中国研究生创新实践系列大赛

“华为杯”第十八届中国研究生

数学建模竞赛

学　　校	哈尔滨工程大学
参赛队号	**21102170024**
队员姓名	**1.赵丹丹**
	2.肖悦
	3.赵梓君

中国研究生创新实践系列大赛
“华为杯”第十八届中国研究生
数学建模竞赛

题 目　　相关矩阵组的低复杂度计算和存储建模

摘　　要：

在信号处理的各个领域如无线通信、阵列雷达、计算机视觉等，矩阵始终都是这些信号进行数学计算与推导的一个重要工具，这一系列的信号矩阵间往往在某些维度会存在一定的关联特性，由相关矩阵组来表示。而在物联网迅猛发展的今天，随着各类传感器阵列规模的持续扩大，常规处理算法对计算和存储的需求成倍增长，给各类大规模系统的实现成本和功耗的需求带来了巨大的挑战。因此，需要充分地挖掘矩阵间的关联性，降低相关矩阵组计算、压缩、解压缩流程中各步骤的运算与存储的复杂度。在不影响结果精度的前提下，建立或优化近似计算相关矩阵组的合适方法和模型，从而使得整体流程的计算复杂度和存储复杂度得到明显的降低，便于工程实践用。

针对问题一：本题不用考虑矩阵压缩与解压缩问题，只需要在满足 $\rho_{\min}(\boldsymbol{W}) \geq \rho_{th} = 0.99$，以及计算复杂度尽可能低的约束条件下，建立近似计算的模型。我们从影响复杂度的几个关键环节出发，设计两步法模型 $\hat{\boldsymbol{V}} = f_1(\boldsymbol{H}), \hat{\boldsymbol{W}} = f_2(\hat{\boldsymbol{V}})$。在该模型中，第一步利用给定矩阵间的关联性，进行加权变步长的合理采样，并在最后通过最邻近插值的方法进行数据补齐，使得整体矩阵组计算量从 K 次运算直接降到了 K/λ 次。第二步，对比多个奇异值分解（SVD）方法，采用 Shlien S 提出的近似奇异值 SVD 分解的方法，将矩阵进行矢量化拼接，进而迭代获得右奇异向量矩阵，使得该部分的计算复杂度由常规方法的 $O(N^3M^3 + M^3)$ 降到了 $O(3N^2M + 3M^2N)$。第三步，对比矩阵求逆的各类求解，我们利用基于 Nuemann 级数 1 阶展开近似矩阵求逆的方法，将复杂度较高的矩阵求逆过程，转换成一个最小二乘法的迭代求解问题，也使得该部分复杂度由 $\mathrm{O}\left((LJ)^3\right)$ 降低到 $\mathrm{O}\left((LJ)^2\right)$。最后，利用所给的 6 组矩阵数据分析发现，与常规算法相比，计算复杂度大约降低 500 倍（10^2 量级），说明了模型设计的有效性。

针对问题二：本题主要在满足误差 $err_{\boldsymbol{H}} \leq E_{th1} = -30\,\mathrm{dB}$、$err_{\boldsymbol{W}} \leq E_{th2} = -30\,\mathrm{dB}$ 的约束条件下，设计矩阵组的压缩与解压缩模型，使得模型的存储和计算复杂度都最低。我们主要借助于稀疏化和压缩感知的相关理论进行模型设计。对于压缩模型 $P_1(\bullet)$、$P_2(\bullet)$，我们先进行一次自适应 Contourlet 变换操作实现矩阵稀疏化，再引入利用概率划分稀疏域（SDP）理论的压缩感知算法完成矩阵内元素的数据压缩，最后采取类似于视频信号的帧间压缩的方法，通过相关性补偿，实现矩阵间的压缩处理。而对于解压缩模型 $G_1(\bullet)$、$G_2(\bullet)$,我们利用矩阵间、矩阵内的相关度，设置合适的软硬阈值，采取 MH-BCS-SPL 这一匹配追踪算法重构原始数据，实现数据的解压缩。最后，利用本文所给的 6 组数据进行分析，发现在满足规定误差的条件下，存储复杂度降低约 70%，计算复杂度也降低约 15%，再次验证了模型设计的有效性。

针对问题三：该题仅有 $\rho_{\min}(\boldsymbol{W}) \geq \rho_{th} = 0.99$ 这个约束条件，需要在使得存储和计算复杂度尽可能低的情况下，进行整个流程的黑盒模型的想象与设计，有较大的开放性。针对该黑盒模型，我们主要有以下几种想法。一是在问题 1 和 2 的基础上，将 1 和 2 的模型，进行简单的叠加，自然符合题目要求。二是借鉴求解半光滑方程组的正则化牛顿法，对问题 1 和 2 的模型，进行合适的权值匹配，实现整体流程的存储和计算复杂度的优化。三是，完全打破 1 和 2 题的思维局限，结合已有的 H 和 W 的数据，利用合适的神经网络（如 RNN），直接对压缩后数据进行计算处理，并根据数据中保留的特征来进行其余矩阵数据的填补，学习出适合该计算的网络模型，实现大数据的计算和存储。

关键词：相关矩阵组，计算复杂度，存储复杂度，变步长采样，近似奇异值 SVD 分解法，基于 Nuemann 级数展开矩阵求逆，自适应 Contourlet 变换，概率划分稀疏域（SDP）理论，匹配追踪，半光滑牛顿法

目录

1. 问题重述

1.1 问题背景

矩阵作为数学计算与推导中的一个重要工具，在计算机视觉、相控阵雷达、无线通信等领域都有广泛的应用，无论图像，音频，通信信号，最后通常都要呈现为矩阵形式进行计算，而这一系列的矩阵间或者矩阵内通常在某些维度会存在一定的关联特性，因此在数学领域，常常用相关矩阵组来表示。而在物联网迅猛发展的今天，随着各类传感器阵列规模的持续扩大，常规处理算法也对计算和存储的需求成倍增长，对处理器件或算法的实现成本和功耗的需求也随之成倍增长。因此计算复杂度和存储复杂度[1]是大规模器件系统等得以实现的主要挑战。

对于相关矩阵组的运算，以及求解流程，文中所给的常规算法流程的建模描述主要表述如下：

已知一组复数矩阵 $\mathbf{H}=\left\{\mathbf{H}_{j,k}\right\}$, $\mathbf{H}_{j,k}\in\mathbb{C}^{M\times N}, j=1,\ldots,J,k=1,\ldots,K$ 。其中，矩阵间及同矩阵元素之间有一定的关联性，即 $\left\{\mathbf{H}_{j,1},\mathbf{H}_{j,2},\mathbf{H}_{j,3},\cdots,\mathbf{H}_{j,K}\right\}$ 之间存在关联，且矩阵

$$\boldsymbol{H}_{j,k}=\begin{bmatrix} h_{1,1}^{(j,k)} & h_{1,2}^{(j,k)} & h_{1,3}^{(j,k)} & \cdots & h_{1,N}^{(j,k)} \\ h_{2,1}^{(j,k)} & h_{2,2}^{(j,k)} & h_{2,3}^{(j,k)} & \cdots & h_{2,N}^{(j,k)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ h_{M,1}^{(j,k)} & h_{M,2}^{(j,k)} & h_{M,3}^{(j,k)} & \cdots & h_{M,N}^{(j,k)} \end{bmatrix} \tag{1.1}$$

的各个元素间 $\left\{h_{m,n}^{(j,k)}\right\}, m=1,\ldots,M,n=1,\ldots,N$ ，也存在着关联。

本文定义矩阵组 $\mathbf{H}=\left\{\mathbf{H}_{j,k}\right\}$ 的一组数学运算，第一步先通过奇异值分解求解中间矩阵，中间矩阵 $\boldsymbol{V}=\left\{\boldsymbol{V}_{j,k}\right\}$ 由以下公式给出:

$$\begin{gathered}\boldsymbol{V}_{j,k}=svd\left(\boldsymbol{H}_{j,k}\right),\text{ or }\boldsymbol{H}_{j,k}=\boldsymbol{U}_{j,k}\boldsymbol{S}_{j,k}\tilde{\boldsymbol{V}}_{j,k}^{H},\boldsymbol{V}_{j,k}=\tilde{\boldsymbol{V}}_{j,k}^{H}(:,1:L)\\ j=1,\ldots,J;k=1,\ldots,K\end{gathered} \tag{1.2}$$

其中，$svd\left(\bullet\right)$ 是矩阵的奇异值分解中求解右奇异向量的过程，$\boldsymbol{V}_{j,k}$ 是 $\boldsymbol{H}_{j,k}$ 的前 L 个右奇异向量构成的矩阵。

第二步，将 $\boldsymbol{V}_{j,k}$ 中不同 j 下标、相同 k 下标的矩阵块进行横向拼接，得到维度 $N\times LJ$ 的 $\boldsymbol{V}_{k}=\left[\boldsymbol{V}_{1,k},\cdots,\boldsymbol{V}_{j,k},\cdots,\boldsymbol{V}_{J,k}\right]$，然后由下式获得 $\boldsymbol{W}_{k}$：

$$\boldsymbol{W}_{k}=\boldsymbol{V}_{k}\left(\boldsymbol{V}_{k}^{H}\boldsymbol{V}_{k}+\sigma^{2}\boldsymbol{I}\right)^{-1} \tag{1.3}$$

其中，σ^{2} 为常数，$\boldsymbol{I}$ 为 $LJ\times LJ$ 维单位矩阵。

最后，再将各 $\boldsymbol{W}_{k}$ 按照以下的公式进行拆解：

$$\boldsymbol{W}_{k}=\left[\boldsymbol{W}_{1,k},\ldots,\boldsymbol{W}_{j,k},\ldots,\boldsymbol{W}_{J,k}\right] \tag{1.4}$$

其中，$\boldsymbol{W}_{j,k}$ 是维度为 $N\times L$，顺序排列的子矩阵，整合得到的 $\boldsymbol{W}$ 即为最后的输出结果。

在此过程中，还会涉及到数据的存储空间和解压缩问题，因此，相关矩阵组的整体流程如下图 1.1

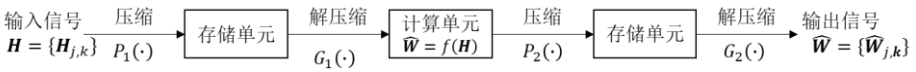


图 1.1 相关矩阵组处理流程

综上，本文的研究目标，主要是在满足相应的最低建模精度，以及数据解压缩误差的前提下，建立合适的模型，实现低复杂度的计算和存储，对实际生活中传感器数量/阵列天线等持续扩大的系统实现及改进，具有十分重要的价值和意义。

1.2 问题的提出

1.2.1 问题 1 内容

实现相关矩阵组计算复杂度的降低。基于本题所给定的六组输入信号数据 $\boldsymbol{H}$，暂不考虑数据 $\boldsymbol{H}$ 的压缩与解压缩，剖析矩阵数据间的关联性，合理设计近似分析模型 $\hat{\boldsymbol{W}} = f(\boldsymbol{H})$ 或者 $\hat{\boldsymbol{V}} = f_1(\boldsymbol{H}), \hat{\boldsymbol{W}} = f_2(\hat{\boldsymbol{V}})$，使得在满足相应的最低建模精度 $\rho_{\min}(\boldsymbol{W}) \geq \rho_{th} = 0.99$ 的情况下，整体流程的计算复杂度最低。

优化目标：无论采用直接建模设计，还是采用两步法经过中间矩阵 $\boldsymbol{V}$ 间接建模估计，整体估计流程的计算复杂度尽可能达到最低。

约束条件：

1) 如果是不考虑中间结果 $\boldsymbol{V}$，直接设计运算模型 $\hat{\boldsymbol{W}} = f(\boldsymbol{H})$，需要满足 $\rho_{\min}(\boldsymbol{W}) \geq \rho_{th} = 0.99$ 建模精度要求；

2)如果考虑中间结果 $\boldsymbol{V}$，两步法设计分析模型 $\hat{\boldsymbol{V}} = f_1(\boldsymbol{H}), \hat{\boldsymbol{W}} = f_2(\hat{\boldsymbol{V}})$，需要同时满足 $\rho_{\min}(\boldsymbol{V}) \geq \rho_{th} = 0.99, \rho_{\min}(\boldsymbol{W}) \geq \rho_{th} = 0.99$ 的建模估计精度要求。

3)暂不考虑数据压缩与解压缩的影响.。

相关定义补充：

计算复杂度的定义：表示由矩阵组 $\boldsymbol{H}$ 近似估计输出结果 $\boldsymbol{W}$ 整体流程所需的总计算复杂度，并将复数矩阵运算，合理拆解到基本的实数运算，进而根据下述表 1.1，得到计算复杂度的量化结果。

表 1.1 实数基本运算的计算复杂度衡量

运算类型	计算复杂度量化
加(减)法（+-）	1
乘法(*)	3
倒数(/)	25
平方根(n^2)	25
自然指数(a^n)	25
自然对数(ln/log)	25
正弦(sin)	25
余弦(cos)	25
其它	100

$\boldsymbol{W}$ 建模估计精度[1]由以下公式定义：

$$\rho_{l,j,k}(\boldsymbol{W}) = \frac{\left\| \hat{\boldsymbol{W}}_{l,j,k}^{H} \boldsymbol{W}_{l,j,k} \right\|_2}{\left\| \hat{\boldsymbol{W}}_{l,j,k} \right\|_2 \left\| \boldsymbol{W}_{l,j,k} \right\|_2}, l = 1, \ldots, L \tag{1.5}$$

其中，$\|\bullet\|_2$ 为矢量的欧几里得范数，$\boldsymbol{W}_{l,j,k}$ 表示 $\boldsymbol{W}_{j,k}$ 的第 l 列。

同时额外定义 $\boldsymbol{W}$ 的最低建模精度 $\rho_{\min}(\boldsymbol{W})$ 为：

$$\rho_{\min}(\boldsymbol{W}) \triangleq \min_{\substack{l\in\{1,2,\ldots,L\}\\ j\in\{1,2,\ldots,J\}\\ k\in\{1,2,\ldots,K\}}} \rho_{l,j,k}(\boldsymbol{W}) \tag{1.6}$$

这里，$\min_{l,j,k}(\bullet)$ 表示取 l,j,k 三个维度上的最小值。中间结果 $\boldsymbol{V}$ 最低建模精度 $\rho_{\min}(\boldsymbol{V})$ 的定义与此相同。

1.2.2 问题 2 内容

实现相关矩阵组存储复杂度的降低。基于本题所给定的六组输入矩阵数据 $\boldsymbol{H}$，暂不考虑分析模型 $\hat{\boldsymbol{W}} = f(\boldsymbol{H})$ 对输出结果的影响，设计合理的压缩 $P_1(\bullet)$、$P_2(\bullet)$ 和解压缩 $G_1(\bullet)$、$G_2(\bullet)$ 模型，使得在满足给定的误差范围 $err_{\boldsymbol{H}} \le E_{th1} = -30\,\text{dB}$、$err_{\boldsymbol{W}} \le E_{th2} = -30\,\text{dB}$ 的条件下，相关矩阵组估计的存储复杂度以及压缩与解压缩的计算复杂度都尽可能最低。

优化目标：在估计结果 $\hat{\boldsymbol{W}}$ 和标准结果 $\boldsymbol{W}$ 近似相等的基础上，整体运算流程的存储复杂度，以及压缩与解压缩的计算复杂度尽可能达到最低

约束条件：1)误差限制 $err_{\boldsymbol{H}} \le E_{th1} = -30\,\text{dB}$、$err_{\boldsymbol{W}} \le E_{th2} = -30\,\text{dB}$

2)分析模型 $\hat{\boldsymbol{W}} = f(\boldsymbol{H})$ 对输出结果的影响不考虑

定义补充：

压缩与解压缩误差定义，公式如下：

$$err_H = 10*\log_{10}\frac{E\left\{\|\hat{H}_{j,k} - \boldsymbol{H}_{j,k}\|_F^2\right\}}{E\left\{\|\boldsymbol{H}_{j,k}\|_F^2\right\}}, err_W = 10*\log_{10}\frac{E\left\{\|\hat{\boldsymbol{W}}_{j,k} - \boldsymbol{W}_{j,k}\|_F^2\right\}}{E\left\{\|\boldsymbol{W}_{j,k}\|_F^2\right\}} \tag{1.7}$$

其中，$E\{\cdot\}$表示期望，$\|\cdot\|_F$表示矩阵的 Frobenius 范数；

存储复杂度定义：矩阵数据占用存储空间的大小，单位为比特。在本文中，文件中的矩阵数据为复数浮点数，因此本文中的矩阵数据所占用的存储空间为$64\times$矩阵维度。

1.2.3 问题 3 内容

实现相关矩阵组计算和存储复杂度的同时降低。在本题中，可以不受文中所给流程的限制，将由输入矩阵 $\boldsymbol{H}$ 获取近似输出矩阵 $\hat{\boldsymbol{W}}$，直接看成一个端到端的流程，从整体去把握，分析数据之间的关联情况，并在满足 $\rho_{\min}(\boldsymbol{W}) \ge \rho_{th} = 0.99$ 建模精度的要求下，设计出一个整体方案，使得相关矩阵组的运算过程中，计算和存储的复杂度都尽可能低。

