j \\ 1 & \frac{\sqrt{2}}{2}(j+1) & j & \frac{\sqrt{2}}{2}(j-1) & -1 & -\frac{\sqrt{2}}{2}(1+j) & -j & \frac{\sqrt{2}}{2}(1-j) \end{bmatrix} \tag{7.71}$$

根据式 (5.5) 可以计算得到当前的 ADFT 矩阵的损失为 $\mathrm{RMSE}(F_8, \beta\overset{*}{F}_8) = 0.894968919$。

表 7.4　指数 t 与相关数据的统计（$\mathcal{S}-\mathcal{Q}$ 算法）

t	1	2	3	4	5
β	0.015625	0.015625	0.008051958	0.003709898	0.011447678
$d(\mathcal{A},\beta)$	0	0	0.894968919	1.031063153	1.104393708
$C_{\mathcal{S}-\mathcal{Q}}$	0	0	3	513	1917
C_B	0	0	90	588	2700

对于 t 为其他数值的情况，我们给出了相应的定性和定量结果，其中定量结果如表6.3所示。定性结果给出了最小误差 $d(\mathcal{A},\beta)$ 随指数 t 的变化如图7.10所示。执行 $\mathcal{S}-\mathcal{Q}$ 算法硬件复杂度 $C_{\mathcal{S}-\mathcal{Q}}$ 以及未执行算法的硬件复杂度 C_{B} 随时域点数 $N=2^t$ 的变化如图7.11所示，为了彰显差异，我们在原始坐标系和对数坐标系内分别进行了绘制，对应图7.11a和图7.11b。

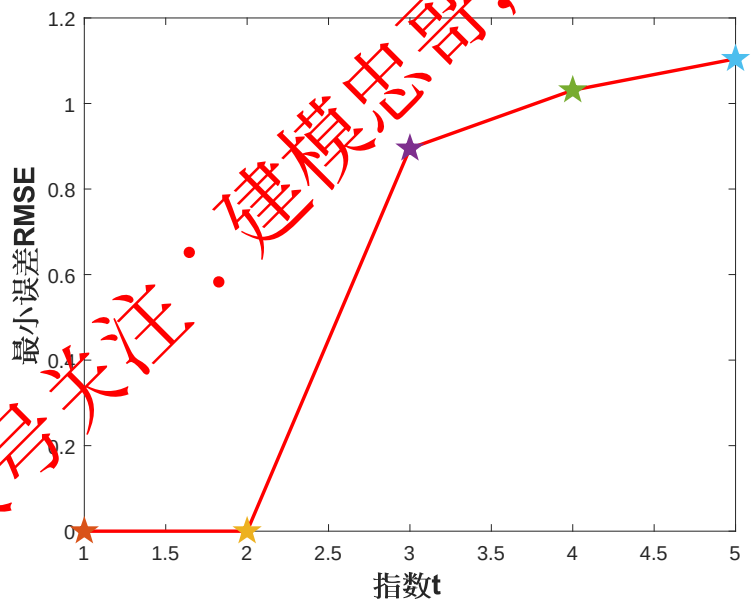


图 7.10　$d(\mathcal{A},\beta)$ 随时域点数 $N=2^t$ 的变化（$\mathcal{S}-\mathcal{Q}$ 算法）

7.3.2 $\mathcal{Q}-\mathcal{S}$ 优化

根据方案 (II) 所设计的算法5，对 N 维 DFT 矩阵进行蝶形-奇异值分解算法可将 F_2 分解为：

$$F_2 = W_2 U_2 S_2 V_2^T P_2,$$

再先利用算法3对前述分解后的矩阵进行量化操作，最后利用算法1量化后的新矩阵完成稀疏处理, 可得：

$$\overset{*}{F}_2 = \overset{*}{W}_2 \overset{*}{U}_2 \overset{*}{S}_2 \overset{*}{V}_2^T \overset{*}{P}_2,$$

$$\overset{*}{W}_2 = 4\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}, \overset{*}{U}_2 = 4\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}, \overset{*}{S}_2 = \begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}, \overset{*}{V}_2^T = 4\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}, \overset{*}{P}_2 = \begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}, \tag{7.72}$$

所求的 $\beta = 0.015625$ 依据式 (5.28) 兼并上述矩阵可得合并后的矩阵 A_1, A_2, A_3 分别为:

$$A_1 = 4\begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}, A_2 = 4\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}, A_3 = 4\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}, \tag{7.73}$$

上述分解的矩阵严格符合约束 1 和约束 2，且满足：

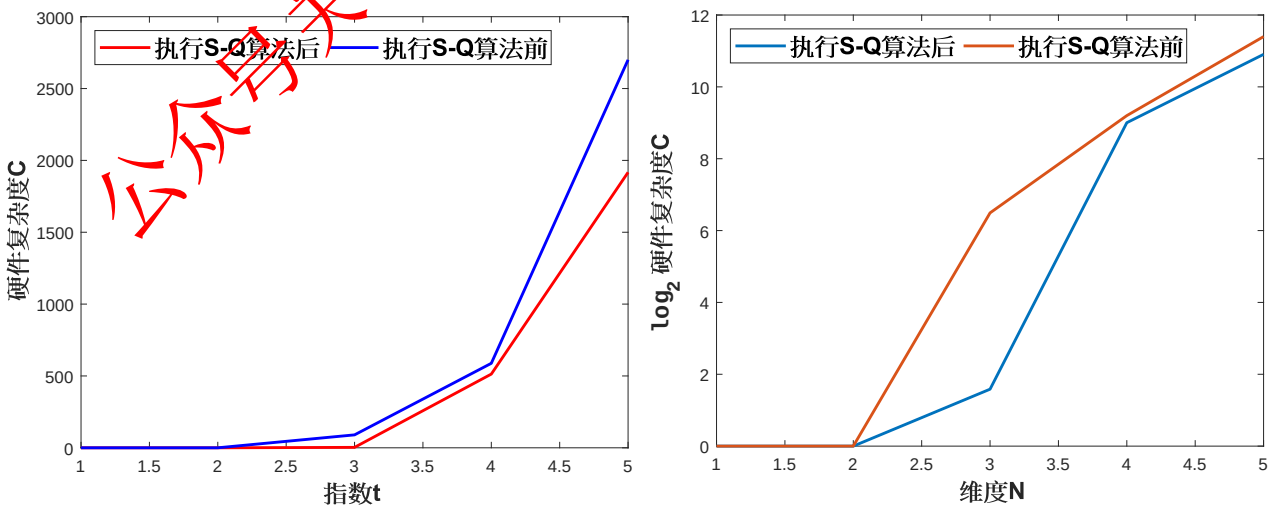
$$F_2 = \beta \overset{*}{F}_2 = A_1 A_2 A_3 = \begin{bmatrix}1 & 1\\ 1 & -1\end{bmatrix}, \tag{7.74}$$

根据式 (5.5) 可以计算得到当前的 ADFT 矩阵的损失为 $\mathrm{RMSE}(F_2, \beta \overset{*}{F}_2) = 0$。当 $t = 2$ 时，$N = 4$，重复执行基于 BSVD 奇异值分解算法可将 F_4 分解为：

$$F_4 = W_4 U_4 S_4 V_4^T P_4,$$

再先利用算法3对前述分解后的矩阵进行量化操作，最后利用算法1量化后的新矩阵完成稀疏处理, 可得：

$$\overset{*}{F}_4 = \overset{*}{W}_4 \overset{*}{U}_4 \overset{*}{S}_4 \overset{*}{V}_4^T \overset{*}{P}_4,$$



(a) 原始坐标系　　(b) 对数坐标系

图 7.11　硬件复杂度 C 随时域点数 $N = 2^t$ 的变化（$\mathcal{S} - \mathcal{Q}$ 算法）

$$\overset{*}{W}_4 = 4\begin{bmatrix}1&0&1&0\\0&1&0&-j\\1&0&-1&0\\0&1&0&j\end{bmatrix}, \overset{*}{U}_4 = \begin{bmatrix}-2&0&-2&0\\-2&0&2&0\\0&-2&0&-2\\0&-2&0&2\end{bmatrix},$$

$$\overset{*}{S}_4 = \begin{bmatrix}1&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix}, \overset{*}{V}_4^T = \begin{bmatrix}-1&0&0&0\\0&0&-1&0\\0&-1&0&0\\0&0&0&-1\end{bmatrix}, \overset{*}{P}_4 = \begin{bmatrix}1&0&0&0\\0&0&1&0\\0&1&0&0\\0&0&0&1\end{bmatrix}, \tag{7.75}$$

所求的 $\beta = 0.015625$ 依据式 (5.28) 兼并上述矩阵可得合并后的矩阵 A_1, A_2, A_3 分别为:

$$A_1 = \begin{bmatrix}1&0&1&0\\0&1&0&0\\1&0&-1&0\\0&1&0&0\end{bmatrix}, \quad A_2 = \begin{bmatrix}-2&0&-4&0\\-2&0&2&0\\0&-2&0&-4\\0&-2&0&2\end{bmatrix}, A_3 = \begin{bmatrix}-4&0&0&0\\1&-4&0&0\\0&0&-4&0\\0&0&1&-4\end{bmatrix} \tag{7.76}$$

上述分解的矩阵严格符合约束 1 和约束 2，且满足：

$$F_4 = \beta\overset{*}{F}_4 = \beta A_1 A_2 A_3 = \begin{bmatrix}1&1&1&1\\1&-1&-1&j\\1&-1&1&-1\\1&j&-1&-j\end{bmatrix} \tag{7.77}$$

根据式 (5.5) 可以计算得到当前的 ADFT 矩阵的损失为 $\mathrm{RMSE}(F_4, \beta\overset{*}{F}_4) = 0$。当 $t = 3$ 时，$N = 8$，对 F_8 进行分解可得：

$$F_8 = W_8 U_8 S_8 V_8^T P_8$$

再先利用算法3对前述分解后的矩阵进行量化操作，最后利用算法1量化后的新矩阵完成稀疏处理, 可得:

$$\overset{*}{F}_8 = \overset{*}{W}_8 \overset{*}{U}_8 \overset{*}{S}_8 \overset{*}{V}_8^T \overset{*}{P}_8,$$

$$\overset{*}{W}_8 = \begin{bmatrix}4&0&0&0&4&0&0&0\\0&4&0&0&0&2-2j&0&0\\0&0&4&0&0&0&-4j&0\\0&0&0&4&0&0&0&-2-2j\\4&0&0&0&-4&0&0&0\\0&4&0&0&0&-2+2j&0&0\\0&0&4&0&0&0&4j&0\\0&0&0&4&0&0&0&2+2j\end{bmatrix}, \overset{*}{U}_8 = \begin{bmatrix}0&0&0&0&2&2&0&0\\-2&0&0&0&0&-2&0&0\\0&0&0&0&-2&2-j&0&0\\-2&0&0&0&-2j&0&0&0\\0&0&-2&2&0&0&0&0\\0&0&0&0&0&0&2j&-4\\0&0&-2&0&0&0&-2&0\\0&0&-2&0&0&0&-2j&0\end{bmatrix}$$

$$V_8^{*T}=\begin{bmatrix}-4&0&0&0&0&0&0&0\\0&4&0&0&0&0&0&0\\0&0&0&0&-4&0&0&0\\0&0&0&0&0&4&0&0\\0&0&j&4&0&0&0&0\\0&0&4+j&0&0&0&0&0\\0&0&0&0&0&0&j&4\\0&0&0&0&0&0&4&0\end{bmatrix},S_8^{*}=\begin{bmatrix}2&0&0&0&0&0&0&0\\0&2&0&0&0&0&0&0\\0&0&2&0&0&0&0&0\\0&0&0&2&0&0&0&0\\0&0&0&0&2&0&0&0\\0&0&0&0&0&2&0&0\\0&0&0&0&0&0&2&0\\0&0&0&0&0&0&0&2,\end{bmatrix},P_8^{*}=\begin{bmatrix}1&0&0&0&0&0&0&0\\0&0&1&0&0&0&0&0\\0&0&0&0&1&0&0&0\\0&0&0&0&0&0&1&0\\0&1&0&0&0&0&0&0\\0&0&0&1&0&0&0&0\\0&0&0&0&0&1&0&0\\0&0&0&0&0&0&0&1\end{bmatrix}\tag{7.78}$$

所求的 $\beta=0.011798229$ 依据式 (5.28) 兼并上述矩阵可得合并后的矩阵 A_1,A_2,A_3 分别为:

$$A_1=\begin{bmatrix}4&0&0&0&4&0&0&0\\0&4&0&0&0&2-2j&0&0\\0&0&4&0&0&0&-4j&0\\0&0&0&4&0&0&0&-2-2j\\4&0&0&0&-4&0&0&0\\0&4&0&0&0&-2+2j&0&0\\0&0&4&0&0&0&4j&0\\0&0&0&4&0&0&0&2+2j\end{bmatrix},$$

$$A_2=\begin{bmatrix}0&0&0&0&2&2&0&0\\-2&0&0&0&0&-2&0&0\\0&0&0&0&-2&2-j&0&0\\-2&0&0&0&-2j&0&0&0\\0&0&-2&2&0&0&0&0\\0&0&0&0&0&0&2j&-4\\0&0&-2&0&0&0&-2&0\\0&0&-2&0&0&0&-2j&0\end{bmatrix},A_3=\begin{bmatrix}-4&0&0&0&0&0&0&0\\0&0&4&0&0&0&0&0\\0&-4&0&0&0&0&0&0\\0&0&0&4&0&0&0&0\\0&0&0&0&0-j&0&4&0\\0&0&0&0&4+j&0&0&0\\0&0&0&0&0&0-j&0&4\\0&0&0&0&0&4&0&0\end{bmatrix},$$

上述分解的矩阵严格符合约束 1 和约束 2，可以求得估计的 ADFT 矩阵 $\overset{*}{F}_8$ 为：

$$\overset{*}{F}_8 = \begin{bmatrix} 0 & 32 & 0 & 32 & 32 & 0 & 32 & 0 \\ 32 & 0 & 0 & 0 & -32-8j & -28+28j & 0 & 16+16j \\ 0 & -32j & 0 & 0 & 36 & 8 & -32 & 32j \\ 32 & -16-16j & 0 & 0 & -8 & 4+4j & -32j & -16+16j \\ 0 & -32 & 0 & -32 & 32 & 0 & 32 & 0 \\ 32 & 0 & 0 & 0 & -32-8j & 28-28j & 0 & -16-16j \\ 0 & 32j & 0 & 0 & 36 & -8 & -32 & -32j \\ 32 & 16+16j & 0 & 0 & -8 & -4-4j & -32j & 16-16j \end{bmatrix}, \tag{7.79}$$

且满足：

$$\beta\overset{*}{F}_8 \approx F_8 = \begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \frac{\sqrt{2}}{2}(1-j) & -j & -\frac{\sqrt{2}}{2}(1+j) & -1 & \frac{\sqrt{2}}{2}(j-1) & j & \frac{\sqrt{2}}{2}(j+1) \\ 1 & -j & -1 & j & 1 & -j & -1 & j \\ 1 & -\frac{\sqrt{2}}{2}(1+j) & j & \frac{\sqrt{2}}{2}(1-j) & -1 & \frac{\sqrt{2}}{2}(j+1) & -j & \frac{\sqrt{2}}{2}(j-1) \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & \frac{\sqrt{2}}{2}(j-1) & -j & \frac{\sqrt{2}}{2}(j+1) & -1 & \frac{\sqrt{2}}{2}(1-j) & j & -\frac{\sqrt{2}}{2}(1+j) \\ 1 & j & -1 & -j & 1 & j & -1 & -j \\ 1 & \frac{\sqrt{2}}{2}(j+1) & j & \frac{\sqrt{2}}{2}(j-1) & -1 & -\frac{\sqrt{2}}{2}(1+j) & -j & \frac{\sqrt{2}}{2}(1-j) \end{bmatrix}, \tag{7.80}$$

根据式 (5.5) 可以计算得到当前的 ADFT 矩阵的损失为 $\text{RMSE}(F_8, \beta\overset{*}{F}_8) = 0.851951297$。

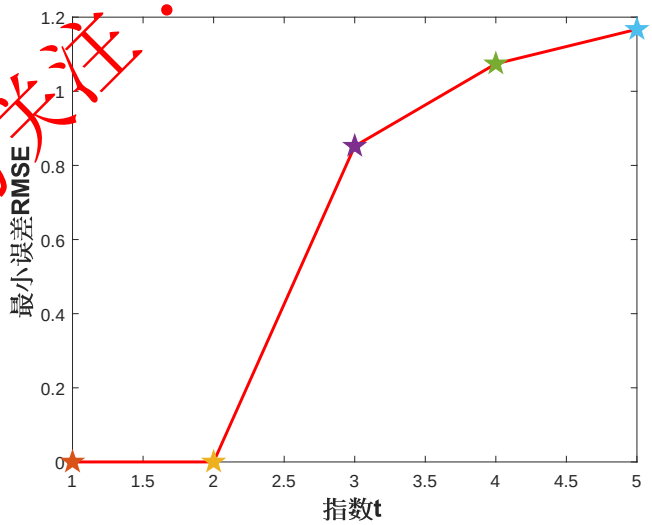
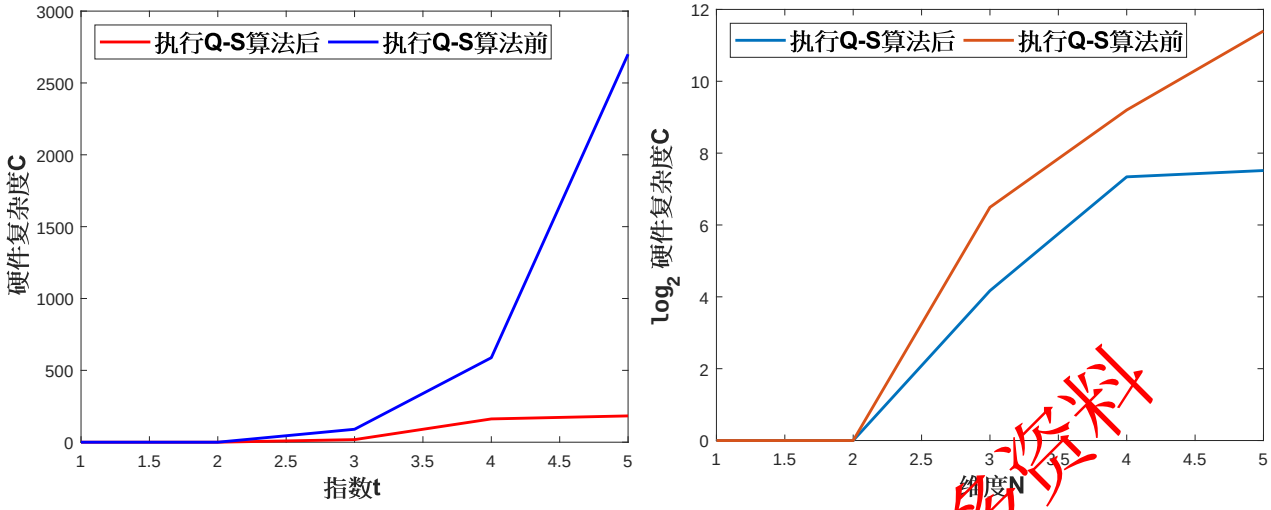


