成绩评定： 分

北京理工大學

本科生研究型课程专题报告

题　目： 任意点数 FFT 算法探究

课程名称：	信号处理理论与技术 II
报告形式：	☑ 个人　□小组
姓　　名：	[redacted]
学　　号：	[redacted]
学　　院：	信息与电子学院
专　　业：	电子信息工程
班　　级：	[redacted]
授课教师：	[redacted]

[redacted]

任意点数 FFT 算法探究

摘要

基 2-FFT 算法因要求序列点数必须是 2 的幂次，难以满足任意点数 DFT 的高效求解需求。本文围绕任意点数 FFT 算法展开深入探究，核心思路是采用时域抽取法将基 2-FFT 算法推广为基 p-FFT 算法，并以基 3-FFT 算法为例完成理论推导、蝶形运算单元分析，并给出递归编程实现算法。在此基础上，针对序列长度为复合数的情况，基于素因数分解提出混合基 FFT 算法，通过组合小点数 DFT 实现高效计算。针对序列长度为质数或含大质因子的场景，引入雷德算法将素数点 DFT 转化为循环相关，结合混合基 FFT 算法进一步优化计算速度。本文以单边指数序列为仿真对象，通过 MATLAB 仿真对比内置 fft 函数、DFT 直接计算与所提算法的频谱及运算时间，验证了算法的正确性，且所提算法运算速度远优于直接 DFT 计算。本文的最后总结了不同序列长度对应的最优算法策略，为信号处理、数字通信等领域的任意点数 DFT 高效求解提供了可行方案。

1. 研究背景与意义

在课堂上我们学习的是序列点数$N = 2^L$，即以 2 为基数的 FFT 算法。这种算法具有程序简单、效率高、使用方便的优点，因而得到了广泛的应用。

若序列的点数不满足$N = 2^L$，可以在原序列后面补 0，使序列点数满足$N = 2^L$，然后对其使用基 2-FFT 算法求解。由 DFT 的性质可知，有限长序列补 0 并不影响频谱的包络，只是频率采样点数增加了。但有时补 0 点数太多，造成计算量增加太多，从而使求解速度变慢。

在有些情况下，我们需要求解准确的 N 点 DFT，并且 MATLAB 中 fft 函数也能对任意点数的序列求解 FFT。在这种情况下基 2-FFT 算法不再适用，如果使用 DFT 的定义直接进行计算，计算量将非常庞大，因此需要对任意点数 N 的 FFT 快速算法进行探索。

FFT 混合基算法是 Cooley-Tukey 算法的通用形式，其核心思想是将一个长度为复合数 N 的 DFT，递归地分解为多个更小长度的 DFT 组合进行计算。它不像基-2 算法那样要求长度必须是 2 的幂，而是能灵活高效地处理任意包含较小质因子的序列，从而在通用性和效率间取得平衡。

因此，若想实现混合基算法，首先需要对基 2-FFT 算法进行推广，得到以任意点数 p 为基数的 FFT 算法，即基 p-FFT 算法。在此基础上对序列长度 N 进行素因数分解，逐层使用基 p-FFT 算法进行求解。基 p-FFT 算法仍分为时间抽取和频率抽取两种方式，两种方式计算量相同。为了方便使用递归的算法编写程序进行验证，本次报告采用时间抽取的方法

2. 基 p-FFT 算法

2.1 基 3-FFT 算法

2.1.1 基 3-FFT 算法理论推导

在推导基 p-FFT 算法时，首先对基 3-FFT 算法进行推导。已知长度为N的序列$x[n]$的 DFT 计算公式为：

$$X[k]=\sum_{n=0}^{N-1}x[n]W_N^{kn} \tag{1}$$

设序列$x[n]$的长度N满足$N=3^L$，将$x[n]$分成 3 个$N/3$的子序列：

$$x_0[n]=x[3n],\ x_1[n]=x[3n+1],\ x_2[n]=x[3n+2] \quad n=0,1,\cdots,\frac{N}{3}-1 \tag{2}$$

x$x[n]$的 DFT 可以表示为：

$$\begin{aligned}X[k]&=\sum_{n=0}^{N-1}x[n]W_N^{kn}\\&=\sum_{n=0}^{N/3-1}x[3n]W_N^{k(3n)}+\sum_{n=0}^{N/3-1}x[3n+1]W_N^{k(3n+1)}+\sum_{n=0}^{N/3-1}x[3n+2]W_N^{k(3n+2)}\\&=\sum_{n=0}^{N/3-1}x_0[n]W_{\frac{N}{3}}^{kn}+W_N^k\sum_{n=0}^{N/3-1}x_1[n]W_{\frac{N}{3}}^{kn}+W_N^{2k}\sum_{n=0}^{N/3-1}x_2[n]W_{\frac{N}{3}}^{kn}\\&=X_0[k]+W_N^kX_1[k]+W_N^{2k}X_2[k] \qquad k=0,1,\cdots,N-1\end{aligned} \tag{3}$$

其中$X_0[k]$、 $X_1[k]$和$X_2[k]$分别是$x_0[n]$、 $x_1[n]$和$x_2[n]$的$N/3$点 DFT，注意到$X_0[k]$、$X_1[k]$和$X_2[k]$均是以$N/3$为周期的，于是可得

$$\begin{aligned}X[k]&=X_0[k]+W_N^kX_1[k]+W_N^{2k}X_2[k],\\X[k+N/3]&=X_0[k]+W_N^{k+N/3}X_1[k]+W_N^{2(k+N/3)}X_2[k],\\X[k+2N/3]&=X_0[k]+W_N^{k+2N/3}X_1[k]+W_N^{2(k+2N/3)}X_2[k],k=0,1,\cdots,N/3-1\end{aligned} \tag{4}$$

注意到$W_N^{N/3}=W_3^1$，$W_N^{2N/3}=W_3^2$，将其带入上式得

$$\begin{aligned}X[k]&=X_0[k]+W_N^kX_1[k]+W_N^{2k}X_2[k],\\X[k+N/3]&=X_0[k]+W_3^1W_N^kX_1[k]+W_3^2W_N^{2k}X_2[k],\\X[k+2N/3]&=X_0[k]+W_3^2W_N^kX_1[k]+W_3^4W_N^{2k}X_2[k],k=0,1,\cdots,N/3-1\end{aligned} \tag{5}$$

$x[n]$的 N 点 DFT 可以转化为$x_0[n]$, $x_1[n]$和$x_2[n]$的$N/3$点 DFT 的加权组合。这种方式是将序列以 3 的模分成 3 个子列，即在时域上进行 3 倍抽取，这就是基-3 时间抽取 FFT 算法。

2.1.2 基 3-FFT 算法蝶形运算单元

将公式(5)写成矩阵形式，即：

$$\boldsymbol{X}=\boldsymbol{W_3D_3X'} \tag{6}$$

其中$\boldsymbol{W_3}$为 3 点 DFT 矩阵，$\boldsymbol{D_3}=\boldsymbol{diag}\left(W_N^0,W_N^k,W_N^{2k}\right)$，$\boldsymbol{X}=\begin{bmatrix}X[k]\\X[k+N/3]\\X[k+2N/3]\end{bmatrix}$，$\boldsymbol{X'}=\begin{bmatrix}X_0[k]\\X_1[k]\\X_2[k]\end{bmatrix}$，$k=0,1,\cdots,N/3-1$。