此问题约束条件仅有估计精度 $\rho_{\min}(\boldsymbol{W}) \ge \rho_{th} = 0.99$ 的要求，一方面可以是在问题 1、2 解决的基础上，将问题 1、2 的模型做一个连接，使得最终的计算和存储复杂度得以降低；另一方面，可以完全将 $\boldsymbol{H}$ 到 $\boldsymbol{W}$ 的整个过程，看成一个黑盒，采用诸如神经网络等人工智能的方法，去学习出相关矩阵组的估算，然后对网络结构进行优化在，达成计算和存储复杂度的降低。

2. 模型假设和符号说明

2.1 模型假设

依据题目和解题要求，本文在建立数学模型前，需要作出如下模型假设：

假设 1 模型仅考虑同一行块内部的矩阵间的相关性，不考虑矩阵组中属于不同行块的矩阵间的相关性。

假设 2 模型忽略数据在传输过程中所造成的误差。

假设 3 模型输入数据矩阵 N 的值大于 M 的取值十倍以上。

假设 4 在未标明矩阵基础计算所使用方法时，模型使用经典基础运算方法。

假设 5 模型仅需考虑各个问题本身申明的建模需求，不需要考虑其他问题建模需求。

2.2 符号说明

由于本文各题之间的约束条件各不相同，因此本文在各题模型建立中有下列符号：

问题 1 的符号说明

符号	意义
$\boldsymbol{H}$	题目定义复数矩阵组，即输入矩阵
$\boldsymbol{V}$	奇异值分解的中间矩阵
$\boldsymbol{W}$	最后的输出结果矩阵
$\hat{\boldsymbol{V}}$	估计模型进行奇异值分解的中间矩阵
$\hat{\boldsymbol{W}}$	估计模型的输出结果矩阵
M,N,L,J,K	题目规定的矩阵维数
$\rho_{\min}(\cdot)$	矩阵的最小估计精度
ρ_{th}	规定的估计精度下限
C	基本采样步长，取 $C=1$
β	采样步长的加权系数
λ	采样步长
$\boldsymbol{\Sigma}$	奇异值对角矩阵
$\boldsymbol{R}$	上三角矩阵
$\boldsymbol{B}$	双对角矩阵
$\boldsymbol{T}$	三对角矩阵
$\boldsymbol{U},\boldsymbol{V}$	待求酉矩阵

问题 2 的符号说明

符号	意义
$P_1(\bullet),P_2(\bullet)$	设计的压缩模型函数
$G_1(\bullet),G_2(\bullet)$	设计的解压缩模型函数
$err_{\boldsymbol{H}}$	压缩误差
$err_{\boldsymbol{W}}$	解压缩误差
E_{th1}	题目给定的压缩误差上限
E_{th2}	题目给定的解压缩误差上限
$E\{\bullet\}$	数学期望

$\|\bullet\|_F$	矩阵的 Frobenius 范数
$\boldsymbol{\Phi}$	测量矩阵
$\boldsymbol{H}'$	原始矩阵经过压缩后得到的结果矩阵

问题 3 的符号说明

符号	意义
$\rho_{\min}(\cdot)$	矩阵的最小估计精度
ρ_{th}	规定的估计精度下限
$\boldsymbol{V}$	奇异值分解的中间矩阵
$\boldsymbol{W}$	最后的输出结果矩阵
$\hat{\boldsymbol{V}}$	估计模型进行奇异值分解的中间矩阵
$\hat{\boldsymbol{W}}$	估计模型的输出结果矩阵

3. 问题 1 的模型建立与求解

3.1 问题 1 分析

问题 1 主要是通过建模，尽可能降低相关矩阵组求解过程中的计算复杂度，此题中不需要考虑数据存储复杂度问题，即无需考虑数据解压缩的影响，只需要估计结果$\hat{\boldsymbol{W}}$ 和标准结果$\boldsymbol{W}$ 满足相应的估计建模精度。

因此，问题 1 所需满足的要求及约束条件如下：

- 如果是不考虑中间结果$\boldsymbol{V}$，直接设计运算模型$\hat{\boldsymbol{W}}=f(\boldsymbol{H})$，需要满足$\rho_{\min}(\boldsymbol{W})\geq\rho_{th}=0.99$建模精度要求；
- 如果考虑中间结果$K/2$，两步法设计分析模型$\hat{\boldsymbol{V}}=f_1(\boldsymbol{H}),\hat{\boldsymbol{W}}=f_2(\hat{\boldsymbol{V}})$，需要同时满足$\rho_{\min}(\boldsymbol{V})\geq\rho_{th}=0.99,\rho_{\min}(\boldsymbol{W})\geq\rho_{th}=0.99$的建模估计精度要求。
- 暂不考虑数据压缩与解压缩的影响

在满足上述条件的情况下，根据本文提示和相关的参考文献，针对问题 1，本文主要设计两步法的模型$\hat{\boldsymbol{V}}=f_1(\boldsymbol{H}),\hat{\boldsymbol{W}}=f_2(\hat{\boldsymbol{V}})$，从求取$\boldsymbol{V}$ 和$\boldsymbol{W}$ 的各个求解步骤出发，进行建模优化，降低各部分的计算复杂度，从而使得整体运算流程的计算复杂度得以明显降低。

主要考虑的改进切入点如下：

1)挖掘矩阵组$\boldsymbol{H}$ 各个矩阵之间，或矩阵$\mathbf{H}_{j,k}$ 各个矩阵元素之间的关联情况，例如各矩阵之间通过拟合求解，符合线性相关的情况，即可以通过分组，或是将乘法计算经过线性规划的处理，变为一系列的加法运算，减少需要处理的矩阵个数，可能由K 个矩阵，转变为只处理$K/2$ 个矩阵，最后利用合适的插值方法，获取最终的输出矩阵$\hat{\boldsymbol{W}}$ 。这样，就可以在满足估计精度要求的前提下，经过前期的预处理，减少需要计算的数据量，从而降低复杂度。

2)从求解右奇异向量的过程考虑，改进常规的 SVD 分解的操作流程，利用双对角化，三对角化结合 QR 分解或者 Givens 旋转等方法，寻求更低复杂度的 SVD 分解流程，引入到中间矩阵 V 的求解中来，实现拆分过程中第一步$\hat{\boldsymbol{V}}=f_1(\boldsymbol{H})$ 计算复杂度的降低。

3)从矩阵求逆的过程考虑，矩阵的求逆运算，往往会使计算复杂度大量增长。可以利用比如高斯消元法通过对矩阵每一列的数据进行处理，得到上三角矩阵，再对矩阵进行回减操作最终达到消元的目的，此时模型的运算复杂度为$\mathrm{O}\left((LJ)^3\right)$，又或者利用应用分治法求解的$LJ\times LJ$ 维矩阵的逆的复杂度近似为$\mathrm{O}\left((LJ)^{2.807}\right)$。在这些基础上，我们可以利用矩阵求逆引理，大大降低求逆矩阵的维度，并将求逆运算，由最小二乘、梯度下降等方法来取代，进一步的去降低求逆过程，也就是建模第二步$\hat{\boldsymbol{W}}=f_2(\hat{\boldsymbol{V}})$ 过程的运算复杂度。

3.2 问题 1 数学模型建立与求解分析

3.2.1 针对矩阵间的关联情况，进行矩阵数据的合理预处理

在进行后续奇异值分解，矩阵求逆计算之前，可以先根据矩阵组$\boldsymbol{H}$ 间的相关性，对矩阵组$\boldsymbol{H}$ 进行合理采样，减少需要进行运算的矩阵数量，再进行后续奇异值分解和求逆的操作，这可以大大降低计算复杂度。如在高光谱图像[2]处理领域，常常由于相关系数矩阵计算量过大的问题，会先对高光谱图像的某个维度如空间域进行均匀采样等方法，之后再利用

采样后的矩阵数据，来代替整幅图像的相关系数矩阵。通过该手段，可以在保持图像压缩性能的前提下，十分有效地大大降低计算量，保证预处理算法可实时性操作。

在高光谱图像处理中，能够使用均匀采样的方法以较少的数据代表原图像数据的前提是，对矩阵数据有一定先验知识，如该图像的地物变化不那么剧烈，或图像数据的地物内容没有过于丰富和零散，也就是矩阵间的相关度都比较高，可以简单粗暴地选取一定步长进行均匀采样，从而大幅地降低计算复杂度。因此，对于本题所给的 6 组 $\boldsymbol{H}$ 矩阵数据，我们对每组数据，计算输入矩阵组 $\mathbf{H}_{j,k}$ 两两矩阵间的相关系数。

由于计算过程中发现，该 6 组数据各自的矩阵间关联性基本一致，可以猜想该 6 组数据是由一个黑盒过程，产生的同类 6 组数据，因此，我们选取第一组为代表，得到矩阵组各个 $\mathbf{H}_{j,k}$ 之间的相关系数结果如下图 3.1 所示：

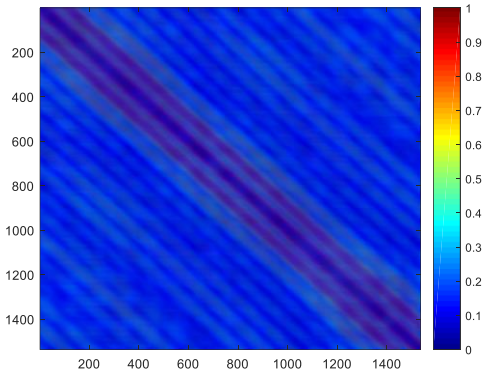


图 3.1 第一组矩阵数据间的相关系数

由上图可以看出，输入矩阵组 $\boldsymbol{H}$ 的矩阵间相关系数大小不是均匀分布，有强有弱，因此在本文中，不能直接采用均匀采样的方法，去除冗余数据。但是，采样和插值的思想，依旧是大大减少运算量的一个良好手段。

那么，根据本文数据，我们可以采用相关系数重排列分组[3]（如图 3.2 示意），各组加权变步长采样和插值的方法，使得近似算法的计算复杂度大大降低。我们假设一个基本的采样步长为 $C(C=1)$ 即原始不抽样算法，然后计算各矩阵间的相关系数，并根据相关系数从大到小的顺序，对 $\mathbf{H}_{j,k}$ 进行一个重排列，之后在根据相关系数所属区间，如 [0 0.2),[0.2 0.4),...,[0.8 1]，对不同区间内的矩阵组选取合适的采样权值 β，按照步长 $\lambda=C*\beta$ 进行采样（在满足估计建模精度 $\rho_{\min}(\boldsymbol{W})\geq\rho_{th}=0.99$ 的条件下，采样步长 λ 尽可能大），从而明显降低计算复杂度。

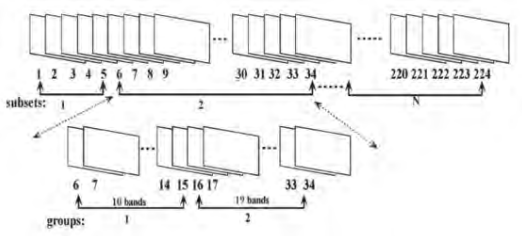


图 3.2 相关系数区间矩阵分组排序

关于采样步长加权系数 β 的选择，我们以第一组输入矩阵按照矩阵间相关度计算、排序、分组之后，在区间 [0.8 1] 上的矩阵数据为例，说明不同步长对建模估计精度的影响，如下表 3.1 所示。

表 3.1 近似算法不同采样步长下的建模估计精度

加权系数 β 步长 λ=$C*\beta, C=1$	1	2	4	8	16	32	64
估计建模精度 $\rho_{\min}(\boldsymbol{W})$	0.996	0.995	0.993	0.992	0.991	0.990	0.989

由上表可以看出，在该区间矩阵组中，可以选取采样步长 λ=32，进行合理的矩阵数据预处理，此时相关矩阵组估计近似算法性能损失不大，但同时计算量可以从 K 次计算，直接降到了 K/λ 次计算，最终的量化计算复杂度可能只是原来的千分之一，非常有效地降低计算负担。

同理，如果矩阵 $\boldsymbol{H}_{j,k}$ 内部矩阵元素之间，也有比较明显的关联特性，那我也可以进行合适的采样，降低后续奇异值分解，矩阵求逆运算的运算量，以便于计算复杂度的有效降低。理论上，假设对某一矩阵 $\boldsymbol{H}_{j,k}$ 的每 μ 行 μ 列进行，得到采样后的矩阵维度从 $M\times N$，变成 $M'\times N'(M'=M/L, N'=N/L)$，那么直接用常规法求解与采样法近似计算相关矩阵的计算量之比如下表 3.2，运算量的降低效果会非常显著。

表 3.2 相关矩阵组复数计算量

计算量	复数加法次数	复数乘法次数
直接法	$5MNJK/2$	$3MNJK/2$
采样法	$MNJK/2\mu^2$	$MNJK/2\mu^2$
直接法：采样法	$5\mu^2$	$3\mu^2$

由上表可知，如果各个矩阵内部元素之间也有很好的关联特性，对其进行合适的采样，也能很好减少运算量。但是经过第一组数据的取样分析，各矩阵内部元素之间的关联度变化较大（见附录），要在满足估计建模精度的要求下，进行合理采样和插值的操作不是特别简便，因此在本题中，利用矩阵内部元素间的关联性，降低求解的计算复杂度，不太好操作。

综上所述，我们主要借助输入矩阵组 $\boldsymbol{H}$ 各矩阵之间的相关度，采用加权变步长采样的方法，适当选取采样步长，能在几乎不影响建模估计精度的情况下，大幅降低计算量，以便于最终计算复杂度的降低。

3.2.2 奇异值分解过程的低复杂度建模优化

矩阵的奇异值分解是特征分解在任意矩阵上的推广，一个任意维的复数矩阵 $\boldsymbol{A}\in\mathbb{C}^{m\times n}$，可以利用两个酉矩阵 $\boldsymbol{U}\in\mathbb{C}^{m\times m}$ 和 $\boldsymbol{V}\in\mathbb{C}^{n\times n}$ 进行分解，形式如下：

$$\boldsymbol{A}=\boldsymbol{U}\begin{bmatrix}\boldsymbol{\Sigma} & \boldsymbol{0}\\ \boldsymbol{0} & \boldsymbol{0}\end{bmatrix}\boldsymbol{V}^H \tag{3.1}$$

式中 $\boldsymbol{\Sigma}$=diag$\left(\Sigma_1 \quad \cdots \quad \Sigma_r\right)$ 是一个与 $\boldsymbol{A}$ 有着相同维度的对角矩阵，r 为矩阵 $\boldsymbol{A}$ 的秩，其对角元素是矩阵 $\left(\boldsymbol{A}\cdot\boldsymbol{A}^H\right)$ 或矩阵 $\left(\boldsymbol{A}^H\cdot\boldsymbol{A}\right)$ 特征值的非负平方根，即矩阵 $\boldsymbol{A}$ 的奇异值，且按照由大到小的顺序排列，即 $\Sigma_1\geq\Sigma_2\geq\cdots\geq\Sigma_r$，酉矩阵 $\boldsymbol{U}$ 和 $\boldsymbol{V}$ 分别为矩阵 $\boldsymbol{A}$ 的左奇异向量和右奇异向量构成的矩阵。

根据处理实际问题的经验可知，矩阵 $\boldsymbol{A}$ 的奇异值一般差别较大，数值较大的部分奇异值对应的奇异向量对矩阵的影响较大，而数值较小的部分奇异值对应的奇异向量对矩阵 $\boldsymbol{A}$ 的影响较小，故在实际的矩阵处理中，通常只考虑前 L 个奇异值及其对应的奇异向量。

为降低传统 SVD 分解的计算复杂度，现考虑以下四种方法。

3.2.2.1 基于双对角化和 QR 分解的 SVD 分解

文献[4]提出了一种基于双对角化和 QR 分解的 SVD 分解方法。对于一个任意复数矩阵 $\boldsymbol{A}\in\mathbb{C}^{M\times N}$,不妨假设 $M\geq N$，则可以通过双对角化结合 QR 分解的方法来完成 SVD 分解过程并降低计算复杂度，基本步骤如下：

1)对矩阵 $\boldsymbol{A}$ 进行 QR 分解如下：

$$\boldsymbol{A}=\boldsymbol{Q}\boldsymbol{R} \tag{3.2}$$

式中 $\boldsymbol{Q}\in\mathbb{C}^{M\times M}$ 为酉矩阵；$\boldsymbol{R}$ 为维度与矩阵 $\boldsymbol{A}$ 相同的上三角矩阵，即矩阵 $\boldsymbol{R}$ 中非零元素仅在其对角线及对角线以上的部分。

2)对矩阵 $\boldsymbol{R}$ 进行双对角化如下：

$$\boldsymbol{R}=\boldsymbol{U}_1\boldsymbol{B}\boldsymbol{V}_1^H \tag{3.3}$$

式中 $\boldsymbol{U}_1\in\mathbb{C}^{M\times M}$ 和 $\boldsymbol{V}_1\in\mathbb{C}^{N\times N}$ 均为酉矩阵；$\boldsymbol{B}$ 为维度与矩阵 $\boldsymbol{A}$ 相同的双对角矩阵，即在两个对角线上存在非零元素的矩阵。

Golub-Kahan 双对角化方法是基于 Lanczos 双对角化方法的拓展，基本思想是利用迭代的方式构造若干个向量，Lanczos 双对角化方法的具体步骤如下。