图 7.12　$d(\mathcal{A}, \beta)$ 随时域点数 $N = 2^t$ 的变化（$\mathcal{Q}-\mathcal{S}$ 算法）



(a) 原始坐标系　　(b) 对数坐标系

图 7.13　硬件复杂度 C 随时域点数 $N = 2^t$ 的变化（$\mathcal{Q} - \mathcal{S}$ 算法）

对于 t 为其他数值的情况，我们给出了相应的定性和定量结果，其中定量结果如表7.5所示。定性结果给出了最小误差 $d(\mathcal{A}, \beta)$ 随指数 t 的变化如图7.13所示。执行 $\mathcal{Q} - \mathcal{S}$ 算法硬件复杂度 $C_{\mathcal{Q}-\mathcal{S}}$ 以及未执行算法的硬件复杂度 C_{B} 随时域点数 $N = 2^t$ 的变化如图7.11所示，为了彰显差异，我们在原始坐标系和对数坐标系内分别进行了绘制，对应图7.13a和图7.13b。

表 7.5　指数 t 与相关数据的统计（$\mathcal{Q} - \mathcal{S}$ 算法）

t	1	2	3	4	5
β	0.015625	0.015625	0.011798229	0.010498037	0.014008124
$d(\mathcal{A}, \beta)$	0	0	0.851951297	1.074202401	1.167236415
$C_{\mathcal{Q}-\mathcal{S}}$	0	0	18	162	183
C_{B}	0	0	90	588	2700

7.4 结果分析

对于 $\mathcal{S} - \mathcal{Q}$ 算法，由图 (7.10) 图 (7.11) 和表 (7.4) 可知，与未实施 $\mathcal{S} - \mathcal{Q}$ 算法的模型对比，执行该算法后的 BSVD 模型在复杂度上具有极其显著的优势。整体上，实施稀疏量化后的算法复杂度远低于未实施稀疏量化的模型。且当时域点 $N = 2^t$ 增加时，这种优势更加明显。例如，在时域点 $N = 2^5$ 时，未实施稀疏量化的算法模型复杂度为 2700，而实施稀疏量化后的算法模型复杂度为 312。

对于 $\mathcal{Q}-\mathcal{S}$ 算法，由图 (7.12) 图 (7.13) 和表 (7.5) 可知，对比于未实施 $\mathcal{Q}-\mathcal{S}$ 算法的模型，执行该算法后的 BSVD 模型在复杂度上也同样具有极其显著的优势。整体上，实施量化稀疏后的算法复杂度远低于未实施量化稀疏的模型。此时，执行 $\mathcal{Q}-\mathcal{S}$ 算法前后，模型复杂度变化最大的地方出现在时域点 $N=2^5$ 时，未实施量化稀疏的算法模型复杂度为 2700，而实施量化稀疏后的算法模型复杂度为 159。

对于先稀疏后量化 $\mathcal{S}-\mathcal{Q}$ 和先量化后稀疏 $\mathcal{Q}-\mathcal{S}$ 两种算法而言，先稀疏后量化 $\mathcal{S}-\mathcal{Q}$ 在复杂度上具有一定优势，但其精度却小于先量化后稀疏 $\mathcal{Q}-\mathcal{S}$ 的算法。例如时域点 $N=2^5$ 时，先稀疏后量化 $\mathcal{S}-\mathcal{Q}$ 的算法模型复杂度为 159，最小误差为 1.4013，而先量化后稀疏 $\mathcal{Q}-\mathcal{S}$ 的算法模型复杂度为 312，最小误差为 1.1381。因此，可以再次发现，当复杂度降低时，模型的精度也会下降。

总的来说，$\mathcal{S}-\mathcal{Q}$ 算法在一定程度上能够更有效的减少运算的硬件复杂度，但在降低复杂度的同时，也会损失模型的精度。类似地，$\mathcal{Q}-\mathcal{S}$ 算法在一定程度上能够尽量的保证模型的精度，但同时也会增加模型的计算复杂度。实际上，$\mathcal{S}-\mathcal{Q}$ 算法和 $\mathcal{Q}-\mathcal{S}$ 算法的最主要区别在于：稀疏是信息丢失，量化是信息损失。由于稀疏强制把一些元素变为 0，而量化是将元素变为与其最近邻的整数值。因此，在多数时候，先量化后稀疏 $\mathcal{Q}-\mathcal{S}$ 的算法比先稀疏后量化 $\mathcal{S}-\mathcal{Q}$ 的算法精度更高，这也是我们得出的一个重要结论。

8 问题四：*Kronecker* 积与稀疏量化约束条件下的 DFT 矩阵估计

8.1 问题四的分析

问题四要求考虑矩阵 $F_N = F_{N_1} \otimes F_{N_2}$ 在稀疏约束与量化约束条件下的近似求解，其中 F_{N_1} 和 F_{N_2} 分别是 N_1 和 N_2 维的 DFT 矩阵，$\otimes$ 表示 *Kronecker* 积。此外，本题设定的时域点维度分别为 $N_1=4, N_2=8$，此时的 DFT 矩阵表述为 $F_N = F_4 \otimes F_8$。在同时满足约束 1 和 2 的条件下，我们需要对 $\mathcal{A}$ 和 β 进行优化，并优化方案的计算最小误差和硬件复杂度 C。

8.2 问题四的建模

给定的 DFT 矩阵 $F_N = F_4 \otimes F_8$，其中 $\otimes$ 为克罗内克 (*Kronecker*) 积。为了在稀疏约束条件和量化约束条件下求取相应的近似 DFT 矩阵，我们首先分析 *Kronecker* 积的性质。设任意矩阵 A 是一个 $m \times n$ 的矩阵，B 是一个 $p \times q$ 的矩阵，*Kronecker* 积定义为矩阵 $A_{m\times n}$ 的各个元素分别乘以矩阵 $B_{p\times q}$，即：

$$A \otimes B = \begin{bmatrix} a_{11}B & a_{12}B & \cdots & a_{1n}B \\ a_{21}B & a_{22}B & \cdots & a_{2n}B \\ \cdots & \cdots & \ddots & \cdots \\ a_{m1}B & a_{m2}B & \cdots & a_{mn}B \end{bmatrix}, \tag{8.81}$$

不难发现，$A \otimes B$ 的最终结果是一个大小为 $mp \times nq$ 的矩阵。*Kronecker* 积的一个重要性质被称为“混合乘积性”。该性质混合了普通的矩阵乘积以及 *Kronecker* 积的运算。假设 A, B, C 和 D 是四个具有相同维度的方型矩阵，且矩阵乘积 AB 和 CD 存在，那么 *Kronecker* 混合乘积满足如下等式：

$$AB \otimes CD = (A \otimes C)(B \otimes D) \tag{8.82}$$

由上述性质可知，基于 *Kronecker* 积生成的矩阵 $F_N = F_{N_1} \otimes F_{N_2}, N_1 = 4, N_2 = 8$，是一个行列大小为 32 的矩阵。由于矩阵的 *Kronecker* 积运算与寻常的矩阵乘积不同，它并非简单的矩阵乘法，但本质上其可以看作是矩阵的扩张，并且作用完后得到的结果仍为矩阵。这表明，我们可以从两个角度来逼近目标矩阵 F_N，即考虑先求解矩阵的 *Kronecker* 积，再进行矩阵分解，还是先考虑矩阵分解，再求解其 *Kronecker* 积。故我们给出了以下三种方案来求解在稀疏约束条件及量化约束条件下目标矩阵 F_N 的近似估计 $\hat{F}_N^*$：

- **方案一:** 求出 $F_4 \otimes F_8$ 作用后的矩阵，再对该矩阵进行量化和稀疏。
- **方案二:** 对 F_4 和 F_8 分别进行量化和稀疏，然后求解稀疏矩阵的 *Kronecker* 积。
- **方案三:** 先对 F_4 和 F_8 进行分解，并利用 *Kronecker* 混合积的性质对矩阵乘积进行简化，最后再对简化后的矩阵进行量化和稀疏，并求解最终的 *Kronecker* 积。

分析可知，方案一和方案二未涉及到 *Kronecker* 积的性质，只需要依次执行量化和稀疏过程即可。下面我们的把重点聚焦在方案三，由 *Kronecker* 混合积性质有：

$$\begin{aligned} F_N &= F_4 \otimes F_8 \\ &= (W_4 R_4 P_4) \otimes (W_8 R_8 P_8) \\ &= (W_4 \otimes W_8)(R_4 \otimes R_8)(P_4 \otimes P_8) \\ &= (W_4 \otimes W_8)(U_4 S_4 {V_4}^T) \otimes (U_8 S_8 {V_8}^T)(P_4 \otimes P_8) \\ &= (W_4 \otimes W_8)(U_4 \otimes {V_8}^T)(S_4 \otimes S_8)(V_4^T \otimes {V_8}^T)(P_4 \otimes P_8) \end{aligned} \tag{8.83}$$

其中，$W_{2n} = \begin{bmatrix} I_n & D_n \\ I_n & -D_n \end{bmatrix}$，$I_n$ 为单位矩阵，$D_n = diag(1, \omega^1, ..., \omega^{n-1})$，$P_{2n}$ 为排序矩阵。U_{2n}，S_{2n} 和 ${V_{2n}}^T$ 是对矩阵 R_{2n} 进行 SVD 分解后的矩阵，n 取值为 $2, 4$。注意到矩阵 P_{2n} 和 S_{2n} 具有天然的稀疏性、正交性和元素非复数性（即矩阵每行每列同时满足只有一个非零的实数元素，其余位置都为零值），所以 $(S_4 \otimes S_8)$ 和 $(P_4 \otimes P_8)$ 的最终结果必定仍为满足稀疏条件约束以及量化条件约束。因此这两项 *Kronecker* 积不用考虑后续的量化和稀疏处理，而仅需考虑对矩阵 $(R_4 \otimes R_8)$，$(U_4 \otimes U_8)$ 和 $(V_4^T \otimes {V_8}^T)$ 进行稀疏和量化处理，然后再求解分解后的最小损失并计算其复杂度。

总的来说，方案三执行的步骤如下：

(1). 对矩阵 F_4 和 F_8 进行分别进行蝶形-奇异值（BSVD）分解;

(2). 依次求解 $(R_4 \otimes R_8)$，$(U_4 \otimes U_8)$ 和 $(V_4^T \otimes {V_8}^T)$ 的 *Kronecker* 积结果，可得三个矩阵;

(4). 使用算法1，对 *Kronecker* 积作用后的三个结果矩阵进行量化处理;

(5). 使用算法3，对量化后的矩阵进行稀疏处理。

表 8.6　不同方案的相关数据的统计

	方案一	方案二	方案三
β	1	1	0.015625
$d(\mathcal{A},\beta)$	1.224744871391589	1.224744871391589	0.9741532781728846
C	0	0	36

8.3 问题四的求解

方案一与方案二的求解不涉及中间变量，属于端到端的求解模式，其定量结果如表8.6所示。

下面我们详细讨论方案三的计算过程，首先对于 4 维的 DFT 矩阵 F_4，我们进行蝶形-奇异值分解得到下式及对应的五个子矩阵：

$$F_4 = W_4 U_4 S_4 V_4^T P_4,$$

$$A_4 = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & -j \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & j \end{bmatrix}, U_4 = \begin{bmatrix} -\frac{\sqrt{2}}{2} & 0 & -\frac{\sqrt{2}}{2} & 0 \\ -\frac{\sqrt{2}}{2} & 0 & \frac{\sqrt{2}}{2} & 0 \\ 0 & -\frac{\sqrt{2}}{2} & 0 & -\frac{\sqrt{2}}{2} \\ 0 & -\frac{\sqrt{2}}{2} & 0 & \frac{\sqrt{2}}{2} \end{bmatrix},$$

$$S_4 = \sqrt{2}\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, V_4 = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix}, P_4 = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{8.84}$$

类似的，对于 8 维度的 DFT 矩阵 F_8，其分解为：

$$F_8 = W_8 U_8 S_8 V_8^T P_8,$$

$$W_8=\begin{bmatrix}1&0&0&0&1&0&0&0\\0&1&0&0&0&\frac{\sqrt{2}}{2}(j-1)&0&0\\0&0&1&0&0&0&-j&0\\0&0&0&1&0&0&0&-\frac{\sqrt{2}}{2}(1+j)\\1&0&0&0&-1&0&0&0\\0&1&0&0&0&\frac{\sqrt{2}}{2}(j-1)&0&0\\0&0&1&0&0&0&j&0\\0&0&0&1&0&0&0&\frac{\sqrt{2}}{2}(1+j)\end{bmatrix},S_8=\begin{bmatrix}2&0&0&0&0&0&0&0\\0&2&0&0&0&0&0&0\\0&0&2&0&0&0&0&0\\0&0&0&2&0&0&0&0\\0&0&0&0&2&0&0&0\\0&0&0&0&0&2&0&0\\0&0&0&0&0&0&2&0\\0&0&0&0&0&0&0&2\end{bmatrix},$$

$$U_8=\begin{bmatrix}-0.5&0.5&0&0&0.490+0.098j&0.499+0.028j&0&0\\-0.5&-0.5j&0&0&0.392j&-0.582+0.083j&0&0\\-0.5&-0.5&0&0&-0.490+0.098j&0.471-0.167j&0&0\\-0.5&0.500j&0&0&-0.000-0.588j&-0.388+0.056j&0&0\\0&0&-0.5&0.5&0&0&0.490+0.098j&0.499+0.028j\\0&0&-0.5&-0.5j&0&0&0.392j&-0.582+0.083j\\0&0&-0.5&-0.5&0&0&-0.490+0.098j&0.471-0.167j\\0&0&-0.5&0.500j&0&0&-0.000-0.588j&-0.388+0.056j\end{bmatrix},\tag{8.85}$$

$$V_8=\begin{bmatrix}-1&0&0&0&0&0&0&0\\0&1&0&0&0&0&0&0\\0&0&0&0&0.196j&0.971-0.139j&0&0\\0&0&0&0&0.981&0.028+0.194j&0&0\\0&0&-1&0&0&0&0&0\\0&0&0&1&0&0&0&0\\0&0&0&0&0&0&0.196j&0.971-0.139j\\0&0&0&0&0&0&0.981&0.028+0.194j\end{bmatrix},P_8=\begin{bmatrix}1&0&0&0&0&0&0&0\\0&0&1&0&0&0&0&0\\0&0&0&0&1&0&0&0\\0&0&0&0&0&0&1&0\\0&1&0&0&0&0&0&0\\0&0&0&1&0&0&0&0\\0&0&0&0&0&1&0&0\\0&0&0&0&0&0&0&1\end{bmatrix},$$

在根据式 (8.83) 进行对矩阵 W,U,S,V,P 进行张量积的计算，得到相应的矩阵记为：

$$W_N=W_4\otimes W_8,U_N=U_4\otimes U_8,S_N=S_4\otimes S_8,V_N\ =V_4^T\otimes V_8^T,P_N=P_4\otimes P_8,\tag{8.86}$$

再对上述矩阵进行稀疏和量化得到对应的近似矩阵 $\overset{*}{W}_N,U_N,\overset{*}{S}_N,\overset{*}{V}_N^T,\overset{*}{P}_N$，最终经过求解可得估计矩阵的近似矩阵为 $\overset{*}{F}_N=\overset{*}{W}_N\overset{*}{U}_N\overset{*}{S}_N\overset{*}{V}_N^T\overset{*}{P}_N$。

8.4 结果分析

由表8.6可知，对于三种方案，精度最高的是方案三，即利用 *Kronecker* 混合积性质对矩阵乘积进行简化，然后再执行量化和稀疏以及 *Kronecker* 积的求解。此时，方案三的 RMSE

为 0.9742，而其余两种方法 RMSE 值都为 1.2447。尽管方案三的硬件复杂度为 36，其余两种方法复杂度都为 0。但对比于精度的提升，该计算的复杂度并不算太大，基本上可以忽略。实际上，方案一和方案二本质上都没有对矩阵进行分解，而是直接对 *Kronecker* 积矩阵进行稀疏和量化。由矩阵 F_4，F_8 的特殊形式以及 *Kronecker* 积性质可知，先执行 *Kronecker* 积或者先对 F_4 和 F_8 进行量化和稀疏并不会对其复杂度造成影响，故此时它们的计算复杂度都为 0。因此，在该条件下，我们的算法能够有效的提高模型精度，同时并不会带来较大的硬件复杂度。

9 问题五：稀疏量化约束下限制精度的 DFT 矩阵估计

9.1 问题五的分析

在降低硬件复杂度的同时，无法避免的回带来一定的精度误差，从而影响芯片功能的正常执行。因此，我们希望能够在可接受的误差范围内对 DFT 矩阵进行近似，即进一步研究硬件复杂度 C 和最小误差 $d(\mathcal{A},\beta)$ 之间的权衡关系。我们需要在稀疏约束条件及量化约束条件下，对 $\mathcal{A}$ 和 $\beta,\mathcal{P}$ 进行优化，并尽可能的降低该方案的硬件复杂度 C 并将求解精度限制在 0.1 以内，满足即 $\text{RMSE}(\mathcal{A},\beta)\leq 0.1$。

