
信号与系统 · 课程设计

快速傅里叶变换的实现和优化



姓名:

李昕

学号:

3230103034

任课教师:

胡浩基

日期:

2025 年 6 月 22 日

Table of Contents

| | |
|---|----|
| 1 绪论 | 4 |
| 1.1 离散傅里叶变换及研究意义 | 4 |
| 1.2 本文主要内容 | 5 |
| 2 FFT 基础算法实现与迭代优化 | 5 |
| 2.1 Radix-2 CT 算法 | 5 |
| 2.1.1 Cooley-Turkey 递归算法 | 5 |
| 2.2.2 补零：一般序列长度的 FFT 实现 | 6 |
| 2.2.3 蝶形网络和 Radix-2 CT 算法迭代实现 | 7 |
| 3 更高级 FFT 优化策略 | 10 |
| 3.1 高基算法：Radix-4 CT 算法 | 10 |
| 3.2 内存访问优化：Stockham 自动排序思想实现 | 14 |
| 3.3.1 算法推导及实现 | 14 |
| 3.3 运算量优化：分裂基 FFT 算法实现 | 16 |
| 3.3.1 分裂基算法简介 | 16 |
| 3.3.2 分裂基算法的数学优越性与适用环境 | 19 |
| 4 性能分析与混合算法的构建 | 20 |
| 4.1 实验环境与评测标准 | 20 |
| 硬件环境 | 20 |
| 软件环境 | 20 |
| 4.2 算法测试、性能对比 | 20 |
| 4.2.1 Radix-2 迭代算法显著快于递归算法 | 21 |
| 4.2.2 Radix-4 迭代算法优于 Radix-2 迭代算法 | 22 |
| 4.2.3 Stockham 算法在大规模数据下显著优于 Radix-2 迭代算法 | 22 |
| 4.2.4 分裂基(Split-Radix)算法与 Radix-4 算法的比较 | 22 |
| 4.3 HS-FFT：混合串行算法实现 | 23 |
| 5 应用：基于优化 FFT 的快速卷积 | 25 |
| 6 讨论与总结 | 26 |
| 6.1 关于任意长度序列处理的讨论 | 26 |
| 6.2 局限性与未来展望 | 27 |
| References | 27 |

| | |
|---|----|
| 附录 A: 部分数学证明与算法推导 | 28 |
| 附录 A.1: 系数矩阵 E 的可逆性证明 | 28 |
| 附录 A.2: 从离散傅里叶级数 a_k 到 $x[n]$ 的推导 | 28 |
| 附录 A.3: Cooley-Tukey Radix-2 递归算法推导 | 29 |
| 附录 A.4: Cooley-Tukey Radix-4 递归算法推导 | 30 |
| 附录 B: 文中算法的 Matlab 代码实现 | 32 |
| 附录 B.1: Radix-2 CT 递归算法 Matlab 代码 | 32 |
| 附录 B.2: Radix-2 CT 迭代算法 Matlab 代码 | 32 |
| 附录 B.3: Radix-4 CT 递归算法 Matlab 代码 | 33 |
| 附录 B.4: Radix-4 CT 迭代算法 Matlab 代码 | 35 |
| 附录 B.5: Stockham FFT 的 Matlab 实现 | 36 |
| 附录 B.6: 分裂基 FFT 的 Matlab 实现 | 37 |
| 附录 B.5: HS-FFT 函数定义及实现 | 38 |
| 附录 B.6: 利用优化 FFT 进行快速卷积的 Matlab 实现 | 38 |

快速傅里叶变换的实现和优化

李昕 3230103034

[摘要] 本文系统介绍了快速傅里叶变换(FFT)及其多种优化算法的原理、推导与实现，包括Radix-2、Radix-4、Stockham自动排序和分裂基FFT等。通过理论分析与Matlab实验对比，评估了不同算法在计算复杂度和实际性能上的优劣，并提出了基于Radix-4与Stockham的混合串行FFT方案。最后，展示了FFT在快速卷积等信号处理中的应用，并讨论了算法的局限性与未来优化方向。

[关键词] FFT, Radix-2, Radix-4, Stockham自动排序, 分裂基FFT, 性能分析, 混合算法

1 绪论

1.1 离散傅里叶变换及研究意义

在数字信号处理中，为了实现从离散信号到离散信号的变换，两位美国科学家和工程师库利(J. W. Cooley)和图基(J. W. Tukey)提出了离散傅里叶级数(DFS)[1]。其基本思想是，基于离散信号的傅里叶变换 $X(e^{j\omega}) = \sum_{n=-\infty}^{+\infty} x[n]e^{-jn\omega}$ ，在 $X(e^{j\omega})$ 的一个周期内任取 N 个点 $\omega_0, \omega_1, \omega_2, \dots, \omega_{N-1} \in [0, 2\pi]$ ，得到如下方程组：

$$\begin{bmatrix} X(e^{j\omega_0}) \\ X(e^{j\omega_1}) \\ X(e^{j\omega_2}) \\ \vdots \\ X(e^{j\omega_{N-1}}) \end{bmatrix} = \begin{bmatrix} 1 & e^{-j\omega_0} & e^{-j2\omega_0} & \dots & e^{-j(N-1)\omega_0} \\ 1 & e^{-j\omega_1} & e^{-j2\omega_1} & \dots & e^{-j(N-1)\omega_1} \\ 1 & e^{-j\omega_2} & e^{-j2\omega_2} & \dots & e^{-j(N-1)\omega_2} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ 1 & e^{-j\omega_{N-1}} & e^{-j2\omega_{N-1}} & \dots & e^{-j(N-1)\omega_{N-1}} \end{bmatrix} \begin{bmatrix} x[0] \\ x[1] \\ x[2] \\ \vdots \\ x[N-1] \end{bmatrix} \quad (1)$$

由于系数矩阵 \mathbf{E} 是可逆的(其证明见附录A.1)，因此可以通过矩阵求逆来唯一确定 $x[n]$ ，即 $\mathbf{x} = \mathbf{E}^{-1}\mathbf{X}$ 。若利用高斯消元法对 N 阶矩阵求逆，时间复杂度为 $O(N^3)$ ，在如今不断优化之后，最多可以优化至 $O(n^{2.37188})$ ，计算代价巨大。

此后，库利和图基提出若在频域区间 $[0, 2\pi]$ 上取 N 个均匀分布的采样点， $\omega_k = \frac{2\pi}{N} \cdot k$ ，将矩阵求逆的时间复杂度会降为 $O(N^2)$ ，此时采样点即为离散傅里叶级数 a_k ，表达式为：

$$a_k = X\left(e^{j\frac{2\pi}{N}k}\right) = \sum_{n=0}^{N-1} x[n]e^{-j(\frac{2\pi}{N})nk}, k = 0, 1, 2, \dots, N-1 \quad (2)$$

对应的，我们可以通过式2推导 $x[n]$ (具体过程见附录A.2)：

$$x[n] = \frac{1}{N} \sum_{k=0}^{N-1} a_k e^{j(\frac{2\pi}{N})nk}, k = 0, 1, 2, \dots, N-1 \quad (3)$$

由于 $o(N^2)$ 的时间复杂度仍然较高，随后 Cooley 和 Tukey 提出了快速傅里叶变换(FFT)，将计算时间复杂度降低至 $o(N \log N)$ ，极大推动了信号频域分析的广泛应用。围绕着 FFT 算法，存在一系列进步和改善，不断提升算法的运行速度。

由离散傅里叶级数的表达式，我们发现无论是从 $x[n]$ 推导 a_k ，还是从 a_k 推导 $x[n]$ ，其基本思路和计算是类似的，所以我们在本文主要讨论 $a_k = X(e^{j\frac{2\pi}{N}k}) = \sum_{n=0}^{N-1} x[n]e^{-j(\frac{2\pi}{N})nk}$ 的算法。

1.2 本文主要内容

本文从最基础的 FFT 算法入手，详细推导了经典的 Cooley-Tukey Radix-2 递归算法的原理。我们基于蝶形网络结构，将其改进为迭代形式，为后续优化奠定基础。

其次，为了追求更高的计算性能，本文探索了多种高级 FFT 优化策略。首先，实现了高基(Radix-4)算法，通过四分治策略进一步减少蝶形运算中的乘法次数；其次，针对迭代 FFT 中非连续内存访问导致的性能瓶颈，引入并实现了 Stockham 自动排序算法，以优化数据局部性，提升缓存命中率。此外，本文还研究了目前已知算术运算量最低的分裂基(Split-Radix)算法，并分析了其局限性。

接着，本文对上述多种算法进行了性能分析与比较。通过实验测试，明确了不同算法在不同数据规模下的性能优劣及交叉点。基于此分析，本文设计并实现了一种结合 **Radix-4 迭代算法** 与高基 **Stockham 算法** 优点的混合串行算法(HS-FFT)，并与 Matlab 自带的 fft 函数进行了比较。

最后，我们将 FFT 应用卷积任务中，对任意长度序列处理等问题进行了讨论，并对工作的局限性与未来研究方向进行了展望。

2 FFT 基础算法实现与迭代优化

2.1 Radix-2 CT 算法

2.1.1 Cooley-Tukey 递归算法

Cooley-Tukey Radix-2 算法[1]，通过将长度为 $N = 2^p$ 的输入序列 $x[n]$ 分解为偶数索引 $x_{e[m]}$ 和奇数索引 $x_{o[m]}$ ，从而通过分治将一个 N 点的 DFT 问题转化成两个 $\frac{N}{2}$ 的 DFT 问题。

通过数学推导（详见附录 A.3），可以证明

$$\begin{cases} a_k = x_e[k] + e^{-j\frac{2\pi k}{N}} x_o[k] \\ a_{k+\frac{N}{2}} = x_e[k] - e^{-j\frac{2\pi k}{N}} x_o[k] \end{cases} \quad k = 0, 1, \dots, N/2 - 1 \quad (4)$$

式 4 清晰地展示了算法的递归结构。其中，递归的底层为 2 点 DFT，表达式为：

$$\begin{aligned} a_0 &= x[0] + x[1] \\ a_1 &= x[0] - x[1] \end{aligned} \quad (5)$$

其伪代码如下，Matlab 实现见附录 B.1。

```

1 function RecursiveFFT(x)
2 N = length(X)
3 if N = 2 then
4     Let Y[0] be x[0] + x[1]
5     Let Y[1] be x[0] - x[1]
6     return Y
7 else
8     x_even ← all even-indexed elements of x //偶序列
9     x_odd ← all odd-indexed elements of x //奇序列
10    Y_even ← the result of RecursiveFFT(x_even)
11    Y_odd ← the result of RecursiveFFT(x_odd)
12    Y ← a new sequence of length N
13    //下面通过蝶形运算将 Y_odd[k]和 Y_even[k]合并到 Y
14    for k from 0 to N/2 - 1
15        w =  $e^{-j\frac{2\pi k}{N}}$  //旋转因子
16        t = w · Y_odd[k]
17        Y[k] = Y_even[k] + t
18        Y[k + N/2] = Y_even[k] - t
19    end for
20    return Y
21 end if
22 end function

```

图 1 递归实现的 Radix-2 FFT

通过递归分解，运行时间 $T(N)$ 满足关系式 $T(N) = 2T(\frac{N}{2}) + O(N)$ ，由递归树高度 $\log_2 N$ ，可递推得到总时间复杂度 $O(N \log N)$ ，优于直接计算 DFT 所需的 $O(N^2)$ 。

2.2.2 补零：一般序列长度的 FFT 实现

实际信号处理中，序列长度 N 往往 $\neq 2^p$ ，我们会对其进行时域补零，即

$$x'[n] = \begin{cases} x[n], & 0 \leq n \leq N-1 \\ 0, & N \leq n \leq N'-1 \end{cases} \quad (6)$$