首先定义矩阵 $\boldsymbol{W}$ 如下，矩阵 $\boldsymbol{W}$ 的第 j 列表示 Lanczos 算法过程中的第 $i=j+1$ 次迭代；

$$\begin{aligned}\boldsymbol{W}&=\begin{bmatrix}v_0 & 0 & w_2 & \cdots & w_{2k-1} & 0\\ 0 & w_1 & 0 & \cdots & 0 & w_{2k}\end{bmatrix}\\&=\begin{bmatrix}u_1 & 0 & u_2 & \cdots & u_k & 0\\ 0 & v_1 & 0 & \cdots & 0 & v_k\end{bmatrix}\end{aligned} \tag{3.4}$$

则三对角矩阵 $\boldsymbol{T}$ 可表示为:

$$\boldsymbol{T}=\begin{bmatrix}0 & \alpha_1 & 0 & \cdots & \cdots & \cdots & 0\\ \alpha_1 & 0 & \beta_2 & 0 & \cdots & \cdots & 0\\ 0 & \beta_2 & 0 & \alpha_2 & 0 & \cdots & 0\\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots\\ 0 & \cdots & \cdots & \cdots & \cdots & \alpha_k & 0\end{bmatrix} \tag{3.5}$$

设置待求的奇异矩阵分别为 $\boldsymbol{U}\in\mathbb{C}^{M\times L}$ 和 $\boldsymbol{V}\in\mathbb{C}^{L\times N}$，表示为：

$$\begin{cases}\boldsymbol{U}_k=\left[u_1,u_2,\cdots u_k\right]\\ \boldsymbol{V}_k=\left[v_1,v_2,\cdots v_k\right]\end{cases} \tag{3.6}$$

且两个矩阵满足 $\boldsymbol{U}_k^H\boldsymbol{U}_k=\boldsymbol{I}$，$\boldsymbol{V}_k^H\boldsymbol{V}_k=\boldsymbol{I}$ 的规范正交关系。

设定初始矢量 $u_1=b/\beta_1$，则有：

$$\begin{cases}\boldsymbol{A}^H u_1=\alpha_1 v_1\\ \beta_{i+1}u_{i+1}=\boldsymbol{A}v_i-\alpha_i u_i\\ \alpha_{i+1}v_{i+1}=\boldsymbol{A}^H u_{i+1}-\beta_{i+1}v_i\end{cases} \tag{3.7}$$

将上述方程组改写为矩阵形式，则有：

$$\begin{cases}\boldsymbol{U}_{k+1}\left(\beta_1 e_1\right)=b\\ \boldsymbol{A}\boldsymbol{V}_k=\boldsymbol{U}_{k+1}\boldsymbol{B}_k\\ \boldsymbol{A}^H\boldsymbol{U}_{k+1}=\boldsymbol{V}_k\boldsymbol{B}_k^H+\alpha_{k+1}v_{k+1}e_{k+1}^H\end{cases} \tag{3.8}$$

其中，双对角矩阵 $\boldsymbol{B}$ 表示为：

$$\boldsymbol{B}_k=\begin{bmatrix}\alpha_1 & 0 & \cdots & \cdots & \cdots & 0\\ \beta_2 & \alpha_2 & 0 & \cdots & \cdots & 0\\ 0 & \beta_3 & \alpha_3 & 0 & \cdots & 0\\ \cdots & \cdots & \cdots & \cdots & \cdots & \alpha_k\\ 0 & \cdots & \cdots & \cdots & 0 & \beta_{k+1}\end{bmatrix} \tag{3.9}$$

Golub-Kahan 双对角化的具体流程算法如下：

> Golub-Kahan 双对角化算法：
> 1. 初始化 i=1，v_0=0，$\alpha_1 v_1 = b$，$\boldsymbol{A}^H u_1 = \alpha_1 v_1$；
> 2. 迭代开始，当 i 为偶数时，$\alpha_i = 0$，且满足：
> $$\beta_i v_i = \boldsymbol{A} v_{i-1} - \beta_{i-1} v_{i-2}$$
> 3. 当 i 为偶数时，$\alpha_i = 0$，且满足：
> $$\beta_i v_i = \boldsymbol{A}^H v_{i-1} - \beta_{i-1} v_{i-2}，\ \alpha_{i+1} v_{i+1} = \boldsymbol{A}^H u_{i+1} - \beta_{i+1} v_i$$
> 4. i=i+1，直至结束。

3)通过 QR 迭代的方式将双对角矩阵 $\boldsymbol{B}$ 转换为对角矩阵 $\boldsymbol{S}$，形式如下：

$$\boldsymbol{B} = \boldsymbol{U}_2 \boldsymbol{S} \boldsymbol{V}_2^H \tag{3.10}$$

式中 $\boldsymbol{S}$ 为维度与矩阵 $\boldsymbol{A}$ 相同的对角矩阵，$\boldsymbol{U}_2 \in \mathbb{C}^{M\times M}$ 和 $\boldsymbol{V}_2 \in \mathbb{C}^{N\times N}$ 均为酉矩阵；基于迭代的 QR 分解过程，$\boldsymbol{U}_2$ 和 $\boldsymbol{V}_2$ 其实是多个酉矩阵的乘积，即：

$$\begin{cases}\boldsymbol{U}_2 = \boldsymbol{U}_2^{(k)} \boldsymbol{U}_2^{(k-1)} \cdots \boldsymbol{U}_2^{(1)}\\ \boldsymbol{V}_2 = \boldsymbol{V}_2^{(k)} \boldsymbol{V}_2^{(k-1)} \cdots \boldsymbol{V}_2^{(1)}\end{cases} \tag{3.11}$$

综合上述步骤，可以得到对矩阵 $\boldsymbol{A}$ 的 SVD 分解过程，形式如下：

$$\boldsymbol{A} = \underbrace{\boldsymbol{Q}\boldsymbol{U}_1\boldsymbol{U}_2}_{\boldsymbol{U}} \boldsymbol{S} \underbrace{\left(\boldsymbol{V}_1\boldsymbol{V}_2\right)^H}_{\boldsymbol{V}} = \boldsymbol{U}\boldsymbol{S}\boldsymbol{V}^H \tag{3.12}$$

若矩阵 $\boldsymbol{A}$ 的维数满足 $M \le N$ 的关系，则可将上式改写为：

$$\boldsymbol{A} = \boldsymbol{V}\boldsymbol{S}\boldsymbol{U}^H \tag{3.13}$$

或将步骤 1)中的 QR 分解更改为 LQ 分解，再针对下三角矩阵进行双对角化和 QR 迭代分解的操作。

总结该算法的流程大致如下：

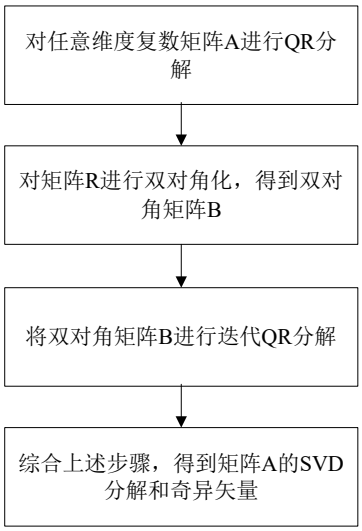


图 3.3 基于双对角化和 QR 分解的 SVD 分解算法流程图

根据实际数据，第(3)步中的 QR 迭代分解至少需要进行 3 次才能满足题目要求的估计误差精度，故在之后的复杂度分析时考虑 3 次 QR 迭代分解的总复杂度。

3.2.2.2 基于随机 SVD 的 SVD 分解

文献[5]提出了一种基于随机方法的 SVD 分解，对于一个任意复数矩阵 $\boldsymbol{A}\in\mathbb{C}^{M\times N}$,不妨假设 $M\geq N$，则找到两个矩阵使下式满足：

$$\boldsymbol{A}\approx\boldsymbol{BC} \tag{3.14}$$

式中矩阵 $\boldsymbol{B}$ 的维度为 $M\times L$，矩阵 $\boldsymbol{C}$ 的维度为 $L\times N$，且满足 $L\ll N$。根据这个思路，可以利用一种随机算法对矩阵 $\boldsymbol{A}$ 进行近似，步骤大致如下：

1)利用随机方法计算矩阵 $\boldsymbol{A}$ 列向量张成的线性空间的近似基，构造由 L 个标准正交列矢量组成的矩阵 $\boldsymbol{Q}$，则矩阵 $\boldsymbol{A}$ 和矩阵 $\boldsymbol{Q}$ 满足如下关系：

$$\boldsymbol{A}\approx\boldsymbol{QQ}^{H}\boldsymbol{A} \tag{3.15}$$

矩阵 $\boldsymbol{Q}$ 的确定依据如下所示的条件，其中 ε 是设定的残差。

$$\left\|\boldsymbol{I}-\boldsymbol{QQ}^{H}\boldsymbol{A}\right\|\leq\varepsilon \tag{3.16}$$

考虑到 L 的取值可能涉及到 $\boldsymbol{A}\approx\boldsymbol{QQ}^{H}\boldsymbol{A}$ 近似过程中所能接受的最大误差，可采用迭代增量的方法来构建矩阵 $\boldsymbol{Q}$。令 $\boldsymbol{Q}^{(i)}$ 表示包含 i 列的迭代标准正交矩阵，$\boldsymbol{Q}^{(0)}$ 为空矩阵，则通过如下迭代算法计算矩阵 $\boldsymbol{Q}$：

迭代法计算矩阵 $\boldsymbol{Q}$：

1. 随机抽取一个维度为 N 的高斯随机矢量 $\boldsymbol{\omega}^{(i)}$，令 $\boldsymbol{y}^{(i)}=\boldsymbol{A\omega}^{(i)}$；
2. 构造 $\tilde{\boldsymbol{q}}^{(i)}=\left(\boldsymbol{I}-\boldsymbol{Q}^{(i-1)}\left(\boldsymbol{Q}^{(i-1)}\right)^{H}\right)\boldsymbol{y}^{(i)}$；
3. 令 $\boldsymbol{q}^{(i)}=\tilde{\boldsymbol{q}}^{(i)}\Big/\left\|\tilde{\boldsymbol{q}}^{(i)}\right\|_{2}$；
4. 构造 $\boldsymbol{Q}^{(i)}=\left[\boldsymbol{Q}^{(i-1)}\quad\boldsymbol{q}^{(i)}\right]$；
5. 持续上述步骤直至 $\left\|\boldsymbol{I}-\boldsymbol{QQ}^{H}\boldsymbol{A}\right\|\leq\varepsilon$ 条件满足。

2)定义 $\boldsymbol{B}=\boldsymbol{Q}^{H}\boldsymbol{A}$，矩阵 $\boldsymbol{B}$ 是一个 $L\times N$ 维矩阵，其行数小于矩阵 $\boldsymbol{A}$，则对矩阵 $\boldsymbol{B}$ 进行 SVD 分解可以降低计算复杂度，对矩阵 $\boldsymbol{B}$ 进行 SVD 分解为：

$$\boldsymbol{B}=\tilde{\boldsymbol{U}}\boldsymbol{SV}^{H} \tag{3.17}$$

结合步骤(1)和(2)，则矩阵 $\boldsymbol{A}$ 的近似 SVD 分解为：

$$\boldsymbol{A}=\boldsymbol{Q}\tilde{\boldsymbol{U}}\,\boldsymbol{SV}=\underset{\boldsymbol{U}}{\boldsymbol{USV}^{H}} \tag{3.18}$$

上式中的矩阵 $\boldsymbol{U}$ 不必要求是酉矩阵。

根据实际数据，每轮迭代平均进行 4 次即可满足收敛条件，故在之后的复杂度分析时考虑 4 次迭代的总复杂度。

3.2.2.3 一种计算部分奇异值分解的方法

论文[6]提出一种计算矩阵部分特征值分解的方法，当矩阵满足低秩条件且只对主要的奇异值感兴趣时，这种算法步骤简单易行，且不会消耗过大的内存资源。

一个任意维的复数矩阵 $\boldsymbol{A}\in\mathbb{C}^{m\times n}$，可以利用两个酉矩阵 $\boldsymbol{U}\in\mathbb{C}^{m\times m}$ 和 $\boldsymbol{V}\in\mathbb{C}^{n\times n}$ 进行分解，形式如下：

$$\boldsymbol{A}=\boldsymbol{U}\begin{bmatrix}\boldsymbol{\Sigma} & \boldsymbol{0}\\ \boldsymbol{0} & \boldsymbol{0}\end{bmatrix}\boldsymbol{V}^{H} \tag{3.19}$$

酉矩阵$\boldsymbol{U}$和$\boldsymbol{V}$分别为矩阵$\boldsymbol{A}$的左奇异向量和右奇异向量构成的矩阵，这两个矩阵分别由左奇异矢量u_i和右奇异矢量v_i构成，即：

$$\begin{cases}\boldsymbol{U}=\begin{bmatrix}u_1 & \cdots & u_i & \cdots & u_M\end{bmatrix}\\ \boldsymbol{V}=\begin{bmatrix}v_1 & \cdots & v_i & \cdots & v_N\end{bmatrix}\end{cases} \tag{3.20}$$

将矩阵$\boldsymbol{A}$的 SVD 分解表达式改写为外部积展开形式如下：

$$\boldsymbol{A}=\sum_{i=1}^{L}\Sigma_i u_i v_i^H \qquad L\le rank\left(\boldsymbol{A}\right) \tag{3.21}$$

矩阵$u_i v_i^H$被称为奇异平面或特征平面。奇异值分解的一重要特性就是截断面上的级数就是通过矩阵$\boldsymbol{A}$的最小二乘近似得到的，其残差与属于奇异平面的奇异值平方根之和成正比。设置迭代是为了解决如下等式所示问题：

$$\boldsymbol{\Sigma}u_i=\boldsymbol{A}v_i \qquad \boldsymbol{\Sigma}v_i=\boldsymbol{A}^H u_i \tag{3.22}$$

随机定义初始右奇异矢量$\boldsymbol{v}^{(0)}$，则可以通过如下迭代方法计算出奇异矢量：

$$\begin{cases}\boldsymbol{u}^{(k+1)}=\left[\boldsymbol{A}\right]\boldsymbol{v}^{(k)}\Big/\left\|\left[\boldsymbol{A}\right]\boldsymbol{v}^{(k)}\right\|\\ \boldsymbol{v}^{(k+1)}=\left[\boldsymbol{A}\right]^H\boldsymbol{u}^{(k+1)}\Big/\left\|\left[\boldsymbol{A}\right]^H\boldsymbol{u}^{(k+1)}\right\|\end{cases} \tag{3.23}$$

当满足$\boldsymbol{u}^{(k+1)}-\boldsymbol{u}^{(k)}<\varepsilon$条件时，迭代终止，式中$\varepsilon$为设置精度。随着迭代的结束，可以计算出第一个奇异值为：

$$\hat{\Sigma}_1=\left\|\left[\boldsymbol{A}\right]^H\boldsymbol{u}^{(k+1)}\right\| \tag{3.24}$$

与之对应的奇异矢量$\hat{u}_1$和$\hat{v}_1$可由最后一次迭代的结果$\boldsymbol{u}^{(k+1)}$和$\boldsymbol{v}^{(k+1)}$所得。为了获得下一组奇异矢量，已估计出的奇异值要从矩阵$\boldsymbol{A}$的奇异平面移除，则更新的矩阵为：

$$\left[\boldsymbol{A}'\right]=\left[\boldsymbol{A}\right]-\hat{\Sigma}_1\hat{u}_1\hat{v}_1^H \tag{3.25}$$

利用得到的矩阵$\left[\boldsymbol{A}'\right]$进行新一轮迭代，从而计算出第二组奇异值$\hat{\Sigma}_2$和相对应的奇异矢量$\hat{u}_2$和$\hat{v}_2$。忽略近似运算时的微小误差，该算法保障了$\hat{u}_1$，$\hat{u}_2$,…的正交性在迭代运算在收敛前终止时也依旧成立，这是因为下式成立：

$$\left[\boldsymbol{A}'\right]\hat{u}_1=0 \tag{3.26}$$

上式的成立表明本次估计得到的奇异矢量与上一个估计得到的奇异矢量线性无关，故经过多次迭代，无论结果是否收敛，都一定能够得到矩阵$\left[\boldsymbol{A}\right]$的近似值。总结该算法的流程大致如下：

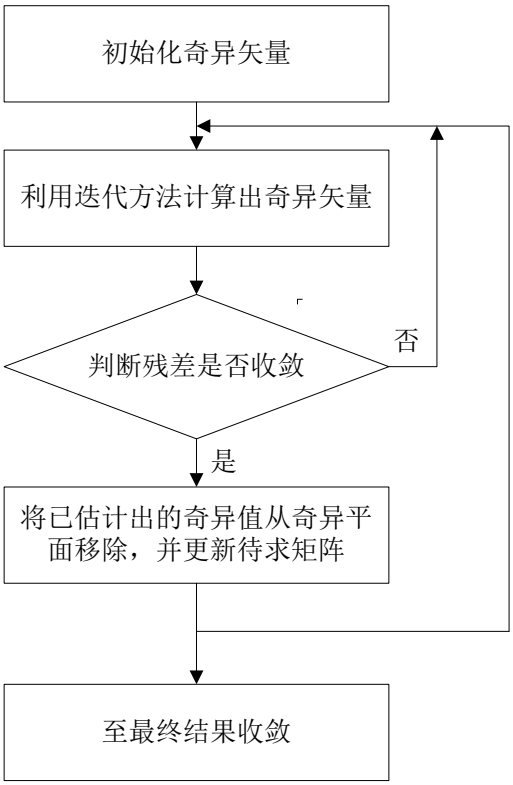


图 3.4 计算部分奇异值分解方法的流程图

根据实际数据，每轮迭代平均进行 4 次即可满足收敛条件；由于题目设定 $L=2$，故只需要进行两轮迭代即可求得右奇异矩阵；在之后的复杂度分析时考虑两轮迭代且每轮迭代次数为 4 的总复杂度。