9.2 问题五的建模

为了尽可能的逼近极限 RMSE 的数值，在降低误差并提升精度的同时，满足矩阵元素分布的稀疏条件以及矩阵元素取值的量化条件，我们首先引入广义逆矩阵的概念与性质：

定义 1 设矩阵 $A\in C^{m\times n}$，若矩阵 $X\in C^{n\times m}$ 满足如下四个 Penrose 方程：

$$AXA=A, \tag{9.87}$$

$$XAX=X, \tag{9.88}$$

$$(AX)^H=AX, \tag{9.89}$$

$$(XA)^H=XA, \tag{9.90}$$

则称 X 为 A 的 Moore-Penrose 逆，记为 A^+。若满足其中的 $(i),(j),\cdots,(l)$ 个方程，则称 X 为 A 的 $i,j,\cdots,l$-逆，记为 $A^{(i,j,\cdots,l)}$，由于这类逆不一定唯一，故将其全体记为 $A\{i,j,\cdots,l\}$。因此，A^+ 也可记为 $A^{(1,2,3,4)}$ 且是唯一的。

性质 1 考虑非齐次线性方程组：

$$Ax=b, \tag{9.91}$$

其中 $A\in C^{m\times n},b\in C^m$ 给定，而 $x\in C^n$ 为待定向量，如果存在向量 x 使方程组成立，则称方程组相容，否则称为不相容或者矛盾方程组。

- 当方程组相容时，$x = A^{(1)}b$ 为方程的特解，$x = A^{(1,4)}b$ 为方程的极小范数解。
- 当方程组矛盾时，则不存在通常意义下的解，通常寻求极值解，满足：

$$x = \arg\min_{x \in C^n} \|Ax - b\|_F^2,$$

$x = A^{(1,3)}b$ 为方程组的最小二乘解，$x = A^{+}b$ 为方程组的最小二乘极小范数解，该解既满足最小二乘，又满足极小范数，是唯一的。

在对 N 维度 DFT 矩阵 F_N 求取满足稀疏约束和量化约束的近似 DFT 矩阵 $\overset{*}{F}_N$ 后，其精度往往受限。于是，需要考虑矩阵簇 $\mathcal{A}$ 内矩阵的数量来进一步提升精度，利用广义逆矩阵的性质，我们构造矩阵 X，使其满足下式：

$$X = \arg\min_{X} \|\beta \overset{*}{F}_N X - F_N\|_F^2, \tag{9.92}$$

可以快速求得满足上式最小二乘误差的矩阵为：

$$X = [\beta \overset{*}{F}_N]^{+} F_N, \tag{9.93}$$

可以看到在不考虑对 X 进行约束的情况下，最小误差为：

$$d(F_N, \overset{*}{F}_N X) = \frac{1}{N}\sqrt{\|F_N - \beta \overset{*}{F}_N X\|_F^2} = \frac{1}{N}\sqrt{\|F_N - \beta \overset{*}{F}_N [\beta \overset{*}{F}_N]^{+} F_N\|_F^2}, \tag{9.94}$$

原本的构造误差为：

$$d(F_N, \overset{*}{F}_N) = \frac{1}{N}\sqrt{\|F_N - \beta \overset{*}{F}_N\|_F^2}, \tag{9.95}$$

满足:

$$d(F_N, \overset{*}{F}_N X) \le d(F_N, \overset{*}{F}_N) \tag{9.96}$$

因此，我们的目标在于寻找一个近似 X 的矩阵 $\overset{*}{X}$，满足以下优化目标：

$$\begin{aligned} &\overset{*}{X} = \arg_{\overset{*}{X}} d(F_N, \overset{*}{F}_N X) \le d(F_N, \overset{*}{X}) \le d(F_N, \overset{*}{F}_N) \\ &s.t. \qquad G(\overset{*}{X}[l]) \le 2, l = 0, 1, \cdots, N-1, \\ &\qquad\quad \overset{*}{X}[l, m] \in \{x + yj | x, y \in \{2^0, 2^1, \cdots, 2^{q-1}\}\}, \end{aligned} \tag{9.97}$$

如此，DFT 矩阵 F_N 更加精确的近似表达应为：

$$F_N \approx \beta \overset{*}{F}_N \overset{*}{X}, \tag{9.98}$$

记录增加的硬件复杂度为 $C_{\text{add}} \ge 0$，将矩阵 $\overset{*}{F}_N = \overset{*}{F}_N \overset{*}{X}$ 视作新的估计矩阵，重复式 (9.92)-式 (9.97)，即可逐渐降低最小误差，统计每次增加的计算复杂度，即可绘制出最小误差 $d(\mathcal{A}, \beta)$ 与计算复杂度之间的关系，从而制定相应的方案，在满足最小误差 RMSE 0.1 的情况，合理的选择计算复杂度较低的方案。

上述方案依赖于不断的建立广义逆矩阵来寻找最佳的近似子矩阵以便求取更加精确的近似 DFT 矩阵，即通过对父代的广义逆矩阵进行满足稀疏约束和量化约束的近似，来促进子代的广义逆矩阵生成。将近似的父代广义逆矩阵视作矩阵簇 $\mathcal{A}$ 内的矩阵元素的拓展，从而以增加矩阵 A 的个数达到降低误差提升精度效果的目的。因此，我们称该算法为广义逆反演（Generalized Inverse Generation, GIG）算法。该算法的总体流程框图如算法6所示：

算法 6 基于广义逆繁衍算法的稀疏量化优化求解 DFT 算法

输入: 数据维度 N，数据位数 q。

输出: 量化且稀疏的矩阵簇 $\overset{*}{\mathcal{A}}$，尺度系数 $\overset{*}{\beta}$，最终 RMSE 误差 $\overset{*}{E}$，总硬件复杂度 $\overset{*}{C}$。

根据算法5初始化 DFT 矩阵 F_N 的近似矩阵 $\overset{*}{F}_N$，记录当前硬件复杂度为 C_N，尺度系数 $\beta = 1$，误差 RMSE = Inf;
while RMSE > 0.1 **do**
    根据广义逆的定义求 $X = [\beta \overset{*}{F}_N]^{+} F_N$;
    更新 β 的值，令 $\beta = \beta . * \max(\|X\|_2)$;
    对 X 内的元素规范化 $\bar{X} = X./\max(\|X\|_2)$;
    根据算法5对 $\bar{X}$ 进行稀疏和量化得到近似矩阵 $\overset{*}{X}$;
    更新矩阵簇 $\mathcal{A}$ 内的元素，即添加 $\overset{*}{X}$;
    更新近似矩阵 $\overset{*}{F}_N$，令 $\overset{*}{F}_N = \overset{*}{F}_N \bar{X}$，并记录复杂度的增量 C_{ADD};
    更新当前硬件复杂度 $C_N = C_N + C_{\text{ADD}}$;
    计算当前的误差 RMSE $= \sqrt{\|F_N - \beta \overset{*}{F}_N\|_F^2 / N}$;
end
存储矩阵簇 $\overset{*}{\mathcal{A}}$ 以及尺度系数 $\overset{*}{\beta} = \beta$，硬件复杂度 $\overset{*}{C} = C_N$;
更新最终的 RMSE 误差: $\overset{*}{E} = \sqrt{\|F_N - \overset{*}{\beta} = \beta \overset{*}{F}_N\|_F^2 / N}$;
返回：程序结束。

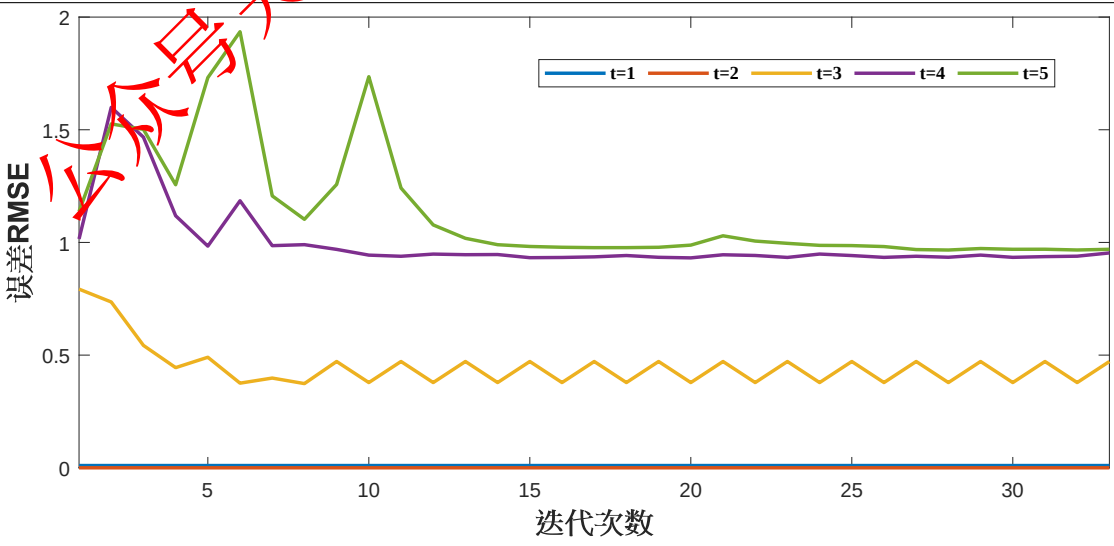


图 9.14　当 $q = 2$ 时，32 次迭代过程中 RMSE 误差变化曲线

9.3 问题五的求解

根据上述的建模，我们计算了在不同 q 值条件下（即不同量化程度约束），32 次迭代过程中尺度系数 β，RMSE 误差 E，总硬件复杂度 C。

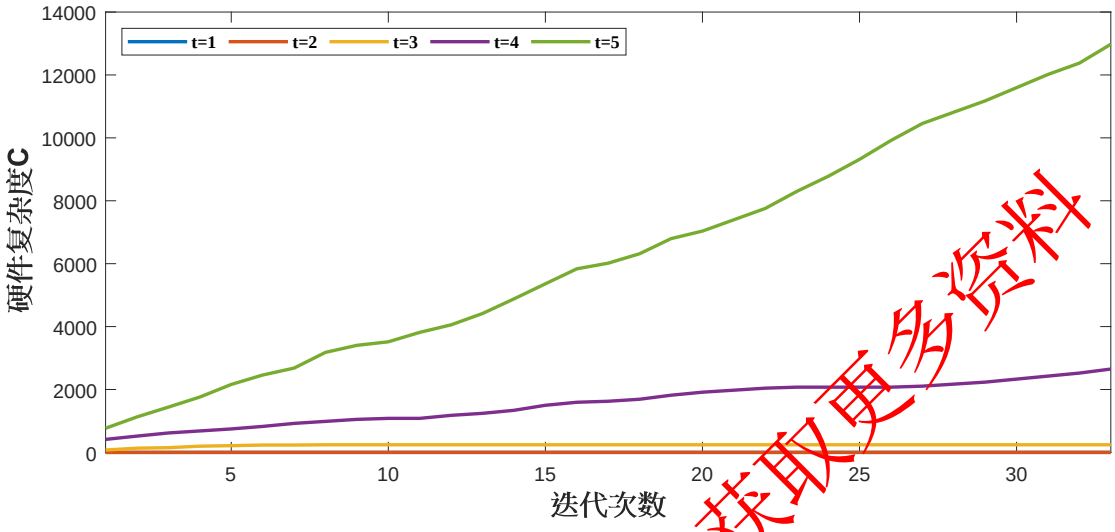


图 9.15　当 $q=2$ 时，32 次迭代过程中总硬件复杂度 C 变化曲线

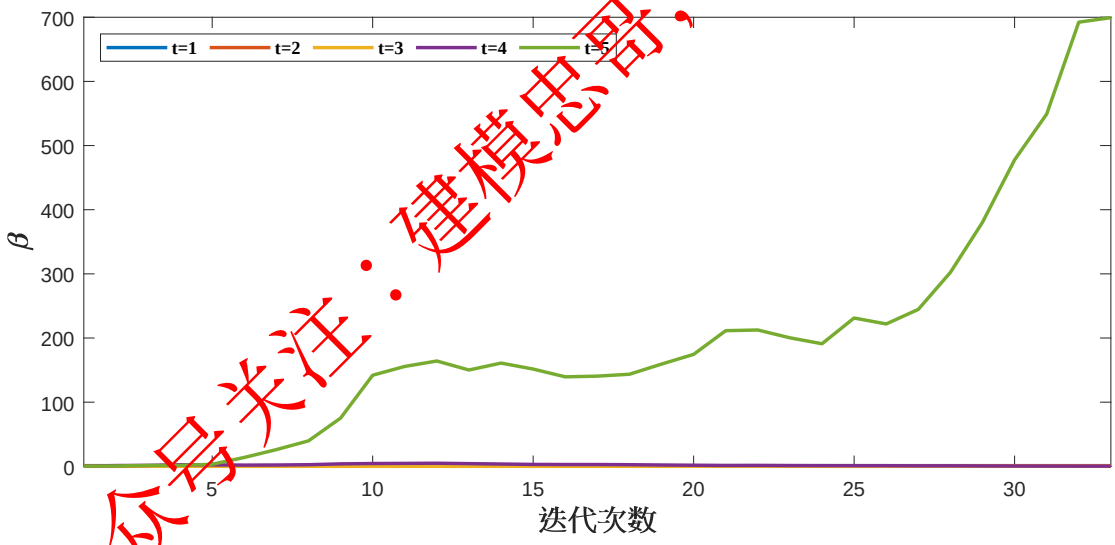


图 9.16　当 $q=2$ 时，32 次迭代过程中尺度系数 β 变化曲线

在在 32 次迭代过程中，图9.14-9.16分别记录了当 $q=2$ 时 RMSE 误差，总硬件复杂度 C 以及尺度系数 β 三个参数的数值变化曲线。图9.17-9.19分别记录了当 $q=3$ 时 RMSE 误差，总硬件复杂度 C 以及尺度系数 β 三个参数的数值变化曲线。图9.20-9.22分别记录了当 $q=4$ 时 RMSE 误差，总硬件复杂度 C 以及尺度系数 β 三个参数的数值变化曲线。图9.23-9.25分别记录了当 $q=5$ 时 RMSE 误差，总硬件复杂度 C 以及尺度系数 β 三个参数的数值变化曲线。图9.26-9.28分别记录了当 $q=6$ 时总硬件复杂度 C 以及尺度系数 β 三个参数的数值变化曲线。