则基 3-FFT 算法蝶形运算单元定义如下图所示

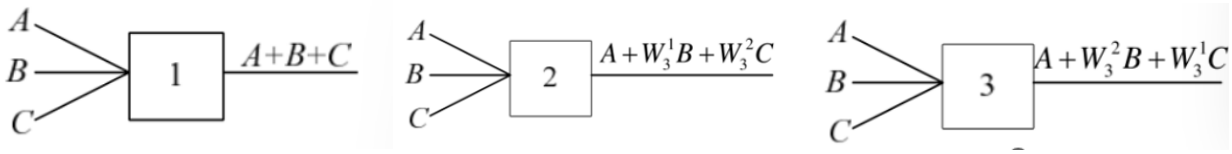

图 1：基 3-FFT 算法蝶形运算单元

上述只是进行一次基 3-FFT 分解，进一步如果当$N/3$也是偶数则可将$N/3$点 DFT 分解为$N/9$点 DFT。以此类推，直到 DFT 点数不是 3 的倍数为止。当点数$N = 3^s$时，可持续进行分解，总共需要 s 级运算。按照上述规律，27 点序列基 3-FFT 算法结构如下图所示：

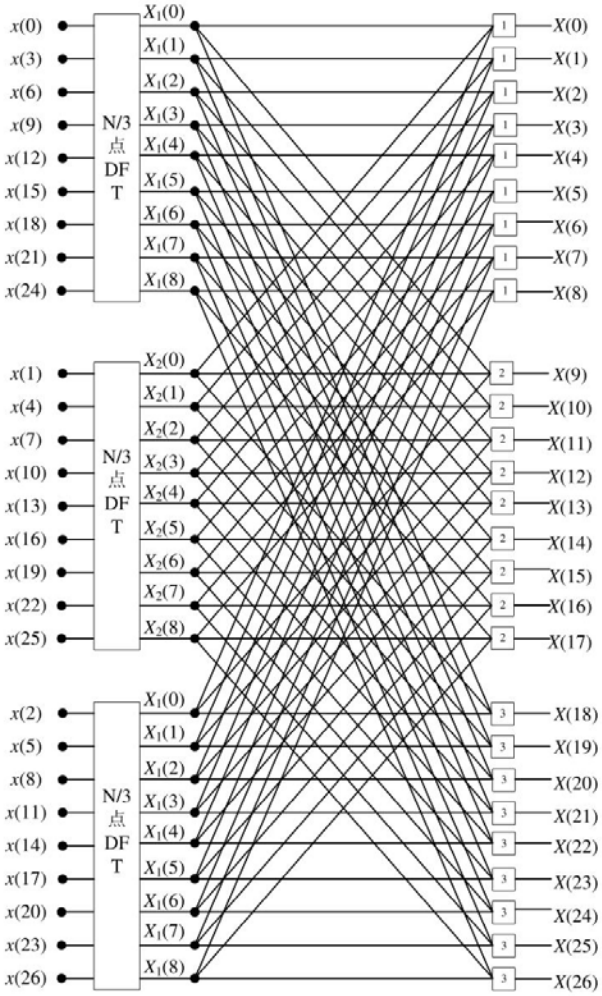

图 2：27 点序列基 3-FFT 算法结构

2.1.3 基 3-FFT 算法输入序列的顺序

与基-2 时间抽取的 FFT 算法类似，上述的基 3-FFT 算法输入端序列也不是按照自然数的顺序排列的，同样可以按照三进制“码位倒置”的方法得到。输入端序列的序号 N 可表示成三进制数$(r_{L-1}, r_{L-2}, \dots, r_1, r_0)_3$，即：

$$N = 3^{L-1}r_{L-1} + 3^{L-2}r_{L-2} + \cdots + 3r_1 + r_0 \tag{7}$$

其中$r_i = 0,1$ 或 2。然后将该三进制数的首尾颠倒为 $(r_0, r_1, \dots, r_{L-2}, r_{L-1})_3$，重新按照上式计算新的序号值，最后转换为十进制数。

在后续的编程实践中，采用上述方法会使算法设计较为复杂。根据基 3-FFT 多级蝶形运算的特点，我们可以采用递归调用的方式对 N 点 DFT 进行逐级分解，最终分解为 3 点

DFT。这种实现方法不需要知道输入序列的序号，既降低了编程难度，也可以提高程序代码的可读性，并且仍可以发挥 FFT 快速计算的性能优势。

2.2 基 p-FFT 算法

基 p-FFT 算法可以看作基 2-FFT 算法和基 3-FFT 算法的推广。将序列以 p 的模分成 p 个子列，即在时域上进行 p 倍抽取，即可得

$$x_0[n]=x[pn],x_1[n]=x[pn+1]\cdots$$
$$x_{p-2}[n]=x[pn+p-2],x_{p-1}[n]=x[pn+p-1]\qquad n=0,1,\cdots,\frac{N}{p}-1 \tag{8}$$

仿照基 3-FFT 算法推导过程，设$X_0[k]$、 $X_1[k]$ ⋯ $X_{p-1}[k]$是$x_0[n]$、 $x_1[n]$ ⋯ $x_{p-1}[n]$的N/p点 DFT，则：

$$X[k]=X_0[k]+W_N^kX_1[k]+\cdots+W_N^{(p-2)k}X_{p-2}[k]+W_N^{(p-1)k}X_{p-1}[k],$$
$$X[k+N/p]=X_0[k]+W_p^1W_N^kX_1[k]+\cdots+W_p^{(p-2)}W_N^{(p-2)k}X_{p-2}[k]+W_p^{(p-1)}W_N^{(p-1)k}X_{p-1}[k]$$
$$\vdots$$
$$X[k+(p-2)N/p]=X_0[k]+W_p^{p-2}W_N^kX_1[k]+\cdots+W_p^{(p-2)(p-2)}W_N^{(p-2)k}X_{p-2}[k]$$
$$+W_p^{(p-2)(p-1)}W_N^{(p-1)k}X_{p-1}[k]$$
$$X[k+(p-1)N/p]=X_0[k]+W_p^{p-1}W_N^kX_1[k]+\cdots+W_p^{(p-1)(p-2)}W_N^{(p-2)k}X_{p-2}[k]$$
$$+W_p^{(p-1)(p-1)}W_N^{(p-1)k}X_{p-1}[k] \tag{9}$$

由此可观察到基 3-FFT 算法是基 p-FFT 算法在 p=3 时的特例， 此时$x[n]$的 N 点 DFT 可以转化为$x_0[n]$, $x_1[n],\cdots,x_{p-1}[n]$的N/p点 DFT 的加权组合。这种方式是将序列以 p 的模分成 p 个子列，即在时域上进行 p 倍抽取，这就是基-p 时间抽取 FFT 算法。