可以证明，时域补零操作不改变原序列的离散时间傅里叶变换。计算新序列 $x'[n]$ 的DTFT：

$$\begin{aligned}
 X'(e^{j\omega}) &= \sum_{n=0}^{N'-1} x'[n]e^{-j\omega n} \\
 &= \sum_{n=0}^{N-1} x'[n]e^{-j\omega n} + \sum_{n=N}^{N'-1} x'[n]e^{-j\omega n} \\
 &= \sum_{n=0}^{N-1} x[n]e^{-j\omega n} + \sum_{n=N}^{N'-1} (0) \cdot e^{-j\omega n} \\
 &= X(e^{j\omega})
 \end{aligned} \tag{7}$$

由此，在实际处理中，我们首先检验序列长度，进行补零。其伪代码如下，Matlab实现见附录 B.1。实际使用中，补零操作在时域进行，其效果等同于对频谱的插值，将序列长度补至 2 的整数次幂的同时，可以提高频谱的表观分辨率。

```

1  Function PaddedFFT(X)
2      N = length(X)
3      L = 2 $\lceil \log_2 N \rceil$ 
4      if N = L then
5          X_padded  $\leftarrow$  X
6      else
7          X_padded  $\leftarrow$  [X, (L-N) zeros]
8      end if
9      Y = RecursiveFFT(X_padded)
10     return Y
11 end Function

```

图 2 补零 FFT

2.2.3 蝶形网络和 Radix-2 CT 算法迭代实现

在 Radix-2 CT 算法的实现中，我们可以通过迭代来代替递归进行优化。由于一个长度为 N 的 FFT 会产生大约 $2N - 1$ 次函数调用，迭代节省了这些过程。同时，考虑到迭代版的循环结构更容易被编译器和程序员优化，及在缓存上减少了 CPU 等待内存时间，我们尝试使用迭代进行优化。

Radix-2 CT 需要使用位反转序列对信号进行重排。逐层分析之前的递归过程，我们可以绘制出 图 3 所示的信号流图。

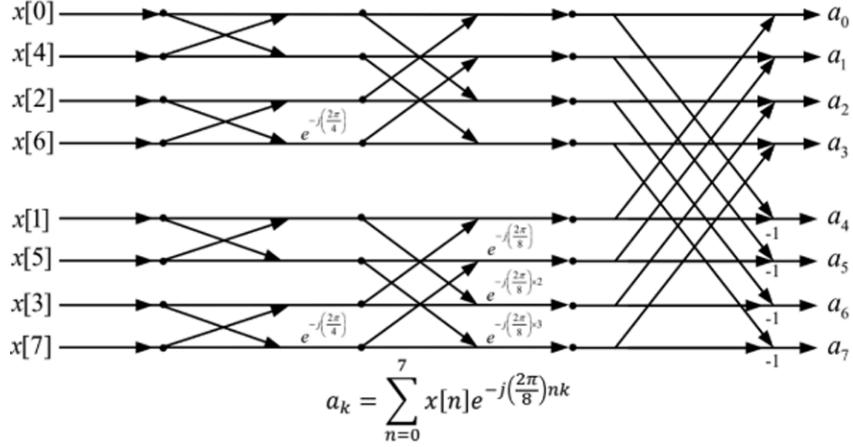


图 3 8 点 FFT 的完整信号流图

由图易知, 以 $n = 8$ 为例, 要想计算 8 点 FFT, 我们首先计算 $(x[0], x[4])$ ($x[2], x[6]$)... 的两点 FFT, 再计算 $(x[0], x[2], x[4], x[6])(x[1], x[3], x[5], x[7])$ 的 4 点 FFT。将各下标化为二进制, 如表 1。

| 原始下标 | 二进制 | 位反转后二进制 | 位反转后十进制 | 最终顺序 |
|------|-----|---------|---------|--------|
| 0 | 000 | 000 | 0 | $x[0]$ |
| 1 | 001 | 100 | 4 | $x[4]$ |
| 2 | 010 | 010 | 2 | $x[2]$ |
| 3 | 011 | 110 | 6 | $x[6]$ |
| 4 | 100 | 001 | 1 | $x[1]$ |
| 5 | 101 | 101 | 5 | $x[5]$ |
| 6 | 110 | 011 | 3 | $x[3]$ |
| 7 | 111 | 111 | 7 | $x[7]$ |

表 1 位反转序列表

观察表中规律, 我们发现原始下标对应的二进制进行位反转后, 化成十进制, 即得到最终顺序。基于这个规律, 我们能够迭代地求解 FFT。其伪代码如下, Matlab 实现见附录 B.2。

```
1  Function IterativeFFT(X)
2      N = length(X)
3      L =  $2^{\lceil \log_2 N_{in} \rceil}$ 
4      P =  $\log_2 L$ 
5      X_padded = [X, (L-N) zeros]
6      Y = a new sequence of length L
7      for i = 0 to L-1 do
8          reversed_i = BitReverse(i, P) //位反转算法
9          Y[reversed_i] = X_padded[i]
10     end for
11     for m = 2, 4, 8, ..., L do
12         h = m / 2
13         dw =  $e^{-j2\frac{\pi}{m}}$  // 当前阶段的基础旋转因子
14         for k = 0 to L-1 in steps of m do // 遍历每一组蝶形
15             w = 1
16             for j = 0 to h-1 do // 对组内元素进行蝶形运算
17                  $i_1 = k + j$ 
18                  $i_2 = k + j + h$ 
19                 temp = Y[i1]
20                 term = w * Y[i2]
21                 Y[i1] = temp + term
22                 Y[i2] = temp - term
23                 w = w * dw // 更新组内的旋转因子
24             end for
25         end for
26     end for
27     return Y
28 end Function
```

图 4 迭代 Radix-2 CT FFT 算法

其中，位反转函数实现如下：

```

1  Function BitReverse(n, p)
2      r = 0
3      for i = 1 to p do
4          lsb = n mod 2
5          r = r * 2 + lsb
6          n = floor(n / 2)
7      end for
8      return r
9  end Function

```

图 5 位反转函数

在实际实现中，由于 Radix-2 的每一级蝶形运算都需要执行一次复数乘法，在现代计算机中较为耗费时间。由此，我们希望能够通过提升算法的 Radix，在一次运算中尽量多的计算数据，从而减少运行时间。因此，产生了更高阶的 CT 算法。

3 更高级 FFT 优化策略

3.1 高基算法：Radix-4 CT 算法

在 Radix-2 算法后，相继有 Radix-4, Radix-8 等高阶 FFT 算法被提出[2]。由于基数越大，相应的算法也会更加复杂，从 Radix-2 到 Radix-4 的提升最为显著，因此，Radix-4 通常被认为是性能和实现复杂度之间的一个绝佳平衡点，本文仅讨论该算法实现及优越性。

Radix-4 算法，通过将长度为 $N = 4^p$ 的输入序列 $x[n]$ 分解为四个子序列，从而通过分治将一个 N 点的 DFT 问题转化成四个 $\frac{N}{4}$ 点的 DFT 问题。

记 $W_N^{nk} = e^{-j\frac{2\pi}{N}nk}$ 是旋转因子， b_k, c_k, d_k, e_k 分别是四个子序列的 $\frac{N}{4}$ 点 DFT。和 Radix-2 类似的，通过数学推导（详见附录 A.4），可以证明对于 $k = 0, 1, \dots, \frac{N}{4} - 1$ ，有如下递推关系：

$$\begin{cases} a_k = b_k + W_N^k c_k + W_N^{2k} d_k + W_N^{3k} e_k \\ a_{k+\frac{N}{4}} = b_k - jW_N^k c_k - W_N^{2k} d_k + jW_N^{3k} e_k \\ a_{k+\frac{2N}{4}} = b_k - W_N^k c_k + W_N^{2k} d_k - W_N^{3k} e_k \\ a_{k+\frac{3N}{4}} = b_k + jW_N^k c_k - W_N^{2k} d_k - jW_N^{3k} e_k \end{cases} \quad (8)$$

递归的底层为 4 点 DFT，表达式为：

$$\begin{aligned}
a_1 &= X_1 + X_2 + X_3 + X_4 \\
a_2 &= X_1 - jX_2 - X_3 + jX_4 \\
a_3 &= X_1 - X_2 + X_3 - X_4 \\
a_4 &= X_1 + jX_2 - X_3 - jX_4
\end{aligned} \tag{9}$$

对于 Radix-4 的 FFT，我们也可以画出和 2 点类似的信号流图 图 6：

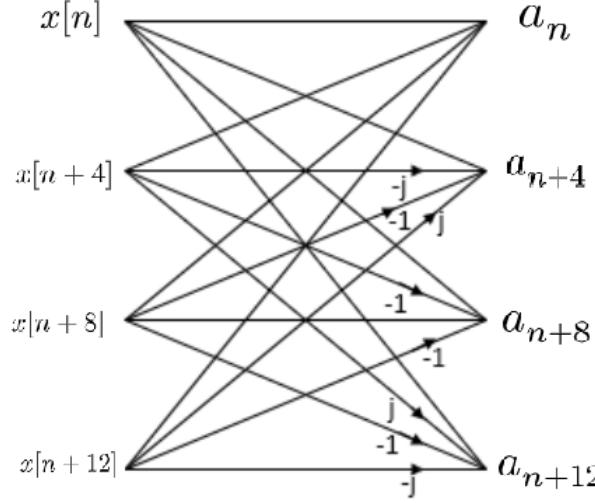


图 6 Radix-4 FFT 信号流图，以 $n = 16$ 为例，来自 [博客](#)

为了减少冗余计算，我们优先计算 $\frac{N}{4}$ DFT 结果和旋转因子的成绩，作为中间变量：

$$T_1 = W_N^k c_k, \quad T_2 = W_N^{2k} d_k, \quad T_3 = W_N^{3k} e_k \tag{10}$$

由此，我们可以将递推式简化为：

$$\begin{cases} a_k = b_k + T_1 + T_2 + T_3 \\ a_{k+\frac{N}{4}} = b_k - jT_1 - T_2 + jT_3 \\ a_{k+\frac{2N}{4}} = b_k - T_1 + T_2 - T_3 \\ a_{k+\frac{3N}{4}} = b_k + jT_1 - T_2 - jT_3 \end{cases} \tag{11}$$

分析复杂度，对于任意 k ，计算 T_1, T_2, T_3 只需要 3 次复数乘法。后续合并过程只涉及复数加减法以及与 $-j, -1, j$ 的相乘。由于和 j 的乘法仅为实部和虚部的交换与变号，其计算开销远小于一次完整的复数乘法，被称为“平凡乘法”，通常在复杂度分析中被忽略。因此，通过该优化，Radix-4 的蝶形运算将原本看似需要多次乘法的过程，优化为 3 次（非平凡）复数乘法和 8 次复数加减法。此外，由于 Radix-4 在每一步中同时处理四个数据点，其数据访问模式相比 Radix-2 更为紧凑，提高了内存的访问速度。