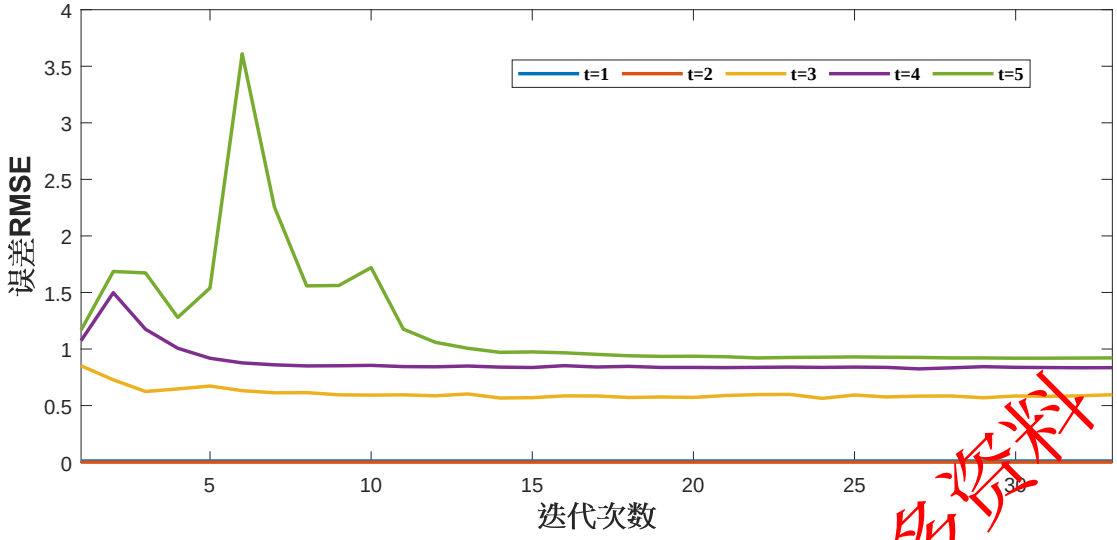


图 9.17　当 $q=3$ 时，32 次迭代过程中 RMSE 误差变化曲线

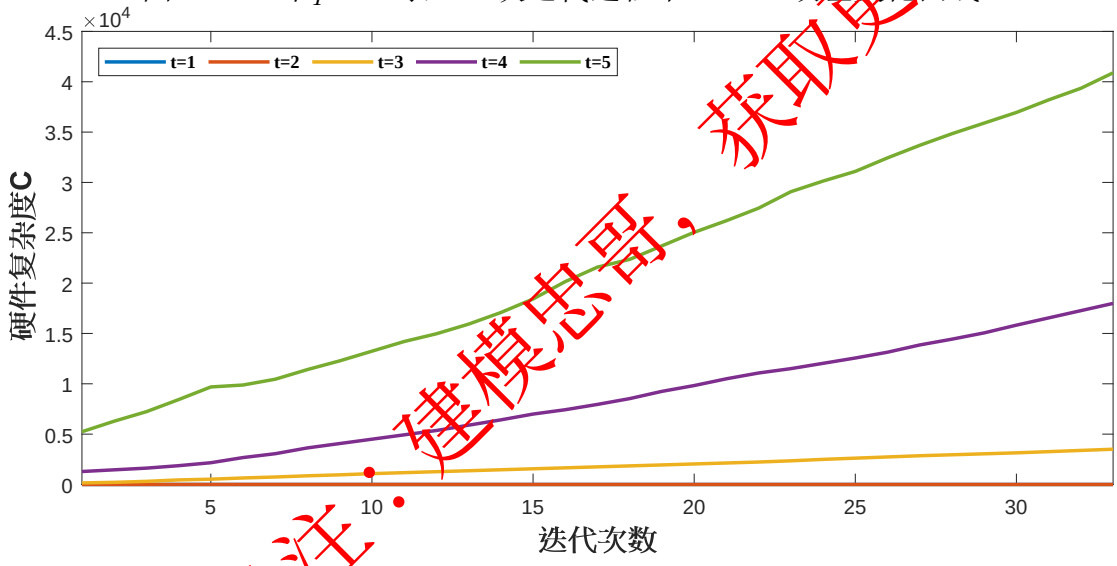


图 9.18　当 $q=3$ 时，32 次迭代过程中总硬件复杂度 C 变化曲线

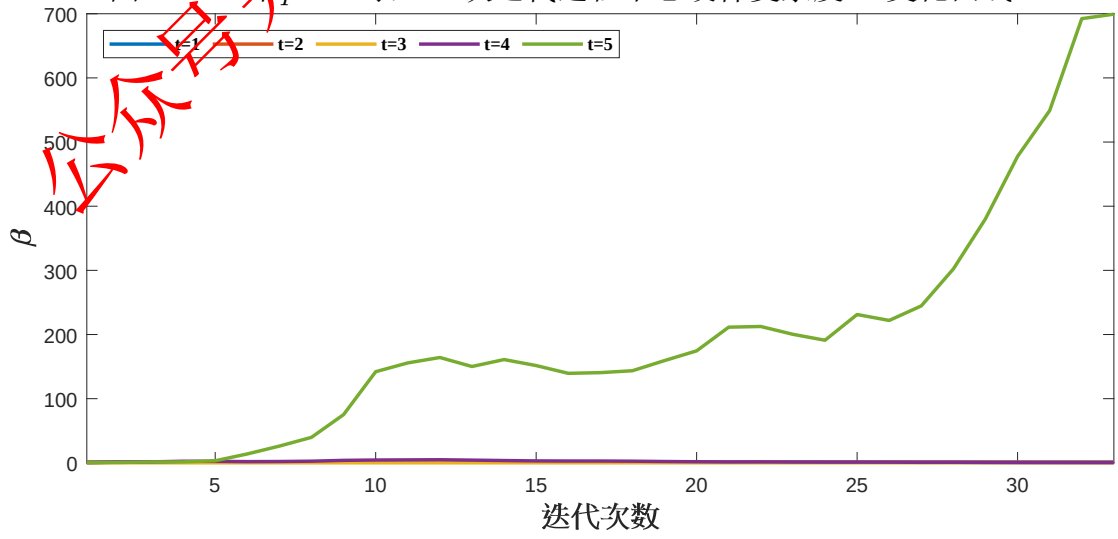


图 9.19　当 $q=3$ 时，32 次迭代过程中尺度系数 β 变化曲线

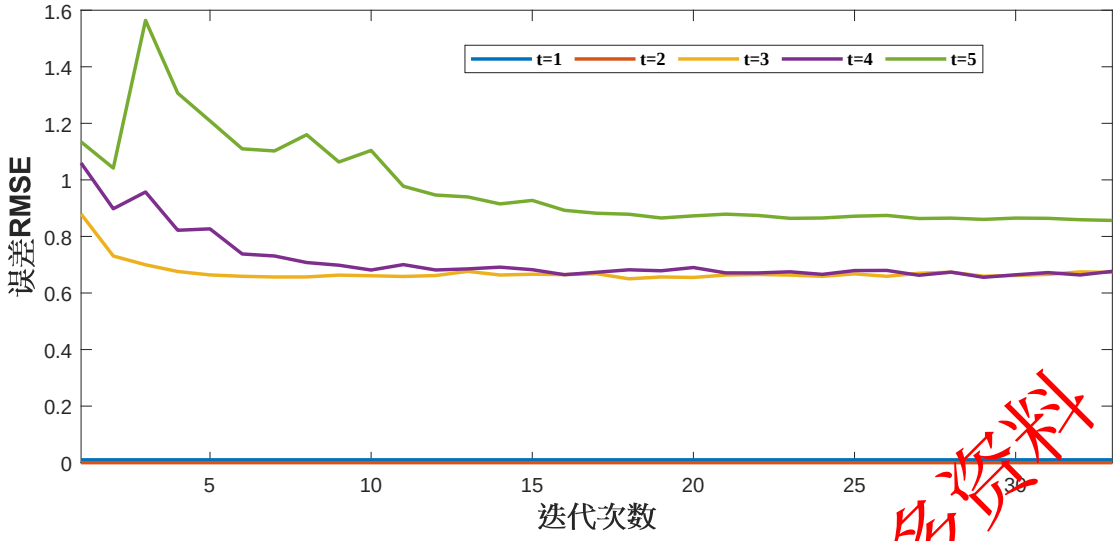


图 9.20　当 $q=4$ 时，32 次迭代过程中 RMSE 误差变化曲线

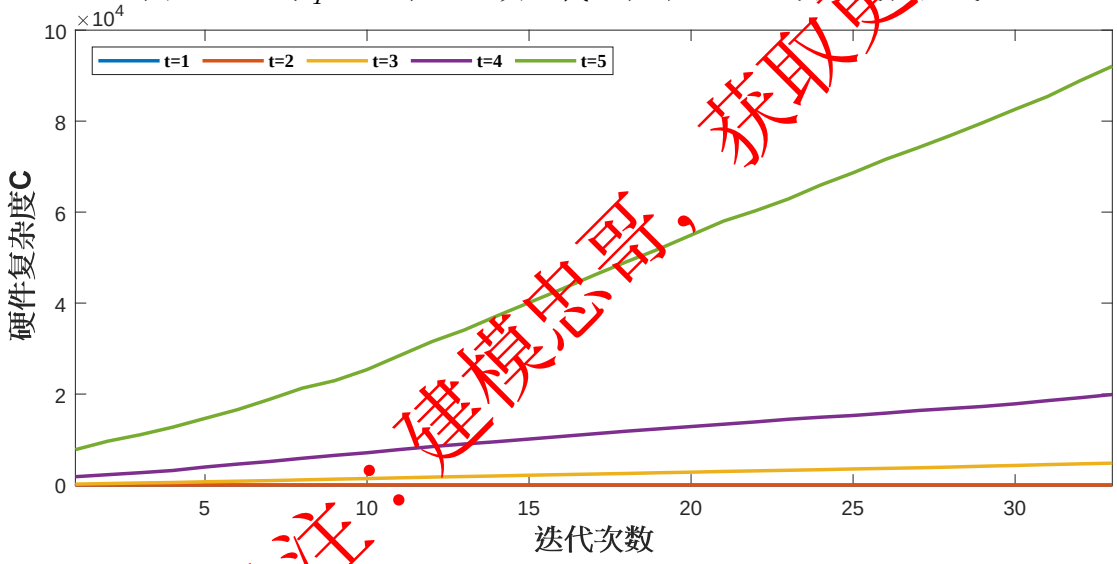


图 9.21　当 $q=4$ 时，32 次迭代过程中总硬件复杂度 C 变化曲线

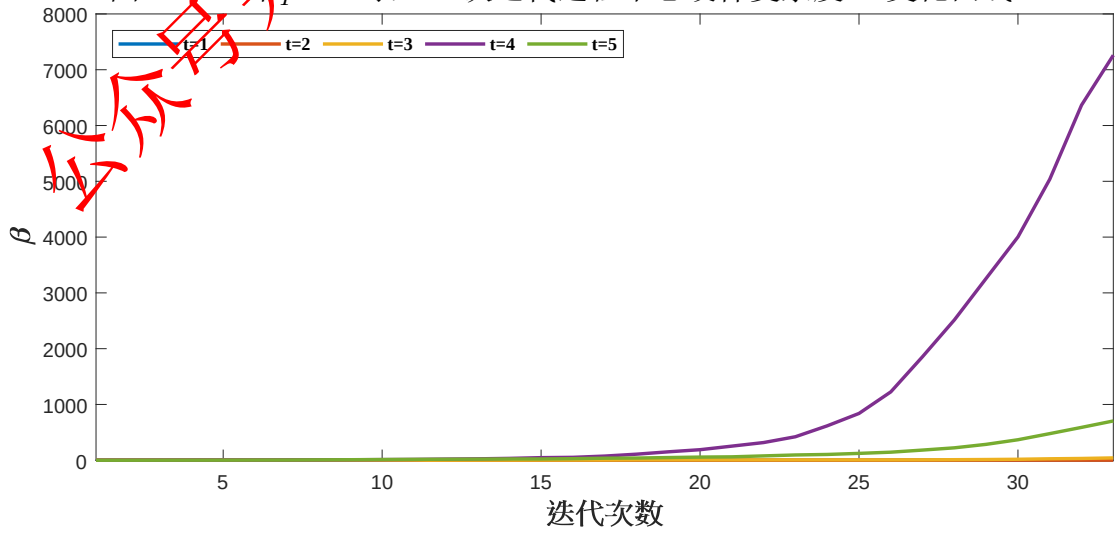


图 9.22　当 $q=4$ 时，32 次迭代过程中尺度系数 β 变化曲线

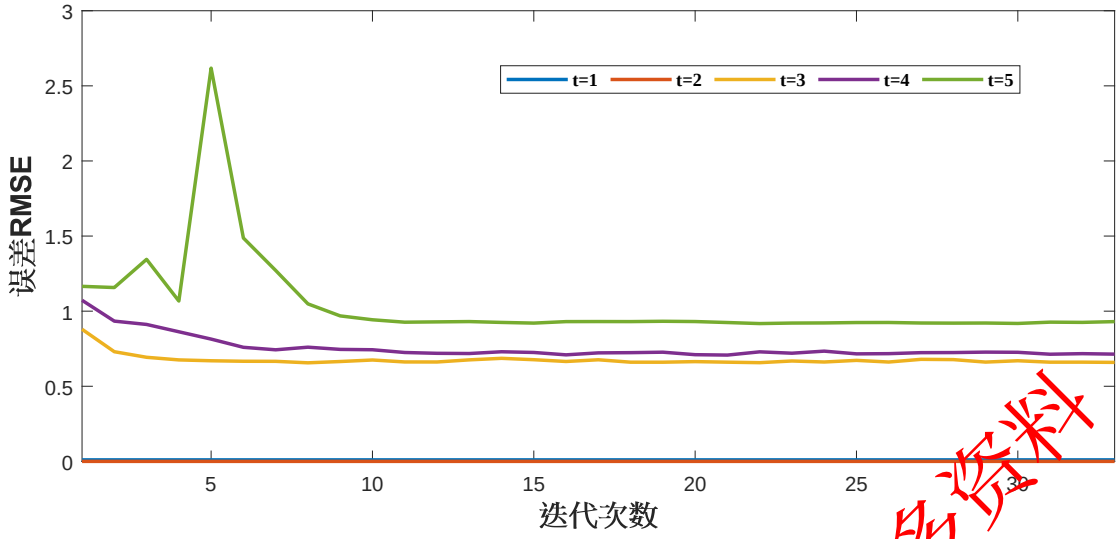


图 9.23　当 $q=5$ 时，32 次迭代过程中 RMSE 误差变化曲线

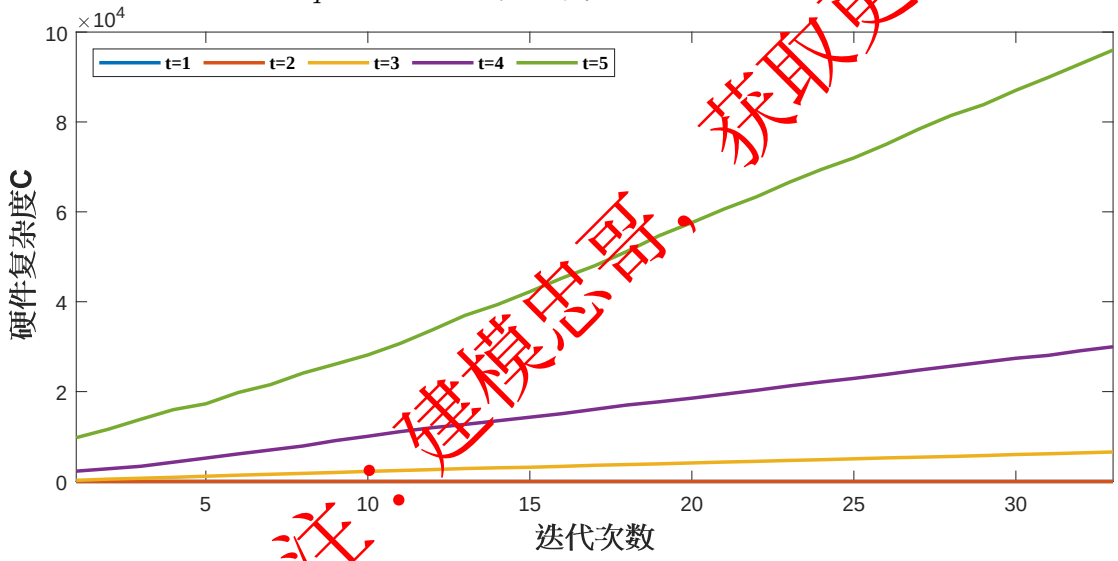


图 9.24　当 $q=5$ 时，32 次迭代过程中总硬件复杂度 C 变化曲线

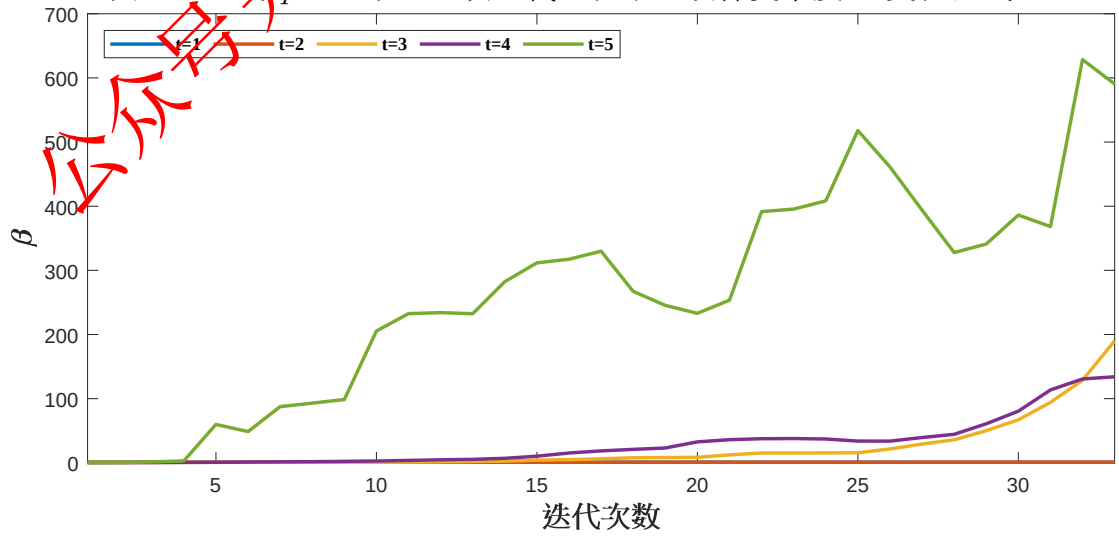


图 9.25　当 $q=5$ 时，32 次迭代过程中尺度系数 β 变化曲线

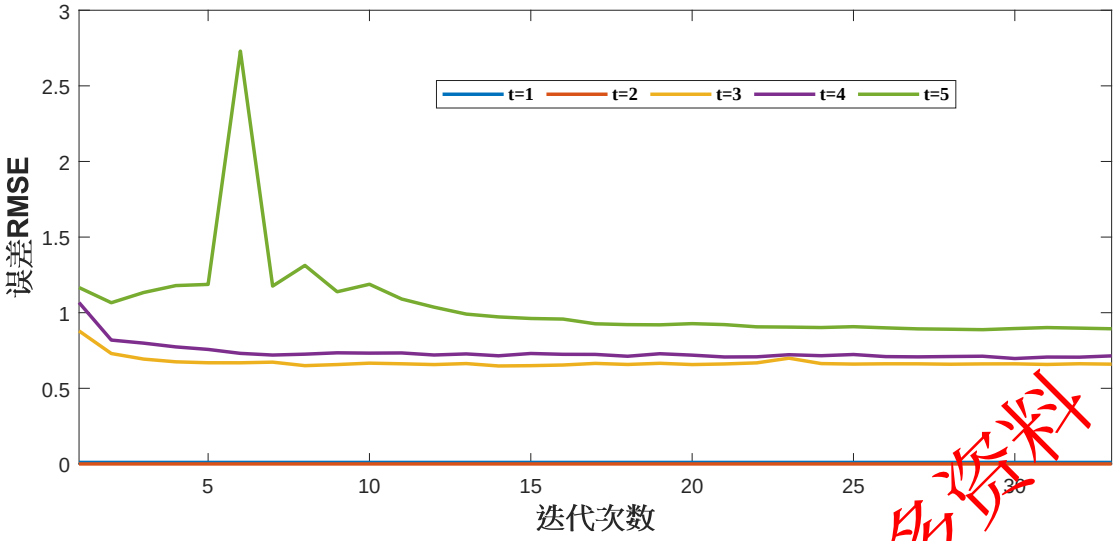


图 9.26　当 $q=6$ 时，32 次迭代过程中 RMSE 误差变化曲线

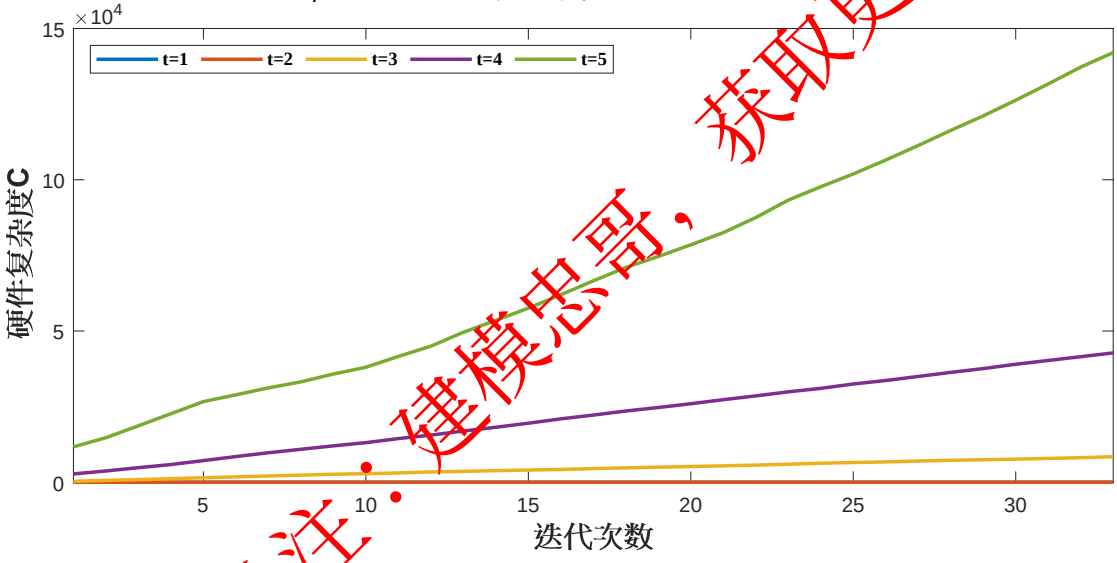


图 9.27　当 $q=6$ 时，32 次迭代过程中总硬件复杂度 C 变化曲线

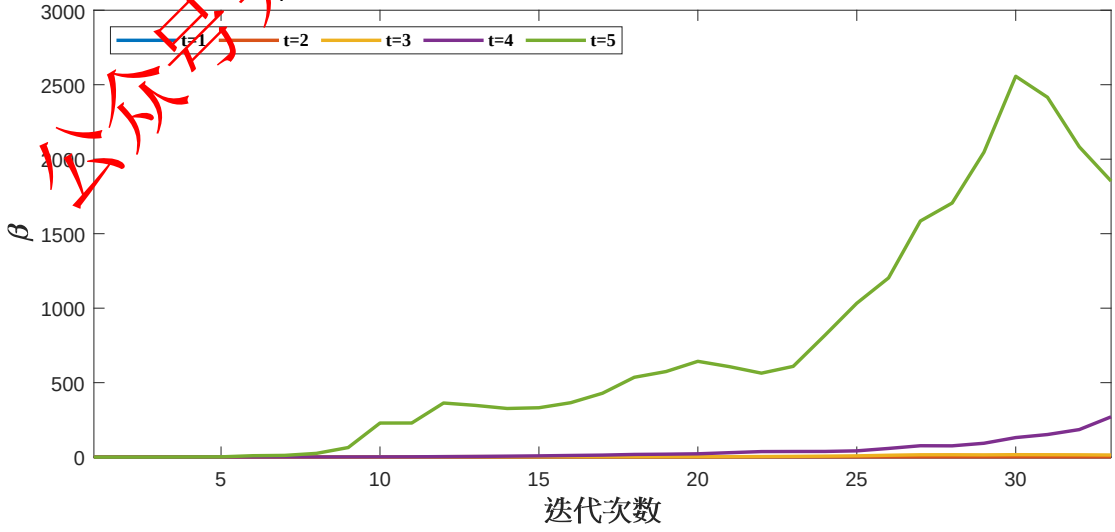


图 9.28　当 $q=6$ 时，32 次迭代过程中尺度系数 β 变化曲线

9.4 结果分析

结合问题三的结果，我们知道，先执行量化再执行稀疏的$\mathcal{Q}-\mathcal{S}$算法具有更高的精度。因此，在满足分解矩阵量化且稀疏的约束条件下，为尽可能满足精度的约束条件，考虑使用$\mathcal{Q}-\mathcal{S}$算法并增加分解子矩阵。由图 (9.14)-图 (9.28) 结果可知，对于$q=2,3,...,6$的情况下，可以观察到，q的大小直接影响到模型的硬件复杂度，但当q取值越大时，其对硬件复杂度的影响似乎不那么明显。一方面，广义逆反演迭代的方法能够有效的提高模型的精度，但当迭代次数较多时，分解的子矩阵也较多，这将导致更高的硬件复杂度和更大的β。且当q取值较小时，模型的误差 RMSE 更小。另一方面，使用广义逆反演法增加分解子矩阵，当迭代次数达到一定值时，其对精度的提升并不明显。经过大量测试，我们发现，我们的结果可能很难满足精度 RMSE<0.1 的要求，但我们给定的广义逆优化算法可以有效的提高我们之前模型的精度。

实际上，我们的 RMSE 是由建模官方组委会新给出的新定义，在这种新定义的目标函数$d(\mathcal{A},\beta)=\sqrt{\|F_N-\beta A_1A_2\cdots A_k\|_F^2}$下，要同时满足分解矩阵是量化且稀疏的条件是相对较难的，尽管如此，可以看到我们提出的 GIG 算法能够在不断的迭代过程中通过增加矩阵的数目来提升精度。这种做法免不了牺牲硬件复杂度，因为每一次的精度的变化，都会引入新的矩阵$\overset{*}{X}$。幸运的是，我们的结果表明，当引入的新矩阵数目超过一定数量时，其对模型的精度影响相对较小，因此，当需要对提出的算法就行优化时，可以考虑引入较少的矩阵。

备注：需要注意的是，现在假定的 RMSE 可能导致了最终计算的误差结果相对较大。例如，对于如下的 4 维 DFT 矩阵F_4和假定的逼近矩阵$\hat{F}$：

$$F_4=\begin{bmatrix}1&1&1&1\\1&-i&-1&i\\1&-1&1&-1\\1&i&-1&-i\end{bmatrix},\hat{F}=\begin{bmatrix}1&1&1&1\\1&-i&-1&i\\1&-1&1&1\\1&i&2&-i\end{bmatrix},|F_4-\hat{F}|=\begin{bmatrix}0&0&0&0\\0&0&0&0\\0&0&0&2\\0&0&3&0\end{bmatrix},\tag{9.99}$$

在式 (9.99) 中，由于F_N和$\hat{F}$的量级相当，因此不妨取$\beta=1$，此时，用$\widehat{F_4}$去逼近F_4的 RMSE 误差为：$\frac{1}{N}\left\|F_4-\beta\widehat{F_4}\right\|_F^2=\frac{\sqrt{13}}{4}=0.9014$。可以发现，尽管用$\widehat{F_4}$去逼近$F_4$的有很多值都完全准确的重合。但当矩阵存在一两个元素估计不准时，模型的 RMSE 也不会很小。

10 模型评价及算法改进

10.1 模型评价

10.1.1 模型优势

经过建模与分析，本文较好地解决了低硬件复杂度的 DFT 类矩阵的整数分解逼近的问题，所提出的模型优势在于：

针对问题一，利用了一种新颖的蝶形-奇异值分解（Butterfly Singular Value Decomposition, BSVD）方法，对 DFT 矩阵蝶形划分并执行奇异值分解，并采用交替方向乘子优化方法对相关连乘矩阵进行优化约束，限制其行非零元素数量（稀疏约束）。为了有效验证设计矩阵的硬件复杂度，提出了基于布尔矩阵复杂度统计算法。实验结果表明，在稀疏约束条件限制下，以 BSVD 方法为前置策略对矩阵进行连乘分解能够尽量在保证精度的同时，极大地降低了模型的复杂度，且当时域点数 N 越大效果越显著。

针对问题二，本文基于提出的快速量化算法，在满足矩阵元素实虚部取值受限的条件下（量化约束），提供了集中量化和分布量化两种策略，对连乘矩阵进行优化约束，并且两种策略均能显著降低硬件复杂度。

针对问题二，在满足矩阵 A_K 元素实虚部受限的条件下，提供了集中量化和分布量化两种策略对 A 和 β 进行优化。所提出的模型在量化约束的条件下能够同时兼顾高精度和低复杂度。

针对问题三，实现了稀疏约束与量化约束下的 DFT 矩阵近似估计，并提出了两种稀疏与量化算法实现优化策略：$\mathcal{S}-\mathcal{Q}$ 算法和 $\mathcal{Q}-\mathcal{S}$ 算法。对于不同的精度和硬件复杂度要求，所提出的两种算法都能满足。对比于精度，$\mathcal{Q}-\mathcal{S}$ 算法具有较大的优势，而对比于硬件复杂度，$\mathcal{S}-\mathcal{Q}$ 具有更好的性能。

针对问题四，本文基于 *Kronecker* 混合积的性质，结合 BSVD 算法对高维目标矩阵的连乘分解进行了简化。简化后的矩阵在计算 *Kronecker* 积时，具有较高的精度，同时其硬件复杂度也保存在较低的水平（复杂度 γ 为 36）。

针对问题五，基于广义逆矩阵是矛盾方程组最小二乘解的性质，本文将满足约束的近似矩阵的广义逆与估计矩阵相乘，作为新增的待优化连乘矩阵。实验证明在不同的参数设定下，该方法能够在精度受限的情况下有效地降低最小误差。

10.1.2 模型局限性

1) 提出的 BSVD 算法，每次分解的矩阵数量个数为五，尽管可以通过稀疏和量化对复杂度或精度进行优化，但分解矩阵数目过少可能会对目标精度造成影响。

2) 对分解矩阵实施稀疏和量化的交替乘子优化方法本质上是一种类似于贪心算法的局部搜索算法，在寻找最优解时可能并没有很大的优势，

10.2 模型改进与推广

未来的研究可以聚焦在对 BSVD 模型进行改进，如对该模型分解后的矩阵 U 和 V^T 进一步执行 SVD 分解，或寻找更优的参数搜索算法等。

参考文献

[1] 张宪超, 武继刚, 蒋增荣等. 离散傅里叶变换的算术傅里叶变换算法 [J]. 电子学报, 2000 (05): 105-107.

[2] Ariyarathna V, Coelho D F G, Pulipati S, et al. Multibeam digital array receiver using a 16-point multiplierless DFT approximation [J]. IEEE Transactions on Antennas and Propagation, 2018, 67 (2): 925-933.

[3] Tablada C J, Bayer F M, Cintra R J. A class of DCT approximations based on the Feig－Winograd algorithm[J]. Signal Processing, 2015, 113: 38-51.

[4] 曹伟丽. 快速傅里叶变换的原理与方法 [J]. 上海电力学院学报, 2006 (02): 192-194.

[5] Cooley J W, Tukey J W. An algorithm for the machine calculation of complex Fourier series[J]. Mathematics of computation, 1965, 19 (90): 297-301.

[6] Suarez D, Cintra J R, Bayer M F, et al. Multi-beam RF aperture using multiplierless FFT approximation [J]. Electronics Letters, 2014, 50 (24): 1788-1790.

[7] Ariyarathna V, Madanayake A, Tang X, et al. Analog approximate-FFT 8/16-beam algorithms, architectures and CMOS circuits for 5G beamforming MIMO transceivers[J]. IEEE Journal on Emerging and Selected Topics in Circuits and Systems, 2018, 8 (3): 466-479.

[8] Zhou J. Realization of Discrete Fourier Transform and Inverse Discrete Fourier Transform on One Single Multimode Interference Coupler[J]. IEEE Photonics Technology Letters, 2011, 23 (5): 302-304.

[9] Pei S C, Yeh M H. Fractional Fourier series expansion for finite signals and dual extension to discrete-time fractional Fourier transform[J]. Signal Processing IEEE Transactions on, 1999, 47(10): 2883-2888.

[10] Blahut R E. Fast algorithms for signal processing[M]. Cambridge University Press, 2010.

[11] Ariyarathna V, Coelho D F G, Pulipati S, et al. Multibeam digital array receiver using a 16-point multiplierless DFT approximation[J]. IEEE Transactions on Antennas and Propagation, 2018, 67 (2): 925-933.

[12] Rao KR, Kim DN, and Hwang JJ, Fast Fourier Transform: Algorithms and Applications[M]. 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.）

[13] Madanayake A, Cintra R J, Akram N, et al. Fast radix-32 approximate DFTs for 1024-beam digital RF beamforming[J]. IEEE Access, 2020, 8: 96613-96627.

[14] Zhai L, Dong L, Liu Y, et al. Adaptive hybrid threshold shrinkage using singular value decomposition[J]. Journal of Electronic Imaging, 2019, 28 (6): 063002.

[15] 赵学智, 叶邦彦. SVD 和小波变换的信号处理效果相似性及其机理分析 [J]. 电子学报, 2008 (08): 1582-1589.

[16] Deng L, Leng K. Application of Singular Value Decomposition and Adaptive Noise Cancellation Method in AUV Sonar Pulse[J]. Journal of Physics: Conference Series, 2023, 2419 (1): 012060.

[17] 高乾坤, 王玉军, 王惊晓. 基于交替方向乘子法的非光滑损失坐标优化算法 [J]. 计算机应用, 2013, 33 (07): 1912-1916.

[18] Essoufi E, Koko J, Zafrar A. Alternating direction method of multiplier for a unilateral contact problem in electro-elastostatics[J]. Computers and Mathematics with Applications, 2017, 73 (8): 1789-1802.

[19] 陈端兵, 黄文奇. 求解矩形 packing 问题的贪心算法 [J]. 计算机工程, 2007 (04): 160-162.

[20] Juedes D W, Jones J S. A generic framework for approximation analysis of greedy algorithms for star bicoloring[J]. Optimization Methods and Software, 2019, 36 (4): 1-22.

附录 A　MATLAB 源程序

A.1　第 1 问程序

code.m

```
clc;
clear all;

%% data generation
% pars : n -> the order of the fourier transform matrix
for iter=1:7
    n=2^iter;
    F_actual=dftmtx(n);
    I=eye(n/2);
    D=zeros(n/2,n/2);P=zeros(n,n);
    w=exp(-2*pi*1i/n);
    for j=1:n/2
        D(j,j)=w.^(j-1);
        P(j,2*j-1)=1;
        P(n/2+j,2*j)=1;
    end
%% loss validation
    F_dec=[dftmtx(n/2) zeros(n/2);zeros(n/2) dftmtx(n/2)];