将公式(8)写成矩阵形式，即：

$$\boldsymbol{X}=\boldsymbol{W_pD_pX'} \tag{10}$$

其中$\boldsymbol{W_p}$为 p 点 DFT 矩阵， $\boldsymbol{D_p}=\boldsymbol{diag}\left(W_N^0,W_N^k,\cdots,W_N^{(p-2)k},W_N^{(p-1)k}\right)$， $\boldsymbol{X}=\begin{bmatrix}X[k]\\X[k+N/p]\\\vdots\\X[k+(p-2)N/p]\\X[k+(p-1)N/p]\end{bmatrix}$, $\boldsymbol{X'}=\begin{bmatrix}X_0[k]\\X_1[k]\\\vdots\\X_{p-2}[k]\\X_{p-1}[k]\end{bmatrix}$, $k=0,1,\cdots,N/p-1$。

基 p-FFT 算法同样具有类似的蝶形运算单元，由于较为复杂这里不再展示。由于基 p-FFT 算法是基 3-FFT 算法的推广，因此当点数$N=p^s$时，可持续按照 p 的倍数逐级分解，总共需要 s 级运算，在具体编程中仍可采取与上述基 3-FFT 算法类似的递归算法实现。

在基 p-FFT 算法中，输入序列的序号同样可以通过 p 进制“码位倒置”的方法得到，即先将序号 N 表示为 p 进制数$(r_{L-1},r_{L-2},\dots,r_1,r_0)_p$，然后将该 p 进制数的首尾颠倒为$(r_0,r_1,\dots,r_{L-2},r_{L-1})_p$，据此计算新的序号值，最后转换为十进制数即可。

3. 混合基 FFT 算法

当点数 N 不再是某个数的幂次时，基 p-FFT 算法的分解不能持续进行。在此情况下可对 N 进行质因数分解，即$N=p_1p_2\cdots p_{L-1}p_L$，其中$p_1,p_2\cdots p_{L-1},p_L$均为质数。因此可以采取混合基算法加快运算速度。

3.1 $N=p_1p_2$形式的混合基算法

3.1.1 $N=p_1p_2$形式的混合基算法公式推导

为了简单起见，首先推导分解成两个质因数的混合基算法。当$N=p_1p_2$时，按照基 p-FFT 算法的思路，将序列以p_2的模分成p_2个子列，即

$$x_0[n]=x[p_2n],x_1[n]=x[p_2n+1],x_{p_2-2}[n]=x[p_2n+p_2-2]\cdots$$
$$x_{p_2-1}[n]=x[p_2n+p_2-1]\qquad n=0,1,\cdots,p_1-1 \tag{11}$$

设$X_0[k]$、 $X_1[k]$ ⋯ $X_{p_2-1}[k]$是$x_0[n]$、 $x_1[n]$ ⋯ $x_{p_2-1}[n]$的p_1点 DFT，重复基 p-FFT 算法推导方法，则有

$$\boldsymbol{X}=\boldsymbol{W}_{p_2}\boldsymbol{D}_{p_2}\boldsymbol{X}' \tag{12}$$

其中$\boldsymbol{W}_{p_2}$为p_2点 DFT 矩阵， $\boldsymbol{D}_{p_2}=\boldsymbol{diag}\left(W_N^0,W_N^k,\cdots,W_N^{(p_2-2)k},W_N^{(p_2-1)k}\right)$， $\boldsymbol{X}=\begin{bmatrix}X[k]\\X[k+p_1]\\\vdots\\X[k+(p_2-2)p_1]\\X[k+(p_2-1)p_1]\end{bmatrix}$， $\boldsymbol{X}'=\begin{bmatrix}X_0[k]\\X_1[k]\\\vdots\\X_{p_2-2}[k]\\X_{p_2-1}[k]\end{bmatrix}$， $k=0,1,\cdots,p_1-1$。

由上述推导可知，$x[n]$的 N 点 DFT 可以转化为$x_0[n]$, $x_1[n],\cdots,x_{p_2-1}[n]$ p_2个子列的p_1点 DFT 的加权组合，由于分解的p_1和p_2并不相同，所以这种算法叫做混合基算法。

3.1.2 $N=p_1p_2$形式的混合基算法运算量分析

由公式(11)可知，混合基 FFT 算法可看作由p_2个子列 DFT 的加权组合和求解这两部分组成。一次公式(11)所示的 DFT 加权组合可看作先乘 N 个旋转因子再进行p_1个p_2点 DFT 操作。在求p_2个子列的p_1点 DFT 时，复数乘法的运算量为$p_2{p_1}^2$，复数乘法的运算量为$p_2p_1(p_1-1)$，因此总计算量为：

$$\text{复数乘法}=N(p_1+p_2+1)，\text{复数加法}=N(p_1+p_2-2) \tag{13}$$

直接计算 DFT 的计算量：复数乘法$=N^2$，复数*加*法$=N(N-1)$，因此混合基运算可节省的计算量倍数为：

$$\text{复数乘法}=N/(p_1+p_2+1)，\text{复数加法}=(N-1)/(p_1+p_2-2) \tag{14}$$

一般情况下$p_1+p_2+1\ll N$，因此使用混合基算法可明显降低计算量

注意到当p_1或p_2等于 2 时上式不再成立，这是由于基 2-FFT 蝶形运算单元运算结构比较简单，每次只需进行 1 次乘法和 2 次加法即可，因此相较于上式运算量还可以进一步降低。进一步推导可知，相较于所有基 p-FFT 算法，基 2-FFT 算法的运算量最低，计算速度最快。

3.2 $N=p_1p_2\cdots p_L$形式的混合基算法

当$N=p_1p_2\cdots p_L$时，可按照$N=p_1p_2$的方法，先将序列以p_L的模分成p_L个子列，则$x[n]$的 N 点 DFT 可以转化为$x_0[n]$, $x_1[n],\cdots,x_{p_L-1}[n]$这$p_L$个子列的$p_1p_2\cdots p_{L-1}$点 DFT 的加权组合，每个子列的$p_1p_2\cdots p_{L-1}$点 DFT 又可按照上述同样的方式继续分解，持续进行下去，直到最后会分解为p_1点 DFT 的加权组合。

此时输入序列的序号仍然满足类似于基 2-FFT 算法的多基多进制“码位倒置”的关系，但是在具体编程过程中我们可以借鉴之前基 3-FFT 的思路，采用递归调用的方式对 N 点 DFT 根据分解的素因子进行混合基分解，最终分解为p_1点 DFT。

与$N = p_1 p_2$的情况类似，$N = p_1 p_2 \cdots p_L$形式的混合基算法的总计算量为：

$$\text{复数乘法} = N\left[\left(\sum_{i=1}^{L} r_i\right) + L - 1\right]，\text{节省的计算量倍数} = \frac{N}{\sum_{i=1}^{L} r_i + L - 1} \tag{15}$$

同样地，上式也必须在$p_1, p_2, \cdots, p_L > 2$时成立。

3.3 混合基算法存在的问题与改进

3.3.1 混合基算法存在的问题

由 3.2 中的分析可知，混合基 FFT 算法可以通过对序列点数 N 进行质因数分解，从而大幅减小计算量。然而当序列长度 N 为质数或质因数分解含有大质数因子，此时 N 不能得到有效分解，混合基算法就变为计算质数点数的 DFT，这会导致计算速度大幅降低，因此必须引入雷德算法进行改进。

3.3.2 雷德算法

当序列长度 N 为素数时，可以通过雷德算法进行改进。雷德算法通过将素数长度的 DFT 转化为循环相关，而循环相关可通过 FFT 高效计算，通过调整计算循环相关的 FFT 点数可以满足快速计算的要求，从而加快质数点数 FFT 的计算速度。