其伪代码如下，Matlab 实现见附录 B.3。

```

1  function RecursiveRadix4FFT(x)
2      N=length(x)
3      if N = 4 then
4          Let Y[0] be x[0] + x[1] + x[2] + x[3]
5          Let Y[1] be x[0] - jx[1] - x[2] + jx[3]
6          Let Y[2] be x[0] - x[1] + x[2] - x[3]
7          Let Y[3] be x[0] + jx[1] - x[2] - jx[3]
8          return Y
9      else
10         x0  $\leftarrow$  elements of x with index  $n \bmod 4 = 0$ 
11         x1  $\leftarrow$  elements of x with index  $n \bmod 4 = 1$ 
12         x2  $\leftarrow$  elements of x with index  $n \bmod 4 = 2$ 
13         x3  $\leftarrow$  elements of x with index  $n \bmod 4 = 3$ 
14         Y0  $\leftarrow$  RecursiveRadix4FFT(x0)
15         Y1  $\leftarrow$  RecursiveRadix4FFT(x1)
16         Y2  $\leftarrow$  RecursiveRadix4FFT(x2)
17         Y3  $\leftarrow$  RecursiveRadix4FFT(x3)
18         Y  $\leftarrow$  a new sequence of length N
19         // 下面通过蝶形运算将 Y0, Y1, Y2, Y3 合并到 Y
20         for k from 0 to N/4 - 1
21             T1 =  $e^{-j\frac{2\pi k}{N}} \cdot Y1[k]$ 
22             T2 =  $e^{-j\frac{2\pi \cdot 2k}{N}} \cdot Y2[k]$ 
23             T3 =  $e^{-j\frac{2\pi \cdot 3k}{N}} \cdot Y3[k]$ 
24             Y[k] = Y0[k] + T1 + T2 + T3
25             Y[k + N/4] = Y0[k] - j dot T1 - T2 + j dot T3
26             Y[k + 2*N/4] = Y0[k] - T1 + T2 - T3
27             Y[k + 3*N/4] = Y0[k] + j dot T1 - T2 - j dot T3
28         end for
29         return Y
30     end if
31 end function

```

图 7 递归实现的 Radix-4 FFT

同样的, Radix-4 也可以通过迭代实现来减少运行时间, 优化方法和理由与 Radix-2 类似, 伪代码如下, Matlab 实现见附录 B.4。

```

1  function IterativeRadix4FFT(X)
2      N ← length(X)
3      p ← ⌈log4(N)⌉
4      L ← 4p
5      if N < L then
6          X_padded ← pad X with L-N zeros to its end
7      else
8          X_padded ← X
9      end if
10     Y ← a new sequence of length L
11     for i from 0 to L - 1
12         Y[Base4Reverse(i, p)] ← X_padded[i]
13     end for
14     m ← 4
15     while m <= L
16         q ← m/4
17         for k from 0 to L - 1 step m
18             for j from 0 to q - 1
19                 w1 ← e-j(2π/m·j) // 计算旋转因子
20                 w2 ← w1 · w1
21                 w3 ← w2 · w1
22                 x0 ← Y[k + j]
23                 x1 ← Y[k + j + q]
24                 x2 ← Y[k + j + 2*q]
25                 x3 ← Y[k + j + 3*q]
26                 t1 ← w1 · x1 // 计算中间项
27                 t2 ← w2 · x2
28                 t3 ← w3 · x3
29                 Y[k + j] ← x0 + t1 + t2 + t3 // 基 4 蝶形运算
30                 Y[k + j + q] ← x0 - jt1 - t2 + jt3
31                 Y[k + j + 2*q] ← x0 - t1 + t2 - t3
32                 Y[k + j + 3*q] ← x0 + jt1 - t2 - jt3
33         end for
34     end for
35     m ← m * 4
36   end while
37   return Y
38 end function
39 function Base4Reverse(n, p) // 基 4 位倒序
40     r ← 0
41     for i from 1 to p
42         least_digit ← n mod 4
43         r ← 4r + least_digit
44         n ← ⌊n/4⌋
45     end for
46     return r
47 end function

```

图 8 迭代实现的 Radix-4 FFT

3.2 内存访问优化: Stockham 自动排序思想实现

3.3.1 算法推导及实现

由于在原始的 CT 算法中, 需要跨越形如 $\frac{N}{2^k}$ 的下标进行数据的计算, Stockham 自动排序算法 [3] 提出通过输入、输出的缓冲来增加程序读取内存的速度。它接受自然顺序的输入, 并在 $\log_2 N$ 个计算阶段中, 利用两个缓冲区进行“乒乓”操作。在第 s (从 1 到 $\log_2 N$) 个阶段, 算法处理的蝶形运算步长为 $L_{\text{half}} = 2^{\{s-1\}}$ 。这意味着计算模式是从短步长 (第一阶段步长为 1) 向长步长 (最后阶段步长为 $\frac{N}{2}$) 演进。从位操作的视角看, 这相当于 Stockham 算法的计算顺序是从处理索引的最低有效位 (LSB) 开始, 逐级向最高有效位 (MSB) 推进。

以 Radix-2 CT 算法为基准, 以 $N = 16$, 输入 $x[6]$ 为例, 跟踪数据流的流程如下:

- $x[6] = x[(110)_2]$ 与 $x[7] = x[(111)_2]$ 蝶形运算, 存入 $y[6], y[7]$
- $y[6] = y[(110)_2]$ 与 $y[4] = y[(100)_2]$ 蝶形运算, 存入 $x[4], x[6]$
- $x[6] = x[(110)_2]$ 和 $x[2] = x[(010)_2]$ 蝶形运算, 存入 $y[2], y[6]$

容易发现, Stockham 的流程省去了位倒序中, 首先对于 $x[n]$ 的序列进行的重排, 从而减少了内存访问的时间, 减少了算法的运行时间。

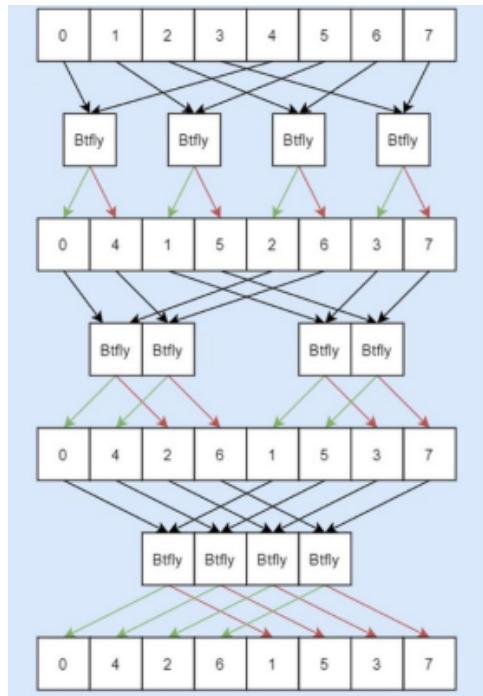


图 9 Stockham 示意图, 来自 [csdn](#)

Stockham 的算法实现较为复杂, 其伪代码如下, Matlab 实现见附录 B.5。

```

1  function IterativeStockhamFFT(x)
2      N = length(X)
3      L =  $2^{\lceil \log_2 N_{in} \rceil}$ 
4      P =  $\log_2 L$ 
5      x = X_padded = [X, (L-N) zeros]
6      let in_buf be a new sequence of length N
7      let out_buf be a new sequence of length N
8      in_buf  $\leftarrow$  x
9      for stage from 0 to log2(N) - 1
10     s  $\leftarrow$   $2^{stage}$ 
11     n  $\leftarrow$  N / s
12     m  $\leftarrow$  n / 2
13     for p from 0 to m - 1
14     wp  $\leftarrow e^{-j(2\pi \frac{p}{n})}$ 
15     for q from 0 to s - 1
16     a  $\leftarrow$  in_buf[q + sp]
17     b  $\leftarrow$  in_buf[q + s(p + m)]
18     out_buf[q + s·(2p)]  $\leftarrow$  a+b
19     out_buf[q + s·(2p + 1)]  $\leftarrow$  (a-b)wp
20     end for
21   end for
22   Swap(in_buf, out_buf)
23 end for
24 return in_buf
25 end function

```

图 10 迭代式 Stockham 自排序 FFT

3.3.2 优势及使用场景

Stockham 算法是一种典型的用空间（需要一个额外的缓冲区）和略微复杂的控制逻辑来换取时间（特别是内存访问时间）的优化策略。在处理小规模数据时，位倒序操作的开销并不明显，此时 Stockham 算法相比其他算术运算量更低的算法可能不具备性能优势。当 FFT 的输入序列长度 N 非常大，以至于数据无法完全装入 CPU 的高速缓存时，内存访问的延迟会成为主导性能的因素。此时，Stockham 算法优秀的内存访问模式能够带来显著的性能提升，远超传统需要位倒序的迭代算法。

我们将在实验中验证 Stockham 在较大数据输入时带来的性能提升。

3.3 运算量优化：分裂基 FFT 算法实现

3.3.1 分裂基算法简介

分裂基算法[4]将长度为 $N = 4^p$ 的输入序列分解为长度为 $\frac{N}{2}$, $\frac{N}{4}$ 和 $\frac{N}{4}$ 的输入序列。以 $x[16]$ 为例，在 a_k 的计算中，先将 $x[n]$ 分成偶数和奇数部分，即

$$a_k = \sum_{m=0}^7 x[2m]e^{-j\frac{2\pi(2m)k}{16}} + \sum_{m=0}^7 x[2m+1]e^{-j\frac{2\pi(2m+1)k}{16}} \quad (12)$$

先计算偶数部分：

$$\begin{aligned} a_e &= x[0] + x[2]e^{-j(\frac{2\pi}{16}) \times 2k} + x[4]e^{-j(\frac{2\pi}{16}) \times 4k} + x[6]e^{-j(\frac{2\pi}{16}) \times 6k} + x[8]e^{-j(\frac{2\pi}{16}) \times 8k} + x[10]e^{-j(\frac{2\pi}{16}) \times 10k} \\ &\quad + x[12]e^{-j(\frac{2\pi}{16}) \times 12k} + x[14]e^{-j(\frac{2\pi}{16}) \times 14k} \\ &= x[0] + x[2]e^{-j(\frac{2\pi}{8}) \times k} + x[4]e^{-j(\frac{2\pi}{8}) \times 2k} + x[6]e^{-j(\frac{2\pi}{8}) \times 3k} + x[8]e^{-j(\frac{2\pi}{8}) \times 4k} + x[10]e^{-j(\frac{2\pi}{8}) \times 5k} \\ &\quad + x[12]e^{-j(\frac{2\pi}{8}) \times 6k} + x[14]e^{-j(\frac{2\pi}{8}) \times 7k} \\ &= \sum_{m=0}^7 x[2m]e^{-j\frac{2\pi km}{8}} \end{aligned}$$

上面的偶数项视作 FFT8，进行递归求解。

剩下的奇数项，提出一个 $e^{-j\frac{2\pi k}{16}}$ ，仿照偶数项写成类似的形式，并拆解成更小的 FFT：

$$\begin{aligned} a_o &= \sum_{m=0}^7 x[2m+1]e^{-j\frac{2\pi km}{8}} e^{-j\frac{2\pi k}{16}} \\ &= e^{-j(\frac{2\pi}{16}) \times k} (x[1] + x[5]e^{-j(\frac{2\pi}{16}) \times 4k} + x[9]e^{-j(\frac{2\pi}{16}) \times 8k} + x[13]e^{-j(\frac{2\pi}{16}) \times 12k}) \text{ (FFT-4)} \\ &\quad + e^{-j(\frac{2\pi}{16}) \times 3k} (x[3] + x[7]e^{-j(\frac{2\pi}{16}) \times 4k} + x[11]e^{-j(\frac{2\pi}{16}) \times 8k} + x[15]e^{-j(\frac{2\pi}{16}) \times 12k}) \text{ (FFT-4)} \\ &= e^{-j(\frac{2\pi}{16}) \times k} \sum_{p=0}^3 x[4p+1]e^{-j\frac{2\pi pk}{4}} + e^{-j(\frac{2\pi}{16}) \times 3k} \sum_{p=0}^3 x[4p+3]e^{-j\frac{2\pi pk}{4}} \end{aligned}$$