    A=[I D; I -D];
    [U,S,V] = svd(F_dec);
    F_hat=A*U*S*V'*P;
    beta(iter)=norm(F_actual,'fro')./norm(F_hat,'fro'); %
       initialize as 1
    loss_valid=1/n*sqrt(norm(F_actual-beta(iter)*F_hat,'fro').^2);

%% matrix sparser
    % retrain large rmse loss
    u_first_flag = 1;
    [u_sparse,v_sparse,beta(iter)] = svd_sparser(A,S,U,V,P,n,
       F_actual,u_first_flag,beta(iter));
    F_hat=A*u_sparse*S*v_sparse'*P;
    beta(iter)=norm(F_actual,'fro')./norm(F_hat,'fro');
    rmse_loss(iter)=1/n*sqrt(norm(F_actual-beta(iter)*F_hat,'fro')
       .^2);
    disp(n)
```

```
    disp(rmse_loss)

%% Computational complexity
    L   = MyCalComplexity(A,u_sparse,v_sparse');
    C(iter)=16*L;
    FFT_C(iter)=16*CompCal(A,F_dec);
end
%% Save Results
save data_prob_1


function [u_sparse,v_sparse,beta_sparse] = svd_sparser(A,S,U,V,P,n,
   F_actual,u_first_flag,beta)
    if u_first_flag
        mse = zeros(size(U));
        for k=1:n
            for m=1:n
                U_new=U;
                U_new(k,m)=0;
                F_hat=A*U_new*S*V'*P;
                mse(k,m)=1/n*sqrt(norm(F_actual-F_hat.*beta,'fro')
                    .^2);
            end
        end
        clear U_new
        U_sparse=zeros(size(U));
        for k=1:n
            for m=1:n
                mse_line = mse(k,:);
                max_id_1 = find(mse_line == max(mse_line));
                if length(max_id_1) >= 2
                    id1 = max_id_1(1);
                    id2 = max_id_1(2);
                else
                    mse_line(max_id_1) = 0;
                    max_id_2 = find(mse_line == max(mse_line));
                    id1 = max_id_1(1);
                    id2 = max_id_2(1);
                end
```

```
        end
        U_sparse(k,id1)=U(k,id1);
        U_sparse(k,id2)=U(k,id2);
    end

    F_hat=A*U_sparse*S*V'*P;
    beta= norm(F_actual,'fro')./norm(F_hat,'fro');

    mse = zeros(size(V));

    for k=1:n
        for m=1:n
            V_new=V;
            V_new(k,m)=0;
            F_hat=A*U_sparse*S*V_new'*P;
            mse(k,m)=1/n*sqrt(norm(F_actual-F_hat.*beta,'fro'))
                ;
        end
    end
    clear V_new
    V_sparse=zeros(size(V));
    for k=1:n
        for m=1:n
            mse_line = mse(k,:);
            max_id_1 = find(mse_line == max(mse_line));
            if length(max_id_1) >= 2
                id1 = max_id_1(1);
                id2 = max_id_1(2);
            else
                mse_line(max_id_1) = 0;
                max_id_2 = find(mse_line == max(mse_line));
                id1 = max_id_1(1);
                id2 = max_id_2(1);
            end
        end
        V_sparse(k,id1)=V(k,id1);
        V_sparse(k,id2)=V(k,id2);
    end
    F_hat=A*U_sparse*S*V_sparse'*P;
```

```
        beta=norm(F_actual,'fro')/norm(F_hat,'fro');

    else
        mse = zeros(size(V));

        for k=1:n
            for m=1:n
                V_new=V;
                V_new(k,m)=0;
                F_hat=A*U*S*V_new'*P;
                mse(k,m)=1/n*sqrt(norm(F_actual-F_hat.*beta,'fro'))
                    ;
            end
        end
        clear V_new
        V_sparse=zeros(size(V));
        for k=1:n
            for m=1:n
                mse_line = mse(k,:);
                max_id_1 = find(mse_line == max(mse_line));
                if length(max_id_1) >= 2
                    id1 = max_id_1(1);
                    id2 = max_id_1(2);
                else
                    mse_line(max_id_1) = 0;
                    max_id_2 = find(mse_line == max(mse_line));
                    id1 = max_id_1(1);
                    id2 = max_id_2(1);
                end
            end
            V_sparse(k,id1)=V(k,id1);
            V_sparse(k,id2)=V(k,id2);
        end
        F_hat=A*U*S*V_sparse'*P;
        beta=norm(F_actual,'fro')./norm(F_hat,'fro');


        mse = zeros(size(U));
```

```
        for k=1:n
            for m=1:n
                U_new=U;
                U_new(k,m)=0;
                F_hat=A*U_new*S*V_sparse'*P;
                mse(k,m)=1/n*sqrt(norm(F_actual-F_hat.*beta,'fro')
                    .^2);
            end
        end
        clear U_new
        U_sparse=zeros(size(U));
        for k=1:n
            for m=1:n
                mse_line = mse(k,:);
                max_id_1 = find(mse_line == max(mse_line));
                if length(max_id_1) >= 2
                    id1 = max_id_1(1);
                    id2 = max_id_1(2);
                else
                    mse_line(max_id_1) = 0;
                    max_id_2 = find(mse_line == max(mse_line));
                    id1 = max_id_1(1);
                    id2 = max_id_2(1);
                end
            end
            U_sparse(k,id1)=U(k,id1);
            U_sparse(k,id2)=U(k,id2);
        end
        F_hat=A*U_sparse*S*V_sparse'*P;
        beta=norm(F_actual,'fro')./norm(F_hat,'fro');
    end
    % output
    u_sparse = U_sparse;
    v_sparse = V_sparse;
    beta_sparse = beta;
end

function Order_generate = order_generate(n)
wave = linspace(1,n,n);
```

```
wave_group= group( wave);
P = zeros(n,n);
for temp = 1 : 1 : n
P(temp,wave_group(temp)) = 1;
end
Order_generate = P;
end
%%
function wave_group = group( wave)
 len = length(wave);
 pow = nextpow2(len);
 for i = 1 : 1 : pow
 ng = 2^(i-1);%Num of the group
 nl = len/ng;%The length of each group
 StartIndex = 1;
 EndIndex = 1 + nl - 1;
 for j = 1 : 1 : ng
 wave( StartIndex : EndIndex) = sortfft(wave(StartIndex : EndIndex)
    );
 StartIndex = StartIndex + nl;
 EndIndex = EndIndex + nl;
 end
 end
wave_group = wave;
% P=zeros(len,len);
% for temp= 1:1:length(wave)
%     P(temp,wave_group(temp)) = 1;
% end
% Pn = P;
end
%%
function wave_sort = sortfft( wave)
len = length(wave);
odd = zeros(1,len/2);
even = zeros(1,len/2);
for i = 1 : 1 : len
if (mod(i,2) == 1)
even((i+1)/2) = wave(i);
 else
```

```
odd(i/2) = wave(i);
 end
 end
 %The index is 1 , but it's the zeroth num ,  is even
 wave_sort = [even,odd];
end


function Logical=myLogicalize(A)
    for k=1:size(A,1)
        for j=1:size(A,2)
            Logical1(k,j)=~isreal(A(k,j));
            if isequal(A(k,j), 1i) || isequal(A(k,j), -1i) ||
               isequal(A(k,j), 1+1i) || isequal(A(k,j), 1-1i)
                Logical1(k, j) = 0;
            end
            end
    end
    Logical = Logical1;
end


function MyCalComplexity=MyCalComplexity(W,U,V)
    logical_1 =myLogicalize(W);
    logical_2 =myLogicalize(U);
    logical_3 =myLogicalize(V);

    Comlogical_1=logical_1*logical_2;
    Comlogical_2 = Comlogical_1*logical_3;
    CalComplexity_1=nnz(Comlogical_1);
    CalComplexity_2=nnz(Comlogical_2);
    MyCalComplexity =CalComplexity_1+CalComplexity_2;
end

function CalComplexity=CompCal(A,B)
% 转换成逻辑矩阵
    logical_1 =myLogicalize(A);
    logical_2 =myLogicalize(B);
    CalComplexity = nnz(logical_1*logical_2);
```

```
end

function isflag = check_cond_one(F)
    %CHECK_COND_ONE
    mask = (abs(F)~=0);
    mask = int8(mask);
    mask_sum_col = sum(mask,2);
    isflag = all(mask_sum_col(:)<=2);
end
```