3.2.2.4 一种利用近似奇异值的 SVD 分解

论文[7]提出了一种改进的 SVD 分解方法，这种方法依赖一个与矩阵 $\boldsymbol{A}$ 的奇异值矩阵近似的对角矩阵来实现。当矩阵 $\boldsymbol{A}$ 有相同或相近的特征值时，可以采用直接法将 SVD 分解问题分成两部分，一部分包含良好的奇异值，另一部分的奇异值则可以忽略不考虑。

一个任意维的复数矩阵 $\boldsymbol{A}\in\mathbb{C}^{m\times n}$，可以利用两个酉矩阵 $\boldsymbol{U}\in\mathbb{C}^{m\times m}$ 和 $\boldsymbol{V}\in\mathbb{C}^{n\times n}$ 进行分解，形式如下：

$$\boldsymbol{A}=\boldsymbol{U}\begin{bmatrix}\boldsymbol{\Sigma} & \boldsymbol{0}\\ \boldsymbol{0} & \boldsymbol{0}\end{bmatrix}\boldsymbol{V}^{H} \tag{3.27}$$

式中酉矩阵 $\boldsymbol{U}$ 和 $\boldsymbol{V}$ 分别由左奇异矢量 u_i 和右奇异矢量 v_i 构成，即：

$$\begin{cases}\boldsymbol{U}=\begin{bmatrix}u_1 & \cdots & u_i & \cdots & u_M\end{bmatrix}\\ \boldsymbol{V}=\begin{bmatrix}v_1 & \cdots & v_i & \cdots & v_N\end{bmatrix}\end{cases} \tag{3.28}$$

由奇异值分解的定义可知，矩阵 $\boldsymbol{A}$ 和奇异值、奇异矢量满足下述关系：

$$\boldsymbol{A}v_i=\Sigma_i u_i \qquad \boldsymbol{A}^{H}u_i=\Sigma_i v_i \tag{3.29}$$

下式描述了奇异值对加性扰动的敏感性[5]：

$$\left|\Sigma_i\left(\tilde{\boldsymbol{A}}\right)-\Sigma_i\left(\boldsymbol{A}\right)\right|\le\left\|\delta\boldsymbol{A}\right\|_2 \qquad \tilde{\boldsymbol{A}}=\boldsymbol{A}+\delta\boldsymbol{A} \tag{3.30}$$

上式表明了矩阵 $\boldsymbol{A}$ 的奇异值对小的扰动并不敏感，根据论文[8][9]的结论可知，对于矩阵 $\boldsymbol{A}$ 的一个微小扰动 $\delta\boldsymbol{A}$ 相当于在大小为 $\left\|\delta\boldsymbol{A}\right\|_2/\varepsilon$ 的奇异子空间内，其中 ε 是相关奇异值之间的差值。因此，当 ε 足够小时，矩阵 $\boldsymbol{A}$ 上的微小扰动会导致巨大的偏差。

当 $\left\|\tilde{\boldsymbol{A}}-\boldsymbol{A}\right\|$ 的值很小时，认为矩阵 $\boldsymbol{A}$ 的 SVD 分解结果和矩阵 $\tilde{\boldsymbol{A}}$ 的 SVD 分解结果近似

相等，即两个矩阵有近似相同的奇异值对角阵，即：

$$\boldsymbol{U}^H\boldsymbol{A}\boldsymbol{V}=\boldsymbol{\Sigma}\quad \boldsymbol{U}^H\tilde{\boldsymbol{A}}\boldsymbol{V}=\boldsymbol{\Delta} \tag{3.31}$$

上式中矩阵 $\boldsymbol{\Delta}$ 就是对角阵 $\boldsymbol{\Sigma}$ 的近似值。根据泰勒级数展开可知，酉矩阵 $\boldsymbol{U}$ 和 $\boldsymbol{V}$ 可表示为：

$$\begin{cases}\boldsymbol{U}^T=\boldsymbol{I}+\hat{\boldsymbol{X}}+\dfrac{1}{2}\hat{\boldsymbol{X}}^2+\cdots\\ \boldsymbol{V}^T=\boldsymbol{I}+\hat{\boldsymbol{Y}}+\dfrac{1}{2}\hat{\boldsymbol{Y}}^2+\cdots\end{cases} \tag{3.32}$$

上式中的 $\hat{\boldsymbol{X}}$ 和 $\hat{\boldsymbol{Y}}$ 均为斜对称矩阵，那么近似矩阵 $\boldsymbol{\Delta}$ 可以表示为：

$$\begin{aligned}\boldsymbol{\Delta}=&\boldsymbol{A}+\hat{\boldsymbol{X}}\boldsymbol{A}+\boldsymbol{A}\hat{\boldsymbol{Y}}+\frac{1}{2}\left(\hat{\boldsymbol{X}}^2\boldsymbol{A}+\boldsymbol{A}\hat{\boldsymbol{Y}}^2\right)+\hat{\boldsymbol{X}}\boldsymbol{A}\hat{\boldsymbol{Y}}+O\left(\left\|\hat{\boldsymbol{X}}\right\|^i\|\boldsymbol{A}\|\left\|\hat{\boldsymbol{Y}}\right\|^j\right)\\ =&\boldsymbol{A}_0+\hat{\boldsymbol{X}}\boldsymbol{A}_0+\boldsymbol{A}_0\hat{\boldsymbol{Y}}+\frac{1}{2}\left(\hat{\boldsymbol{X}}^2\boldsymbol{A}_0+\boldsymbol{A}_0\hat{\boldsymbol{Y}}^2\right)+\hat{\boldsymbol{X}}\boldsymbol{A}_0\hat{\boldsymbol{Y}}\\ &+\boldsymbol{A}_1+\hat{\boldsymbol{X}}\boldsymbol{A}_1+\boldsymbol{A}_1\hat{\boldsymbol{Y}}+\frac{1}{2}\left(\hat{\boldsymbol{X}}^2\boldsymbol{A}_1+\boldsymbol{A}_1\hat{\boldsymbol{Y}}^2\right)+\hat{\boldsymbol{X}}\boldsymbol{A}_1\hat{\boldsymbol{Y}}+O\left(\left\|\hat{\boldsymbol{X}}\right\|^i\|\boldsymbol{A}\|\left\|\hat{\boldsymbol{Y}}\right\|^j\right)\end{aligned} \tag{3.33}$$

上式中 $\boldsymbol{A}_0$ 表示矩阵 $\boldsymbol{A}$ 的对角线部分， $\boldsymbol{A}_1$ 表示矩阵 $\boldsymbol{A}$ 的非对角线部分； $\hat{\boldsymbol{X}}$ 和 $\hat{\boldsymbol{Y}}$ 是两个斜对称矩阵的和，即：

$$\begin{cases}\hat{\boldsymbol{X}}=\hat{\boldsymbol{X}}_1+\hat{\boldsymbol{X}}_2\\ \hat{\boldsymbol{Y}}=\hat{\boldsymbol{Y}}_1+\hat{\boldsymbol{Y}}_2\end{cases} \tag{3.34}$$

定义 $\hat{\boldsymbol{X}}_1$ 和 $\hat{\boldsymbol{Y}}_1$ 满足以下关系：

$$\hat{\boldsymbol{X}}_1\boldsymbol{A}_0+\boldsymbol{A}_0\hat{\boldsymbol{Y}}_1=-\boldsymbol{A}_1 \tag{3.35}$$

将矩阵 $\boldsymbol{A}$ 改写成如下形式：

$$\boldsymbol{A}=\begin{matrix}N\\ M-N\end{matrix}\begin{bmatrix}\boldsymbol{B}_0+\boldsymbol{B}_1\\ \underset{N}{\boldsymbol{C}_1}\end{bmatrix} \tag{3.36}$$

上式中矩阵 $\boldsymbol{B}_0$ 是矩阵 $\boldsymbol{A}_{1:N,1:N}$ 的对角线部分，矩阵 $\boldsymbol{B}_1$ 是矩阵 $\boldsymbol{A}_{1:N,1:N}$ 的非对角线部分。则上式可改写为：

$$\begin{vmatrix}\boldsymbol{X}_1\boldsymbol{B}_0\\ \tilde{\boldsymbol{X}}_1\boldsymbol{B}_0\end{vmatrix}+\begin{vmatrix}\boldsymbol{B}_0\boldsymbol{Y}_1\\ 0\end{vmatrix}=-\begin{vmatrix}\boldsymbol{B}_1\\ \boldsymbol{C}_1\end{vmatrix} \tag{3.37}$$

将上述矩阵运算改写为方程组，即：

$$\begin{cases}\boldsymbol{X}_1\boldsymbol{B}_0+\boldsymbol{B}_0\boldsymbol{Y}_1=-\boldsymbol{B}_1\\ \tilde{\boldsymbol{X}}_1\boldsymbol{B}_0=-\boldsymbol{C}_1\end{cases} \tag{3.38}$$

由于矩阵 $\boldsymbol{B}_0$ 是一个对角阵，则上述方程的解为：

$$\left(\boldsymbol{X}_1\right)_{ij}=\begin{cases}\dfrac{a_{ij}a_{jj}+a_{ji}a_{ii}}{a_{ii}^2-a_{jj}^2}, & i\neq j\\ 0, & i=j\end{cases}\qquad \left(\boldsymbol{Y}_1\right)_{ij}=\begin{cases}\dfrac{a_{ij}a_{ii}+a_{ji}a_{jj}}{a_{jj}^2-a_{ii}^2}, & i\neq j\\ 0, & i=j\end{cases} \tag{3.39}$$

定义矩阵 $\boldsymbol{D}$ 为：

$$\boldsymbol{D}=\frac{1}{2}\left(\hat{\boldsymbol{X}}_1\boldsymbol{A}_1+\boldsymbol{A}_1\hat{\boldsymbol{Y}}_1\right) \tag{3.40}$$

同理可求得 $\boldsymbol{X}_2$ 和 $\boldsymbol{Y}_2$ 为：

$$\left(\boldsymbol{X}_2\right)_{ij}=\begin{cases}\dfrac{d_{ij}a_{jj}+d_{ji}a_{ii}}{a_{ii}^2-a_{jj}^2}, & i\neq j\\ 0, & i=j\end{cases}\qquad \left(\boldsymbol{Y}_2\right)_{ij}=\begin{cases}\dfrac{d_{ij}a_{ii}+d_{ji}a_{jj}}{a_{jj}^2-a_{ii}^2}, & i\neq j\\ 0, & i=j\end{cases} \tag{3.41}$$

从而求得酉矩阵 $\boldsymbol{U}$ 和 $\boldsymbol{V}$ 为：

$$\begin{cases}\boldsymbol{U}^T=\boldsymbol{I}+\hat{\boldsymbol{X}}_1+\hat{\boldsymbol{X}}_2+\dfrac{1}{2}\hat{\boldsymbol{X}}_1^2\\ \boldsymbol{V}^T=\boldsymbol{I}+\hat{\boldsymbol{Y}}_1+\hat{\boldsymbol{Y}}_2+\dfrac{1}{2}\hat{\boldsymbol{Y}}_1^2\end{cases} \tag{3.42}$$

这种算法的主要计算复杂度体现在四次矩阵乘法，矩阵 $\hat{\boldsymbol{X}}_1\in\mathbb{C}^{M\times M}$ 做乘法的复杂度为 $O\left(M^3\right)$，由于矩阵 $\hat{\boldsymbol{X}}_1^2$ 是具有特殊的结构的对称阵，故算法复杂度可降低至 $O\left(M^2N+N^2\left(M-N\right)\right)$，则该算法的总复杂度为 $O\left(3M^2N+3MN^2\right)$。

对比上述的四种算法的运算复杂度，根据实际数据，平均 4 次迭代即可满足收敛条件，对 $M\times N$ 维的矩阵 $\boldsymbol{H}_{j,k}$ 进行改进的 SVD 分解分别需要的运算复杂度如下表 3.3 和图 3.4 所示：

表 3.3 四种方法的运算复杂度对比

输入矩阵组 $\boldsymbol{H}_{j,k}\in\mathbb{C}^{M\times N}$	运算类型 计算复杂度	加法 2	乘法 14	总复杂度 求和
方法 1 基于双对角化和 QR 分解 SVD	次数	19968	21504	41472
	计算量	39936	301056	340992
方法 2 基于随机算法的 SVD 分解	次数	15050	19866	34916
	计算量	30100	278124	308224
方法 3 一种计算部分奇异值分解的方法	次数	14686	17624	32310
	计算量	29372	246736	276108
方法 4 一种利用近似奇异值的 SVD 分解	次数	18496	16088	34584
	计算量	36992	225232	262224

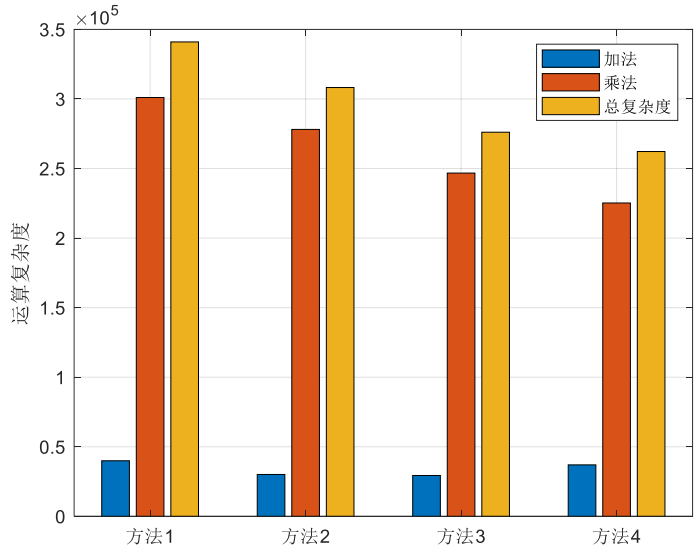


图 3.5 四种方法的运算复杂度对比

根据上述表 3.3 和图 3.5，可以更直观地看出，在 SVD 分解这一步利用近似奇异值分解法求矩阵 $\boldsymbol{H}_{j,k}$ 的右奇异向量，可以在满足建模估计精度要求的情况下，尽可能多地降低该步骤的计算复杂度。

3.2.3 针对计算流程进行合理的构造和转化

通过对计算流程进行简化，避免高复杂度的矩阵运算，可以有效降低模型的运算复杂度。第二步建模过程可以表示为

$$\boldsymbol{W} = f_2\left(\boldsymbol{V}\right) \tag{3.43}$$

其中，$f_2\left(\bullet\right)$ 表示从第一步建模结果 $\boldsymbol{V}$ 到最终结果 $\boldsymbol{W}$ 的建模过程。

由前文所述，$\boldsymbol{V}$ 是 $N\times L$ 维矩阵，为了得到模型最终的输出结果 $\boldsymbol{W}$，我们将矩阵 $\boldsymbol{V}$ 进行横向拼接，得到 $N\times LJ$ 维的矩阵 $\boldsymbol{V}_k=\left[\boldsymbol{V}_{1,k}\quad\cdots\quad\boldsymbol{V}_{j,k}\quad\cdots\quad\boldsymbol{V}_{J,k}\right]$。根据下式

$$\boldsymbol{W}_k=\boldsymbol{V}_k\left(\boldsymbol{V}_k^H\boldsymbol{V}_k+\sigma^2\boldsymbol{I}\right)^{-1} \tag{3.44}$$

可以得到最终的模型输出矩阵 $\boldsymbol{W}$ 的子矩阵 $\boldsymbol{W}_k$。其中，σ^2 为常数，$\boldsymbol{I}$ 为 $LJ\times LJ$ 维单位矩阵。

第二步建模过程中计算复杂度较高，这是因为模型中应用了矩阵的逆运算。高斯消元法通过对矩阵每一列的数据进行处理，得到上三角矩阵，再对矩阵进行回减操作最终达到消元的目的。求解 $LJ\times LJ$ 维矩阵的逆需要的运算复杂度近似为 $\mathrm{O}\left(\left(LJ\right)^3\right)$。由于矩阵 $\boldsymbol{W}$ 的拆分过程不涉及运算复杂度问题，所以当使用高斯消元法时，模型的运算复杂度为 $\mathrm{O}\left(\left(LJ\right)^3\right)$。