将 FFT4 作为下一层递归的起点进行求解。

下面讨论合并，我们令

$$\begin{aligned} B[k] &= \sum_{m=0}^7 x[2m]e^{-j\frac{2\pi km}{8}} \\ C[k] &= \sum_{p=0}^3 x[4p+1]e^{-j\frac{2\pi pk}{4}} \\ D[k] &= \sum_{p=0}^3 x[4p+3]e^{-j\frac{2\pi pk}{4}} \end{aligned} \quad (13)$$

考虑周期性：

$$\therefore e^{-j\frac{2\pi(k+4)}{16}} = e^{-j\frac{2\pi k}{16}}e^{-j\frac{\pi}{2}} = -je^{-j\frac{2\pi k}{16}} \quad (14)$$

$$e^{-j\frac{2\pi\cdot3(k+4)}{16}} = e^{-j\frac{2\pi\cdot3k}{16}}e^{-j\frac{3\pi}{2}} = je^{-j\frac{2\pi k}{16}} \quad (15)$$

$$C[k+4] = C[k], D[k+4] = D[k], B[k+8] = B[k] \quad (16)$$

因此我们利用周期性进行合并，有：

$$\begin{aligned} a_k &= B[k] + e^{-j\frac{2\pi k}{16}}C[k] + e^{-j\frac{2\pi\cdot3k}{16}}D[k] \\ a_{k+4} &= B[k+4] - je^{-j\frac{2\pi k}{16}}C[k] + je^{-j\frac{2\pi\cdot3k}{16}}D[k] \\ a_{k+8} &= B[k] - e^{-j\frac{2\pi k}{16}}C[k] - e^{-j\frac{2\pi\cdot3k}{16}}D[k] \\ a_{k+12} &= B[k+4] + je^{-j\frac{2\pi k}{16}}C[k] - je^{-j\frac{2\pi\cdot3k}{16}}D[k] \end{aligned} \quad (17)$$

这个过程即是 L 形蝶形运算。整个过程的信号流图如图 11 所示：

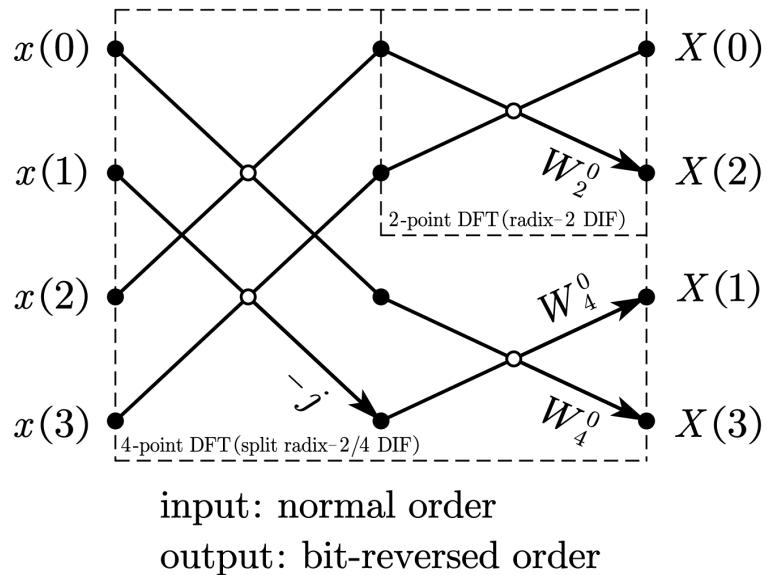


图 11 分裂基 FFT 的信号流图，来自 [Josh](#)

其伪代码如下，matlab 的代码见附录 B.5。

```

1  function fftdivpad(X)
2      N = length(X)
3      p =  $\lceil \log_2(N) \rceil$ 
4      L =  $2^p$ 
5      if N = L then
6          X_padded = X
7      else
8          X_padded = Pad X with (L-N) zeros to length L
9      end if
10     return fftDiv(X_padded)
11 end function
12 function fftDiv(X)
13     N = length(X)
14     if N = 1 then
15         return X
16     else if N = 2 then
17         Let Y[0] be X[0] + X[1]
18         Let Y[1] be X[0] - X[1]
19         return Y
20     else
21         X1  $\leftarrow$  elements of X at indices 2k (i.e., X[0], X[2], ...)
22         X2  $\leftarrow$  elements of X at indices 4k+1 (i.e., X[1], X[5], ...)
23         X3  $\leftarrow$  elements of X at indices 4k+3 (i.e., X[3], X[7], ...)
24         Y1  $\leftarrow$  fftDiv(X1) // 递归
25         Y2  $\leftarrow$  fftDiv(X2)
26         Y3  $\leftarrow$  fftDiv(X3)
27         Y  $\leftarrow$  a new sequence of length N
28         for k from 0 to N/4 - 1
29             T1 =  $e^{-j\frac{2\pi k}{N}} \cdot Y2[k]$ 
30             T2 =  $e^{-j\frac{2\pi*3k}{N}} \cdot Y3[k]$ 
31             Y[k] = Y1[k] + T1 + T2
32             Y[k + N/2] = Y1[k] - T1 - T2
33             Y[k + N/4] = Y1[k + N/4] - j  $\cdot$  (T1 - T2)
34             Y[k + 3N/4] = Y1[k + N/4] + j  $\cdot$  (T1 - T2)
35         end for
36         return Y
37     end if
38 end function

```

图 12 递归实现的分裂基 FFT

3.3.2 分裂基算法的数学优越性与适用环境

分裂基（Split-Radix）FFT 算法在数学推导上具有显著的运算量优势。与传统的 Radix-2 和 Radix-4 算法相比，分裂基算法通过灵活地将序列分解为 $\frac{N}{2}$ 和两个 $\frac{N}{4}$ 的子问题，能够最大限度地减少复数乘法和加法的总次数。

我们进一步分析分裂基 FFT 的运算量。设输入序列长度为 $N = 2^p$ ，记 $M(N)$ 为分裂基 FFT 所需的复数乘法次数， $A(N)$ 为加法次数。分裂基 FFT 的递推关系为：

$$M(N) = M\left(\frac{N}{2}\right) + 2M\left(\frac{N}{4}\right) + \left(3\frac{N}{4}\right) \quad (18)$$

其中 $M\left(\frac{N}{2}\right)$ 对应偶数部分的 FFT， $2M\left(\frac{N}{4}\right)$ 对应两组 $\frac{N}{4}$ 点 FFT， $(3\frac{N}{4})$ 为合并阶段的复数乘法。初始条件为 $M(2) = 0$ ， $M(4) = 4$ 。

利用递推公式展开，经过数学归纳法可得：

$$M(N) = N \log_2 N - 3N + 4 \quad (19)$$

同理，加法次数递推为：

$$A(N) = A\left(\frac{N}{2}\right) + 2A\left(\frac{N}{4}\right) + \frac{3N}{2} \quad (20)$$

解得：

$$A(N) = 3N \log_2 N - 3N + 4 \quad (21)$$

与 Radix-2 算法的 $N \log_2 N$ 次乘法、 $2N \log_2 N$ 次加法相比，分裂基 FFT 在同样规模下显著减少了乘法和加法次数。这一优势在 $N = 16, 32, 64$ 等小中等规模时尤为突出，能够带来更低的理论运算复杂度和更高的算术效率。

分裂基算法的核心在于充分利用 DFT 的对称性和周期性，将部分子问题递归地采用 Radix-2 分解，部分采用 Radix-4 分解，从而实现最优的算术运算量。其信号流图呈现“L 形”结构，合并阶段的旋转因子也更为高效。

然而，分裂基算法的访存模式较为不规则，难以与如 Stockham 自动排序等内存优化技术兼容。因此，在小规模或算术运算为主要瓶颈时，分裂基算法具有明显优势；而在大规模、内存带宽成为瓶颈时，其性能提升可能受限。实际应用中，常将分裂基算法用于串行高性能 FFT 的核心计算，尤其适合嵌入式、DSP 等对算术效率要求极高的场景。

从理论上讲，当输入规模 n 较小时，分裂基 FFT 凭借最优的算术运算量能够获得更高的整体性能；而当 n 较大时，内存访问延迟逐渐成为主要瓶颈，此时 Stockham 自动排序算法凭借优秀的缓存友好性和顺序访存模式，能够显著提升大规模 FFT 的实际运行效率。

由此我们采用分裂基 FFT 处理小规模数据、采用 Stockham 算法处理大规模数据，可能是一种更优的混合策略。两者之间存在一个与硬件架构和实现细节相关的分界点，需要通过实验来确定。

4 性能分析与混合算法的构建

4.1 实验环境与评测标准

硬件环境

实验所用计算机的核心硬件配置如下：

- 处理器 (CPU): 13th Gen Intel(R) Core(TM) i7-13700H
- 关键参数: 搭载高性能核心 (P-core) 与高能效核心 (E-core) 的混合架构，拥有 24 MB 的 L3 共享缓存。
- 物理内存 (RAM): 16 GB
- 操作系统: Microsoft Windows 11

软件环境

- MATLAB R2024b

4.2 算法测试、性能对比

采用上述环境对算法进行测试（代码见附件），测试结果整理至表 2，并绘制折形图 13。其中， N 为 4 的 4 次幂至 10 次幂，误差则为与 matlab 自带的 fft 进行比较的结果。

