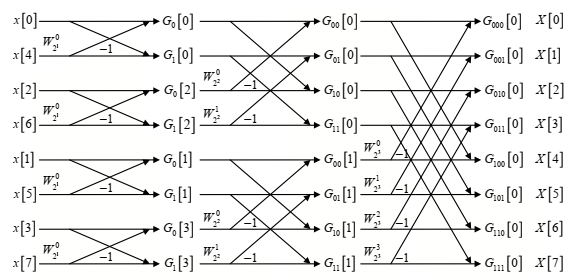
# 第一次课程设计报告

姓名：李科 班级：无02 学号：2020010678

## 设计目的和原理

通过设计基2 DIT-FFT和DIF-FFT比较直接进行DFT和使用FFT的效率差异。N点DFT的原理为：

其中，因此只需要定义好W矩阵就可以直接运算。FFT的思路是利用W矩阵的冗余（循环特性以及周期特性等），将原序列拆分为小规模的序列分别做DFT然后相加。当N为2的整数次幂时，刚好可以拆分为级，这也是基2的FFT原理。具体算法实现可以用递归，但是要比较最终运行时间，如果用递归则无法体现出FFT的优势，于是使用循环。原理可以参考下图（DIT的原理，DIF类似）：



只需要按照上图实现即可。

## 代码实现

使用的编程语言为python，实现过程中调用了numpy库（一些矩阵的操作便于实现）。

### 直接DFT

最简单的方法就是直接定义好W矩阵，然后与序列相乘即可。W矩阵定义如下：

*# 定义W矩阵*

    W\_N = np.exp(-2j \* np.pi / dft\_len)

    matrix\_W = W\_N \*\* np.matmul(

        np.expand\_dims(np.arange(dft\_len), 1), np.expand\_dims(np.arange(dft\_len), 0)

    )

利用矩阵乘法得到W矩阵，最后与x序列相乘可以得到结果。

### 基2 DIT-FFT

原理可以见第一部分。实现的主要困难为两部分：将输入序列按照二进制倒排；每一步定义好蝶形运算。

1. **二进制倒排**

最直接的方法就是先将十进制数转化为二进制数，然后倒序，再转化为十进制。但是我们知道计算机中本身就是按照二进制存储的，因此无需自己转化，而是将数字按照二进制打印出来。实现代码：

bits = "{:0{}b}".format(*x*, *N*)  *# N位二进制数*

return int(bits[::-1], 2)  *# 逆序后转化为10进制数*

第一行将数x按照N位二进制数打印出来，第二行逆序后，转化为十进制数。最终为了简化代码，使用map函数将二进制倒排函数应用到索引序列：

*# 将输入序列按照二进制倒排*

    input\_x = [

        real\_x[i]

        for i in list(

            map(lambda *x*: int("{:0{}b}".format(*x*, *N*)[::-1], 2), np.arange(fft\_len))

        )

    ]

1. **实现m级蝶形运算**

对于每一级都有个蝶形，我们需要确定每一步划分的DFT点数以及蝶形的两个元素间隔以及旋转因子取值，最终实现为：

for i in range(*N*):

        vec\_W\_N = np.exp(-2j \* np.pi / (2 \*\* (i + 1)) \* np.arange(2\*\*i))

        for j in range(0, fft\_len, 2 \*\* (i + 1)):

*# 遍历 2^(N-i) 个蝶形*

            for k in range(2\*\*i):

*# 遍历一个里面的 2^i 个蝶形*

                input\_x[j + k], input\_x[j + k + 2\*\*i] = (

                    input\_x[j + k] + vec\_W\_N[k] \* input\_x[j + k + 2\*\*i],

                    input\_x[j + k] - vec\_W\_N[k] \* input\_x[j + k + 2\*\*i],

                )

上面的为长度的对数。

### 基2 DIF-FFT

DIF实现的过程与DIT基本一致（因为二者的信号流图互为转置）。只不过输入序列不需要二进制倒排，蝶形运算后再倒排。蝶形的运算实现：

for i in range(*N*):

        vec\_W\_N = np.exp(-2j \* np.pi / (2 \*\* (*N* - i)) \* np.arange(2 \*\* (*N* - 1 - i)))

        for j in range(0, fft\_len, 2 \*\* (*N* - i)):

            for k in range(2 \*\* (*N* - 1 - i)):

*# 遍历一个里面的 2^i 个蝶形*

                real\_x[j + k], real\_x[j + k + 2 \*\* (*N* - 1 - i)] = (

                    real\_x[j + k] + real\_x[j + k + 2 \*\* (*N* - 1 - i)] \* complex(1, 0),

                    (real\_x[j + k] - real\_x[j + k + 2 \*\* (*N* - 1 - i)]) \* vec\_W\_N[k],

                )