为后续公式推导需要，定义：

$$\langle X \rangle_N = X \; mod N \tag{16}$$

若 N 为素数，则存在一个数 g(可能不唯一)，使得整数$i = 1, \cdots, N-1$与整数$j = 1, \cdots, N-1$存在一一映射关系，映射关系可表示为：

$$j = \langle g^i \rangle_N \tag{17}$$

数论中 g 被称为 N 的原根，一般情况下素数的原根可查表获得。

对于序列$x[n]$，其 DFT 序列为$X[k]$，则$X[0]$可由下式计算：

$$X[0] = \sum_{n=0}^{N-1} x[n] \tag{18}$$

对于其他$X[k]$，注意到$x[0]$不参加乘法运算，可以在求和最后加入，因此对序列$X[k]$的计算可转化为对序列$X[k] - x[0]$的计算，即

$$X[k] - x[0] = \sum_{n=1}^{N-1} x[n] W^{nk} \tag{19}$$

对求和项进行置换，通过下列映射关系改变方程顺序：

$$\begin{aligned} n &\to \langle g^n \rangle_N \\ k &\to \langle g^k \rangle_N \end{aligned} \tag{20}$$

注意到$\langle g^{N-1} \rangle_N = \langle g^0 \rangle_N$，因此式(18)可化为：

$$X[\langle g^k \rangle_N] - x[0] = \sum_{n=1}^{N-1} x[\langle g^n \rangle_N] W^{g^{n+k}} \tag{21}$$

此时可发现，序列$X[\langle g^k\rangle_N]-x[0]$是序列$x[\langle g^n\rangle_N]$和$W^{g^n}$的循环相关，而利用 FFT 算法可大幅减少循环相关的计算量，具体可通过以下方法实现：

由于 N 是素数，则$N-1$必为合数。若$N-1$是高度复合的合数，则式(20)中的$N-1$点循环相关可表示为$x[\langle g^{-n}\rangle_N]$和$W^{g^n}$的 DFT 的乘积的逆 DFT，所有 DFT 运算均可通过 FFT 算法实现：

$$X[\langle g^k\rangle_N]-x[0]=DFT^{-1}\left[DFT\left[x[\langle g^{-n}\rangle_N]\right]\left(DFT\left[W^{g^n}\right]\right)\right] \tag{22}$$

这一种方法仅在$N-1$为高度合数时有效。若$N-1$仅为一般合数，则由公式(15)给出的运算量可知 FFT 算法计算效率并不是很高。对于点数非高度合数的循环相关或卷积，可将其作为更大点数的循环卷积的一部分进行计算。设N'为任意大于$2N-4$的高度合数，在$x[\langle g^n\rangle_N]$的第 0 个元素和第 1 个元素之间插入$N'-N+1$个 0，可构造N'点序列$b[n]$。将$N-1$点序列W^{g^n}周期性重复，直至达到N'个点，可构造第二个N'点序列$c[n]$。此时在$b[n]$的 DFT 与$c[n]$的 DFT 的乘积的逆 DFT 中，前$N-1$个点即为$X[\langle g^k\rangle_N]-x[0]$。由于$N'$可选择为高度合数，甚至是 2 的幂次，因此可通过 FFT 算法计算上述 DFT。

无论采用哪种方法，若预计算W^{g^n}的变换，可节省约 1/3 的计算量。第一种方法的计算量与$N-1$乘以$N-1$的因子和成正比；第二种方法的计算量与$N'logN'$成正比。此外，式(18)中的求和运算以及将$x[0]$加到每一个$X[k]$的运算，可利用 FFT 计算相关时得到的中间量完成，额外计算量可忽略不计。

4. 仿真验证

4.1 DTFT 频谱

根据上述对任意点数 FFT 算法原理的介绍，可以利用递归方式对基 p-FFT 算法、混合基算法和雷德算法进行编程实现。在这里仿真验证中，为了使结果更直观，我们以有限长的单边指数序列为例，探究不同 FFT 算法的运行速度。单边指数序列为形如下式的序列：

$$x[n]=a^n u[n],\qquad |a|<1 \tag{23}$$

仿真验证时取$a=0.8e^{j\frac{\pi}{5}}$，单边指数序列 DTFT 频谱图如下图所示：

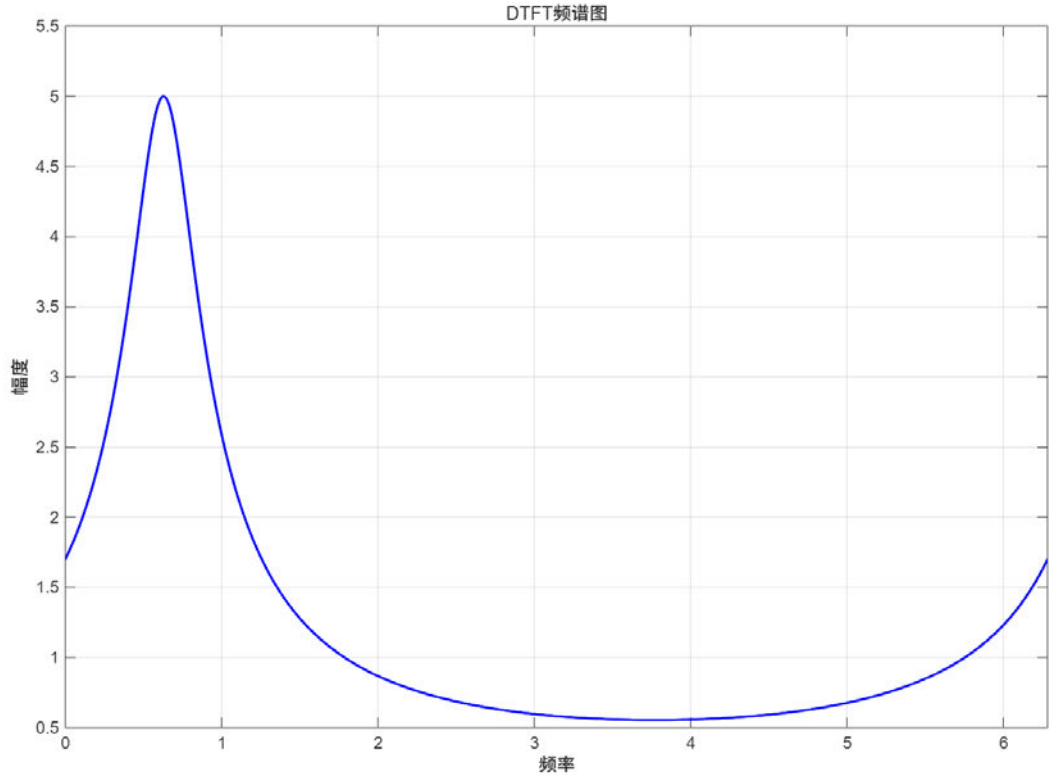

图 3：DTFT 频谱图

当序列较长时后面的项几乎趋近于 0，对 DFT 的求解结果影响可以忽略不计，并且 DFT 是对 DTFT 采样的结果，因此绘制的 DFT 频谱图的形状也应与上图图相同。