可以使用 Strassen 提出的方法来代替高斯消元法[10]进行矩阵求逆运算，算法通过分治法来直接降低运算数目的要求。对于阶数为 2^x 的复数矩阵乘法：

$$
\begin{aligned}
\boldsymbol{C} &= \begin{bmatrix} \boldsymbol{C}_{11} & \boldsymbol{C}_{12} \\ \boldsymbol{C}_{21} & \boldsymbol{C}_{22} \end{bmatrix} \\
&= \boldsymbol{AB} = \begin{bmatrix} \boldsymbol{A}_{11} & \boldsymbol{A}_{12} \\ \boldsymbol{A}_{21} & \boldsymbol{A}_{22} \end{bmatrix} \begin{bmatrix} \boldsymbol{B}_{11} & \boldsymbol{B}_{12} \\ \boldsymbol{B}_{21} & \boldsymbol{B}_{22} \end{bmatrix} \\
&= \begin{bmatrix} \boldsymbol{M}_1 + \boldsymbol{M}_4 - \boldsymbol{M}_5 + \boldsymbol{M}_7 & \boldsymbol{M}_3 + \boldsymbol{M}_5 \\ \boldsymbol{M}_2 + \boldsymbol{M}_4 & \boldsymbol{M}_1 - \boldsymbol{M}_2 + \boldsymbol{M}_3 + \boldsymbol{M}_6 \end{bmatrix}
\end{aligned} \tag{3.45}
$$

$$
\begin{cases}
\boldsymbol{M}_1 = \left(\boldsymbol{A}_{11} + \boldsymbol{A}_{22}\right)\left(\boldsymbol{B}_{11} + \boldsymbol{B}_{22}\right) \\
\boldsymbol{M}_2 = \left(\boldsymbol{A}_{21} + \boldsymbol{A}_{22}\right)\boldsymbol{B}_{11} \\
\boldsymbol{M}_3 = \boldsymbol{A}_{11}\left(\boldsymbol{B}_{12} - \boldsymbol{B}_{22}\right) \\
\boldsymbol{M}_4 = \boldsymbol{A}_{22}\left(\boldsymbol{B}_{21} - \boldsymbol{B}_{11}\right) \\
\boldsymbol{M}_5 = \left(\boldsymbol{A}_{11} + \boldsymbol{A}_{12}\right)\boldsymbol{B}_{22} \\
\boldsymbol{M}_6 = \left(\boldsymbol{A}_{21} - \boldsymbol{A}_{11}\right)\left(\boldsymbol{B}_{11} + \boldsymbol{B}_{12}\right) \\
\boldsymbol{M}_7 = \left(\mathbf{A}_{12} - \mathbf{A}_{22}\right)\left(\boldsymbol{B}_{21} + \boldsymbol{B}_{22}\right)
\end{cases} \tag{3.46}
$$

可以通过复数加法来代替。此时， $LJ \times LJ$ 维复数矩阵乘法的运算复杂度为 $\left(LJ\right)^{\log_2^7}$，因此可以有效降低复数矩阵的逆运算。应用分治法求解的 $LJ \times LJ$ 维矩阵的逆的复杂度近似为 $\mathrm{O}\left(\left(LJ\right)^{2.807}\right)$。

如果对此进行进一步简化，将矩阵的逆运算与 Coppersmith Winograd[11]所提算法相结合，通过构造网格避免了直接进行的乘法运算。算法中将矩阵进行分块操作

$$
\begin{aligned}
\boldsymbol{X}^{[0]}\boldsymbol{Y}^{[1]}\boldsymbol{Z}^{[1]} &= \sum_{i=1}^{q} \boldsymbol{X}_0\boldsymbol{Y}_i\boldsymbol{Z}_i \\
\boldsymbol{X}^{[1]}\boldsymbol{Y}^{[0]}\boldsymbol{Z}^{[1]} &= \sum_{i=1}^{q} \boldsymbol{X}_i\boldsymbol{Y}_0\boldsymbol{Z}_i \\
\boldsymbol{X}^{[1]}\boldsymbol{Y}^{[1]}\boldsymbol{Z}^{[0]} &= \sum_{i=1}^{q} \boldsymbol{X}_i\boldsymbol{Y}_i\boldsymbol{Z}_0
\end{aligned} \tag{3.47}
$$

定义

$$
\begin{aligned}
\boldsymbol{p}(\boldsymbol{X},\boldsymbol{Y},\boldsymbol{Z}) = \sum_{i=1}^{q}\big(&\boldsymbol{X}_0\boldsymbol{Y}_i\boldsymbol{Z}_i + \boldsymbol{X}_i\boldsymbol{Y}_0\boldsymbol{Z}_i + \boldsymbol{X}_i\boldsymbol{Y}_i\boldsymbol{Z}_0\big) \\
&+ \boldsymbol{X}_0\boldsymbol{Y}_0\boldsymbol{Z}_{q+1} + \boldsymbol{X}_0\boldsymbol{Y}_{q+1}\boldsymbol{Z}_0 + \boldsymbol{X}_{q+1}\boldsymbol{Y}_0\boldsymbol{Z}_0
\end{aligned} \tag{3.48}
$$

即

$$
\begin{aligned}
p(\boldsymbol{X},\boldsymbol{Y},\boldsymbol{Z}) = &\boldsymbol{X}^{[0]}\boldsymbol{Y}^{[1]}\boldsymbol{Z}^{[1]} + \boldsymbol{X}^{[1]}\boldsymbol{Y}^{[0]}\boldsymbol{Z}^{[1]} + \boldsymbol{X}^{[1]}\boldsymbol{Y}^{[1]}\boldsymbol{Z}^{[0]} \\
&+ \boldsymbol{X}^{[0]}\boldsymbol{Y}^{[0]}\boldsymbol{Z}^{[2]} + \boldsymbol{X}^{[0]}\boldsymbol{Y}^{[2]}\boldsymbol{Z}^{[0]} + \boldsymbol{X}^{[2]}\boldsymbol{Y}^{[0]}\boldsymbol{Z}^{[0]}
\end{aligned} \tag{3.49}
$$

$$\begin{cases} \boldsymbol{X}^{[0]}\boldsymbol{Y}^{[0]}\boldsymbol{Z}^{[1]} = \sum_{i=1}^{q} \boldsymbol{X}_0\boldsymbol{Y}_i\boldsymbol{Z}_i \\ \boldsymbol{X}^{[1]}\boldsymbol{Y}^{[0]}\boldsymbol{Z}^{[1]} = \sum_{i=1}^{q} \boldsymbol{X}_i\boldsymbol{Y}_0\boldsymbol{Z}_i \\ \boldsymbol{X}^{[1]}\boldsymbol{Y}^{[1]}\boldsymbol{Z}^{[0]} = \sum_{i=1}^{q} \boldsymbol{X}_i\boldsymbol{Y}_i\boldsymbol{Z}_0 \\ \boldsymbol{X}^{[0]}\boldsymbol{Y}^{[0]}\boldsymbol{Z}^{[2]} = \boldsymbol{X}_0\boldsymbol{Y}_0\boldsymbol{Z}_{q+1} \\ \boldsymbol{X}^{[0]}\boldsymbol{Y}^{[2]}\boldsymbol{Z}^{[0]} = \boldsymbol{X}_0\boldsymbol{Y}_{q+1}\boldsymbol{Z}_0 \\ \boldsymbol{X}^{[2]}\boldsymbol{Y}^{[0]}\boldsymbol{Z}^{[0]} = \boldsymbol{X}_{q+1}\boldsymbol{Y}_0\boldsymbol{Z}_0 \end{cases} \tag{3.50}$$

通过应用该算法，第二步建模过程的运算复杂度可以降低到$\mathrm{O}\left(\left(LJ\right)^{2.376}\right)$。

但是，上述方法仅仅是在降低复数矩阵逆运算的层面上降低了运算复杂度。实际上，我们可以通过对计算流程的合理构造和转化，直接避免高运算量的逆运算。

我们重新考虑第二步建模过程$\boldsymbol{W} = f_2\left(\boldsymbol{V}\right)$，得到

$$\boldsymbol{W}_k\left(\boldsymbol{V}_k^H\boldsymbol{V}_k + \sigma^2\boldsymbol{I}\right) = \boldsymbol{V}_k\left(\boldsymbol{V}_k^H\boldsymbol{V}_k + \sigma^2\boldsymbol{I}\right)^{-1}\left(\boldsymbol{V}_k^H\boldsymbol{V}_k + \sigma^2\boldsymbol{I}\right) \tag{3.51}$$

即

$$\left(\boldsymbol{V}_k^H\boldsymbol{V}_k + \sigma^2\boldsymbol{I}\right)\boldsymbol{W}_k^H = \boldsymbol{V}_k^H \tag{3.52}$$

此时我们可以构造残差函数：

$$F_k(\boldsymbol{W}) = \left\|\left(\boldsymbol{V}_k^H\boldsymbol{V}_k + \sigma^2\boldsymbol{I}\right)\boldsymbol{W}_k^H - \boldsymbol{V}_k^H\right\|_2^2 \tag{3.53}$$

从而将一个矩阵求逆的问题，转换成了最小二乘梯度迭代下降问题，利用文献[12]当中的基于 Nuemann 级数 1 阶展开近似求逆的方法，可以在保证矩阵求逆结果满足精度要求情况下，使得计算复杂度降低到$\mathrm{O}\left(\left(LJ\right)^2\right)$，具体的复杂度以及该方法的收敛性详细推导可见文献[12]。

已知复数加法的计算复杂度为 2，复数乘法的计算复杂度为 14。由于拼接后矩阵为$N \times LJ$维复数矩阵，当模型应用 data.mat 文件给出的数据（$N = 64$，$LJ = 2 \times 4 = 8$，$K = 384$）时，第二步求逆运算建模的一次运算复杂度减为$\left(64^2 \times 8\right) \times 14 + 8^2 \times 2 = 458880$，因此模型第二部分共需要的总运算复杂度为$458880 \times 384 = 176209920$。

3.4 问题 1 最终计算复杂度结果分析

综上所述，对影响相关矩阵组近似求解计算复杂度的重要过程，我们从矩阵间的关联性、奇异值分解优化、矩阵求逆过程迭代替换等角度，进行了模型的优化及计算复杂度对比，最终在满足估计建模精度要求$\rho_{\min}(\boldsymbol{V}) \geq \rho_{th} = 0.99$，$\rho_{\min}(\boldsymbol{W}) \geq \rho_{th} = 0.99$的情况下，归纳设计出总体计算复杂度最低的相关矩阵组整体求解流程，以及各步骤涉及的计算复杂度情况，如下图 3.5。图中虚线框即表示整个计算分析模型$\hat{\boldsymbol{W}} = f(\boldsymbol{H})$。

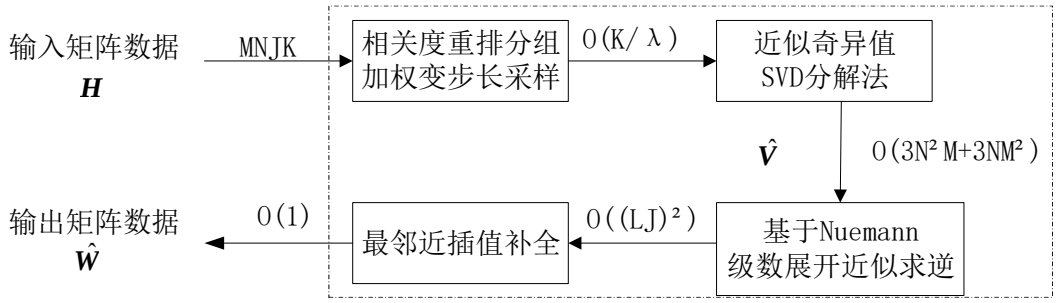


图 3.6 本文改进的最终模型计算流程及运算复杂度

与此同时，本文中所提的相关矩阵组常规求解算法的整体模型流程和相应的运算复杂度，也可以用下图 3.7 表示。

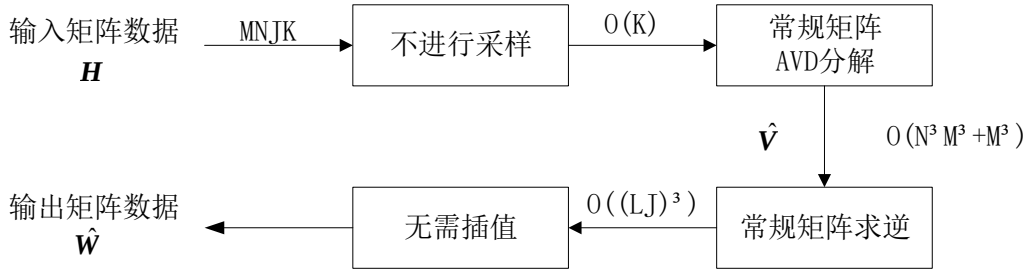


图 3.7 常规模型流程及运算复杂度

通过对比图 3.6 和图 3.7 可以看出，相比于相关矩阵组的常规求解方法，本文改进的设计分析模型，首先利用给定数据矩阵间的关联性，进行加权变步长的合理采样，并在最后通过最邻近插值的方法进行矩阵组补齐，这样可以使得计算量可以从 K 次直接降到了 K/λ 次。其次，本模型中，采用了文献[13]提出的近似奇异值 SVD 分解的方法，将矩阵进行矢量化拼接，通过迭代的方法求得最后的右奇异向量矩阵，使得该部分的计算复杂度由常规方法的 $O(N^3M^3+M^3)$ 降到了 $O(3N^2M+3M^2N)$。最后，因矩阵求逆的复杂度相当之高，我们将其转成一个 LMS 的求解问题，利用基于 Nuemann 级数 1 阶展开近似求逆的方法，使得该部分运算复杂度由原来的 $\mathrm{O}\left((LJ)^3\right)$ 降低到 $\mathrm{O}\left((LJ)^2\right)$。综上，经过这三个部分的改进，使得在保证估计矩阵精度的要求下，大大降低了相关矩阵组求解的运算复杂度。

进一步，根据本文计算复杂度的定义，使用 data.mat 文件中的六组数据，按照关联性选取合适的系数 λ，得到六组数据利用改进模型和常规模型算法流程的计算复杂度结果对比如下表 3.4 所示：

表 3.4 相关矩阵组的计算复杂度对比

计算量	常规模型	本文模型	常规：本文（倍）
第一组	6,442,647,552	10,813,440	595
第二组	6,442,647,552	10,919,741	590
第三组	6,442,647,552	10,737,745	600
第四组	6,442,647,552	10,561,717	610
第五组	6,442,647,552	10,813,440	590
第六组	6,442,647,552	10,475,849	615

由此表可以看出，针对问题 1，我们所搭建的相关矩阵组近似求解的直接模型设计 $\hat{\boldsymbol{W}} = f(\boldsymbol{H})$，在保证了估计建模精度的前提下，整体流程的计算复杂度也降低了大约 600 倍（10^2 数量级），体现了模型的有效性。

4. 问题 2 的模型建立与分析

4.1 问题 2 分析

问题 2 是在问题 1 的基础上，考虑数据的存储问题。由于给定的复数矩阵数据间具有一定的关联性，因此可以通过对矩阵数据的压缩运算，达到降低存储复杂度的目的。同时，压缩后的数据仍然保存原始数据的特征信息，并确保可以通过解压缩运算进行恢复。

问题 2 中需要设计关于初始数据矩阵 $\boldsymbol{H}$ 和最终结果数据矩阵 $\boldsymbol{W}$ 相对应的压缩和解压缩模型，使得其压缩和解压缩过程的计算复杂度最低。其中，计算复杂度同问题 1。由于本问题中不考虑分析模型对输出矩阵的影响，因此，问题 2 可以看作是有约束的优化问题，对计算复杂度进行最小化计算，约束条件为误差 $err \le -30\text{dB}$，同时，应尽量使压缩后的存储复杂度尽可能的小。优化过程中应注意，由于两个建模的优化目标的优先级相同，即存储复杂度和压缩与解压缩的计算复杂度的优先级相同，模型的权重系数应该一致。

本章的主要工作是设计压缩与解压缩函数模型，计算模型存储复杂度及运算复杂度并对模型进行优化。

4.2 问题 2 数学模型的建立

4.2.1 压缩模型

本文将存储复杂度定义为矩阵数据占用存储空间的大小，单位为比特。由于 data.mat 文件中的矩阵数据为复数浮点数，因此本文中的矩阵数据所占用的存储空间为 $64\times$ 矩阵维度。由于 data.mat 文件中的 $M=4$，$N=64$，$J=4$，$K=384$，$L=2$，因此，$M\times N\times J\times K$ 维矩阵 $\boldsymbol{H}$ 所占用的存储空间为 $64MNJK=25165824$ 比特，$N\times L\times J\times K$ 维数据矩阵 $\boldsymbol{W}$ 所占用的存储空间为 $64NLJK=12582912$ 比特。

由于数据矩阵占用存储空间过大，我们考虑分别对初始数据矩阵 $\boldsymbol{H}$ 和最终结果数据矩阵 $\boldsymbol{W}$ 进行压缩处理。

压缩过程可以表示为

$$\begin{aligned}\boldsymbol{H}' &= P_1(\boldsymbol{H})\\ \boldsymbol{W}' &= P_2(\boldsymbol{W})\end{aligned} \tag{4.1}$$

其中，$P_1(\bullet)$ 和 $P_2(\bullet)$ 分别表示关于矩阵 $\boldsymbol{H}$ 和 $\boldsymbol{W}$ 的压缩函数，$\boldsymbol{H}'$ 和 $\boldsymbol{W}'$ 表示矩阵压缩后的结果，即需要占用存储空间的矩阵数据。

复数矩阵数据的压缩处理示意图如下

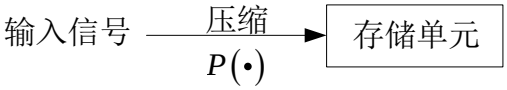


图 4.1 压缩处理示意图

本节借鉴图像处理领域的压缩算法对数据矩阵进行压缩，由于 data.mat 文件中的数据不符合压缩感知等算法要求数据具有稀疏性的先决条件，我们先进行一次自适应 Contourlet 变换操作[13]。

Contourlet 变换模型组合了 ADFB 与 LP 滤波器组，可以对数据矩阵进行非线性近似，具有很小的误差衰减和计算复杂度。自适应 Contourlet 变换模型的结构示意图如下：

