结果及讨论

A.DFT和FFT

1.编写一个DFT程序并验证

如下所示，编写 my_dft 函数，实现对输入的数组进行DFT计算，并返回结果。

```
def my_dft(x_array)->np.ndarray:
    N = len(x_array)
    n = np.arange(N)
    k = n.reshape((N,1))
    M = np.exp(-2j * np.pi * k * n / N)/N**0.5
    return np.dot(M,x_array)
```

使用一个八维的随机复向量作为测试，并且使用 np.fft 作为标准包函数，测试数据、 my_dft 、 np.fft 给出的结果以及二者差值如下：

```
Testing data [0.5488135 +0.96366276j 0.71518937+0.38344152j 0.60276338+0.79172504j
 0.54488318+0.52889492j 0.4236548 +0.56804456j 0.64589411+0.92559664j
 0.43758721+0.07103606j 0.891773  +0.0871293j ]

My DFT result [ 1.70078929+1.52718476j  0.37800121-0.09510798j  0.22101742+0.26322719j
 -0.33969539+0.51364957j -0.2775114 +0.1659601j   0.22010342+0.25805511j
 -0.26901744+0.20978922j -0.08140811-0.11710808j]

Numpy FFT result [ 1.70078929+1.52718476j  0.37800121-0.09510798j  0.22101742+0.26322719j
 -0.33969539+0.51364957j -0.2775114 +0.1659601j   0.22010342+0.25805511j
 -0.26901744+0.20978922j -0.08140811-0.11710808j]

My DFT result - Numpy FFT result [ 2.22044605e-16+2.22044605e-16j  0.00000000e+00+1.38777878e-17j
 -8.32667268e-17+5.55111512e-17j -2.77555756e-16+0.00000000e+00j
  5.55111512e-17-3.05311332e-16j -1.38777878e-16-5.55111512e-16j
 -7.77156117e-16-9.71445147e-16j  1.52655666e-15+9.29811783e-16j]
```

由上图结果可以看出，自己编写的函数和包函数给出的结果差值极小，可以认为自己编写的dft函数具有好的离散傅里叶变换的功能。

2.编写一个FFT程序并验证

递归算法

运用递归思想，写出Base2 FFT程序如下：

```
def fft(x):
    n = len(x)
    assert is_power_of_two(n), "输入数组的长度必须为2的幂次"
    if n==1:
        return x
    even = fft(x[0::2])
    odd = fft(x[1::2])
    T = [cmath.exp(-2j * cmath.pi * k / n) * odd[k] for k in range(n // 2)]
    # 利用离散傅里叶变换的性质:傅里叶变换的结果中前一半和后一半互为共轭
    return [even[k] + T[k] for k in range(n // 2)] + \
           [even[k] - T[k] for k in range(n // 2)]
```

首先验证程序的正确性：
使用一个八维的随机复向量作为测试，并与使用 `np.fft` 的结果进行比较，测试数据、 `fft` 、 `np.fft` 给出的结果以及二者差值如下：

```
Testing data [0.5488135 +0.96366276j 0.71518937+0.38344152j 0.60276338+0.79172504j
 0.54488318+0.52889492j 0.4236548 +0.56804456j 0.64589411+0.92559664j
 0.43758721+0.07103606j 0.891773  +0.0871293j ]

My FFT result [(4.810558553818618+4.319530794448397j), (1.0691488746735744-0.2690059935013657j), (0.6251316535958867+0.
7445189296543112j), (-0.9608036570900079+1.4528203902434071j), (-0.784920772617486+0.4694060413026291j), (0.622546494272821+0.
729890062697657j), (-0.7608962217321006+0.5933735209745091j), (-0.23025689350270728-0.33123166181131O7j)]

FFTW result [ 4.81055855+4.31953079j  1.06914887-0.26900599j  0.62513165+0.74451893j
 -0.96080366+1.45282039j -0.78492077+0.46940604j  0.62254649+0.72989006j
 -0.76089622+0.59337352j -0.23025689-0.33123166j]
```

由上图结果可以看出，自己编写的函数和包函数给出的结果差值极小，可以认为自己编写的fft函数具有好的快速离散傅里叶变换的功能。
在验证了程序的正确性后给出数组大小从2^4到2^{12}的测试样例，记录程序用时。

```
my fft for size16,exection time:0.0339867000002414
np.fft for size16,exection time:0.00159440000042509288
my fft for size32,exection time:0.07550659999833442
np.fft for size32,exection time:0.0019245000148657771
my fft for size64,exection time:0.16855230001965538
np.fft for size64,exection time:0.0016230999899562448
my fft for size128,exection time:0.3851587999961339
np.fft for size128,exection time:0.0019041000050492585
my fft for size256,exection time:0.7883889000222553
np.fft for size256,exection time:0.0032485999981872737
my fft for size512,exection time:1.6509786999959033
np.fft for size512,exection time:0.0041893000015988895
my fft for size1024,exection time:3.3842918000009377
np.fft for size1024,exection time:0.007627399987541139
my fft for size2048,exection time:7.111340300005395
np.fft for size2048,exection time:0.015973100002156557
my fft for size4096,exection time:14.849382999993395
np.fft for size4096,exection time:0.029052300000330433
```

由上图可以看出，我所编写的fft函数用时多于标准库的用时。
接下来分析我的代码的时间复杂度：

1. 递归调用的次数为$log_2(n)$，每次递归调用的

```
T = [cmath.exp(-2j * cmath.pi * k / n) * odd[k] for k in range(n // 2)]
    # 利用离散傅里叶变换的性质:傅里叶变换的结果中前一半和后一半互为共轭
    return [even[k] + T[k] for k in range(n // 2)] + \
           [even[k] - T[k] for k in range(n // 2)]
```

语句的时间复杂度为$O（\frac{n}{2}）$（此处考虑到离散傅里叶变换的形式，即前一半与后一半互为复共轭），
因此时间复杂度为$O(\frac{nlog_2(n)}{2}) = O(nlog_2(n))$。
而这也是理论上最小的时间复杂度。
我的程序用时多于标准库函数的原因分析如下：

1. 我的程序中使用了for循环，时间上有较大的开销，即使我使用了list-generator语法，但是for循环的开销还是较大的。

2. python内置list比较慢
3. 我的程序是递归算法，多次递归进入函数也产生了一定的时间开销。

因此，我运用“蝴蝶运算”，改变了数组顺序，时间复杂度仍未$O(nlog_2(n))$，但是空间复杂度降为$O(1)$。