A.2 第 2 问程序

code.m

```
clc;
clear all;
close all;
%%% Quantizer for A U V
%% data generation
% pars : n -> the order of the fourier transform matrix
N_MAX = 5;
beta_container = ones([N_MAX,1]);
for iter=1:1:N_MAX
    n=2^iter;
    F_actual=dftmtx(n);
    I=eye(n/2);
    D=zeros(n/2,n/2);P=zeros(n,n);
    w=exp(-2*pi*1i/n);
    for j=1:n/2
        D(j,j)=w.^(j-1);
        P(j,2*j-1)=1;
        P(n/2+j,2*j)=1;
    end
%% loss validation
F_dec=[dftmtx(n/2) zeros(n/2);zeros(n/2) dftmtx(n/2)];
A=[I D; I -D];
[U,S,V] = svd(F_dec);
F_hat=A*U*S*V'*P;
%% quantization
q=3;
```

```
value_set = generate_valset(q);
scale_factor = 2.^(q-1);
A_proc = scale_factor .* A;
U_proc = scale_factor .* U;
V_proc = scale_factor .* V;
F_actual_proc = scale_factor .* scale_factor .* scale_factor .*
    F_hat;
% valid
F_hat_proc = A_proc*U_proc*S*V_proc'*P;beta = 1;
loss_valid=1/n*sqrt(norm(F_actual_proc-beta.*F_hat_proc,'fro').^2);
% every elements shall be quantized
[m,n] = size(F_hat);
A_quant_container = A_proc;U_quant_container = U_proc;
    V_quant_container = V_proc;
for col = 1 : 1 : m
    for row = 1 : 1 : n
        A_quant_container = quantizer(A_quant_container,S,
            U_quant_container,V_quant_container,P,F_actual_proc,beta
            ,value_set,col,row,2);
        U_quant_container = quantizer(A_quant_container,S,
            U_quant_container,V_quant_container,P,F_actual_proc,beta
            ,value_set,col,row,1);
        V_quant_container = quantizer(A_quant_container,S,
            U_quant_container,V_quant_container,P,F_actual_proc,beta
            ,value_set,col,row,0);
    end
end
% scale_out
A_quant = A_quant_container./ scale_factor;
U_quant = U_quant_container./ scale_factor;
V_quant = V_quant_container./ scale_factor;
F_hat=A_quant*U_quant*S*V_quant'*P;
beta=norm(F_actual,'fro')/norm(F_hat,'fro');
rmse_loss(iter)=1/n*sqrt(norm(F_actual-beta.*F_hat,'fro').^2);
beta_container(iter)=beta;
%% Computational complexity
L  = MyCalComplexity(A_quant_container,U_quant_container,
    V_quant_container');
C(iter)=q*L;
```

```
FFT_C(iter)=q*MyCalComplexity(A,U,V');
end
%% Save Results
save data_prob_distributed



clc;
clear all;
close all;
%%% Quantizer for F_actual
%% data generation
% pars : n -> the order of the fourier transform matrix
N_MAX = 5;
beta_container = ones([N_MAX,1]);
for iter=1:N_MAX
n=2^iter;
F_actual=dftmtx(n);
I=eye(n/2);
D=zeros(n/2,n/2);P=zeros(n,n);
w=exp(-2*pi*1i/n);
for j=1:n/2
    D(j,j)=w.^(j-1);
    P(j,2*j-1)=1;
    P(n/2+j,2*j)=1;
end
%% quantization
q=3;
value_set = generate_valset(q);
scale_factor = 2.^(q-1);
F_quant_container = scale_factor .* F_actual;
F_actual_proc = scale_factor .* F_actual;
% quantize
beta = 1;
for col = 1 : 1 : n
    for row = 1 : 1 : n
        F_quant_container = quantizer_FN(F_quant_container,
           F_actual_proc,beta,value_set,col,row);
        % beta = norm(F_quant_container,'fro')/norm(F_actual_proc,'
```

```
            fro');
    end
end
% scale_out
F_quant = F_quant_container./ scale_factor;
beta=norm(F_actual,'fro')./norm(F_quant,'fro');
rmse_loss(iter)=1/n*sqrt(norm(F_actual-beta.*F_quant,'fro').^2)
beta_container(iter)=beta;
C(iter)=0;
FFT_C(iter)=0;
end
%% Save Results
save data_prob_focus

function mat_qt = quantizer_FN(F,F_actual,beta,value_set,col,row)
    % u sub-problem
    [~,n] = size(F);
    len = length(value_set);
    mse = zeros(len);
    for iter = 1 : 1 : len
        F_new = F;
        F_new(col,row) = value_set(iter);
        F_hat=F_new;
        mse(iter)=1/n*sqrt(norm(F_actual-beta * F_hat,'fro').^2);
    end
    minimum = min(mse);
    minimum_id = find(mse == minimum);
    minimum_id = minimum_id(1);
    mat_qt = F;
    mat_qt(col,row) = value_set(minimum_id);
end

function mat_qt = quantizer_FFT(A,FFT,P,F_actual,beta,value_set,col
   ,row,mode)
    % MODE 1 for FFT , 2 for A
    % quantizer for ele[col,row]
    if mode == 1
        % u sub-problem
        [~,n] = size(FFT);
```

```
        len = length(value_set);
        mse = zeros(len);
        for iter = 1 : 1 : len
            FFT_new = FFT;
            FFT_new(col,row) = value_set(iter);
            FFT_hat=A*FFT_new*P;
            mse(iter)=1/n*sqrt(norm(beta * F_actual-FFT_hat,'fro')
                .^2);
        end
        minimum = min(mse);
        minimum_id = find(mse == minimum);
        minimum_id = minimum_id(1);
        mat_qt = FFT;
        mat_qt(col,row) = value_set(minimum_id);
    elseif mode == 2
        % a sub-problem
        [~,n] = size(A);
        len = length(value_set);
        mse = zeros(len);
        for iter = 1 : 1 : len
            A_new = A;
            A_new(col,row) = value_set(iter);
            F_hat=A_new*FFT*P;
            mse(iter)=1/n*sqrt(norm(beta * F_actual-F_hat,'fro')
                .^2);
        end
        minimum = min(mse);
        minimum_id = find(mse == minimum);
        minimum_id = minimum_id(1);
        mat_qt = A;
        mat_qt(col,row) = value_set(minimum_id);
    end
end

function mat_qt = quantizer(A,S,U,V,P,F_actual,beta,value_set,col,
    row,mode)
    % MODE 1 for u , 0 for v , 2 for A
    % quantizer for ele[col,row]
    if mode == 1
```

```
        % u sub-problem
        [~,n] = size(U);
        len = length(value_set);
        mse = zeros(len);
        for iter = 1 : 1 : len
            U_new = U;
            U_new(col,row) = value_set(iter);
            F_hat=A*U_new*S*V'*P;
            mse(iter)=1/n*sqrt(norm(F_actual-beta .* F_hat,'fro')
                .^2);
        end
        minimum = min(mse);
        minimum_id = find(mse == minimum);
        minimum_id = minimum_id(1);
        mat_qt = U;
        mat_qt(col,row) = value_set(minimum_id);
    elseif mode == 0
        % v sub-problem
        [~,n] = size(V);
        len = length(value_set);
        mse = zeros(len);
        for iter = 1 : 1 : len
            V_new = V;
            V_new(col,row) = value_set(iter);
            F_hat=A*U*S*V_new'*P;
            mse(iter)=1/n*sqrt(norm(F_actual-beta .* F_hat,'fro')
                .^2);
        end
        minimum = min(mse);
        minimum_id = find(mse == minimum);
        minimum_id = minimum_id(1);
        mat_qt = V;
        mat_qt(col,row) = value_set(minimum_id);
    elseif mode == 2
        % a sub-problem
        [~,n] = size(A);
        len = length(value_set);
        mse = zeros(len);
        for iter = 1 : 1 : len
```

```
            A_new = A;
            A_new(col,row) = value_set(iter);
            F_hat=A_new*U*S*V'*P;
            mse(iter)=1/n*sqrt(norm(F_actual-beta .* F_hat,'fro')
                .^2);
        end
        minimum = min(mse);
        minimum_id = find(mse == minimum);
        minimum_id = minimum_id(1);
        mat_qt = A;
        mat_qt(col,row) = value_set(minimum_id);

    end
end

function value_set = generate_valset(q)
    t = 0 : 1 : q-1;
    x = [0 2.^t -2.^t];
    y = [0 2.^t.*1i -2.^t.*1i];y=y';
    value_set = x+y;
    [m,n] = size(value_set);
    value_set = reshape(value_set,[m*n,1]);
end

function MyCalComplexity=MyCalComplexity(W,U,V)
    logical_1 =myLogicalize(W);
    logical_2 =myLogicalize(U);
    logical_3 =myLogicalize(V);

    Comlogical_1=logical_1*logical_2;
    Comlogical_2 = Comlogical_1*logical_3;
    CalComplexity_1=nnz(Comlogical_1);
    CalComplexity_2=nnz(Comlogical_2);
    MyCalComplexity =CalComplexity_1+CalComplexity_2;
end

function Logical=myLogicalize(A)
    for k=1:size(A,1)
        for j=1:size(A,2)
```

```
            Logical1(k,j)=~isreal(A(k,j));
            if isequal(A(k,j), 1i) || isequal(A(k,j), -1i) ||
               isequal(A(k,j), 1+1i) || isequal(A(k,j), 1-1i)||
               isequal(A(k,j), -1-1i)|| isequal(A(k,j), -1-1i)
                Logical1(k, j) = 0;
            end
            end
    end
    Logical = Logical1;
end

function CalComplexity=CompCal(A,B)
% 转换成逻辑矩阵
    logical_1 =myLogicalize(A);
    logical_2 =myLogicalize(B);
    CalComplexity = nnz(logical_1*logical_2);
end
```