4.2 基 p-FFT 算法

分别取序列的长度为3^7和5^5，利用 MATLAB 内置的 fft 函数、DFT 定义和基 p-FFT 算法分别求序列的 DFT，三种求解方式绘制的频谱图如右图所示。

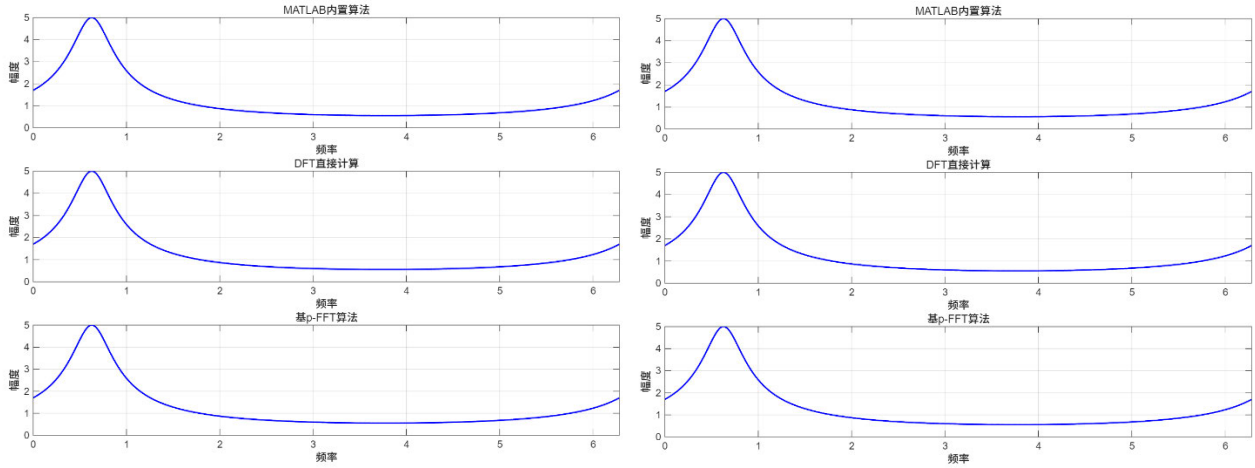

图 4：基 p-FFT 算法频谱图(左图序列点数 2187，右图序列点数 3125)

三种求解方式消耗的时间如下所示。

```
序列的长度N=2187
MATLAB内置fft函数历时 0.000087 秒。
DFT直接计算历时 1.147994 秒。
基p-FFT算法历时 0.006452 秒。
```

```
序列的长度N=3125
MATLAB内置fft函数历时 0.000134 秒。
DFT直接计算历时 2.438041 秒。
基p-FFT算法历时 0.009380 秒。
```

图 5：基 p-FFT 算法运算时间(左图序列点数 2187，右图序列点数 3125)

由上图可知，基 p-FFT 算法运算速度虽然不及 MATLAB 内置 fft 函数，但是相较于通过 DFT 定义直接计算要快得多，因此对于长度满足$N = p^L$*高度*复合数序列，基 p-FFT 算法可以大幅提升运算速度。

4.3 混合基 FFT 算法

取序列的长度为 3000，其中$3000 = 2^3 \times 3 \times 5^3$，在此情况下基 p-FFT 算法不再适用，需借助混合基算法。利用 MATLAB 内置的 fft 函数、DFT 定义和混合基算法分别求序列的 DFT，三种求解方式绘制的频谱图以及计算时间如下所示：

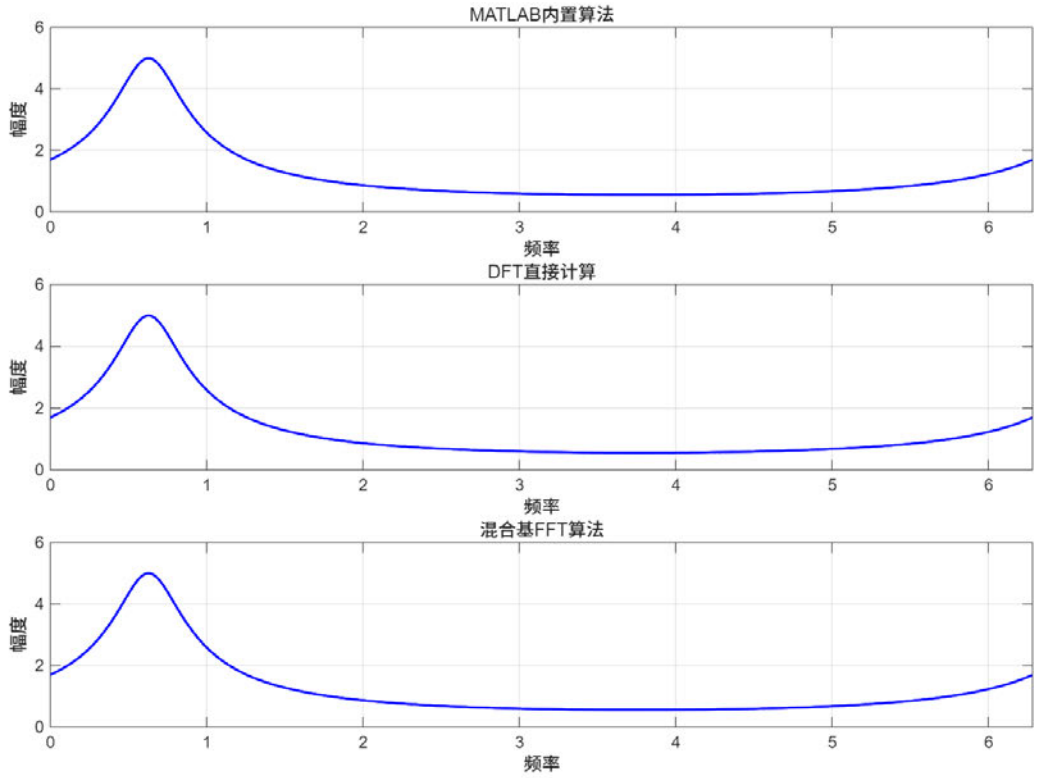

图 6：混合基 FFT 算法频谱图

图 7：混合基 FFT 算法运算时间

可见三种求解方式绘制的频谱图几乎完全一样，并于 DTFT 的形状相符，验证了求解结果的正确性。

由上图可知，编写的混合基算法运算速度虽然不及 MATLAB 内置 fft 函数，但是相较于通过 DFT 定义直接计算要快得多，因此对许长度满足$N = p_1 p_2 \cdots p_L$的序列，混合基算法可以大幅提升运算速度。

4.4 雷德-混合基 FFT 算法

取序列的长度为 3001，其中 3001 是个质数。利用 MATLAB 内置的 fft 函数、DFT 定义、混合基算法分别求序列的 DFT，三种求解方式绘制的频谱图以及消耗的时间如下所示：