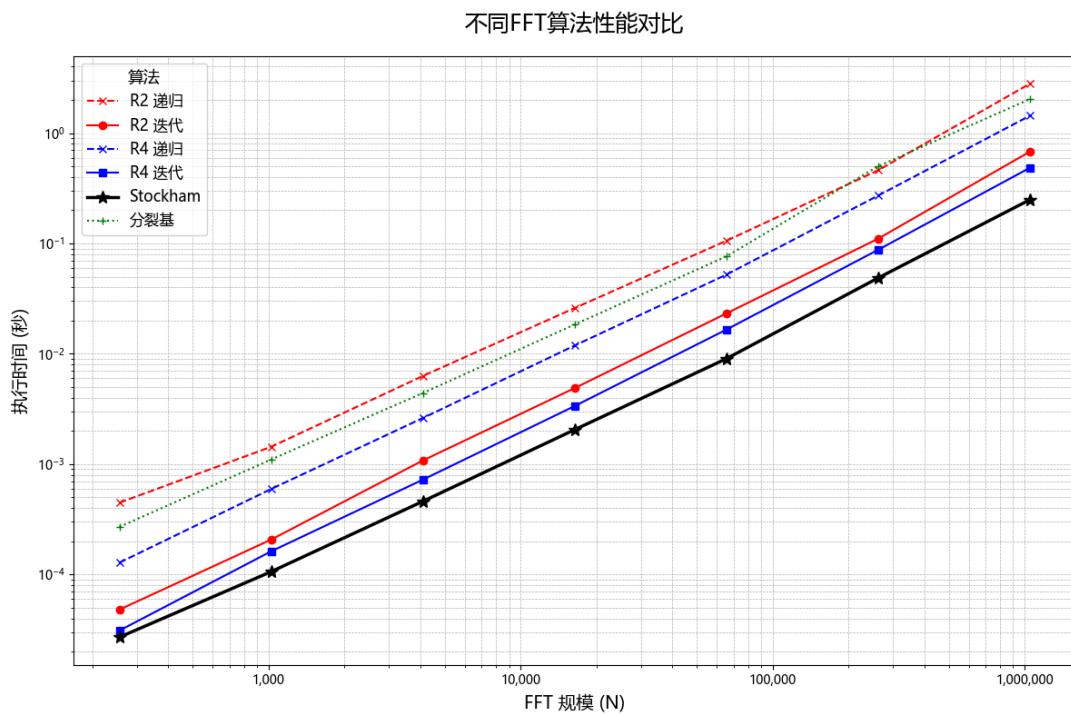


图 13 8 点 FFT 的完整信号流图

| N | 度量 | R2 递归 | R2 迭代 | R4 递归 | R4 迭代 | Stockham | 分裂基 |
|---------|--------|----------|----------|----------|----------|-----------------|----------|
| 256 | 时间 (s) | 0.000447 | 0.000048 | 0.000128 | 0.000031 | 0.000027 | 0.000270 |
| | 误差 | 3.75e-16 | 1.68e-15 | 3.52e-16 | 4.07e-16 | 3.82e-16 | 3.21e-16 |
| 1024 | 时间 (s) | 0.001436 | 0.000207 | 0.000595 | 0.000162 | 0.000106 | 0.001101 |
| | 误差 | 4.56e-16 | 9.31e-15 | 3.92e-16 | 4.61e-16 | 4.49e-16 | 3.78e-16 |
| 4096 | 时间 (s) | 0.006296 | 0.001079 | 0.002617 | 0.000723 | 0.000460 | 0.004412 |
| | 误差 | 5.26e-16 | 2.98e-14 | 4.69e-16 | 5.32e-16 | 5.24e-16 | 4.37e-16 |
| 16384 | 时间 (s) | 0.025971 | 0.004883 | 0.011875 | 0.003359 | 0.002052 | 0.018413 |
| | 误差 | 5.96e-16 | 1.35e-13 | 5.17e-16 | 5.89e-16 | 6.00e-16 | 4.90e-16 |
| 65536 | 时间 (s) | 0.105696 | 0.023212 | 0.052161 | 0.016565 | 0.009027 | 0.076353 |
| | 误差 | 6.60e-16 | 5.40e-13 | 5.69e-16 | 6.39e-16 | 6.62e-16 | 5.39e-16 |
| 262144 | 时间 (s) | 0.460368 | 0.110409 | 0.271529 | 0.087611 | 0.048674 | 0.500915 |
| | 误差 | 7.40e-16 | 2.16e-12 | 6.29e-16 | 7.03e-16 | 7.40e-16 | 5.96e-16 |
| 1048576 | 时间 (s) | 2.803964 | 0.679159 | 1.436747 | 0.484633 | 0.249002 | 2.036109 |
| | 误差 | 7.99e-16 | 9.05e-12 | 6.79e-16 | 7.51e-16 | 8.00e-16 | 6.41e-16 |

表 2 FFT 算法性能与正确性综合数据总表

根据上述数据，可以得出如下结论：

4.2.1 Radix-2 迭代算法显著快于递归算法

| N | R2-Recursive 时间 (s) | R2-Iterative 时间 (s) | 加速比 (递归/迭代) |
|---------|---------------------|---------------------|---------------|
| 256 | 0.000447 | 0.000048 | 9.31 x |
| 1024 | 0.001436 | 0.000207 | 6.94 x |
| 4096 | 0.006296 | 0.001079 | 5.84 x |
| 16384 | 0.025971 | 0.004883 | 5.32 x |
| 65536 | 0.105696 | 0.023212 | 4.55 x |
| 262144 | 0.460368 | 0.110409 | 4.17 x |
| 1048576 | 2.803964 | 0.679159 | 4.13 x |

表 3 数据对比表 1: Radix-2 递归 vs. 迭代

实验数据明确地表明，迭代版本的 Radix-2 FFT 比递归版本快 4 到 9 倍。这是因为递归实现会产生大量的函数调用开销（压栈、出栈、参数传递等），而迭代实现将所有计算置于循环内，避免了这些开销，执行效率更高，理论分析得到的结论得到证实。

4.2.2 Radix-4 迭代算法优于 Radix-2 迭代算法

| N | R2-Iterative 时间 (s) | R4-Iterative 时间 (s) | 加速比 (R2/R4) |
|---------|---------------------|---------------------|---------------|
| 256 | 0.000048 | 0.000031 | 1.55 x |
| 1024 | 0.000207 | 0.000162 | 1.28 x |
| 4096 | 0.001079 | 0.000723 | 1.49 x |
| 16384 | 0.004883 | 0.003359 | 1.45 x |
| 65536 | 0.023212 | 0.016565 | 1.40 x |
| 262144 | 0.110409 | 0.087611 | 1.26 x |
| 1048576 | 0.679159 | 0.484633 | 1.40 x |

表 4 数据对比表 2: Radix-2 迭代 vs. Radix-4 迭代

数据显示, 由于 Radix-4 的蝶形运算包含更少的运算量, 更少的计算级数, Radix-4 迭代算法始终比 Radix-2 迭代算法快约 **25%-55%**。

4.2.3 Stockham 算法在大规模数据下显著优于 Radix-2 迭代算法

| N | R2-Iterative 时间 (s) | Stockham 时间 (s) | 加速比 (R2/Stockham) |
|---------|---------------------|-----------------|-------------------|
| 256 | 0.000048 | 0.000027 | 1.78 x |
| 1024 | 0.000207 | 0.000106 | 1.95 x |
| 4096 | 0.001079 | 0.000460 | 2.35 x |
| 16384 | 0.004883 | 0.002052 | 2.38 x |
| 65536 | 0.023212 | 0.009027 | 2.57 x |
| 262144 | 0.110409 | 0.048674 | 2.27 x |
| 1048576 | 0.679159 | 0.249002 | 2.73 x |

表 5 数据对比表 3: Radix-2 迭代 vs. Stockham

数据显示, Stockham 算法快于 Radix-2 迭代算法, 并且其优势随着 N 的增大而愈发明显, 在 N=256 时, 它快 1.78 倍; 当 N 达到一百万时, 它快了 2.73 倍。

4.2.4 分裂基(Split-Radix)算法与 Radix-4 算法的比较

| N | R4-Iterative 时间 (s) | Split-Radix 时间 (s) | 性能对比 |
|---------|---------------------|--------------------|----------------------|
| 256 | 0.000031 | 0.000270 | R4-Iterative 快 8.7 倍 |
| 1024 | 0.000162 | 0.001101 | R4-Iterative 快 6.8 倍 |
| 4096 | 0.000723 | 0.004412 | R4-Iterative 快 6.1 倍 |
| 16384 | 0.003359 | 0.018413 | R4-Iterative 快 5.5 倍 |
| 65536 | 0.016565 | 0.076353 | R4-Iterative 快 4.6 倍 |
| 262144 | 0.087611 | 0.500915 | R4-Iterative 快 5.7 倍 |
| 1048576 | 0.484633 | 2.036109 | R4-Iterative 快 4.2 倍 |

表 6 数据对比表 4: Radix-4 迭代 vs. 分裂基

实验结果与理论预期有出入。数据显示，在本次测试中，**Radix-4** 迭代算法在所有规模上都显著优于我们实现的分裂基算法。这是因为我们实现的分裂基算法是递归的，这引入了巨大的函数调用开销，掩盖了其在算术运算量上的优势。此外，分裂基的不对称分解 ($N \rightarrow \frac{N}{2}, \frac{N}{4}, \frac{N}{4}$) 使得代码逻辑复杂，不利于自动优化。

因此，尽管分裂基算法在理论上拥有最少的算术运算次数，但一个迭代的、结构规整的 Radix-4 算法，由于其实现效率和对编译优化的友好性，在实际表现中超过了一个递归的、结构复杂的分裂基算法。因此，要发挥分裂基的威力，可能需要一个高度优化的迭代式实现。

由此，我们在下面的实现中，主要使用 Radix-4 和 Stockham 进行混合串行，并尝试找到规模的分界点。

4.3 HS-FFT：混合串行算法实现

根据上述结论，我们在 Matlab 中定义 Hybrid-Serial（混合串行算法实现），当规模 $N <$ 某个临界值时运行 Radix-4，大于其时运行 Stockham（代码见附录 B.7）。图 14 和图 15 分别展示了在大范围的 N 和小范围的 N 中搜索时的结果。由图我们发现最理想的分界点 $N = 416$ 。

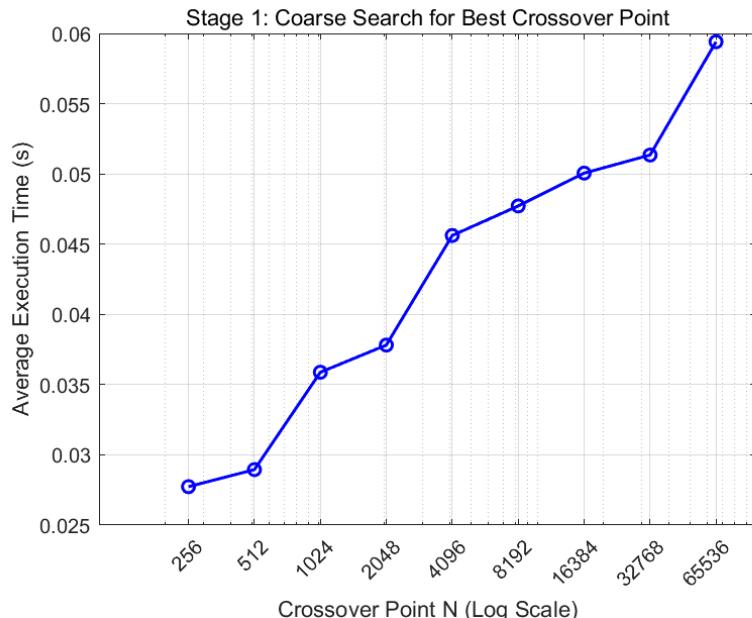


图 14 大规模 N 搜索

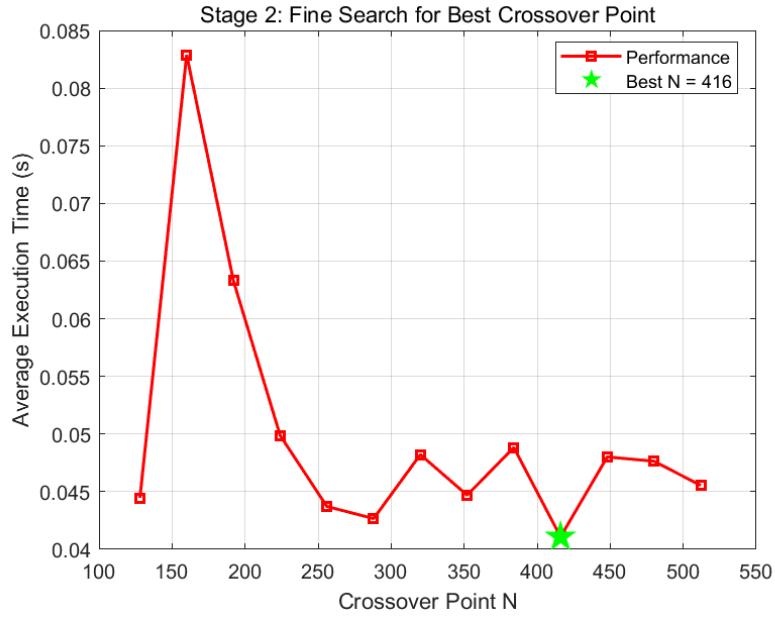


图 15 小规模 N 搜索

我们通过测试代码，将分界点设为 416，将 Hybrid-Serial FFT 与 Radix-4-Iterative FFT CT，及内置 fft 进行比较。可以发现我们的 FFT 实现了一定程度的改善，但与 matlab 的 fft 仍有较大差距。