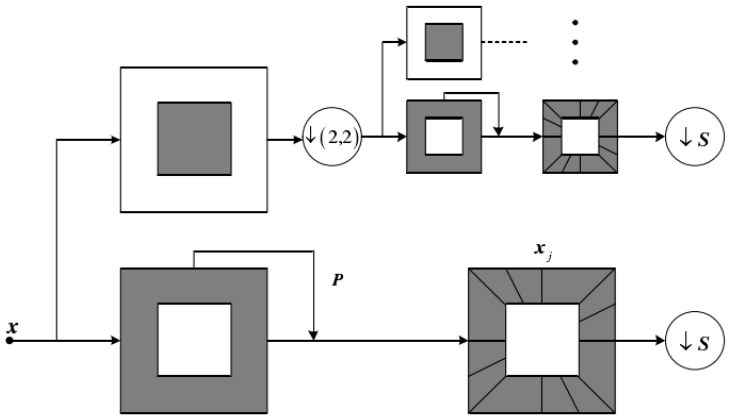


图 4.2 自适应 Contourlet 变换模型的结构示意图

由于 LP 滤波器组存在大小为4/3的冗余，因此 Contourlet 变换中存在相同的冗余度。本文选用自适应 Contourlet 变换模型来降低这个冗余度，从而使压缩后得到的数据矩阵更加稀疏。

在得到稀疏数据之后，本文应用压缩感知理论来进行数据的压缩处理，模型引入利用概率划分稀疏域[14]的压缩感知算法。算法将数据矩阵压缩成 $\boldsymbol{H}=\boldsymbol{\Phi}\boldsymbol{H}'=\begin{bmatrix}\boldsymbol{I} & \\ & \boldsymbol{\Phi}_{part}\end{bmatrix}\boldsymbol{H}'$ 的形式，$\boldsymbol{\Phi}$ 为测量矩阵。数据矩阵压缩过程可以表示为：

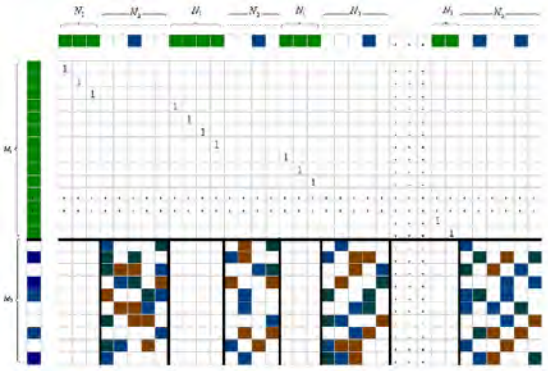


图 4.3 SDP 理论下的压缩感知测量矩阵

上图中的 $\boldsymbol{M}_1$ 部分表示经过正交变换后的稀疏数据，其中绿色部分表示该位置上的数据高于阈值。图中右侧为原始矩阵数据 $\boldsymbol{H}$，中间为得到的测量矩阵 $\boldsymbol{\Phi}$，左侧为压缩之后的矩阵 $\boldsymbol{H}'$。

利用概率划分稀疏域的压缩感知模型的流程图为

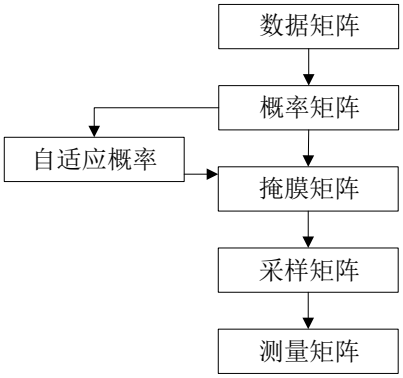


图 4.4 利用概率划分稀疏域的压缩感知模型的流程图

以上我们就完成了压缩的第一部分，也就是矩阵内数据压缩。下面我们对矩阵间数据进行压缩处理，类似于视频信号的帧间压缩。

在本部分模型中，矩阵间数据的压缩采用以下整体架构

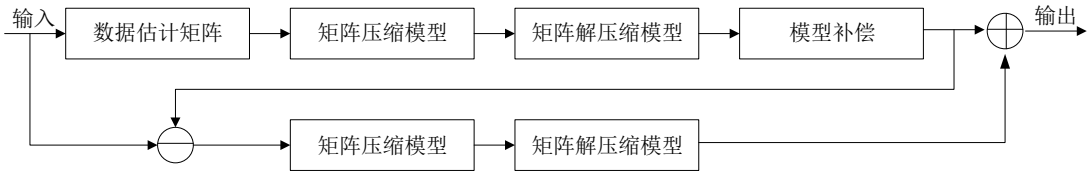


图 4.5 矩阵间数据的压缩整体架构图

其中，输入为当前待压缩的矩阵块数据，模型补偿输出为根据参考矩阵和特征信息得到的数据矩阵。根据当前矩阵数据以及参考矩阵估计，以及解压缩后矩阵与原始矩阵间的残差，再次进行数据的解压缩处理，最终得到回复的输出矩阵数据。

基于上述压缩模型 $P_1(\bullet)$ 和 $P_2(\bullet)$ 的运算复杂度分别为 $\mathrm{O}\left(5/2\,MNK^2\right)$ 和 $\mathrm{O}\left(5/2\,NLK^2\right)$。

4.2.2 解压缩模型

初始数据矩阵 $\boldsymbol{H}$ 和最终结果数据矩阵 $\boldsymbol{W}$ 以压缩的形式 $\boldsymbol{H}'$ 和 $\boldsymbol{W}'$ 存储在储存器中。在进行处理和最终输出时，需要从存储器中读取并进行解压缩处理，使其恢复为原始数据。

解压缩过程可以表示为

$$\begin{aligned}\hat{\boldsymbol{H}} &= G_1\left(\boldsymbol{H}'\right) = G_1\left(P_1\left(\boldsymbol{H}\right)\right)\\ \hat{\boldsymbol{W}} &= G_2\left(\boldsymbol{W}'\right) = G_2\left(P_2\left(\boldsymbol{W}\right)\right)\end{aligned} \tag{4.2}$$

其中，$G_1(\bullet)$ 和 $G_2(\bullet)$ 分别表示关于矩阵 $\boldsymbol{H}'$ 和 $\boldsymbol{W}'$ 的解压缩函数，$\hat{\boldsymbol{H}}$ 和 $\hat{\boldsymbol{W}}$ 表示矩阵压缩后的结果，即需要被处理或输出的矩阵数据。

复数矩阵数据的解压缩处理示意图如下

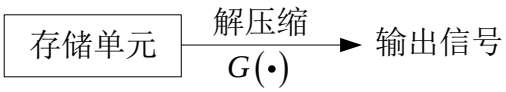


图 4.6 解压缩处理示意图

由于解压缩过程可以看作是压缩的逆过程，我们同样使用压缩感知模型来进行数据的恢复。

由于恢复数据的过程为求解欠定方程组的过程，即已知方程数量少于待恢复的矩阵数据维度，因此这里采用压缩采样匹配追踪算法来进行数据矩阵的重构。

模型通过设立软阈值判断原子集合，并利用最小二乘法求出待解压缩的数据矩阵在所得支撑集上的系数。在每次迭代过程中，取前两个最大元所对应的支撑与上一次迭代所得的解的支撑的平均值作为当前迭代的候选支撑，进而求解相应支撑集上的系数估计。

模型流程图如下所示：

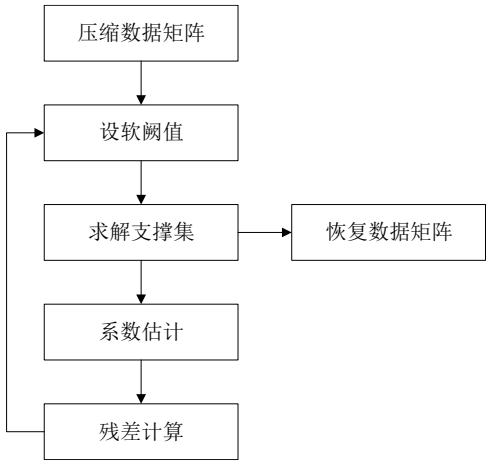


图 4.7 重构模型流程图

将数据矩阵进行分组处理，由于数据矩阵具有一定的非局部自相似性，我们可以将结构相似的矩阵块组成整体再进行解压缩。矩阵间具有一定相关性，因此我们可以选取相关度较高的矩阵作为参考矩阵，在矩阵内部和矩阵之间进行搜索相似矩阵块操作，从而将其组合，并通过结构相似性的大小来选取与矩阵块更为相似的匹配块。

在数据输出端，对压缩数据矩阵使用 MH-BCS-SPL[15]重构算法进行初始化恢复，得到矩阵 $\boldsymbol{H}^{(0)}$，其中，上标数字代表迭代次数。对 $\boldsymbol{H}^{(0)}$ 进行重叠分块操作，得到 D 个重叠块。对于每个块 $\boldsymbol{H}_d^{(0)}$ 进行搜索，并找到最小的矩阵块后组合得到相似数据块组。

模型在每一次计算过程中通过奇异值分解得到自适应字典，而相似数据块组中的所有数据块公用同一个字典，降低了计算复杂度并提高了模型的鲁棒性。

本文首先选取第一个维度的矩阵作为参考矩阵，进行独立矩阵内解压缩重构计算。对于其余维度矩阵，用已经恢复的高相关度矩阵作为参考矩阵，对当前矩阵进行初始重构，在进行矩阵间的联合分组解压缩重构。

本部分模型的具体流程图如下：

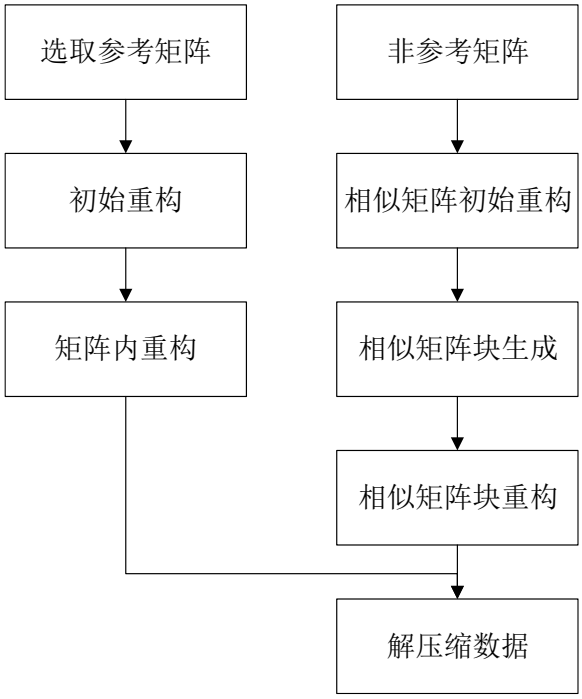


图 4.8 分组解压缩重构模型的流程图

为了保证压缩和解压缩过程的准确性，使参考矩阵包含的特征尽可能的多，上述模型中的参考矩阵采用基于自适应阈值设置[15]和阈值递减方案。由于固定阈值无法将数据矩阵的全部特征保留，因此，我们选取当前已恢复的矩阵块组的参考矩阵以及迭代中的矩阵块的均值作为下一次迭代的参考矩阵。同时，根据每次迭代后求得的误差进行对初始阈值自适应设置，其中，阈值为自适应阶梯递减的。本文将解压缩之后的误差定义为

$$err = 10\log_{10}\frac{E\left\{\left\|\hat{\boldsymbol{H}}_{j,k}-\boldsymbol{H}_{j,k}\right\|_F^2\right\}}{E\left\{\left\|\boldsymbol{H}_{j,k}\right\|_F^2\right\}} \tag{4.3}$$

其中，$\|\bullet\|_F$ 表示 Frobenius 范数，$E\{\bullet\}$ 表示矩阵的期望运算。

下图说明了迭代过程中非参考矩阵的解压缩过程

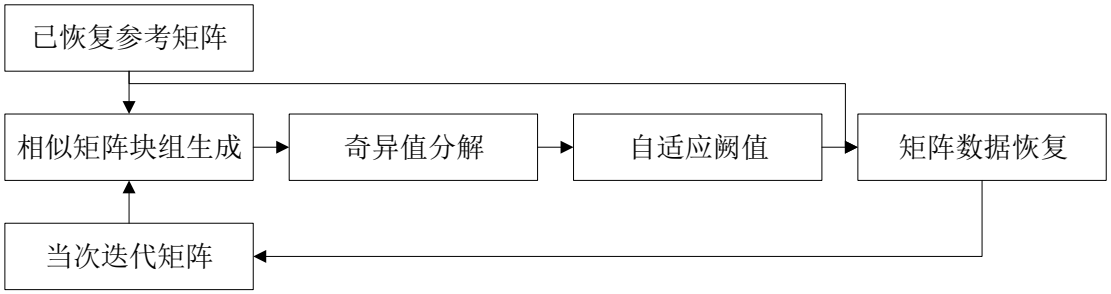


图 4.9 非参考矩阵的解压缩流程图

在矩阵数据解压缩过程中，同样应用了奇异值分解过程。模型对已分组的 $\boldsymbol{H}_{g_k}$ 进行奇异值分解，其中下标表示当前矩阵块在分组中的位置。分解后的奇异值按照由大到小沿矩主对角线排列，其中，最大的奇异值对应的特征向量包含了整个相似矩阵块中最多的特征信息。奇异值越大，数据矩阵块在其对应的特征方向上的投影越长，即具有高度相关性。而小奇异值所对应的通过解压缩恢复出来的是细节特征。

同样的，这里我们为了减少运算复杂度，采用设置阈值的方法来过滤掉较小的奇异值，只保留数据矩阵得到的较大奇异值。这样既可以提高解压缩模型的运算速度，又可以降低模型数据占用的存储空间。

上述基于解压缩模型 $G_1(\bullet)$ 和 $G_2(\bullet)$ 的计算复杂度分别为 $\mathrm{O}\left(MN\log\left(MN\right)\right)$ 和 $\mathrm{O}\left(NL\log\left(NL\right)\right)$。

4.3 问题 2 复杂度分析

4.3.1 存储复杂度

问题 2 中只使用 data.mat 文件中的六组数据中输入矩阵组 $\boldsymbol{H}$，以及输出矩阵组 $\boldsymbol{W}$ 部分，数据采用十进制格式。其中，$M=4$，$N=64$，$J=4$，$K=384$，$L=2$，$\sigma^2=0.01$ 矩阵 $\boldsymbol{H}$ 为 $M\times N\times J\times K$ 维复数浮点数矩阵数据，矩阵 $\boldsymbol{W}$ 为 $N\times L\times J\times K$ 维复数浮点数矩阵。

在满足误差 $err_{\boldsymbol{H}}\le E_{th1}=-30\text{dB}$ 及 $err_{\boldsymbol{W}}\le E_{th2}=-30\text{dB}$ 的情况下，本文所用压缩模型的存储复杂度如下表 4.1：

表 4.1 矩阵压缩前后占用存储空间

	输入矩阵组 $\boldsymbol{H}$		输出矩阵组 $\boldsymbol{W}$	
	压缩前	压缩后	压缩前	压缩后
第一组	25165824	7569408	12582912	4325376
第二组	25165824	8429568	12582912	4521984

接上表 4.1

第三组	25165824	7010304	12582912	3489792
第四组	25165824	7225344	12582912	3833856
第五组	25165824	6408192	12582912	3391488
第六组	25165824	7827456	12582912	3883008

本文所用解压缩模型的存储复杂度为

表 4.2 矩阵解压缩前后占用存储空间

	输入矩阵组 $\boldsymbol{H}$		输出矩阵组 $\boldsymbol{W}$	
	解压缩前	解压缩后	解压缩前	解压缩后
第一组	7569408	25165824	4325376	12582912
第二组	8429568	25165824	4521984	12582912
第三组	7010304	25165824	3489792	12582912
第四组	7225344	25165824	3833856	12582912
第五组	6408192	25165824	3391488	12582912
第六组	7827456	25165824	3883008	12582912

从上述表格 4.2 可以看出，显然，矩阵数据在解压缩后所暂用的存储空间与原始数据占用的存储空间相同，这是由于解压缩过程可以看作是压缩过程的逆过程所导致的。

下表 4.3 说明了数据矩阵的压缩率以及解压缩恢复的数据与原始 data.mat 文件中数据的误差

表 4.3 矩阵数据压缩率及误差

	输入矩阵组 $\boldsymbol{H}$		输出矩阵组 $\boldsymbol{W}$	
	压缩率	误差	压缩率	误差
第一组	69.922%	-30.12dB	65.625%	-30.23dB
第二组	66.504%	-30.09dB	64.063%	-30.17dB
第三组	72.144%	-31.27dB	72.266%	-31.06dB
第四组	71.290%	-31.10dB	69.531%	-30.84dB
第五组	74.536%	-30.82dB	73.047%	-30.68dB
第六组	68.896%	-31.04dB	69.141%	-30.76dB

上述结果表明，本文压缩及解压缩模型可以很好的进行数据压缩，并且可以将压缩后的数据矩阵还原。

4.3.2 运算复杂度

由于问题 2 中要求两个建模优化目标的优先级相同，即压缩和解压缩模型需要兼顾模型存储复杂度，压缩与解压缩的计算复杂度两个方面，下面我们对压缩模型的运算复杂度进行讨论

表 4.4 压缩模型运算复杂度

	运算类型	加法	乘法	逆	指数
	计算复杂度	2	14	238	1402
输入矩阵组	次数	838608	4412276	0	8256
$\boldsymbol{H}$	计算量	1677216	61771864	0	11574912
输出矩阵组	次数	464872	2218954	0	4152
$\boldsymbol{W}$	计算量	929744	31065356	0	5821104