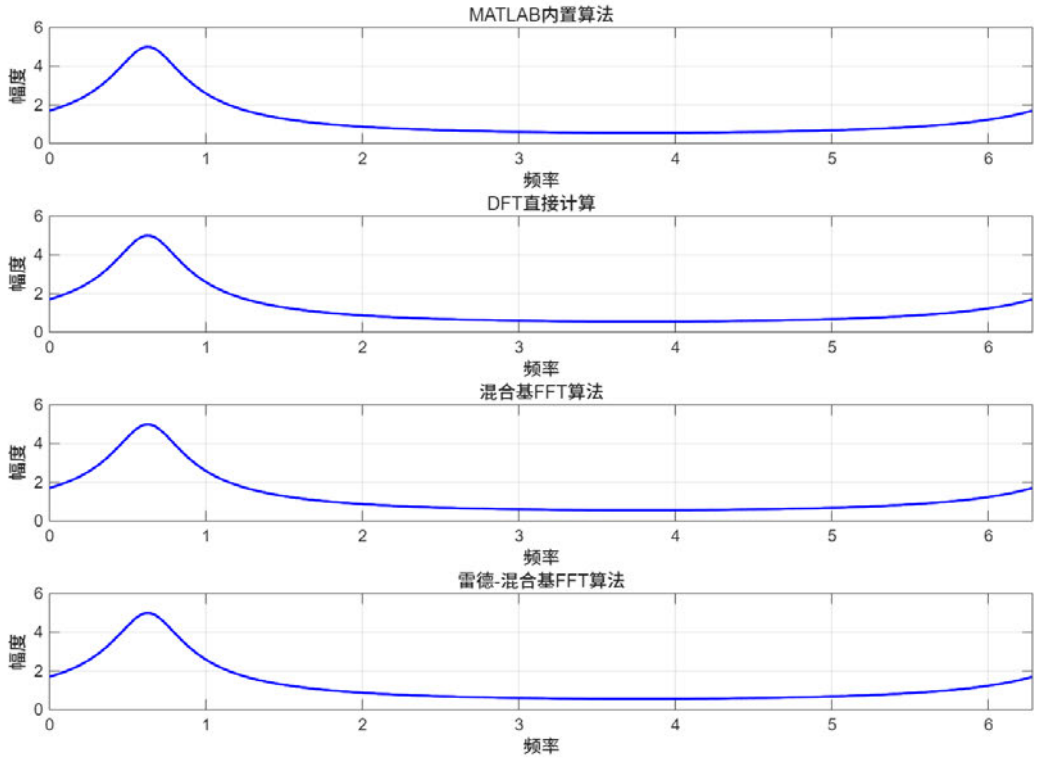

图 8：雷德-混合基 FFT 算法频谱图

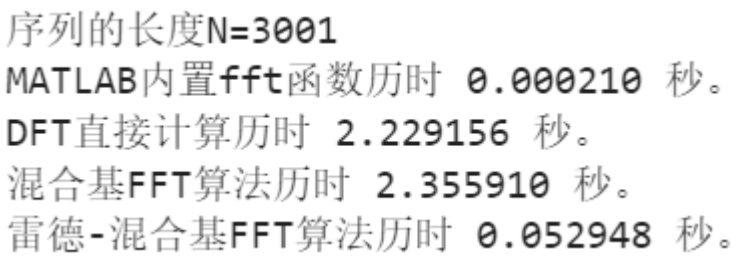

图 9：雷德-混合基 FFT 算法运算时间

由上图可以看出：

- 当序列长度为 3001 时，混合基算法和通过 DFT 直接计算用时几乎直接相等。
- 这说明当 DFT 的点数为质数时，混合基算法就相当于直接求解 DFT，不能加快计算速度。
- 当使用雷德算法改进后，计算速度显著提升，这与仿真预期相符，但计算速度仍不及 MATLAB 内置 fft 函数。

5. 总结

综合以上探究，可以采取以下算法策略实现任意点数 FFT 的快速计算：

1. 当序列长度$N = 2^L$时，可以直接采用基 2-FFT 算法进行计算，此时计算速度最快。
2. 当序列长度$N \neq 2^L$但$N = p^L$(p 为质数)时，应采用基 p-FFT 算法。

3. 当序列长度$N = p_1 p_2 \cdots p_{L-1} p_L$ ($p_1, p_2 \cdots p_{L-1}, p_L$均为质数)时，应对 N 进行质因数分解并采用混合基算法。
4. 当序列长度 N 为质数或质因数分解含有较大素数，应在混合基算法的基础上使用雷德算法进行优化。

MATLAB 内置函数 fft 主要基于 FFTW 来实现快速傅里叶变换，根据输入序列的长度不同采取不同的算法。FFTW 在计算 fft 仍采用了混合基算法和雷德算法等，但其算法种类更多样，使用更灵活，并且底层采用 C 语言编写。这大大提升了 fft 函数的运行速度，与仿真实验中实现的 FFT 算法相比具有更高的效率。混合基算法可以高效计算任意长度数据序列的 DFT，因此在信号处理、数字通信、音频处理等领域得到了广泛的应用。

通过此次任意点数 FFT 算法探究，我对数字信号处理中 FFT 算法有了更深刻的认识与理解。本次探究从基 2-FFT 算法的局限性出发，逐步推广到基 p-FFT、混合基算法，再到雷德算法优化质数点数场景，整个过程让我对 FFT 算法加速运算的原理有了更具体的认识，而不再局限于课本中的基 2-FFT 分解的方法。在理论推导过程中，基 3-FFT 算法的蝶形运算单元分析、混合基的质因数分解思路，锻炼了我的抽象思维与数学建模能力。MATLAB 仿真验证则让我感受到理论与实践结合的重要性，通过对比不同算法的运算速度与频谱效果，直观展现了 FFT 算法加快运算速度的效果。此外，雷德算法中数论知识的应用，让我认识到跨学科融合对解决复杂问题的意义。这次研究不仅提升了我的科研能力，更锻炼了我提出、分析和解决问题的能力，为后续信号处理相关学习与工程应用奠定了坚实基础。

此外在本课题的探究中，仍有许多值得我反思和改进的地方。在进行 PPT 课堂展示的过程中，由于我未能充分从听众的角度出发、对报告时间把握不恰当、对汇报重点的理解存在偏差，从而在一定程度上影响了课堂汇报的效果。在此我也深切感谢[redacted]老师对我课堂展示的不足之处进行了批评指正，我将认真听从您的意见和教导，吸取此次翻转课堂的经验教训，在以后的汇报和展示中注重时间的把控和重点的表达，不断提升自己的演讲汇报能力。

参考文献

[1] Cooley, James W.,Tukey,et al. An algorithm for the machine calculation of complex Fourier series[J]. Mathematics of Computation,1965,Vol.19(90): 297-301
[2] Rader, C.M. Discrete Fourier transforms when the number of data samples is prime[J]. Proceedings of the IEEE,1968,Vol.56(6): 1107-1108.
[3] Frigo, M.,Johnson,et al. FFTW: an adaptive software architecture for the FFT[C]//Proceedings of the 1998 IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP '98 (Cat. No.98CH36181). 1998.
[4] Oppenheim, A. V., Schafer, R. W. Discrete-Time Signal Processing (3rd ed.)[M]. Upper Saddle River (N.J.): Prentice Hall, 2010. ISBN 9780132067099.
[5] 程佩青编著. 数字信号处理教程 经典版[M]. 北京：清华大学出版社,2015
[6] 冷狗。自己实现混合基 fft，适用于任意长度序列 [EB/OL]. (2025-01-17) [2025-12-6]. https://zhuanlan.zhihu.com/p/6871666826.
[7] 通信考研加油哥。数字信号处理 - 基 3 的 FFT 算法 [EB/OL]. (2025-09-02) [2025-12-6]. https://zhuanlan.zhihu.com/p/1946000854160250159.