分析原因：我们撰写的 Radix-4-Iterative FFT 及 Stockham 仍有优化空间，此外我们没有进行并行优化。

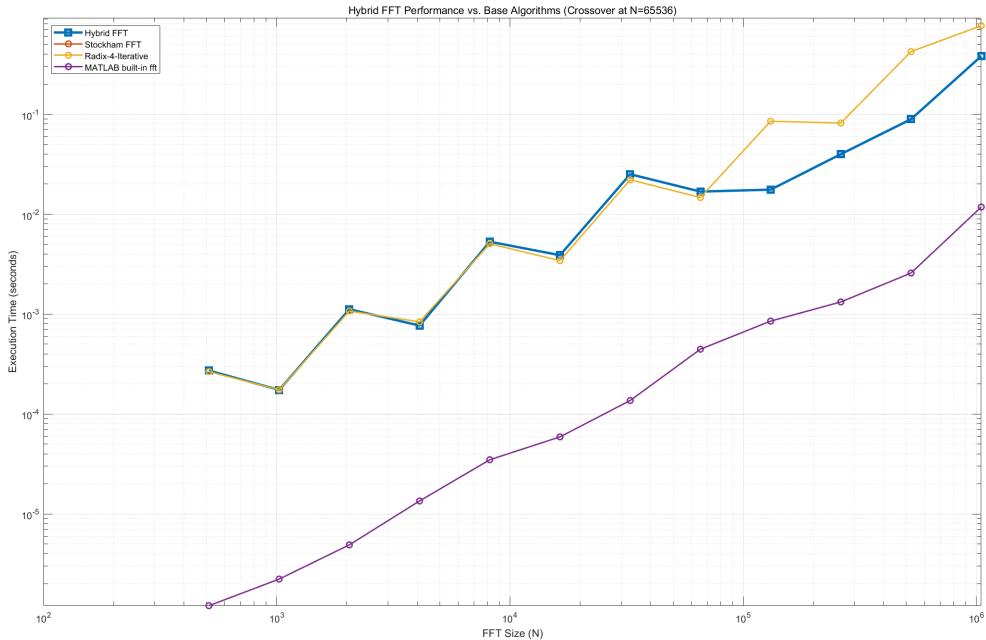


图 16 混合 HS-FFT 测试

5 应用：基于优化 FFT 的快速卷积

在实际信号处理中，我们可以基于快速傅里叶变换（FFT）进行快速卷积。两个信号在时域上的卷积运算，等价于它们各自的傅里叶变换在频域上的乘积运算。用公式表达，为

$$f[n] * g[n] = \mathcal{F}^{-1}\{\mathcal{F}\{f[n]\} \cdot F\{g[n]\}\} \quad (22)$$

基于上述定理，基于优化 FFT 实现快速卷积的伪代码如下（Matlab 代码见附录 B.8）

```
1  Function fast_conv(f, g, crossover_n)
2      Nf = length(f)
3      Ng = length(g)
4      L = Nf + Ng - 1
5      Nfft = 2ceil(log2 L)
6      fp = Pad f with zeros to length Nfft
7      gp = Pad g with zeros to length Nfft
8      Fp = hybrid_fft(fp, crossover_n)
9      Gp = hybrid_fft(gp, crossover_n)
10     Yp = Fp · Gp
11     Y_p_conj = conjugate(Yp)
12     Y_conj_fft = hybrid_fft(Y_p_conj, crossover_n)
13     yp = (1 / Nfft) conjugate(Y_conj_fft)
14     y = real_part(yp[1 to L])
15     return y
16 end Function
```

图 17 快速卷积 (fast_conv) 算法

运行代码，实验效果如图 18，实验数据如表 7。可以看到我们的 FFT 与内置函数比较，计算误差不大，但运行时间上仍有优化空间。

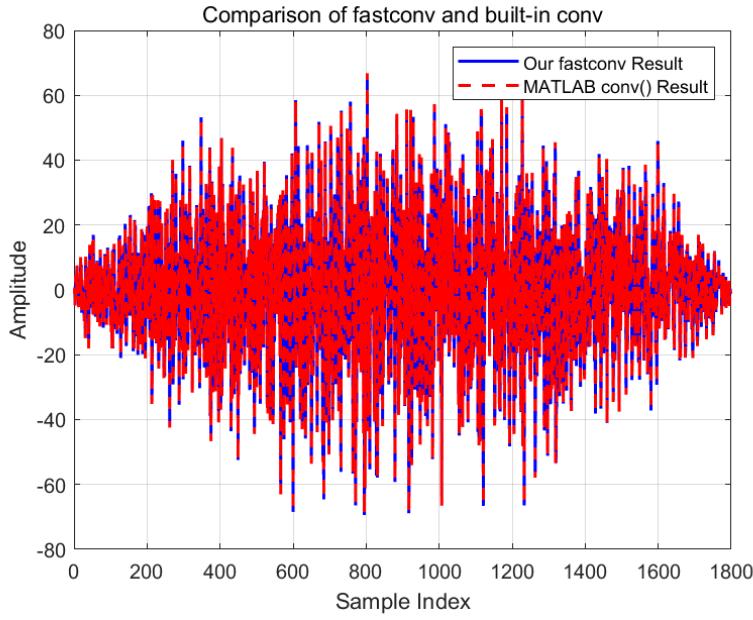


图 18 快速卷积结果

| 算法 | 运行时间 |
|---------------|---------------------|
| 自定义 fast_conv | 0.030207s |
| Matlab conv | 0.000507s |
| 误差 | 8.589013e-16 |

表 7 conv 运行比较

6 讨论与总结

6.1 关于任意长度序列处理的讨论

在实际信号处理中，我们经常需要对任意长度序列进行傅里叶变换。虽然 Cooley-Tukey 等经典 FFT 算法通常要求序列长度为 2 的幂次，但仍有许多方法可以处理非 2 的幂次长度序列[5]，或者通过优化算法来加速特定条件下的 FFT 计算。

- **Chirp-Z 变换：** CZT 是一种广义的傅里叶变换，它通过将 DFT（离散傅里叶变换）表达为卷积的形式来实现。它允许在 Z 平面上的螺旋路径上计算 Z 变换的值，从而能够计算任意长度序列的 DFT，而不仅仅局限于 2 的幂次。简单来说，CZT 通过一个巧妙的数学技巧，将原本复杂的任意长度 DFT 转化为一系列更容易计算的乘法和卷积操作。
- **混合基 FFT 算法：** 这类算法能够处理长度为多个小素数乘积的序列，例如，如果序列长度是 $N = P_1 \times P_2 \times \dots \times P_k$ ，其中 P_i 是小素数，混合基 FFT 可以将一个大长度的 DFT 分解成多个小长度的 DFT，然后将这些小 DFT 的结果组合起来。它结合了不同基数（Radix）的 FFT 算法，例如 2 和 3，以适应各种长度的序列。

除了 Cooley-Tukey 算法，还有一些针对特定情况或优化某些计算量的 DFT 快速算法，例如互素因子算法、Rader-Brenner 算法等。

然而，对于本文讨论的卷积方面的应用，由于补零是在频域上进行插值，将所有序列通过补零扩展到 2 的幂次长度仍然是首选策略。

6.2 局限性与未来展望

尽管本项目实现了多种高效的串行 FFT 算法，但仍存在一些局限性。首先，受时间限制，本实验并未针对分裂基进行迭代实现，导致分裂基的实验结果和理论值相差较大。其次，由于实验误差较大，混合算法的实现效果也不好。

此外，本实验尚未针对 SIMD 指令集（如 AVX、SSE）进行底层优化，也未在 GPU 等异构平台上实现并行 FFT。此外，OpenMPI 等分布式并行方案尚未涉及。未来工作可进一步深入 SIMD 优化、GPU 加速和分布式并行，实现更大规模和更高性能的 FFT 计算。

对于任意长度的信号序列，本文并未探索 Chirp-Z 变换、混合基等较为常用的算法，在某些情况下，补零可能会导致计算量的巨大提升，导致运行时间变长。也可以在判断序列长度上进行改善。

References

- [1] J. W. Cooley and J. W. Tukey, “An algorithm for the machine calculation of complex Fourier series,” *Math. Comput.*, vol. 19, no. 90, pp. 297–301, 1965.
- [2] W. M. Gentleman and G. Sande, “Fast Fourier transforms—for fun and profit,” in AFIPS ‘66 (Fall) Joint Computer Conference, Nov. 1966, pp. 563–578.
- [3] T. G. Stockham, Jr., “High-speed convolution and correlation,” in *Digital Processing of Signals*, B. Gold and C. Rader, Eds. New York: McGraw-Hill, 1969, ch. 7.
- [4] P. Duhamel and H. Hollmann, “Split-radix FFT algorithm,” *Electron. Lett.*, vol. 20, no. 1, pp. 14–16, Jan. 1984.
- [5] “Fast Fourier transform,” in Wikipedia, The Free Encyclopedia, Nov. 20, 2023. [Online]. Available: https://en.wikipedia.org/wiki/Fast_Fourier_transform.

附录 A：部分数学证明与算法推导

附录 A.1：系数矩阵 E 的可逆性证明

我们定义系数矩阵

$$V = \begin{bmatrix} 1 & e^{-j\omega_0} & e^{-j2\omega_0} & \dots & e^{-j(N-1)\omega_0} \\ 1 & e^{-j\omega_1} & e^{-j2\omega_1} & \dots & e^{-j(N-1)\omega_1} \\ 1 & e^{-j\omega_2} & e^{-j2\omega_2} & \dots & e^{-j(N-1)\omega_2} \\ \vdots & \vdots & \vdots & \dots & \vdots \\ 1 & e^{-j\omega_{N-1}} & e^{-j2\omega_{N-1}} & \dots & e^{-j(N-1)\omega_{N-1}} \end{bmatrix} \quad (23)$$

由线性代数的知识可知：矩阵 V 可逆的充要条件为其行列式 $|V| \neq 0$ 。为了验证 V 是否可逆，考察其行列式 $|V|$ 。设

$$v_0 = e^{-j\omega_0}, v_1 = e^{-j\omega_1}, v_2 = e^{-j\omega_2}, \dots, v_{N-1} = e^{-j\omega_{N-1}} \quad (24)$$

由范德蒙行列式的计算公式，得：

$$|V| = \begin{vmatrix} 1 & v_0 & v_0^2 & \dots & v_0^{N-1} \\ 1 & v_1 & v_1^2 & \dots & v_1^{N-1} \\ 1 & v_2 & v_2^2 & \dots & v_2^{N-1} \\ \vdots & \vdots & \vdots & \dots & \vdots \\ 1 & v_{N-1} & v_{N-1}^2 & \dots & v_{N-1}^{N-1} \end{vmatrix} = \prod_{0 \leq i < j \leq N-1} (v_j - v_i) \quad (25)$$

若 $\omega_0, \omega_1, \omega_2, \dots, \omega_{N-1}$ 取 $[0, 2\pi)$ 上两两不等的值，则有

$$(v_i = e^{-j\omega_i}) \neq (v_j = e^{-j\omega_j}), \quad 0 \leq i < j \leq N-1 \quad (26)$$

$\Rightarrow |V| \neq 0$ 。

$\therefore V$ 可逆。

附录 A.2：从离散傅里叶级数 a_k 到 $x[n]$ 的推导

我们首先证明 $\left\{e^{j\frac{2\pi}{N}nk}\right\}_{k=0 \sim (N-1)}$ 在 $\langle x[n], y[n] \rangle = \sum_{n=0}^{N-1} x[n]\overline{y[n]}$ 运算下是一组正交基：

$$\begin{aligned} \langle e^{j\frac{2\pi}{N}nk}, e^{j\frac{2\pi}{N}nl} \rangle &= \sum_{n=0}^{N-1} e^{j\frac{2\pi}{N}nk} \overline{e^{j\frac{2\pi}{N}nl}} = \sum_{n=0}^{N-1} e^{j\frac{2\pi}{N}nk} e^{-j\frac{2\pi}{N}nl} \\ &= \sum_{n=0}^{N-1} e^{j\frac{2\pi}{N}n(k-l)} = \begin{cases} \sum_{n=0}^{N-1} 1 = N & , \quad k = l \\ \sum_{n=0}^{N-1} e^{j\frac{2\pi}{N}n(k-l)} = \frac{1-e^{j2\pi(k-l)}}{1-e^{j\frac{2\pi}{N}(k-l)}} = 0 & , \quad k \neq l \end{cases} \end{aligned} \quad (27)$$