其中一步中乘以complex(1,0)是因为会自动将复数的虚部忽略，需要告诉python这是复数序列。后来使用astype解决：

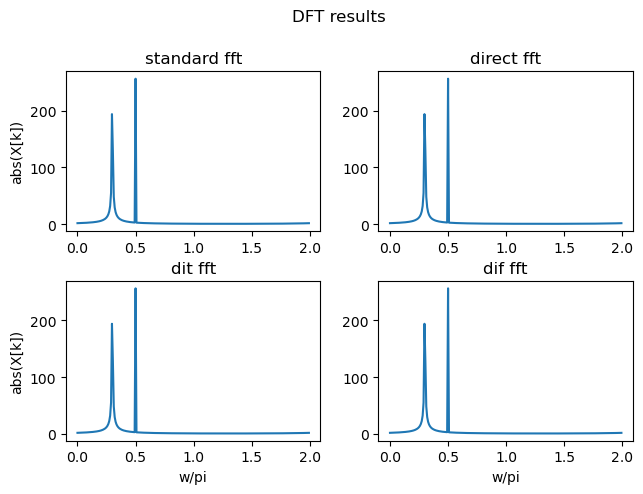
real\_x = real\_x.astype(np.complex\_)

## 实验结果及分析

这里的代码可见fft\_demo.ipynb。图像主要调用matplotlib库实现。

### 结果正确性分析

为了方便，定义信号为，长度定为。分别比较四种情况的dft结果：



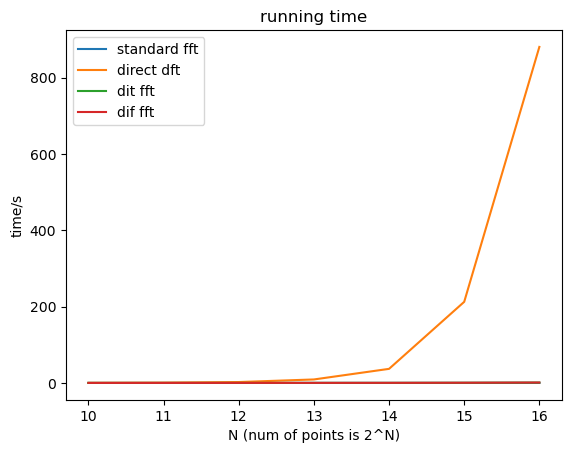
可以看出实现的fft与标准的结果完全一致。

### 运行时间分析

比较FFT与直接计算DFT的运行时间的差异，这里我们同样比较四种情况：（详细结果记录在result.txt中，这里只保留两位有效数字）

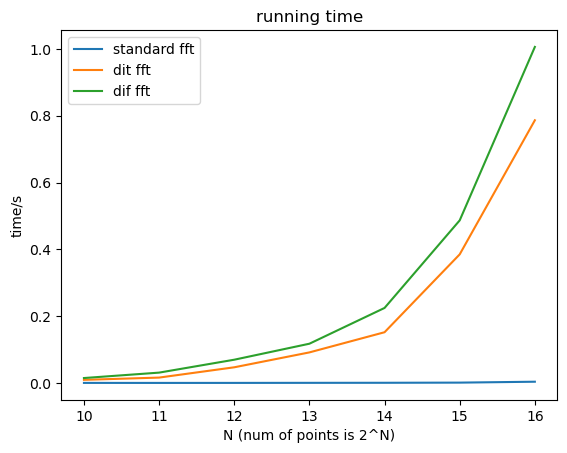
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| N | standard fft | direct fft | dit fft | dif fft |
| 10 | 1.6e-4 | 1.7e-1 | 9.3e-3 | 1.5e-2 |
| 11 | 8.9e-5 | 7.0e-1 | 1.6e-2 | 3.1e-2 |
| 12 | 1.4e-4 | 2.3 | 4.7e-2 | 7.0e-2 |
| 13 | 3.2e-4 | 9.1 | 9.1e-2 | 1.1e-1 |
| 14 | 5.0e-4 | 3.7e1 | 1.5e-1 | 2.2e-1 |
| 15 | 9.9e-4 | 2.1e2 | 3.9e-1 | 4.9e-1 |
| 16 | 3.7e-3 | 8.8e2 | 7.9e-1 | 1.0 |

我们可以看图：



可以看到相比于其他三种方法，直接计算dft的复杂度（对于点数的对数）明显呈指数级增长，这是符合理论分析的，因为dft的复杂度为，关于而言为。

去掉直接计算，剩余三种的曲线为：



可以看到标准库中的运行时间基本不变（增长很小），而自己实现的两种基2FFT都会有上升。这可能是标准库的实现可能基于分裂基等算法，降低复杂度，而自己实现的复杂度约为，仍然呈指数。比较值得讨论的是理论上应该差不多的dit和dif两种方法，dif比dit运行时间略长。个人认为是因为dit计算没有强制转化为复数这一步（与复数运算自动转化），但是dif中python会自动将复数的虚部忽略，因此不得不强制将原序列转化为复序列，这一显式转化会占用一定时间。

## 实验结论与感想

在直接计算dft时，当N达到15时，直接计算矩阵所占的存储十分大（16GB），因此最后不得不一行一行计算W矩阵，这也导致后面的直接dft耗时增加很多。我们看到直接计算dft，当N=16时，直接占用了15min，如果继续增加，可能一天都计算不完。而fft占用时间最后只有1s，这是效率的极大提升，可见fft的重要意义。

## 文件清单

* 第一次课程设计.docx
* images：存放结果图片
* code --

- fft\_demo.ipynb：运行实验结果（文中各种图片获得都在本文件中）

- standard\_fft.py：直接计算dft（没有起名为direct\_fft属于历史遗留问题，后来就懒得改了）

- dit\_fft.py：基2 DIT FFT算法

- dif\_fft.py：基2 DIF FFT算法

- result.txt：实验结果（各种条件下运行时间记录）