A.3 第 3 问程序

code.m

```
clc;
clear all;

%% data generation
% pars : n -> the order of the fourier transform matrix
N_MAX = 5;beta_container = ones([N_MAX,1]);
beta_quant = ones([N_MAX,1]);beta_sparse = ones([N_MAX,1]);
for iter=1:N_MAX
n=2^iter;
F_actual=dftmtx(n);
I=eye(n/2);
D=zeros(n/2,n/2);P=zeros(n,n);
w=exp(-2*pi*1i/n);
for j=1:n/2
    D(j,j)=w.^(j-1);
    P(j,2*j-1)=1;
    P(n/2+j,2*j)=1;
end
```

```
%% loss validation
F_dec=[dftmtx(n/2) zeros(n/2);zeros(n/2) dftmtx(n/2)];
A=[I D; I -D];
[U,S,V] = svd(F_dec);
%% matrix sparser
% retrain large rmse loss
u_first_flag = 1;
[u_sparse,v_sparse,beta_sparse(iter)] = svd_sparser(A,S,U,V,P,n,
   F_actual,u_first_flag,beta_sparse(iter));
F_hat=A*u_sparse*S*v_sparse'*P;
%% quantization
q=3;
value_set = generate_valset(q);
scale_factor = 2.^(q-1);
A_proc = scale_factor .* A;
U_proc = scale_factor .* U;
V_proc = scale_factor .* V;
F_actual_proc = 1./ beta_sparse(iter) .* scale_factor .*
   scale_factor .* scale_factor .* F_hat;
% valid
F_hat_proc = A_proc*U_proc*S*v_proc'*P;
loss_valid=1/n*sqrt(norm(F_actual_proc-F_hat_proc.*beta_quant(iter)
   ,'fro').^2);
% every elements shall be quantized
[m,n] = size(F_hat);
A_quant_container = A_proc;U_quant_container = U_proc;
   V_quant_container = V_proc;
for col = 1 : 2 : m
    for row = 1 : 1 : n
        A_quant_container = quantizer(A_quant_container,S,
            U_quant_container,V_quant_container,P,F_actual_proc,1,
            value_set,col,row,2);
        U_quant_container = quantizer(A_quant_container,S,
            U_quant_container,V_quant_container,P,F_actual_proc,1,
            value_set,col,row,1);
        V_quant_container = quantizer(A_quant_container,S,
            U_quant_container,V_quant_container,P,F_actual_proc,1,
            value_set,col,row,0);
    end
```

```
end
% scale_out
A_quant = A_quant_container./ scale_factor;
U_quant = U_quant_container./ scale_factor;
V_quant = V_quant_container./ scale_factor;
F_hat=A_quant*U_quant*S*V_quant'*P;
de(iter) = S(1,1)
beta_quant(iter)=norm(F_actual,'fro')./norm(F_hat.*beta_sparse(iter
    ),'fro');
beta_container(iter)=beta_quant(iter) .* beta_sparse(iter);
rmse_loss(iter)=1/n*sqrt(norm(F_actual-F_hat.*beta_container(iter),
    'fro').^2);
%% Computational complexity
L  = MyCalComplexity(A_quant_container,U_quant_container,
    V_quant_container');
C(iter)=q*L;
FFT_C(iter)=q*MyCalComplexity(A,U,V');
% FFT_C意味着
end
%% Save Results
disp('beta_value')
beta_container' .* de./64
disp('rmse_loss')
rmse_loss
save data_prob_sq



clc;
clear all;

%% data generation
% pars : n -> the order of the fourier transform matrix
N_MAX = 5;
beta_container = ones([N_MAX,1]);
beta_quant = ones([N_MAX,1]);beta_sparse = ones([N_MAX,1]);
for iter=1:1:N_MAX
n=2^iter;
F_actual=dftmtx(n);
I=eye(n/2);
```

```
D=zeros(n/2,n/2);P=zeros(n,n);
w=exp(-2*pi*1i/n);
for j=1:n/2
    D(j,j)=w.^(j-1);
    P(j,2*j-1)=1;
    P(n/2+j,2*j)=1;
end
%% loss validation
F_dec=[dftmtx(n/2) zeros(n/2);zeros(n/2) dftmtx(n/2)];
A=[I D; I -D];
[U,S,V] = svd(F_dec);
F_hat=A*U*S*V'*P;
%% quantization
q=3;
value_set = generate_valset(q);
scale_factor = 2.^(q-1);
A_proc = scale_factor .* A;
U_proc = scale_factor .* U;
V_proc = scale_factor .* V;
F_actual_proc = scale_factor .* scale_factor .* scale_factor .*
   F_hat;
% valid
F_hat_proc = A_proc*U_proc*S*V_proc'*P;beta = 1;
loss_valid=1/n*sqrt(norm(F_actual_proc-beta.*F_hat_proc,'fro').^2);
% every elements shall be quantized
[m,n] = size(F_hat);
A_quant_container = A_proc;U_quant_container = U_proc;
   V_quant_container = V_proc;
for col = 1 : 1 : m
    for row = 1 : 1 : n
        A_quant_container = quantizer(A_quant_container,S,
           U_quant_container,V_quant_container,P,F_actual_proc,beta
           ,value_set,col,row,2);
        U_quant_container = quantizer(A_quant_container,S,
           U_quant_container,V_quant_container,P,F_actual_proc,beta
           ,value_set,col,row,1);
        V_quant_container = quantizer(A_quant_container,S,
           U_quant_container,V_quant_container,P,F_actual_proc,beta
           ,value_set,col,row,0);
```

```
    end
end
% scale_out
A_quant = A_quant_container./ scale_factor;
U_quant = U_quant_container./ scale_factor;
V_quant = V_quant_container./ scale_factor;
F_hat=A_quant*U_quant*S*V_quant'*P;
% beta=norm(F_hat,'fro')/norm(F_actual,'fro');
rmse_loss(iter)=1/n*sqrt(norm(F_actual-beta.*F_hat,'fro').^2);
beta_container(iter)=norm(F_actual,'fro')/norm(F_hat,'fro');

%% matrix sparser
% retrain large rmse loss
u_first_flag = 1;
[u_sparse,v_sparse,beta_container(iter)] = svd_sparser(A_quant,S,
   U_quant,V_quant,P,n,F_actual,u_first_flag,beta_container(iter));
F_hat=A_quant*u_sparse*S*v_sparse'*P;
beta_container(iter)=norm(F_actual,'fro')./norm(F_hat,'fro');
rmse_loss(iter)=1/n*sqrt(norm(F_actual-beta_container(iter).*F_hat,
   'fro').^2);
disp(n)
disp(rmse_loss)
de(iter) = S(1,1)
%% Computational complexity
L  = MyCalComplexity(A_quant_container,u_sparse,v_sparse');
C(iter)=q*L;
FFT_C(iter)=q*MyCalComplexity(A,U,V');
% FFT_C意味着未优化的
end
%% Save Results
disp('beta_value')
beta_container' .* de./64
disp('rmse_loss')
rmse_loss
save data_prob_qs



function [u_sparse,v_sparse,beta_sparse] = svd_sparser(A,S,U,V,P,n,
```

```
F_actual,u_first_flag,beta)
 if u_first_flag
     mse = zeros(size(U));
     for k=1:n
         for m=1:n
             U_new=U;
             U_new(k,m)=0;
             F_hat=A*U_new*S*V'*P;
             mse(k,m)=1/n*sqrt(norm(F_actual-F_hat.*beta,'fro')
                 .^2);
         end
     end
     clear U_new
     U_sparse=zeros(size(U));
     for k=1:n
         for m=1:n
             mse_line = mse(k,:);
             max_id_1 = find(mse_line == max(mse_line));
             if length(max_id_1) >= 2
                 id1 = max_id_1(1);
                 id2 = max_id_1(2);
             else
                 mse_line(max_id_1) = 0;
                 max_id_2 = find(mse_line == max(mse_line));
                 id1 = max_id_1(1);
                 id2 = max_id_2(1);
             end
         end
         U_sparse(k,id1)=U(k,id1);
         U_sparse(k,id2)=U(k,id2);
     end

     F_hat=A*U_sparse*S*V'*P;
     beta= norm(F_actual,'fro')./norm(F_hat,'fro');

     mse = zeros(size(V));

     for k=1:n
         for m=1:n
```

```
            V_new=V;
            V_new(k,m)=0;
            F_hat=A*U_sparse*S*V_new'*P;
            mse(k,m)=1/n*sqrt(norm(F_actual-F_hat.*beta,'fro'))
                ;
        end
    end
    clear V_new
    V_sparse=zeros(size(V));
    for k=1:n
        for m=1:n
            mse_line = mse(k,:);
            max_id_1 = find(mse_line == max(mse_line));
            if length(max_id_1) >= 2
                id1 = max_id_1(1);
                id2 = max_id_1(2);
            else
                mse_line(max_id_1) = 0;
                max_id_2 = find(mse_line == max(mse_line));
                id1 = max_id_1(1);
                id2 = max_id_2(1);
            end
        end
        V_sparse(k,id1)=V(k,id1);
        V_sparse(k,id2)=V(k,id2);
    end
    F_hat=A*U_sparse*S*V_sparse'*P;
    beta=norm(F_actual,'fro')/norm(F_hat,'fro');

else
    mse = zeros(size(V));

    for k=1:n
        for m=1:n
            V_new=V;
            V_new(k,m)=0;
            F_hat=A*U*S*V_new'*P;
            mse(k,m)=1/n*sqrt(norm(F_actual-F_hat.*beta,'fro'))
                ;
```

```
        end
    end
    clear V_new
    V_sparse=zeros(size(V));
    for k=1:n
        for m=1:n
            mse_line = mse(k,:);
            max_id_1 = find(mse_line == max(mse_line));
            if length(max_id_1) >= 2
                id1 = max_id_1(1);
                id2 = max_id_1(2);
            else
                mse_line(max_id_1) = 0;
                max_id_2 = find(mse_line == max(mse_line));
                id1 = max_id_1(1);
                id2 = max_id_2(1);
            end
        end
        V_sparse(k,id1)=V(k,id1);
        V_sparse(k,id2)=V(k,id2);
    end
    F_hat=A*U*S*V_sparse'*P;
    beta=norm(F_actual,'fro')./norm(F_hat,'fro');


    mse = zeros(size(U));

    for k=1:n
        for m=1:n
            U_new=U;
            U_new(k,m)=0;
            F_hat=A*U_new*S*V_sparse'*P;
            mse(k,m)=1/n*sqrt(norm(F_actual-F_hat.*beta,'fro')
                .^2);
        end
    end
    clear U_new
    U_sparse=zeros(size(U));
    for k=1:n
```

```
            for m=1:n
                mse_line = mse(k,:);
                max_id_1 = find(mse_line == max(mse_line));
                if length(max_id_1) >= 2
                    id1 = max_id_1(1);
                    id2 = max_id_1(2);
                else
                    mse_line(max_id_1) = 0;
                    max_id_2 = find(mse_line == max(mse_line));
                    id1 = max_id_1(1);
                    id2 = max_id_2(1);
                end
            end
            U_sparse(k,id1)=U(k,id1);
            U_sparse(k,id2)=U(k,id2);
        end
        F_hat=A*U_sparse*S*V_sparse'*P;
        beta=norm(F_actual,'fro')./norm(F_hat,'fro');
    end
    % output
    u_sparse = U_sparse;
    v_sparse = V_sparse;
    beta_sparse = beta;
end

function mat_qt = quantizer(A,S,U,V,P,F_actual,beta,value_set,col,
    row,mode)
    % MODE 1 for u , 0 for v , 2 for
    % quantizer for ele[col,row]
    if mode == 1
        % u sub-problem
        if U(col,row) ~= 0
            [~,n] = size(U);
            len = length(value_set);
            mse = zeros(len);
            for iter = 1 : 1 : len
                U_new = U;
                U_new(col,row) = value_set(iter);
                F_hat=A*U_new*S*V'*P;
```

```
            mse(iter)=1/n*sqrt(norm(F_actual-F_hat.*beta,'fro')
                .^2);
        end
        minimum = min(mse);
        minimum_id = find(mse == minimum);
        minimum_id = minimum_id(1);
        mat_qt = U;
        mat_qt(col,row) = value_set(minimum_id);
    else
        mat_qt =   U;
    end
elseif mode == 0
    % v sub-problem
    if V(col,row) ~= 0
        [~,n] = size(V);
        len = length(value_set);
        mse = zeros(len);
        for iter = 1 : 1 : len
            V_new = V;
            V_new(col,row) = value_set(iter);
            F_hat=A*U*S*V_new'*P;
            mse(iter)=1/n*sqrt(norm(F_actual-F_hat.*beta,'fro')
                .^2);
        end
        minimum = min(mse);
        minimum_id = find(mse == minimum);
        minimum_id = minimum_id(1);
        mat_qt = V;
        mat_qt(col,row) = value_set(minimum_id);
    else
        mat_qt =   V;
    end
elseif mode == 2
    if A(col,row) ~= 0
        [~,n] = size(A);
        len = length(value_set);
        mse = zeros(len);
        for iter = 1 : 1 : len
            A_new = A;
```

```
                A_new(col,row) = value_set(iter);
                F_hat=A_new*U*S*V'*P;
                mse(iter)=1/n*sqrt(norm(F_actual-F_hat.*beta,'fro')
                    .^2);
            end
            minimum = min(mse);
            minimum_id = find(mse == minimum);
            minimum_id = minimum_id(1);
            mat_qt = A;
            mat_qt(col,row) = value_set(minimum_id);
        else
            mat_qt = A;
        end
    end
end

function Logical=myLogicalize(A)
    for k=1:size(A,1)
        for j=1:size(A,2)
            Logical1(k,j)=~isreal(A(k,j));
            if isequal(A(k,j), 1i) || isequal(A(k,j), -1i) ||
               isequal(A(k,j), 1+1i) || isequal(A(k,j), 1-1i)||
               isequal(A(k,j), -1-1i)|| isequal(A(k,j), -1-1i)
                Logical1(k, j) = 0;
            end
            end
    end
    Logical = Logical1;
end

function MyCalComplexity=MyCalComplexity(W,U,V)
    logical_1 =myLogicalize(W);
    logical_2 =myLogicalize(U);
    logical_3 =myLogicalize(V);

    Comlogical_1=logical_1*logical_2;
    Comlogical_2 = Comlogical_1*logical_3;
    CalComplexity_1=nnz(Comlogical_1);
    CalComplexity_2=nnz(Comlogical_2);
```

```
    MyCalComplexity =CalComplexity_1+CalComplexity_2;
end

function value_set = generate_valset(q)
    t = 0 : 1 : q-1;
    x = [0 2.^t -2.^t];
    y = [0 2.^t.*1i -2.^t.*1i];y=y';
    value_set = x+y;
    [m,n] = size(value_set);
    value_set = reshape(value_set,[m*n,1]);
end
```

A.4 第 4 问程序

code.m

```
%% data generation
% pars : n -> the order of the fourier transform matrix
clc;clear all;
q = 3;
F4 = dftmtx(4);
F8 = dftmtx(8);
% violence
tic
F = kron(F4,F8);
F_quant = quantizerF(F,1,q);
F_hat = sparser(F_quant,1);
% F_hat = quantizerF(F_sparse,1,q);
beta = norm(F,'fro')./norm(F_hat,'fro');
loss_valid=1/32*sqrt(norm(F-beta*F_hat,'fro').^2);
toc
save violence.mat


%% data generation
% pars : n -> the order of the fourier transform matrix
clc;clear all;
q = 3;
F4 = dftmtx(4);
F8 = dftmtx(8);
```

```
F = kron(F4,F8);
% decom vio
tic
F4_approx = sparser(quantizerF(F4,1,q),1);
F8_approx = sparser(quantizerF(F8,1,q),1);
Fk = kron(F4_approx,F8_approx);
F_hat = sparser(quantizerF(Fk,1,q),1);
beta = norm(F,'fro')/norm(F_hat,'fro');
loss_valid=1/32*sqrt(norm(F-beta*F_hat,'fro').^2);
toc
save decomvio.mat

%% data generation
% pars : n -> the order of the fourier transform matrix
clc;clear all;
q = 3;
F4 = dftmtx(4);
F8 = dftmtx(8);
F = kron(F4,F8);
% decom vio
tic
n=4;I=eye(n/2);D=zeros(n/2,n/2);P=zeros(n,n);w=exp(-2*pi*1i/n);
for j=1:n/2
    D(j,j)=w.^(j-1);
    P(j,2*j-1)=1;
    P(n/2+j,2*j)=1;
end
F_dec_4=[dftmtx(n/2) zeros(n/2);zeros(n/2) dftmtx(n/2)];A4=[I D; I
   -D];P4 = P;
[U4,S4,V4] = svd(F_dec_4);

n=8;I=eye(n/2);D=zeros(n/2,n/2);P=zeros(n,n);w=exp(-2*pi*1i/n);
for j=1:n/2
    D(j,j)=w.^(j-1);
    P(j,2*j-1)=1;
    P(n/2+j,2*j)=1;
end
F_dec_8=[dftmtx(n/2) zeros(n/2);zeros(n/2) dftmtx(n/2)];A8=[I D; I
   -D];P8 = P;
```

```
[U8,S8,V8] = svd(F_dec_8);

A = kron(A4,A8);
U = kron(U4,U8);
S = kron(S4,S8);
V = kron(V4,V8);
P = kron(P4,P8);

A_approx = sparser(quantizerF(A,1,q),1);
U_approx = sparser(quantizerF(U,1,q),1);
S_approx = sparser(quantizerF(S,1,q),1);
V_approx = sparser(quantizerF(V,1,q),1);
P_approx = sparser(quantizerF(P,1,q),1);

F_hat = A_approx*U_approx*S_approx*V_approx*P_approx;
beta = norm(F,'fro')/ norm(F_hat,'fro');
loss_valid=1/32*sqrt(norm(F-F_hat,'fro').^2);
toc
save kronvio.mat


function [u_sparse,v_sparse,beta_sparse] = svd_sparser(A,S,U,V,P,n,
   F_actual,u_first_flag,beta)
    if u_first_flag
        mse = zeros(size(U));
        for k=1:n
            for m=1:n
                U_new=U;
                U_new(k,m)=0;
                F_hat=A*U_new*S*V'*P;
                mse(k,m)=1/n*sqrt(norm(beta * F_actual-F_hat,'fro')
                    .^2);
            end
        end
        clear U_new
        U_sparse=zeros(size(U));
        for k=1:n
            for m=1:n
                mse_line = mse(k,:);
```

```
            max_id_1 = find(mse_line == max(mse_line));
            if length(max_id_1) >= 2
                id1 = max_id_1(1);
                id2 = max_id_1(2);
            else
                mse_line(max_id_1) = 0;
                max_id_2 = find(mse_line == max(mse_line));
                id1 = max_id_1(1);
                id2 = max_id_2(1);
            end
        end
        U_sparse(k,id1)=U(k,id1);
        U_sparse(k,id2)=U(k,id2);
    end

    F_hat=A*U_sparse*S*V'*P;
    beta=norm(F_hat,'fro')/norm(F_actual,'fro');

    mse = zeros(size(V));

    for k=1:n
        for m=1:n
            V_new=V;
            V_new(k,m)=0;
            F_hat=A*U_sparse*S*V_new'*P;
            mse(k,m)=1/n*sqrt(norm(beta * F_actual-F_hat,'fro')
                );
        end
    end
    clear V_new
    V_sparse=zeros(size(V));
    for k=1:n
        for m=1:n
            mse_line = mse(k,:);
            max_id_1 = find(mse_line == max(mse_line));
            if length(max_id_1) >= 2
                id1 = max_id_1(1);
                id2 = max_id_1(2);
            else
```

```
                mse_line(max_id_1) = 0;
                max_id_2 = find(mse_line == max(mse_line));
                id1 = max_id_1(1);
                id2 = max_id_2(1);
            end
        end
        V_sparse(k,id1)=V(k,id1);
        V_sparse(k,id2)=V(k,id2);
    end
    F_hat=A*U_sparse*S*V_sparse'*P;
    beta=norm(F_hat,'fro')/norm(F_actual,'fro');

else
    mse = zeros(size(V));

    for k=1:n
        for m=1:n
            V_new=V;
            V_new(k,m)=0;
            F_hat=A*U*S*V_new'*P;
            mse(k,m)=1/n*sqrt(norm(beta * F_actual-F_hat,'fro')
                );
        end
    end
    clear V_new
    V_sparse=zeros(size(V));
    for k=1:n
        for m=1:n
            mse_line = mse(k,:);
            max_id_1 = find(mse_line == max(mse_line));
            if length(max_id_1) >= 2
                id1 = max_id_1(1);
                id2 = max_id_1(2);
            else
                mse_line(max_id_1) = 0;
                max_id_2 = find(mse_line == max(mse_line));
                id1 = max_id_1(1);
                id2 = max_id_2(1);
            end
```

```
        end
        V_sparse(k,id1)=V(k,id1);
        V_sparse(k,id2)=V(k,id2);
    end
    F_hat=A*U*S*V_sparse'*P;
    beta=norm(F_hat,'fro')/norm(F_actual,'fro');


    mse = zeros(size(U));

    for k=1:n
        for m=1:n
            U_new=U;
            U_new(k,m)=0;
            F_hat=A*U_new*S*V_sparse'*P;
            mse(k,m)=1/n*sqrt(norm(beta*F_actual-F_hat,'fro')
                .^2);
        end
    end
    clear U_new
    U_sparse=zeros(size(U));
    for k=1:n
        for m=1:n
            mse_line = mse(k,:);
            max_id_1 = find(mse_line == max(mse_line));
            if length(max_id_1) >= 2
                id1 = max_id_1(1);
                id2 = max_id_1(2);
            else
                mse_line(max_id_1) = 0;
                max_id_2 = find(mse_line == max(mse_line));
                id1 = max_id_1(1);
                id2 = max_id_2(1);
            end
        end
        U_sparse(k,id1)=U(k,id1);
        U_sparse(k,id2)=U(k,id2);
    end
    F_hat=A*U_sparse*S*V_sparse'*P;
```

```
        beta=norm(F_hat,'fro')/norm(F_actual,'fro');
    end
    % output
    u_sparse = U_sparse;
    v_sparse = V_sparse;
    beta_sparse = beta;
end

function mat_qt = quantizerF(F_unscale,beta,q)
    value_set = generate_valset(q);
    scale_factor = 2.^(q-1);
    F = scale_factor .* F_unscale;
    [~,n] = size(F);
    for col = 1 : 1 :n
        for row = 1: 1 : n
            F_actual = F;
            len = length(value_set);
            mse = zeros(len);
            for iter = 1 : 1 : len
                F_new = F;
                F_new(col,row) = value_set(iter);
                F_hat=F_new;
                mse(iter)=1/n*sqrt(norm(F_actual-beta * F_hat,'fro'
                    ).^2);
            end
            minimum = min(mse);
            minimum_id = find(mse == minimum);
            minimum_id = minimum_id(1);
            F(col,row) = value_set(minimum_id);
        end
    end
    mat_qt = F ./ scale_factor;
end


function F_sparse = sparser(F,beta)
    mse = zeros(size(F));
    F_actual = F;
    [~,n] = size(F);
```

```
    for k=1:n
        for m=1:n
            F_new=F;
            F_new(k,m)=0;
            F_hat=F_new;
            mse(k,m)=1/n*sqrt(norm(beta * F_actual-F_hat,'fro').^2)
                ;
        end
    end
    clear F_new
    F_sparse=zeros(size(F));
    for k=1:n
        for m=1:n
            mse_line = mse(k,:);
            max_id_1 = find(mse_line == max(mse_line));
            if length(max_id_1) >= 2
                id1 = max_id_1(1);
                id2 = max_id_1(2);
            else
                mse_line(max_id_1) = 0;
                max_id_2 = find(mse_line == max(mse_line));
                id1 = max_id_1(1);
                id2 = max_id_2(1);
            end
        end
        F_sparse(k,id1)=F(k,id1);
        F_sparse(k,id2)=F(k,id2);
    end
end

function value_set = generate_valset(q)
    t = 0 : 1 : q-1;
    x = [0 2.^t -2.^t];
    y = [0 2.^t.*1i -2.^t.*1i];y=y';
    value_set = x+y;
    [m,n] = size(value_set);
    value_set = reshape(value_set,[m*n,1]);
end
```