由上表 4.4 可知，压缩输入矩阵组 $\boldsymbol{H}$ 的计算总量为 75023992，压缩输出矩阵组 $\boldsymbol{W}$ 的计算总量为 37816204。

下表说明了对数据矩阵进行解压缩过程的计算量

表 4.5 解压缩模型运算复杂度

	运算类型 计算复杂度	加法 2	乘法 14	逆 238	指数 1402
输入矩阵组	次数	1202176	6313248	8356	17472
$\boldsymbol{H}$	计算量	2404352	88385472	1988728	24495744
输出矩阵组	次数	822784	3165712	4192	8752
$\boldsymbol{W}$	计算量	1645568	44319968	997696	12270304

由上表可知，解压缩输入矩阵组 $\boldsymbol{H}$ 的计算总量为 117274296，压缩输出矩阵组 $\boldsymbol{W}$ 的计算总量为 59233536。

综上可以看出，在问题 2 中所提解压缩模型，在满足压缩误差的情况下，整个解压缩流程的存储复杂度降低了约 30%左右，同时压缩与解压缩的计算复杂度也达到估计精度要求下尽可能低的阈值。

5. 问题 3 的模型建立与分析

5.1 问题 3 分析

问题 3 是在问题 1 和问题 2 的基础上，结合问题 1 和问题 2 中的两个模型来实现从输入矩阵数据 $\boldsymbol{H}$ 到估计所得输出矩阵组 $\hat{\boldsymbol{W}}$ 的整体流程。题目要求不必须经过压缩/解压缩、计算单元、压缩/解压缩的过程，因此问题 3 具有很强的自主性。

问题 3 仅要求考虑 $\rho_{\min}\left(\boldsymbol{W}\right)\geq\rho_{th}=0.99$ 的矩阵数据，矩阵数据间具有较高的相关性，这为我们使用高压缩率模型提供了可能。同时，在保证总体流程输入输出误差的情况下，可以考虑通过给不同模型施加权重来控制最终的输出结果。

本章首先使用结合问题 1 和问题 2 的方法，直接进行流程化计算，得到最终的近似矩阵输出。其次，考虑模型优化，将步骤进行合并及调整，使得在同时保证模型运算复杂度和矩阵数据存储复杂度的同时，尽可能的使总体流程误差最小。

5.2 问题 3 数学模型的建立

5.2.1 加法器模型

本节主要讨论将问题 1 中的计算模型与问题 2 中的压缩解压缩模型直接叠加，得到整体输入输出模型。

整体流程图如下：

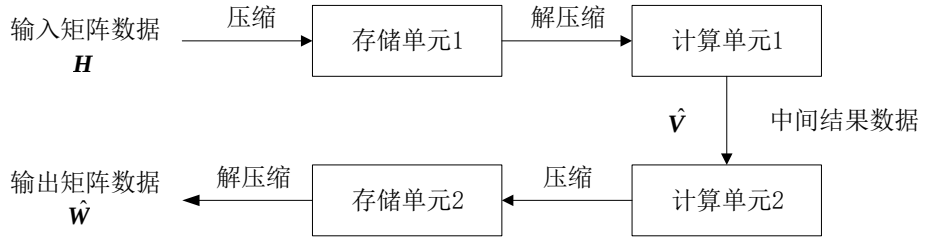


图 5.1 加法器模型整体流程图

由于该模型为问题 1 和问题 2 中模型的简单叠加，所以这里不再赘述，详见第 3 章及第 4 章。

5.2.2 权值改进模型

我们在本节引入权值变量来优化模型的整体流程，这里借鉴求解半光滑方程组[16]的正则化牛顿法来实现。由于系数矩阵的显示表达式不易求解，因此，为了降低模型整体的运算复杂度，这里我们采用共轭梯度法求解线性方程组而非常规迭代求解。

由于我们期望得到一个最佳权重用于给不同模块的模型进行加权处理，因此，我们首先需要设置初始化权值并给定约束条件。在求解过程中，当最优权值在局部收敛域内，才可以保证模型的收敛性。同时，模型应具有适当的步长值，以减少距离最优权值较远时的冗余运算量。本文将题目中最终输出与输入信号的误差值以及相邻两次迭代结果间的比值小于某一硬阈值作为终止条件来进行迭代。

下图是半光滑牛顿法流程图,

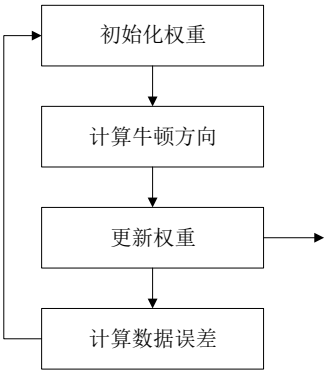


图 5.2 半光滑牛顿法流程图

相对于直接对计算模型以及压缩解压缩模型进行叠加，带有权值的叠加模型可以在一定程度上改善整体流程的运算量。但是加权模型在本质上仍然是对问题 1 和问题 2 中模型的叠加处理，仍然无法避免前述模型由自身高运算量引起的问题。

5.2.3 整体改进模型

前两种模型都是将计算单元进行拆分，分两步进行计算操作。同时，对于输入矩阵数据 $\boldsymbol{H}$ 以及输出矩阵数据 $\boldsymbol{W}$ 的存储，需要分别进行两次压缩及解压缩操作。这样，由初始输入数据得到最终的输出结果，一共需要进行六次运算。由于给定的 data.mat 文件中的数据十分庞大，因此，合理的简化运算次数可以在一定程度上简化总体流程运算量。

根据前述内容已知，压缩后的数据具有整体数据大部分的特征信息，因此，一种合理地想法是将压缩前数据进行舍弃，即直接对压缩后数据进行计算处理，而非在进行解压缩操作后再处理。解压缩过程是根据压缩后的数据中保留的特征来进行其余矩阵数据的填补，因此可以认为填补后的数据与压缩数据矩阵具有一定的等价性。

然而，显然对于输出数据矩阵来说，数据的压缩及解压缩是不可忽略的，这是因为最后面向用户的数据矩阵应具有一定的规范性。当然，如果在本文模型后续仍有其它运算步骤时，即本文所输出结果处于整体流程的中间步骤时，可以考虑适当缩减输出数据矩阵的解压缩过程。

对于计算单元，本文在第 3 章中采用分步计算的方式，将整体流程拆分成两个步骤。这里我们也可以考虑将计算单元进行优化处理，直接得到输出矩阵数据 $\boldsymbol{W}$ 。

$$\hat{\boldsymbol{W}} = f_2\left(\hat{\boldsymbol{V}}\right) = f_2\left(f_1\left(\boldsymbol{H}\right)\right) = f\left(\boldsymbol{H}\right) \tag{5.1}$$

这样既简化了模型计算步骤，又可以有效的降低由中间结果矩阵数据 $\hat{\boldsymbol{V}}$ 的估计以及传输时造成的加性和乘性误差。

下图 5.3 是整体改进之后的模型流程图

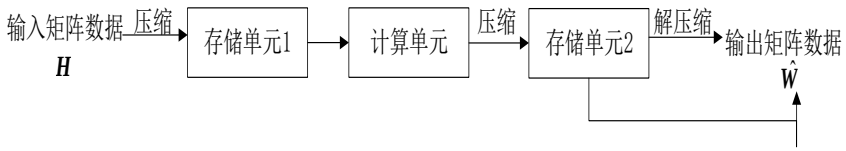


图 5.3 整体改进模型流程图

上图与直接叠加的模型相比，有效的减少了计算步骤，仅需要四步甚至三步运算便可得到最终的输出数据矩阵。对于庞大的输入数据矩阵来说，其实际的运算复杂度是呈倍数减少的。

6. 模型的讨论与评价

6.1 数学及算法模型的优点

本文提出有关相关矩阵组的低复杂度计算及存储模型，相较于原始分步计算模型以及直接存储矩阵数据的方式，本文所用模型具有以下优点:

- 考虑矩阵组组各个矩阵间的关联性，减少需要处理的计算单元的数目，使得整体的复杂度降低。
- 针对计算单元，对比各类 SVD 和求逆方法，选择计算量更小的方式来替代原有的直接计算方式。在处理数据时，用低计算量的改进算法来代替常规的奇异值分解计算，并且尽量通过对计算流程的构造和转化，避免高运算量的矩阵求逆等运算，从而达到降低整体运算量的目的。
- 利用稀疏化和压缩感知的原理，模型具有良好的适用性和可移植性。
- 采用压缩模型处理需要存储的矩阵数据，使得在不丢失数据原有特征的情况下，减少所占用的存储空间。同时，确保压缩后的数据可以通过解压缩处理恢复并且输出。
- 模型的整体改进是在对计算单元和存储单元分别进行改进的基础上，对整体流程进行优化。适当减少整体流程步骤，简化冗余，从而实现对给定矩阵数据的低复杂度计算和存储。

6.2 模型的改进

本文采用的计算模型不是典型的奇异值分解计算模型等，而且在此基础上，基于给定 data.mat 文件中矩阵数据高度相关的特点提出的。下面提出几点对于本文模型的改进及展望

- 本文步长的选择是基于矩阵间相关度的倍数的，这需要对矩阵数据进行初步分析。而在未知矩阵数据相关度的条件下，本文模型无法直接选择最优采样步长。因此，可以考虑对于变步长或者自适应步长的引入，进一步优化。
- 本文模型对于误差值得控制十分严格，因此为了降低误差模型不得不考虑保留一些数据的细节特征，这在一定程度上导致了计算的高复杂度。在对于误差值要求较低的情况下，舍弃一些数据，模型可以考虑实现更高的压缩率。

6.3 总结

本文建立了相关矩阵组的低复杂度计算和存储模型，对于给定的矩阵数据，实现了从矩阵输入信号到近似矩阵输出信号估计的整体流程。对于给定数据的处理，模型输出误差、估计精度符合题目要求。同时，由于矩阵数据的计算及压缩问题几乎是所有模型在处理数据时都会遇到的问题，因此，本文模型具有很强的使用价值。

参考文献

[1] Swartzlander, Earl E., and Hani H. Saleh. "Floating-point implementation of complex multiplication." 2009 Conference Record of the Forty-Third Asilomar Conference on Signals, Systems and Computers. IEEE, 2009.

[2] 洪恒,何明一.一种适于高光谱图像压缩的相关系数矩阵近似计算算法[J].电子设计工程,2013,21(12):128-131.

[3] Zhang J, Liu G. An efficient reordering prediction-based lossless compression algorithm for hyperspectral images[J]. IEEE Geoscience and remote sensing letters, 2007, 4(2): 283-287.

[4] Golub G, Kahan W. Calculating the singular values and pseudo-inverse of a matrix[J]. Journal of the Society for Industrial and Applied Mathematics, Series B: Numerical Analysis, 1965, 2(2): 205-224.

[5] Halko N, Martinsson P G, Tropp J A. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions[J]. SIAM review, 2011, 53(2): 217-288.

[6] Shlien S. A method for computing the partial singular value decomposition[J]. IEEE transactions on pattern analysis and machine intelligence, 1982 (6): 671-676.

[7] James R. Bunch and Christopher P. Nielsen. Updating the singular value decomposition[J]. Numerische Mathematik, 1978, 31(2): 111-129.

[8] G.H. Golub, C.F. Van Loan, Matrix Computations, 3rd Edition, Johns Hopkins University Press, Baltimore, MD, USA, 1996.

[9] G.W. Stewart, Error and perturbation bounds for subspaces associated with certain eigenvalue problems, SIAM Rev. 15 (1973) 727-764.

[10] R.D. Fierro, Perturbation theory for two-sided (or complete) orthogonal decompositions, SIAM J. Matrix Anal. Appl. 17 (1996) 383-400.

[11] Strassen, Volker. "Gaussian elimination is not optimal." Numerische mathematik 13.4 (1969): 354-356.

[12] 谭红成. 大规模 MIMO 系统中的低复杂度方法研究[D].电子科技大学,2021.

[13] 李佳. 基于压缩采样的信号重构与分析[D].电子科技大学,2020.

[14] 李金昊. 基于组稀疏的视频压缩感知重构算法研究[D].华南理工大学,2019.

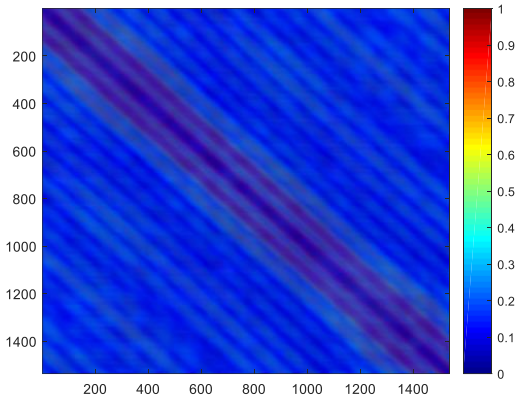
[15] 赵小梅. 基于自适应 Contourlet 变换的压缩感知及其在图像压缩中的应用[D].重庆邮电大学,2017.

[16] 宋君. 图像的压缩感知重构算法研究[D].西安电子科技大学,2013.

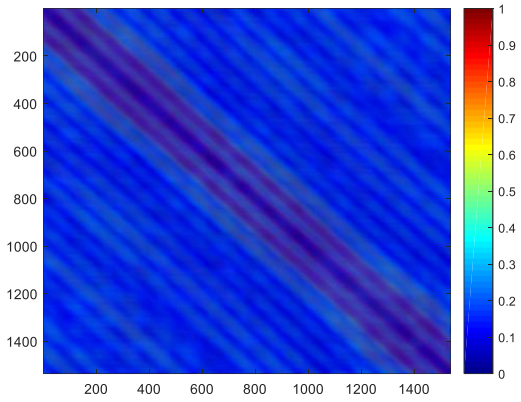
附录

附录 A 矩阵间或矩阵内的相似系数谱图分析

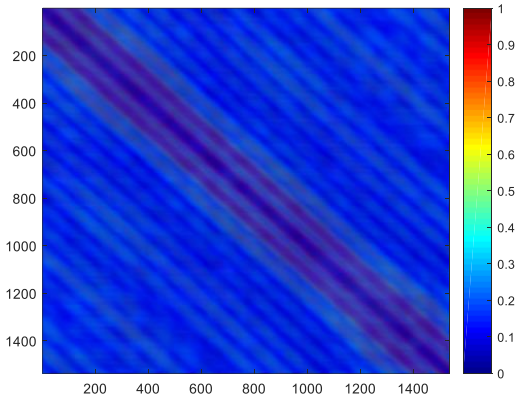
下图 a 为给定 6 组输入矩阵，各组数据矩阵间的相似系数谱图（1-代表完全相关，0-代表不相关）。由该图可以看出，这 6 组的矩阵间关联情况一致，因此可以建立统一的模型，进行复杂度的降低。



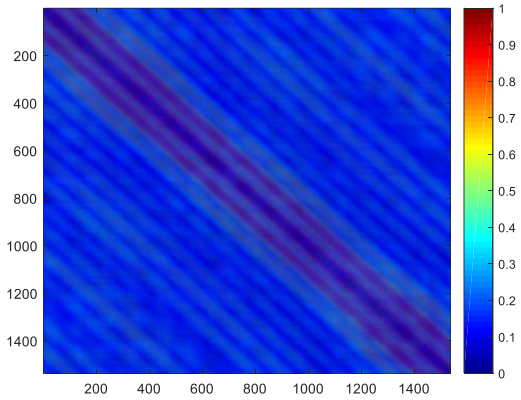
第一组



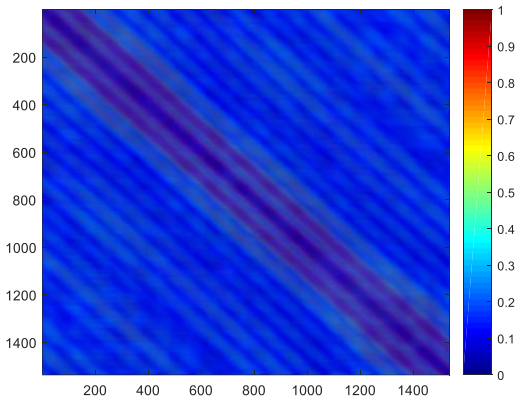
第二组



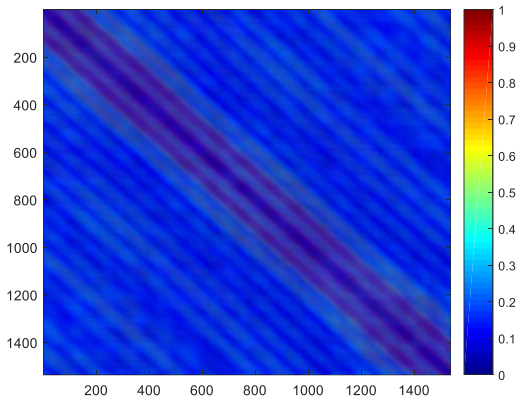
第三组



第四组



第五组

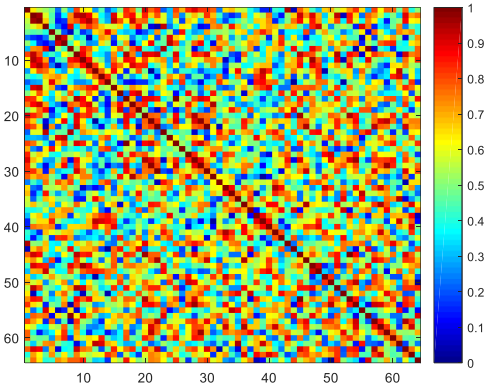


第六组

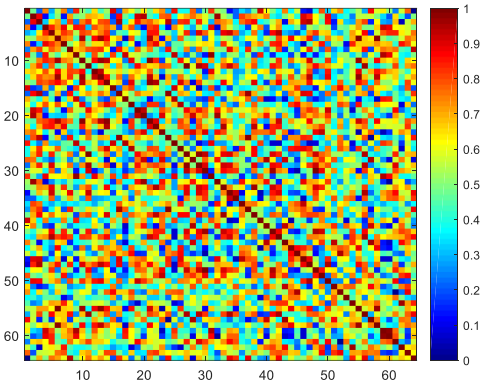
图 a 各组矩阵间相似系数谱图

下图 b 为给定 6 组输入矩阵，随机选取的 2 组内某个矩阵内部行列元素的的相似系数谱图。由图 b 可以看出，矩阵内部元素的关联性差异较大，适合具体情况具体分析，因此本文在降低计算复杂度时，对于矩阵内部元素关联未多做研究。