分工及贡献度

[redacted]

附录（可选）

1. 主代码

```
clc,clear,close all;
%% 生成有限长单边指数序列
a = 0.8*exp(j*pi/5);
L = 3001; % 3^7,5^5,3000,3001
fprintf('序列的长度 N=%d\n',L)
% 生成序列
[n, seq] = gen_seq(a, L);
% 仿真参数
p = 3;
Radix_p_flag = 0;
Radix_mix_flag = 1;
Radix_rader_mix_flag = 1;
t_cnt = 3;
cnt = 1;

% 绘制 DTFT 频谱图
plot_dtft(a)

% MATLAB 内置 fft 函数
fprintf('MATLAB 内置 fft 函数')
tic
fft_base = fft(seq);
toc
figure('Color','w','Name','FFT 频谱图')
plot_fft(t_cnt, cnt, L, fft_base,'MATLAB 内置算法')
cnt = cnt + 1;

% dft 直接计算
fprintf('DFT 直接计算')
tic
dft_test = dft(seq);
toc
plot_fft(t_cnt, cnt, L, dft_test,'DFT 直接计算')
cnt = cnt + 1;

% 基 p-fft 算法
if Radix_p_flag == 1
    fprintf('基 p-FFT 算法')
    tic;
    fft_Radix_p_test = Radix_p_FFT(seq, p);
    toc;
    plot_fft(t_cnt, cnt, L, fft_Radix_p_test,'基 p-FFT 算法')
    diff = fft_base - fft_Radix_p_test;
    cnt = cnt + 1;
end

% 混合基 fft 算法
if Radix_mix_flag == 1
    fprintf('混合基 FFT 算法')
    tic;
    fft_Radix_mix_test = Radix_mix_fft(seq);
    toc;
    plot_fft(t_cnt, cnt, L, fft_Radix_mix_test,'混合基 FFT 算法')
    diff = fft_base - fft_Radix_mix_test;
    cnt = cnt + 1;
end

% 雷德-混合基 fft 算法
if Radix_rader_mix_flag == 1
```

```
    fprintf('雷德-混合基 FFT 算法')
    tic;
    fft_Radix_rader_mix_test = Radix_rader_mix_fft(seq);
    toc;
    plot_fft(t_cnt, cnt, L, fft_Radix_rader_mix_test,'雷德-混合基 FFT 算法')
    diff = fft_base - fft_Radix_rader_mix_test;
    cnt = cnt + 1;
end
saveas(gca,'FFT 频谱图.png')
disp(size(diff))  % 误差的形状
disp(diff(1:5))  % 显示 5 个误差
err = diff.*conj(diff);
mse = mean(err, "all");  % 均方误差
disp(mse)

diff2 = fft_base - dft_test;
disp(size(diff2))
disp(diff2(1:5))
err = diff2.*conj(diff2);
mse = mean(err, "all");  % 均方误差
disp(mse)
```

2. gen_seq 函数

```
function [n, seq] = gen_seq(a, L)
% 生成有限长单边指数序列
n = 0:L-1;
seq = a.^n;
end
```

3. plot_dtft 函数

```
function plot_dtft(a)
% 绘制 DTFT 频谱图
figure('Color','w','Name','DTFT 频谱图')
omega = 0:pi/1500:2*pi;
X = 1./(1 - a*exp(-1j*omega));
plot(omega, abs(X), 'b','LineWidth',1.5);
xlabel('频率'); ylabel('幅度');
grid on;
set(gca, 'FontSize',10);
xlim([0,2*pi])
title('DTFT 频谱图')
saveas(gca,'dtft.png')
saveas(gcf, 'DTFT 频谱图.png')
end
```

4. plot_fft 函数

```
function plot_fft(t_cnt, cnt, L, fft_base,name)
% 绘制 FFT 频谱图
k = 0:L-1;
subplot(t_cnt, 1, cnt)
plot(2*pi*k/L, abs(fft_base), 'b', 'LineWidth',1.5);
xlabel('频率'); ylabel('幅度');
grid on;
set(gca, 'FontSize',10);
xlim([0,2*pi])
title(name)
end
```