A.5 第 5 问程序

code.m

```
clc;
clear all;

%% data generation
% pars : n -> the order of the fourier transform matrix
N_MAX = 5;q= 2;
Ntemp = 32;rmse_loss = zeros([N_MAX,Ntemp+1]);CMultiple=zeros([
    N_MAX,Ntemp+1]);
beta_container = ones([N_MAX,1+Ntemp]);
beta_quant = ones([N_MAX,1]);beta_sparse = ones([N_MAX,1]);
for iter=3:N_MAX
    n=2^iter;
    F_actual=dftmtx(n);
    I=eye(n/2);
    D=zeros(n/2,n/2);P=zeros(n,n);
    w=exp(-2*pi*1i/n);
    for j=1:n/2
        D(j,j)=w.^(j-1);
        P(j,2*j-1)=1;
        P(n/2+j,2*j)=1;
    end
%% loss validation
F_dec=[dftmtx(n/2) zeros(n/2);zeros(n/2) dftmtx(n/2)];
A=[I D; I -D];
[U,S,V] = svd(F_dec);
F_hat=A*U*S*V'*P;
%% quantization
value_set = generate_valset(q);
scale_factor = 2.^(q-1);
A_proc = scale_factor .* A;
U_proc = scale_factor .* U;
V_proc = scale_factor .* V;
F_actual_proc = scale_factor .* scale_factor .* scale_factor .*
    F_actual;
% valid
F_hat_proc = A_proc*U_proc*S*V_proc'*P;
%beta_r是最后beta的倒数
```

```
beta = 1;
loss_valid=1/n*sqrt(norm(beta*F_hat_proc-F_actual_proc,'fro').^2);
% every elements shall be quantized
[m,n] = size(F_hat);
A_quant_container = A_proc;U_quant_container = U_proc;
   V_quant_container = V_proc;
for col = 1 : 1 : m
    for row = 1 : 1 : n
        A_quant_container = quantizer(A_quant_container,S,
           U_quant_container,V_quant_container,P,F_actual_proc,beta
           ,value_set,col,row,2,q);
        U_quant_container = quantizer(A_quant_container,S,
           U_quant_container,V_quant_container,P,F_actual_proc,beta
           ,value_set,col,row,1,q);
        V_quant_container = quantizer(A_quant_container,S,
           U_quant_container,V_quant_container,P,F_actual_proc,beta
           ,value_set,col,row,0,q);
    end
end
% scale_out
A_quant = A_quant_container./scale_factor;
U_quant = U_quant_container./scale_factor;
V_quant = V_quant_container./ scale_factor;
F_hat=A_quant*U_quant*S*V_quant'*P;
% beta=norm(F_hat,'fro')/norm(F_actual,'fro');
%rmse_loss(iter=i/n*sqrt(norm(F_actual-beta.*F_hat,'fro').^2);
beta_container(iter,1)=norm(F_actual,'fro')/norm(F_hat,'fro');

%% matrix sparser
% retrain large rmse loss
A=A_quant;U=U_quant;V=V_quant;
u_first_flag = 1;
[u_sparse,v_sparse,beta_container(iter,1)] = svd_sparser(A,S,U,V,P,
   n,F_actual,u_first_flag,beta_container(iter,1));
F_hat=A*u_sparse*S*v_sparse'*P;
beta_container(iter,1)=norm(F_actual,'fro')./norm(F_hat,'fro');
rmse_loss(iter,1)=1/n*sqrt(norm(F_actual-beta_container(iter,1).*
   F_hat,'fro').^2);
CMultiple(iter,1)=q*MyCalComplexity(A_quant_container,
```

```
    U_quant_container,V_quant_container');
%disp(rmse_loss)
%% satisfy F_actual \approx beta_container(iter).*F_hat
for temp=1:Ntemp
X = 1./beta_container(iter,temp) * pinv(F_hat) * F_actual;
distance = max(max(abs(X)));
X = X ./ distance;
beta_container(iter,1+temp) = beta_container(iter,temp) .* distance
    ;
X_quant=quantizerF(X,1,q);
X = sparser(X_quant,1);
F_hat=F_hat*X;
CMultiple(iter,temp+1)=CMultiple(iter,temp)+q*MyCalComplexity(F_hat
    ,X,eye(2^iter));
rmse_loss(iter,1+temp)=1/n*sqrt(norm(F_actual-beta_container(iter
    ,1+temp).*F_hat,'fro').^2);
end
%% Computational complexity
% L_ini  = MyCalComplexity(A_quant_container,U_quant_container,
    V_quant_container');
%
% C(iter)=q*L;
% FFT_C(iter)=q*MyCalComplexity(A,U,V');
% FFT_C意味着未优化的
end
%% Save Results
save data_prob_qs


function [u_sparse,v_sparse,beta_sparse] = svd_sparser(A,S,U,V,P,n,
    F_actual,u_first_flag,beta)
     if u_first_flag
         mse = zeros(size(U));
         for k=1:n
             for m=1:n
                 U_new=U;
                 U_new(k,m)=0;
                 F_hat=A*U_new*S*V'*P;
                 mse(k,m)=1/n*sqrt(norm(beta *F_hat- F_actual,'fro')
```

```
                .^2);
        end
    end
    clear U_new
    U_sparse=zeros(size(U));
    for k=1:n
        for m=1:n
            mse_line = mse(k,:);
            max_id_1 = find(mse_line == max(mse_line));
            if length(max_id_1) >= 2
                id1 = max_id_1(1);
                id2 = max_id_1(2);
            else
                mse_line(max_id_1) = 0;
                max_id_2 = find(mse_line == max(mse_line));
                id1 = max_id_1(1);
                id2 = max_id_2(1);
            end
        end
        U_sparse(k,id1)=U(k,id1);
        U_sparse(k,id2)=U(k,id2);
    end

    F_hat=A*U_sparse*S*V'*P;
    beta=norm(F_actual,'fro')/norm(F_hat,'fro');

    mse = zeros(size(V));

    for k=1:n
        for m=1:n
            V_new=V;
            V_new(k,m)=0;
            F_hat=A*U_sparse*S*V_new'*P;
            mse(k,m)=1/n*sqrt(norm(beta *F_hat- F_actual,'fro')
                .^2);
        end
    end
    clear V_new
    V_sparse=zeros(size(V));
```

```
        for k=1:n
            for m=1:n
                mse_line = mse(k,:);
                max_id_1 = find(mse_line == max(mse_line));
                if length(max_id_1) >= 2
                    id1 = max_id_1(1);
                    id2 = max_id_1(2);
                else
                    mse_line(max_id_1) = 0;
                    max_id_2 = find(mse_line == max(mse_line));
                    id1 = max_id_1(1);
                    id2 = max_id_2(1);
                end
            end
            V_sparse(k,id1)=V(k,id1);
            V_sparse(k,id2)=V(k,id2);
        end
        F_hat=A*U_sparse*S*V_sparse'*P;
        beta=norm(F_actual,'fro')/norm(F_hat,'fro');

    else
        mse = zeros(size(V));

        for k=1:n
            for m=1:n
                V_new=V;
                V_new(k,m)=0;
                F_hat=A*U*S*V_new'*P;
                mse(k,m)=1/n*sqrt(norm(beta *F_hat- F_actual,'fro')
                    .^2);
            end
        end
        clear V_new
        V_sparse=zeros(size(V));
        for k=1:n
            for m=1:n
                mse_line = mse(k,:);
                max_id_1 = find(mse_line == max(mse_line));
                if length(max_id_1) >= 2
```

```
                id1 = max_id_1(1);
                id2 = max_id_1(2);
            else
                mse_line(max_id_1) = 0;
                max_id_2 = find(mse_line == max(mse_line));
                id1 = max_id_1(1);
                id2 = max_id_2(1);
            end
        end
        V_sparse(k,id1)=V(k,id1);
        V_sparse(k,id2)=V(k,id2);
    end
    F_hat=A*U*S*V_sparse'*P;
    beta=norm(F_actual,'fro')/norm(F_hat,'fro');


    mse = zeros(size(U));

    for k=1:n
        for m=1:n
            U_new=U;
            U_new(k,m)=0;
            F_hat=A*U_new*S*V_sparse'*P;
            mse(k,m)=1/n*sqrt(norm(beta *F_hat- F_actual,'fro')
                .^2);
        end
    end
    clear U_new
    U_sparse=zeros(size(U));
    for k=1:n
        for m=1:n
            mse_line = mse(k,:);
            max_id_1 = find(mse_line == max(mse_line));
            if length(max_id_1) >= 2
                id1 = max_id_1(1);
                id2 = max_id_1(2);
            else
                mse_line(max_id_1) = 0;
                max_id_2 = find(mse_line == max(mse_line));
```

```
                    id1 = max_id_1(1);
                    id2 = max_id_2(1);
                end
            end
            U_sparse(k,id1)=U(k,id1);
            U_sparse(k,id2)=U(k,id2);
        end
        F_hat=A*U_sparse*S*V_sparse'*P;
        beta=norm(F_actual,'fro')/norm(F_hat,'fro');
    end
    % output
    u_sparse = U_sparse;
    v_sparse = V_sparse;
    beta_sparse = beta;
end

function [outputArg1,outputArg2] = svd_quantizer(q,F)

    q=3; value_set = generate_valset(q); scale_factor = 2.^(q-1);
    A_proc = scale_factor .* A; U_proc = scale_factor .* U; V_proc
       = scale_factor .* V;
    F_actual_proc = scale_factor .* scale_factor .* scale_factor .*
        F;
    % every elements shall be quantized
    [m,n] = size(F_hat);
    A_quant_container = A_proc;U_quant_container = U_proc;
       V_quant_container = V_proc;
    for col = 2 : 1 : m
        for row = 1 : 1 : n
            A_quant_container = quantizer(A_quant_container,S,
               U_quant_container,V_quant_container,P,F_actual_proc,
               beta,value_set,col,row,2);
            U_quant_container = quantizer(A_quant_container,S,
               U_quant_container,V_quant_container,P,F_actual_proc,
               beta,value_set,col,row,1);
            V_quant_container = quantizer(A_quant_container,S,
               U_quant_container,V_quant_container,P,F_actual_proc,
               beta,value_set,col,row,0);
        end
```

```
    end
    % scale_out
    A_quant = A_quant_container./ scale_factor;
    U_quant = U_quant_container./ scale_factor;
    V_quant = V_quant_container./ scale_factor;
    F_hat=A_quant*U_quant*S*V_quant'*P;
    beta=norm(F_hat,'fro')/norm(F_actual,'fro');
end

function [F_sparse,beta_sparse] = sparser(F,beta)
    mse = zeros(size(F));
    F_actual = F;
    [~,n] = size(F);
    for k=1:n
        for m=1:n
            F_new=F;
            F_new(k,m)=0;
            F_hat=F_new;
            mse(k,m)=1/n*sqrt(norm(F_actual-beta * F_hat,'fro').^2)
                ;
        end
    end
    clear F_new
    F_sparse=zeros(size(F));
    for k=1:n
        for m=1:n
            mse_line = mse(k,:);
            max_id_1 = find(mse_line == max(mse_line));
            if length(max_id_1) >= 2
                id1 = max_id_1(1);
                id2 = max_id_1(2);
            else
                mse_line(max_id_1) = 0;
                max_id_2 = find(mse_line == max(mse_line));
                id1 = max_id_1(1);
                id2 = max_id_2(1);
            end
        end
        F_sparse(k,id1)=F(k,id1);
```

```
        F_sparse(k,id2)=F(k,id2);
    end
    beta_sparse = 1;
end

function [mat_qt,beta_quant] = quantizerF(F_unscale,beta,q)
    value_set = generate_valset(q);
    scale_factor = 2.^(q-1);
    F = scale_factor .* F_unscale;
    [~,n] = size(F);
    for col = 1 : 1 :n
        for row = 1: 1 : n
            F_actual = F;
            len = length(value_set);
            mse = zeros(len);
            for iter = 1 : 1 : len
                F_new = F;
                F_new(col,row) = value_set(iter);
                F_hat=F_new;
                mse(iter)=1/n*sqrt(norm(F_actual-beta * F_hat,'fro'
                    ).^2);
            end
            minimum = min(mse);
            minimum_id = find(mse == minimum);
            minimum_id = minimum_id(1);
            F(col,row) = value_set(minimum_id);
        end
    end
    mat_qt = F ./ scale_factor;
    beta_quant = norm(F_unscale,'fro')./norm(mat_qt,'fro');
end

function mat_qt = quantizer(A,S,U,V,P,F_actual,beta,value_set,col,
    row,mode,q)
    % MODE 1 for u , 0 for v , 2 for
    % quantizer for ele[col,row]
    if mode == 1
        % u sub-problem
        if U(col,row) ~= 0
```

```
            [~,n] = size(U);
            len = length(value_set);
            mse = zeros([len,1]);
            for iter = 1 : 1 : len
                U_new = U;
                U_new(col,row) = value_set(iter);
                F_hat=A*U_new*S*V'*P;
                mse(iter)=1/n*sqrt(norm(F_actual-1./2.^(q-1)./2.^(q
                    -1)./2.^(q-1)*F_hat.*beta,'fro').^2);
            end
            minimum = min(mse);
            minimum_id = find(mse == minimum);
            minimum_id = minimum_id(1);
            mat_qt = U;
            mat_qt(col,row) = value_set(minimum_id);
        else
            mat_qt =   U;
        end
    elseif mode == 0
        % v sub-problem
        if V(col,row) ~= 0
            [~,n] = size(V);
            len = length(value_set);
            mse = zeros([len,1]);
            for iter = 1 : 1 : len
                V_new = V;
                V_new(col,row) = value_set(iter);
                F_hat=A*U*S*V_new'*P;
                mse(iter)=1/n*sqrt(norm(F_actual-1./2.^(q-1)./2.^(q
                    -1)./2.^(q-1)*F_hat.*beta,'fro').^2);
            end
            minimum = min(mse);
            minimum_id = find(mse == minimum);
            minimum_id = minimum_id(1);
            mat_qt = V;
            mat_qt(col,row) = value_set(minimum_id);
        else
            mat_qt =   V;
        end
```

```
    elseif mode == 2
        if A(col,row) ~= 0
            [~,n] = size(A);
            len = length(value_set);
            mse = zeros([len,1]);
            for iter = 1 : 1 : len
                A_new = A;
                A_new(col,row) = value_set(iter);
                F_hat=A_new*U*S*V'*P;
                mse(iter)=1/n*sqrt(norm(F_actual-1./2.^(q-1)./2.^(q
                    -1)./2.^(q-1)*F_hat.*beta,'fro').^2);
            end
            minimum = min(mse(mse~=0));
            minimum_id = find(mse == minimum);
            minimum_id = minimum_id(1);
            mat_qt = A;
            mat_qt(col,row) = value_set(minimum_id);
        else
            mat_qt = A;
        end
    end
end

function Logical=myLogicalize(A)
    for k=1:size(A,1)
        for j=1:size(A,2)
            Logical1(k,j)=~isreal(A(k,j));
            if isequal(A(k,j), 1i) || isequal(A(k,j), -1i) ||
               isequal(A(k,j), 1+1i) || isequal(A(k,j), 1-1i)||
               isequal(A(k,j), -1+1i)||isequal(A(k,j), -1+1i)
                Logical1(k, j) = 0;
            end
            end
    end
    Logical = Logical1;
end

function MyCalComplexity=MyCalComplexity(W,U,V)
    logical_1 =myLogicalize(W);
```

```
    logical_2 =myLogicalize(U);
    logical_3 =myLogicalize(V);

    Comlogical_1=logical_1*logical_2;
    Comlogical_2 = Comlogical_1*logical_3;
    CalComplexity_1=nnz(Comlogical_1);
    CalComplexity_2=nnz(Comlogical_2);
    MyCalComplexity =CalComplexity_1+CalComplexity_2;
end

function value_set = generate_valset(q)
    t = 0 : 1 : q-1;
    x = [0 2.^t -2.^t];
    y = [0 2.^t.*1i -2.^t.*1i];y=y';
    value_set = x+y;
    [m,n] = size(value_set);
    value_set = reshape(value_set,[m*n,1]);
end

function rmse_loss = CalRMSE(A,B)
    rmse_loss=1/size(A,1)*sqrt(norm(A-B,'fro').^2);
end
```