因此我们证明了正交性，即：

$$\langle e^{j\frac{2\pi}{N}nk}, e^{j\frac{2\pi}{N}nl} \rangle = \begin{cases} 0, & k \neq l \\ N, & k = l \end{cases} \quad (28)$$

由式 2 的正交性质，我们可以将任意序列 $x[n]$ 表示为这些正交基的线性组合，即：

$$x[n] = \sum_{k=0}^{N-1} a_k e^{\frac{j(2\pi)}{N}nk} \quad (29)$$

即证明了逆变换公式。

附录 A.3: Cooley-Tukey Radix-2 递归算法推导

Cooley-Tukey Radix-2 递归算法推导围绕 $a_k = X(e^{j\frac{2\pi}{N}k}) = \sum_{n=0}^{N-1} x[n]e^{-j(\frac{2\pi}{N})nk}$ 式进行化简。

我们以 $n = 8$ 的 8 点 DFS 为例，对于 a_k ，有：

$$\begin{aligned} a_k &= \sum_{n=0}^7 x[n]e^{-j(\frac{2\pi}{8})nk} \\ &= x[0] + x[2]e^{-j(\frac{2\pi}{8})\times 2k} + x[4]e^{-j(\frac{2\pi}{8})\times 4k} + x[6]e^{-j(\frac{2\pi}{8})\times 6k} + \\ &\quad e^{-j(\frac{2\pi}{8})\times k} \cdot (x[1] + x[3]e^{-j(\frac{2\pi}{8})\times 2k} + x[5]e^{-j(\frac{2\pi}{8})\times 4k} + x[7]e^{-j(\frac{2\pi}{8})\times 6k}) \\ &= \text{FFT4}(x[0], x[2], x[4], x[6]) + e^{-j(\frac{2\pi}{8})\times k} \cdot \text{FFT4}(x[1], x[3], x[5], x[7]) \end{aligned} \quad (30)$$

因此可以先计算 $x[0], x[2], x[4], x[6]$ 、 $x[1], x[3], x[5], x[7]$ 的 4 点 FFT 变换。

我们关注合并的过程，设：

$$b_k = x[0] + x[2]e^{-j(\frac{2\pi}{8})\times 2k} + x[4]e^{-j(\frac{2\pi}{8})\times 4k} + x[6]e^{-j(\frac{2\pi}{8})\times 6k} \quad (31)$$

$$c_k = x[1] + x[3]e^{-j(\frac{2\pi}{8})\times 2k} + x[5]e^{-j(\frac{2\pi}{8})\times 4k} + x[7]e^{-j(\frac{2\pi}{8})\times 6k} \quad (32)$$

由式 30，我们知道 $a_k = b_k - e^{-j(\frac{2\pi}{8})\times k} \cdot c_k$ ，可以由复数共轭性进行进一步化简，我们逐一列举 a_k 表达式：

$$\begin{aligned} a_0 &= b_0 + c_0 \\ a_1 &= b_1 + e^{-j(\frac{2\pi}{8})} \cdot c_1 \\ a_2 &= b_2 + e^{-j(\frac{2\pi}{8})\times 2} \cdot c_2 \\ a_3 &= b_3 + e^{-j(\frac{2\pi}{8})\times 3} \cdot c_3 \end{aligned} \quad (33)$$

观察 b_k, c_k 表达式，容易发现周期性。 $k = 4$ 代入解出来的 $b_4 = b_0, c_4 = c_0, 5, 6, 7, \dots$ 同理。

因此，有：

$$\begin{aligned}
a_4 &= b_0 - c_0 \\
a_5 &= b_1 - e^{-j(\frac{2\pi}{8})} \cdot c_1 \\
a_6 &= b_2 - e^{-j(\frac{2\pi}{8}) \times 2} \cdot c_2 \\
a_7 &= b_3 - e^{-j(\frac{2\pi}{8}) \times 3} \cdot c_3
\end{aligned} \tag{34}$$

因此，在实际算法中，可以先通过 4 点 FFT 计算完 b_k 还有 c_k 之后，再同时计算 a_k 及 a_{k+4} ，同时更新 $e^{-j(\frac{2\pi}{8}) \times k}$ 这个旋转因子。

递归底部是两点的 FFT 计算，此时代入公式，则得到：

$$\begin{aligned}
a_0 &= x[0] + x[1] \\
a_1 &= x[0] - x[1]
\end{aligned} \tag{35}$$

即为递归的底层，算法推导完毕。

附录 A.4: Cooley-Tukey Radix-4 递归算法推导

Radix-4 算法推导围绕 $a_k = X(e^{j\frac{2\pi}{N}k}) = \sum_{n=0}^{N-1} x[n]e^{-j(\frac{2\pi}{N})nk}$ 式进行化简。

我们以 $N = 16$ 的 16 点 DFT 为例，对于 a_k ，我们将求和式按 n 模 4 的结果分组：

$$\begin{aligned}
a_k &= \sum_{n=0}^{15} x[n]e^{-j(\frac{2\pi}{16})nk} \\
&= \sum_{m=0}^3 x[4m]e^{-j(\frac{2\pi}{16})(4m)k} + \sum_{m=0}^3 x[4m+1]e^{-j(\frac{2\pi}{16})(4m+1)k} \\
&\quad + \sum_{m=0}^3 x[4m+2]e^{-j(\frac{2\pi}{16})(4m+2)k} + \sum_{m=0}^3 x[4m+3]e^{-j(\frac{2\pi}{16})(4m+3)k}
\end{aligned} \tag{36}$$

提取旋转因子 $W_N^k = e^{-j2\pi\frac{k}{N}}$ ：

$$\begin{aligned}
a_k &= \sum_{m=0}^3 x[4m]W_{16}^{4mk} + W_{16}^k \sum_{m=0}^3 x[4m+1]W_{16}^{4mk} + \\
&\quad W_{16}^{2k} \sum_{m=0}^3 x[4m+2]W_{16}^{4mk} + W_{16}^{3k} \sum_{m=0}^3 x[4m+3]W_{16}^{4mk}
\end{aligned} \tag{37}$$

注意到 $W_{16}^{4mk} = e^{-j(2\frac{\pi}{16})4mk} = e^{-j(2\frac{\pi}{4})mk} = W_4^{mk}$ 。令 b_k, c_k, d_k, e_k 分别是序列 $x[4m], x[4m+1], x[4m+2], x[4m+3]$ 的 4 点 DFT，即：

$$\begin{aligned}
b_k &= \sum_{m=0}^3 x[4m] W_4^{mk} \\
c_k &= \sum_{m=0}^3 x[4m+1] W_4^{mk} \\
d_k &= \sum_{m=0}^3 x[4m+2] W_4^{mk} \\
e_k &= \sum_{m=0}^3 x[4m+3] W_4^{mk}
\end{aligned} \tag{38}$$

$\therefore a_k$ 可由 4 个 4 点 DFT 的结果合并而成:

$$a_k = b_k + W_{16}^k c_k + W_{16}^{2k} d_k + W_{16}^{3k} e_k \tag{39}$$

再考虑周期性, 和 Radix-2 类似的, 4 点 DFT 变换具有周期性, 即 $b_{k+4} = b_k, c_{k+4} = c_k, d_{k+4} = d_k$ 。

利用 b_k, c_k, d_k, e_k 的 4 点周期性和旋转因子的性质 $W_{16}^{4mk} = e^{-j(2\frac{\pi}{16})4mk} = e^{-j(2\frac{\pi}{4})mk} = W_4^{mk}$, 推导 $a_{k+4}, a_{k+8}, a_{k+12}$, 有

$$\begin{aligned}
a_{k+4} &= b_{k+4} + W_{16}^{k+4} c_{k+4} + W_{16}^{2(k+4)} d_{k+4} + W_{16}^{3(k+4)} e_{k+4} \\
&= b_k + W_{16}^k W_{16}^4 c_k + W_{16}^{2k} W_{16}^8 d_k + W_{16}^{3k} W_{16}^{12} e_k \\
&= b_k - jW_{16}^k c_k - W_{16}^{2k} d_k + jW_{16}^{3k} e_k
\end{aligned} \tag{40}$$

$$\begin{aligned}
a_{k+8} &= b_k + W_{16}^k W_{16}^8 c_k + W_{16}^{2k} W_{16}^{16} d_k + W_{16}^{3k} W_{16}^{24} e_k \\
&= b_k - W_{16}^k c_k + W_{16}^{2k} d_k - W_{16}^{3k} e_k
\end{aligned} \tag{41}$$

$$\begin{aligned}
a_{k+12} &= b_k + W_{16}^k W_{16}^{12} c_k + W_{16}^{2k} W_{16}^{24} d_k + W_{16}^{3k} W_{16}^{36} e_k \\
&= b_k + jW_{16}^k c_k - W_{16}^{2k} d_k - jW_{16}^{3k} e_k
\end{aligned} \tag{42}$$

综上, 我们得到了 Radix-4 的蝶形运算公式。

$$\left\{
\begin{array}{l}
a_k = b_k + W_N^k c_k + W_N^{2k} d_k + W_N^{3k} e_k \\
a_{k+\frac{N}{4}} = b_k - jW_N^k c_k - W_N^{2k} d_k + jW_N^{3k} e_k \\
a_{k+\frac{2N}{4}} = b_k - W_N^k c_k + W_N^{2k} d_k - W_N^{3k} e_k \\
a_{k+\frac{3N}{4}} = b_k + jW_N^k c_k - W_N^{2k} d_k - jW_N^{3k} e_k
\end{array}
\right. \tag{43}$$

递归的底部是 4 点 DFT, 其表达式为:

$$\begin{aligned}
 Y_1 &= X_1 + X_2 + X_3 + X_4 \\
 Y_2 &= X_1 - jX_2 - X_3 + jX_4 \\
 Y_3 &= X_1 - X_2 + X_3 - X_4 \\
 Y_4 &= X_1 + jX_2 - X_3 - jX_4
 \end{aligned} \tag{44}$$

算法推导完毕。

附录 B: 文中算法的 Matlab 代码实现

附录 B.1: Radix-2 CT 递归算法 Matlab 代码

该代码主程序部分实现了对 \boxed{x} 的补零，即 $\boxed{x_{\text{padded}}}$ 。后面即递归实现。

```

function Y = fftNewPadded(X)
N=length(X);
p=nextpow2(N);
L=2^p;
if N==L
    X_padded=X;
else
    X_padded=[X,zeros(1,L-N)]; %在X后补L-N个零
end
Y=zeros(1,L);
Y=fftNewV1(X_padded);

function Y = fftNewV1(X)
if length(X) == 2
    Y = zeros(1,2);
    Y(1) = X(1)+X(2);
    Y(2) = X(1)-X(2);
else
    N = 2^floor(log2(length(X)));
    X = X(1:N);%将X的长度变为2^n
    X1 = X([1:2:N]);
    X2 = X([2:2:N]);%假设N=8,那么第一句取了x[1,3,5,7],第二句取了x[2,4,6,8].
    Y1 = fftNewV1(X1);
    Y2 = fftNewV1(X2);%递归。
    %合并两个四点FFT
    Y = zeros(1,N);
    for k = 2:N/2
        Y2(k) = Y2(k)*exp(-1j*2*pi*(k-1)/N);
    end;
    for k = 1:N/2
        Y(k) = Y1(k) + Y2(k);
        Y(k+N/2) = Y1(k) - Y2(k);
    end;
end;