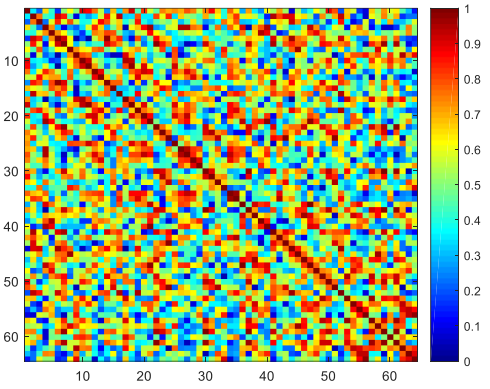
data1_H(:,:,1,1)



data1_H(:,:,2,156)



Data3_H(:,:,1,1)



data3_H(:,:,2,156)

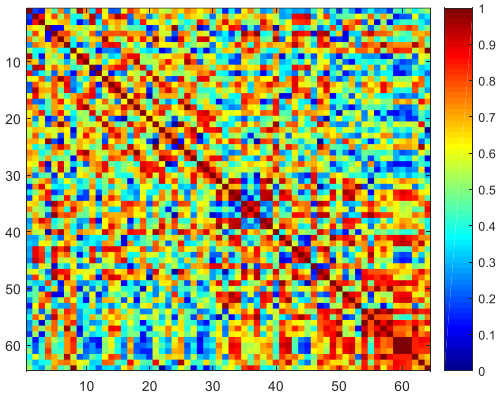


图 b 矩阵内各元素相关系数谱图

附录 B 本文中所涉及的相关代码

3.2.3 节相关代码

```
static void mm_strassen(float* matA, float* matB, float* matC, const int M, const int N, const int K, const int str
{
    if ((M <= 64) || (M%2 != 0 ||N%2 != 0 ||K%2!=0))
    {
        return mm_generate(matA, matB, matC, M, N, K, strideA, strideB, strideC);
    }
    memset(matC, 0, M*strideC*sizeof(float));
    int offset = 0;

    //M1 = (A11+A22)*(B11+B22)
    std::vector<float> M1((M / 2) * (N / 2));
    {
        memset(&M1[0], 0, M1.size()*sizeof(float));
        //M1_0 = (A11+A22)
        std::vector<float> M1_0((M / 2) * (K / 2));
        offset = M*strideA / 2 + K / 2;
        for (int i = 0; i < M / 2; i++)
        {
            for (int j = 0; j < K/2; j++)
            {
                const int baseIdx = i*strideA + j;
                M1_0[i*K/2+j] = matA[baseIdx] + matA[baseIdx + offset];
            }
        }
        //M1_1 = (B11+B22)
        std::vector<float> M1_1((K / 2) * (N / 2));
        offset = K*strideB / 2 + N / 2;
        for (int i = 0; i < K / 2; i++)
        {
            for (int j = 0; j < N / 2; j++)
```

```
            {
                const int baseIdx = i*strideB + j;
                M1_1[i*N/2+j] = matB[baseIdx] + matB[baseIdx + offset];
            }
        }
        mm_strassen(&M1_0[0], &M1_1[0], &M1[0], M / 2, N / 2, K / 2,
            K/2,N/2,N/2);
    }

    //M2 = (A21+A22)*B11
    std::vector<float> M2((M / 2) * (N / 2));
    {
        memset(&M2[0], 0, M2.size()*sizeof(float));
        //M2_0 = (A21+A22)
        std::vector<float> M2_0((M / 2) * (K / 2));
        offset = K / 2;
        for (int i = M / 2; i < M; i++)
        {
            for (int j = 0; j < K / 2; j++)
            {
                const int baseIdx = i*strideA + j;
                M2_0[(i-M/2)*K/2+j] = matA[baseIdx] + matA[baseIdx + offset];
            }
        }
        //M2_2 = B11
        mm_strassen(&M2_0[0], &matB[N / 2], &M2[0], M / 2, N / 2, K / 2,
            K / 2, strideB, N / 2);
    }

    //M3 = A11*(B12-B22)
    std::vector<float> M3((M / 2) * (N / 2));
    {
        memset(&M3[0], 0, M3.size()*sizeof(float));
        //M3_0 = A11
        //M3_1 = (B12-B22)
        std::vector<float> M3_1((K / 2) * (N / 2));
        offset = K*strideB / 2;
        for (int i = 0; i < K/2; i++)
        {
            for (int j = N/2; j < N; j++)
            {
                const int baseIdx = i*strideB + j;
                M3_1[i*N/2+j-N/2] = matB[baseIdx] - matB[baseIdx + offset];
            }
        }
        mm_strassen(matA, &M3_1[0], &M3[0], M / 2, N / 2, K / 2,
            strideA, N / 2, N / 2);
    }

    //M4 = A22*(B21-B11)
    std::vector<float> M4((M / 2) * (N / 2));
    {
        memset(&M4[0], 0, M4.size()*sizeof(float));
        //M4_0 = A22
        //M4_1 = (B12-B22)
        std::vector<float> M4_1((K / 2) * (N / 2));
        offset = K*strideB / 2;
        for (int i = 0; i < K / 2; i++)
        {
            for (int j = N / 2; j < N; j++)
            {
                const int baseIdx = i*strideB + j;
                M4_1[i*N/2+j-N/2] = matB[baseIdx + offset] - matB[baseIdx];
            }
        }
        mm_strassen(matA + M*strideA / 2 + K / 2, &M4_1[0], &M4[0], M / 2, K / 2, N / 2,
            strideA, N / 2, N / 2);
    }

    //M5 = (A11+A12)*B22
    std::vector<float> M5((M / 2) * (N / 2));
    {
        memset(&M5[0], 0, M5.size()*sizeof(float));
        //M5_0 = (A11+A12)
        std::vector<float> M5_0((M / 2) * (K / 2));
        offset = K / 2;
        for (int i = 0; i < M/2; i++)
        {
            for (int j = 0; j < K / 2; j++)
            {
                const int baseIdx = i*strideA + j;
                M5_0[i*K / 2 + j] = matA[baseIdx] + matA[baseIdx + offset];
            }
        }
        //M5_1 = B22
        mm_strassen(&M5_0[0], &matB[K*strideB / 2 + N / 2], &M5[0], M / 2, N / 2, K / 2,
            K / 2, strideB, N / 2);
    }

    //M6 = (A21-A11)*(B11+B12)
```

```
    std::vector<float> M6((M / 2) * (N / 2));
    {
        memset(&M6[0], 0, M6.size()*sizeof(float));
        //M6_0 = (A21-A11)
        std::vector<float> M6_0((M / 2) * (K / 2));
        offset = K*N / 2;
        for (int i = 0; i < M / 2; i++)
        {
            for (int j = 0; j < K/2; j++)
            {
                const int baseIdx = i*strideA + j;
                M6_0[i*K/2+j] = matA[baseIdx + offset] - matA[baseIdx];
            }
        }
        //M6_1 = (B11+B12)
        std::vector<float> M6_1((K / 2) * (N / 2));
        offset = N / 2;
        for (int i = 0; i < K / 2; i++)
        {
            for (int j = 0; j < N/2; j++)
            {
                const int baseIdx = i*strideB + j;
                M6_1[i*N/2+j] = matB[baseIdx] + matB[baseIdx + offset];
            }
        }
        mm_strassen(&M6_0[0], &M6_1[0], &M6[0], M / 2, N / 2, K / 2,
            K / 2, N / 2, N / 2);
    }

    //M7 = (A12-A22)*(B21+B22)
    std::vector<float> M7((M / 2) * (N / 2));
    {
        memset(&M7[0], 0, M7.size()*sizeof(float));
        //M7_0 = (A12-A22)
        std::vector<float> M7_0((M / 2) * (K / 2));
        offset = M*strideA / 2;
        for (int i = 0; i < M / 2; i++)
        {
            for (int j = K/2; j < K; j++)
            {
                const int baseIdx = i*strideA + j;
                M7_0[i*K / 2 + j - K / 2] = matA[baseIdx] - matA[baseIdx + offset];
            }
        }
        //M7_1 = (B21+B22)
        std::vector<float> M7_1((K / 2) * (N / 2));
        offset = N / 2;
        for (int i = K/2; i < K; i++)
        {
            for (int j = 0; j < N / 2; j++)
            {
                const int baseIdx = i*strideB + j;
                M7_1[(i-K/2)*N / 2 + j] = matB[baseIdx] + matB[baseIdx + offset];
            }
        }
        mm_strassen(&M7_0[0], &M7_1[0], &M7[0], M / 2, N / 2, K / 2,
            K / 2, N / 2, N / 2);
    }
    for (int i = 0; i < M / 2;i++)
    {
        for (int j = 0; j < N / 2;j++)
        {
            const int idx = i*N / 2 + j;
            //C11 = M1+M4-M5+M7
            matC[i*strideC + j] = M1[idx] + M4[idx] - M5[idx] + M7[idx];
            //C12 = M3+M5
            matC[i*strideC + j + N/2] = M3[idx] + M5[idx];
            //C21 = M2+M4
            matC[(i+M/2)*strideC + j] = M2[idx] + M4[idx];
            //C22 = M1-M2+M3+M6
            matC[(i+M/2)*strideC + j + N/2] = M1[idx] - M2[idx] + M3[idx] + M6[idx];
        }
    }
}
```

```
static void mm_winograd(float* matA, float* matB, float* matC, const int M, const int N, const int K, const int strideA,
{
    if ((M <= 64) || (M % 2 != 0 || N % 2 != 0 || K % 2 != 0))
    {
        return mm_generate(matA, matB, matC, M, N, K, strideA, strideB, strideC);
    }
    memset(matC, 0, M*strideC*sizeof(float));
    int offset = 0;

    std::vector<float> S1((M / 2) * (K / 2));
    std::vector<float> S2((M / 2) * (K / 2));
    std::vector<float> S3((M / 2) * (K / 2));
    std::vector<float> S4((M / 2) * (K / 2));
    for (int i = 0; i < M / 2;i++)
    {
        for (int j = 0; j < K / 2;j++)
        {
            const int idx = i*K / 2 + j;
            //S1 = A21 + A22
            S1[idx] = matA[(i + M / 2)*strideA + j] + matA[(i + M / 2)*strideA + j + K / 2];
            //S2 = S1 - A11
            S2[idx] = S1[idx] - matA[i*strideA + j];
            //S3 = A11 - A21
            S3[idx] = matA[i*strideA + j] - matA[(i + M / 2)*strideA + j];
            //S4 = A12 - S2
            S4[idx] = matA[i*strideA + j + K / 2] - S2[idx];
        }
    }
    std::vector<float> T1((K / 2) * (N / 2));
    std::vector<float> T2((K / 2) * (N / 2));
    std::vector<float> T3((K / 2) * (N / 2));
    std::vector<float> T4((K / 2) * (N / 2));
    for (int i = 0; i < K / 2; i++)
    {
        for (int j = 0; j < N / 2; j++)
        {
            const int idx = i*N / 2 + j;
            //T1 = B21 - B11
            T1[idx] = matB[(i + K / 2)*strideB + j] - matB[i*strideB + j];
            //T2 = B22 - T1
            T2[idx] = matB[(i + K / 2)*strideB + j + N / 2] - T1[idx];
            //T3 = B22 - B12
            T3[idx] = matB[(i + K / 2)*strideB + j + N / 2] - matB[i*strideB + j + N / 2];
            //T4 = T2 - B21
            T4[idx] = T2[idx] - matB[(i + K / 2)*strideB + j];
        }
    }

    //M1 = A11*B11
    std::vector<float> M1((M / 2) * (N / 2));
    {
        memset(&M1[0], 0, M1.size()*sizeof(float));
        mm_winograd(matA, matB, &M1[0], M / 2, N / 2, K / 2,
            strideA, strideB, N / 2);
    }

    //M2 = A12*B21
    std::vector<float> M2((M / 2) * (N / 2));
    {
        memset(&M2[0], 0, M2.size()*sizeof(float));
        mm_winograd(matA + K / 2, matB + K*strideB/2, &M2[0], M / 2, N / 2, K / 2,
            strideA, strideB, N / 2);
    }

    //M3 = S4*B22
    std::vector<float> M3((M / 2) * (N / 2));
    {
        memset(&M3[0], 0, M3.size()*sizeof(float));
        mm_winograd(&S4[0], matB + K*strideB/2 + N / 2, &M3[0], M / 2, N / 2, K / 2,
            K/2, strideB, N / 2);
    }

    //M4 = A22*T4
    std::vector<float> M4((M / 2) * (N / 2));
    {
        memset(&M4[0], 0, M4.size()*sizeof(float));
        mm_winograd(matA + M*strideA / 2 + K / 2, &T4[0], &M4[0], M / 2, N / 2, K / 2,
            strideA, N / 2, N / 2);
    }

    //M5 = S1*T1
    std::vector<float> M5((M / 2) * (N / 2));
    {
        memset(&M5[0], 0, M5.size()*sizeof(float));
        mm_winograd(&S1[0], &T1[0], &M5[0], M / 2, N / 2, K / 2,
            K / 2, N/2, N / 2);
    }

    //M6 = S2*T2
    std::vector<float> M6((M / 2) * (N / 2));
```

```
    {
        memset(&M6[0], 0, M6.size()*sizeof(float));
        mm_winograd(&S2[0], &T2[0], &M6[0], M / 2, N / 2, K / 2,
            K / 2, N / 2, N / 2);
    }

    //M7 = S3*T3
    std::vector<float> M7((M / 2) * (N / 2));
    {
        memset(&M7[0], 0, M7.size()*sizeof(float));
        mm_winograd(&S3[0], &T3[0], &M7[0], M / 2, N / 2, K / 2,
            K / 2, N / 2, N / 2);
    }

    for (int i = 0; i < M / 2; i++)
    {
        for (int j = 0; j < N / 2; j++)
        {
            const int idx = i*N / 2 + j;
            //U1 = M1 + M2
            const auto U1 = M1[idx] + M2[idx];
            //U2 = M1 + M6
            const auto U2 = M1[idx] + M6[idx];
            //U3 = U2 + M7
            const auto U3 = U2 + M7[idx];
            //U4 = U2 + M5
            const auto U4 = U2 + M5[idx];
            //U5 = U4 + M3
            const auto U5 = U4 + M3[idx];
            //U6 = U3 - M4
            const auto U6 = U3 - M4[idx];
            //U7 = U3 + M5
            const auto U7 = U3 + M5[idx];

            //C11 = U1
            matC[i*strideC + j] = U1;
            //C12 = U5
            matC[i*strideC + j + N / 2] = U5;
            //C21 = U6
            matC[(i + M / 2)*strideC + j] = U6;
            //C22 = U7
            matC[(i + M / 2)*strideC + j + N / 2] = U7;
        }
    }
}
```

4.2.1 节相关代码

```
void CompressiveTracker::sampleRect(Mat& _image, Rect& _objectBox, float _rInner, float _rOuter, int _maxSampleNum, vector<Rect>&
{
    int rowsz = _image.rows - _objectBox.height - 1;
    int colsz = _image.cols - _objectBox.width - 1;

    float inradsq = _rInner*_rInner;
    float outradsq = _rOuter*_rOuter;

    int dist;

    int minrow = max(0,(int)_objectBox.y-(int)_rInner);
    int maxrow = min((int)rowsz-1,(int)_objectBox.y+(int)_rInner);
    int mincol = max(0,(int)_objectBox.x-(int)_rInner);
    int maxcol = min((int)colsz-1,(int)_objectBox.x+(int)_rInner);

    int i = 0;

    float prob = ((float)(_maxSampleNum))/(maxrow-minrow+1)/(maxcol-mincol+1);

    int r;
    int c;

    _sampleBox.clear();//important
    Rect rec(0,0,0,0);

    for( r=minrow; r<=(int)maxrow; r++ )
        for( c=mincol; c<=(int)maxcol; c++ )
        {
            dist = (_objectBox.y-r)*(_objectBox.y-r) + (_objectBox.x-c)*(_objectBox.x-c);

            if( rng.uniform(0.,1.)<prob && dist < inradsq && dist >= outradsq )
            {
                rec.x = c;
                rec.y = r;
                .width = _objectBox.width;
                rec.height= _objectBox.height;

                _sampleBox.push_back(rec);

                i++;
            }
        }

        _sampleBox.resize(i);

}
```