5. RaderFFT 函数

```
function X = RaderFFT(x)
% RaderFFT 实现雷德算法
% 验证输入序列长度是否为质数

N = length(x);
if ~isprime(N)
    error('Input length must be a prime number.');
    % 只处理质数，其它的情况不应该调用该函数
end
```

```
10.
11. % 常量定义
12. X0 = sum(x); % 原点项
13. g = findPrimitiveRoot(N); % 找到模 N 的原根
14. DOUBLE_PI = 2 * pi;
15.
16. % 初始化 a_q 和 b_q
17. aq = zeros(1, N-1);
18. bq = zeros(1, N-1);
19. aqIndex = zeros(1, N-1);
20. bqIndex = zeros(1, N-1);
21.
22. aqIndex(1) = 1; % 起始索引
23. bqIndex(1) = 1; % 起始索引
24. aq(1) = x(2); % 起始值
25. bq(1) = exp(-1i * DOUBLE_PI / N); % 权重初始化
26.
27. % 计算索引和权重
28. currentExp = modExp(g, 1, N); % g^1 mod N
29. currentExpInverse = modExpInverse(g, 1, N); % g^(-1) mod N
30. expInverseBase = currentExpInverse;
31.
32. for index = 2:N-1
33.     aqIndex(index) = currentExp; % 更新 aq 索引
34.     bqIndex(index) = currentExpInverse; % 更新 bq 索引
35.
36.     aq(index) = x(currentExp+1); % 填充 aq 值
37.     bq(index) = exp(-1i * (currentExpInverse * DOUBLE_PI / N)); % 填充 bq 权重
38.
39.     % 更新 currentExp 和 currentExpInverse
40.     currentExp = mod(currentExp * g, N);
41.     currentExpInverse = mod(currentExpInverse * expInverseBase, N);
42. end
43.
44. % 补零到 2 的幂次长度
45. M = 2^nextpow2(2*N-3);
46. if M ~= N-1
47.     aq = [aq(1), zeros(1, M-N+1), aq(2:end)];  % 在第一二元素之间插入连续的零
48.     len = length(bq);  % 记录旧的 bq 的长度，方便之后循环填充
49.     bq = [bq, zeros(1, M-N+1)];  % 循环填充自己到和 aq 一样的长度，务必使用 for 循环，以为可能会循
    环多次，没法直接索引填充
50.     for i=1:1:M-N+1
51.         bq(len+i) = bq(i);
52.     end
53. end
54.
55. % 使用 MATLAB 内置 FFT 计算
56. % 这部分可以使用 matlab 自带的 fft，如果已经实现了混合基 fft，也可以用混合基 fft，此时序列长度是
    2^n，可以高效计算
57. % faq = fft(aq);
58. % fbq = fft(bq);
59. faq = Radix_2_FFT(aq);
60. fbq = Radix_2_FFT(bq);
61. product = faq .* fbq;
62.
63. % 逆 FFT 计算
64. % 这个 ifft，序列长度也是 2^n，所以不用担心质数
65. % inverseDFT = ifft(product);
66. inverseDFT = Radix_2_IFFT(product);
67.
68. % 合并结果，注意不做额外的除法
69. X = zeros(1, N);
70. X(1) = X0; % DC 分量
71. for index = 1:N-1
72.     X(bqIndex(index)+1) = inverseDFT(index) + x(1); % 修复除法问题
73. end
74. end
```

```
75.
76. %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%5
77. % 辅助函数 1:
78. % 快速模幂计算 (a^k) % n
79. function result = modExp(a, k, n)
80. result = 1;
81. base = mod(a, n);
82. while k > 0
83.     if mod(k, 2) == 1
84.         result = mod(result * base, n);
85.     end
86.     base = mod(base^2, n);
87.     k = floor(k / 2);
88. end
89. end
90.
91. % 辅助函数 2:
92. % 模逆运算 (a^(-k)) % n
93. function result = modExpInverse(a, k, n)
94. result = modExp(a, n-1-k, n); % 使用费马小定理
95. end
96.
97. % 辅助函数 3:
98. % 找到质数 N 的一个原根
99. function g = findPrimitiveRoot(N)
100. for g = 2:N-1
101.     if isPrimitiveRoot(g, N)
102.         return;
103.     end
104. end
105. error('No primitive root found.');
106. end
107.
108. % 辅助函数 4:
109. % 判断一个数是否是模 N 的原根
110. function isRoot = isPrimitiveRoot(g, N)
111. tot = N - 1; % 欧拉函数值 (N 为质数时等于 N-1)
112. factors = factor(tot); % 对 tot 进行因式分解
113. isRoot = true;
114.
115. % 检查 g^((tot)/pi) (mod N) 是否为 1
116. for pi = factors
117.     if modExp(g, tot / pi, N) == 1
118.         isRoot = false;
119.         return;
120.     end
121. end
122. end
123.
124. % 辅助函数 5:
125. % 基-2 的 fft
126. function fft_x = Radix_2_FFT(x)
127. N = length(x);
128. if N == 1
129.     fft_x = x;
130.     return;
131. end
132. % 奇数项和偶数项
133. X_even = Radix_2_FFT(x(1:2:N));  % 偶数项
134. X_odd = Radix_2_FFT(x(2:2:N));  % 奇数项
135.
136. % 旋转因子
137. W = exp(-2*pi*1i/N * (0:N/2-1));
138. fft_x = [X_even + W .* X_odd, X_even - W .* X_odd];  % 合并结果
139. end
140.
141.
```

```
% 辅助函数 6-part1：
% 基-2 的 ifft，结果除以 N，是通过调用结果不除以 N 的 ifft 实现的
function fft_x = Radix_2_IFFT(x)
% 对 ifft 的结果除以 N
N_const = length(x);
fft_x = radix_2_ifft(x);
fft_x = fft_x./N_const;
end

% 辅助函数 6-part2
% 基-2 的 ifft，不除以 N，要算完单独除以 N
function fft_x = radix_2_ifft(x)
N = length(x);
if N == 1
    fft_x = x;
    return;
end
% 奇数项和偶数项
X_even = radix_2_ifft(x(1:2:N));  % 偶数项
X_odd = radix_2_ifft(x(2:2:N));  % 奇数项

% 旋转因子
W = exp(2*pi*1i/N * (0:N/2-1));
fft_x = [X_even + W .* X_odd, X_even - W .* X_odd];  % 合并结果
end
```

6. Radix_mix_fft 函数

```
function fft_x = Radix_mix_fft(x)
% 混合基 fft
% 基-p 混合基 fft，可以处理任意长度

% 获取数据长度，之后对它进行质因数分解
N = length(x);
factor_list = factor(N);  % 质因数分解

if factor_list(1) == 1  % 递归到底了
    fft_x = x;
    return;
end

p = factor_list(1);  % 取最小的那个
% 将数据划分为 p 段，每一段长度为 N/p
X_p_array = zeros(p, N/p);  % 每一行就是一段

for i = 1:1:p
    % 调用自己
    X_p_array(i, :) = Radix_mix_fft(x(i:p:N));
end
WN = exp(-2*pi*1i / N);  % 基于 N 的旋转因子
Wp = exp(-2*pi*1i / p);  % 基于 p 的旋转因子

fft_x = zeros(1, N);  % 初始化输出
for k = 0:1:N/p-1
    for i=1:1:p  % 列循环
        for j=1:1:p  % 行循环
            fft_x(k+1+(i-1)*N/p) = fft_x(k+1+(i-1)*N/p) + X_p_array(j, k+1) * WN^((j-1)*k) * Wp^((i-1)*(j-1));
        end
    end
end
end
```

7. Radix_p_FFT 函数

```
function fft_x = Radix_p_FFT(x, p)
% 基 p 的 fft
% 需要保证 x 的长度为 p^n
N = length(x);
```

```
    if N == 1
        fft_x = x;
        return;
    end
    X_p_array = zeros(p, N/p);  % 每一行就是一段
    for i = 1:1:p
        X_p_array(i, :) = Radix_p_FFT(x(i:p:N), p);  % 索引为
    0, p, 2p, ... // 1, p+1, 2p+1, ...
    end
    WN = exp(-2*pi*1i / N);  % 基于 N 的旋转因子
    Wp = exp(-2*pi*1i / p);  % 基于 p 的旋转因子

    fft_x = zeros(1, N);  % 初始化输出
    for k = 0:1:N/p-1
        for i=1:1:p  % 列循环
            for j=1:1:p  % 行循环
                fft_x(k+1+(i-1)*N/p) = fft_x(k+1+(i-1)*N/p) + X_p_array(j, k+1) * WN^((j-
    1)*k) * Wp^((i-1)*(j-1));
            end
        end
    end
    end
```

8. Radix_rader_mix_fft 函数

```
    function fft_x = Radix_rader_mix_fft(x)
    % 加入了雷德算法的混合基 fft，可以处理质数情况
    % 获取数据长度，之后对它进行质因数分解
    N = length(x);
    factor_list = factor(N);  % 质因数分解
    if factor_list(1) == 1  % 递归到底了
        fft_x = x;
        return;
    end

    % 对于小质数长度的序列，直接用定义比用雷德算法快
    if isscalar(factor_list) && factor_list(1) > 32  % 如果序列长度是质数，则调用雷德算法返回 fft 结
    果
        fft_x =  RaderFFT(x);
        return;
    end

    p = factor_list(1);  % 取最小的那个
    X_p_array = zeros(p, N/p);  % 每一行就是一段

    for i = 1:1:p
        X_p_array(i, :) = Radix_rader_mix_fft(x(i:p:N));  % 索引为
    0, p, 2p, ... // 1, p+1, 2p+1, ...
    end
    WN = exp(-2*pi*1i / N);  % 基于 N 的旋转因子
    Wp = exp(-2*pi*1i / p);  % 基于 p 的旋转因子

    fft_x = zeros(1, N);  % 初始化输出
    for k = 0:1:N/p-1
        for i=1:1:p  % 列循环
            for j=1:1:p  % 行循环
                fft_x(k+1+(i-1)*N/p) = fft_x(k+1+(i-1)*N/p) + X_p_array(j, k+1) * WN^((j-
    1)*k) * Wp^((i-1)*(j-1));
            end
        end
    end

    end
```