```

附录 B.2: Radix-2 CT 迭代算法 Matlab 代码

```

function Y = fft_iterative(X)
%和之前一样先补零

```

```

N=length(X);
p=nextpow2(N);
L=2^p;
if N==L
    X_padded=X;
else
    X_padded=[X,zeros(1,L-N)]; %在X后补L-N个零
end

X_reversed=zeros(1,L);
for i=1:L
    X_reversed(reverse(i-1,p)+1)=X_padded(i);
end
Y=X_reversed;
%m代表FFT的规模，逐渐变大
m=2;
while m<=L
    h=m/2;
    %旋转因子
    dw=exp(-1j*2*pi/m);
    %对整个序列以m个为一组进行蝶形计算
    for k=1:m:L
        w=1;
        for j=0:h-1
            i1=k+j;
            i2=k+j+h;
            %一个小组里的蝶形计算
            temp = Y(i1);
            term = w * Y(i2);
            Y(i1) = temp + term;
            Y(i2) = temp - term;
            w=w*dw; %更新旋转因子
        end
    end
    m=m*2;
end
end

```

```

%实现位反转
function r = reverse(n,p)
r=0;
for i=1:p
    l=mod(n,2);
    r=r*2+l;
    n=floor(n/2);
end
end

```

附录 B.3: Radix-4 CT 递归算法 Matlab 代码

由于是 Radix-4 算法，我们在迭代时将长度补为 4^p 。

```

function Y = fft_radix4(X)
N=length(X);
%先补零
N=length(X);

```

```

p=ceil(log2(N)/2);
L=4^p;
if N==L
    X_padded=X;
else
    X_padded=[X,zeros(1,L-N)]; %在X后补L-N个零
end
Y=zeros(1,L);
Y = fft_radix4rec(X_padded)
end

function Y = fft_radix4rec(X)
N = length(X);

% 递归终止条件: 4点FFT
if N == 4
    Y = zeros(1, 4);
    Y(1) = X(1) + X(2) + X(3) + X(4);
    Y(2) = X(1) - 1j*X(2) - X(3) + 1j*X(4);
    Y(3) = X(1) - X(2) + X(3) - X(4);
    Y(4) = X(1) + 1j*X(2) - X(3) - 1j*X(4);
    return;
end

% 1. 分解成4个子序列
X0 = X(1:4:N); % X[0], X[4], X[8], ...
X1 = X(2:4:N); % X[1], X[5], X[9], ...
X2 = X(3:4:N); % X[2], X[6], X[10], ...
X3 = X(4:4:N); % X[3], X[7], X[11], ...

% 2. 四个子序列递归进行FFT运算
Y0 = fft_radix4rec(X0); % b_k
Y1 = fft_radix4rec(X1); % c_k
Y2 = fft_radix4rec(X2); % d_k
Y3 = fft_radix4rec(X3); % e_k

% 3. 蝶形合并
Y = zeros(1, N);

% 对每个k进行合并 (k从0开始, 但MATLAB索引从1开始)
for k = 0:(N/4-1)
    % 计算旋转因子
    W1 = exp(-1j * 2 * pi * k / N); % w_N^k
    W2 = exp(-1j * 2 * pi * 2 * k / N); % w_N^(2k)
    W3 = exp(-1j * 2 * pi * 3 * k / N); % w_N^(3k)

    % 索引转换: k -> k+1 (MATLAB索引)
    idx = k + 1;

    % 按照推导的合并公式计算4个输出点
    Y(idx)= Y0(idx) + W1 * Y1(idx) + W2 * Y2(idx) + W3 * Y3(idx);
    Y(idx + N/4) = Y0(idx) + W1 * (-1j) * Y1(idx) + W2 * (-1) * Y2(idx) + W3 * (1j) *
    Y3(idx);
    Y(idx + N/2) = Y0(idx) + W1 * (-1) * Y1(idx) + W2 * (1) * Y2(idx) + W3 * (-1) *
    Y3(idx);
    Y(idx + 3*N/4) = Y0(idx) + W1 * (1j) * Y1(idx) + W2 * (-1) * Y2(idx) + W3 * (-1j) *

```

```

Y3(idx);
end
end

```

附录 B.4: Radix-4 CT 迭代算法 Matlab 代码

```

function Y = fft_radix4_iterative(X)
N=length(X);
%先补零
N=length(X);
p=ceil(log2(N)/2);
L=4^p;
if N==L
    X_padded=X;
else
    X_padded=[X,zeros(1,L-N)]; %在X后补L-N个零
end
Y=zeros(1,L);

X_reversed=zeros(1,L);
for i=1:L
    X_reversed(reverse(i-1,p)+1)=X_padded(i);
end
Y=X_reversed;
%m代表FFT的规模, 逐渐变大
m=4;
while m<=L
    q=m/4;
    % 预计算当前阶段 m 所需的所有旋转因子
    W_m = zeros(1, q);
    for j = 0 : q-1
        W_m(j+1) = exp(-1j * 2 * pi * j / m);
    end

    % 对整个序列以m个为一组进行蝶形计算
    for k=1:m:L
        for j=0:q-1
            % 不再计算exp(), 而是直接从表中查找!
            w1 = W_m(j+1);
            w2 = w1*w1;
            w3 = w2*w1;

            x0=Y(k+j);
            x1=Y(k+j+q);
            x2=Y(k+j+2*q);
            x3=Y(k+j+3*q);

            T1=w1*x1;
            T2=w2*x2;
            T3=w3*x3;

            Y(k+j) = x0+T1+T2+T3;
            Y(k+j+q) = x0 - 1j*T1 - T2 + 1j*T3;
            Y(k+j+2*q) = x0 - T1 + T2 - T3;
            Y(k+j+3*q) = x0 + 1j*T1 - T2 - 1j*T3;
        end
    end
end

```

```

    end
    m=m*4;
end
end

```

```

%实现位反转
function r = reverse(n,p)
r=0;
for i=1:p
    l=mod(n,4);
    r=r*4+l;
    n=floor(n/4);
end
end

```

附录 B.5: Stockham FFT 的 Matlab 实现

本代码参考了 [fft.c](#) 进行撰写。

```

function y = stockham_fft(x)
% --- 1. 初始化和输入检查 ---
x = x(:); % 确保x是一个列向量
N = length(x);
p=nextpow2(N);
L=2^p;
if N==L
    X_padded=x;
else
    X_padded=[x,zeros(1,L-N)]; %在X后补L-N个零
end
x=X_padded;

% --- 2. 设置 "乒乓" 缓冲区 ---
buffers = zeros(N, 2, 'like', x);
buffers(:, 1) = x; % 将输入数据复制到第一个缓冲区
in_idx = 1; % 当前输入缓冲区的列索引 (1 或 2)
out_idx = 2; % 当前输出缓冲区的列索引 (2 或 1)
% --- 3. 迭代执行 FFT 阶段 ---
% 总共有 log2(N) 个阶段
for stage = 0:(p - 1)
    s = 2^stage;
    n = N / s;
    m = n / 2;
    % 对所有子问题进行蝶形运算
    for p = 0:(m - 1)
        % 计算旋转因子 W_n^p = exp(-j*2*pi*p/n)
        wp = exp(-1i * 2 * pi * p / n);
        for q = 0:(s - 1)
            idx_a = q + s*p + 1;
            idx_b = q + s*(p+m) + 1;

            a = buffers(idx_a, in_idx);
            b = buffers(idx_b, in_idx);

            % 蝶形运算 (DIF - Decimation In Frequency)
    end
end

```

```

    % out_buf[even] = a + b
    % out_buf[odd]  = (a - b) * W
    idx_out_even = q + s*(2*p) + 1;
    idx_out_odd  = q + s*(2*p+1) + 1;

    buffers(idx_out_even, out_idx) = a + b;
    buffers(idx_out_odd, out_idx) = (a - b) * wp;
end
end

% "乒乓": 交换输入和输出缓冲区的角色, 为下一阶段做准备
temp_idx = in_idx;
in_idx = out_idx;
out_idx = temp_idx;
end

% --- 4. 返回最终结果 ---
% 经过 log2(N) 个阶段后, 最终结果位于 'in_idx' 指向的缓冲区中
y = buffers(:, in_idx);
end

```

附录 B.6: 分裂基 FFT 的 Matlab 实现

```

function Y = fftdivpad(X)
N=length(X);
p=nextpow2(N);
L=2^p;
if N==L
    X_padded=X;
else
    X_padded=[X,zeros(1,L-N)]; %在X后补L-N个零
end
Y=zeros(1,L);
Y=fftDiv(X_padded);
end

function Y = fftDiv(X)
if length(X) == 2
    Y = zeros(1,2);
    Y(1) = X(1)+X(2);
    Y(2) = X(1)-X(2);
elseif length(X)==1
    Y=zeros(1,1);
    Y(1)=X(1);
else
    N=length(X);
    X1 = X([1:2:N]);%假设N=16,那么第一句取了x[1,3,5,7,9,...15],0开始就是0,2,4,...14
    X2 = X([2:4:N]);%取1,5,9,13
    X3 = X([4:4:N]);%3,7,11,15
    Y1 = fftDiv(X1);
    Y2 = fftDiv(X2);
    Y3 = fftDiv(X3);%递归。
    %下面这一段讲的就是合并
    Y = zeros(1,N);

    for k=1:N/4
        T1=Y2(k)*exp(-1j*2*pi*(k-1)/N);

```

```

T2=Y3(k)*exp(-1j*2*pi*3*(k-1)/N);
Y(k)=Y1(k)+T1+T2;
Y(k+N/2)=Y1(k)-T1-T2;
Y(k+N/4)=Y1(k+N/4)-1j*(T1-T2);
Y(k+3*N/4)=Y1(k+N/4)+1j*(T1-T2);
end
end
end

```

附录 B.5: HS-FFT 函数定义及实现

```

% --- 混合FFT函数定义 ---
% 这个函数被上面的测试循环所调用
function Y = hybrid_fft(X, crossover_n)
    % HYBRID_FFT: 根据crossover_n在Stockham和Radix-4-iterative间切换
    % 假设: N > crossover_n 使用 Stockham (适合大规模)
    %       N <= crossover_n 使用 Split-Radix (适合小规模)
    % !!! 注意: 请根据你的实际结论调整这个逻辑 !!!
    % 如果你的结论是反过来的, 请交换下面if/else里的函数调用

N = length(X);

if N > crossover_n
    % 大规模 -> 调用Stockham
    % Stockham函数期望列向量, 需要转置
    Y = stockham_fft(X.');
    Y = Y.'; % 转置回来
else
    % 小规模 -> 调用radix4
    Y = fft_radix4_iterative(X);
end
end

```

附录 B.6: 利用优化 FFT 进行快速卷积的 Matlab 实现

```

function y = fast_conv(f, g, crossover_n)
Nf = length(f);
Ng = length(g);
%计算线性卷积结果的长度
L = Nf + Ng - 1;
N_fft = 2^nextpow2(L);
fp = [f, zeros(1, N_fft - Nf)];
gp = [g, zeros(1, N_fft - Ng)];
%用hybrid_fft正向FFT
Fp = hybrid_fft(fp, crossover_n);
Gp = hybrid_fft(gp, crossover_n);
Yp = Fp .* Gp;
%傅里叶逆变换
Y_conj_fft = hybrid_fft(conj(Yp), crossover_n);
yp = (1/N_fft) * conj(Y_conj_fft);
%最终的线性卷积结果, 取实部并截取前 L 个点
y = real(yp(1:L));
end

```