快速傅里叶变换 FFT 算法

摘要

数字信号处理（DSP）芯片是高级消费电子产品中的核心。数字信号处理芯片的特点是它的快速数值计算的能力，包括快速 Fourier 变换（FFT）。Fourier 变换在执行插值中相当有效，并且在现代信号处理的数据密集型应用中是不可代替的。Cooley 和 Tukey 在计算上的突破被称为快速 Fourier 变换（FFT）、意味着可以非常廉价地计算离散 Fourier 变换（DFT）。

本文主要推演出 FFT 算法并采用 numpy 实现。

关键词

快速 Fourier 变换(FFT) 离散 Fourier 变换(DFT) numpy

法国数学家 Jean Baptiste Joseph Fourier 在研究热传导理论时，为了使这种理论起作用，他需要扩张函数，不是像 Taylor 级数那样用多项式来表示，而是用首先由 Euler 和 Bernoulli 建立的一种革命性的方法——用正弦和余弦函数来表示。虽然当时被认为不够严谨而受到主流数学家的反对，但今天 Fourier 方法已渗透到应用数学、物理和工程的许多领域。

通过采用复数的语言可以大大简化三角函数的记法要求。每个复数具有形式 $z=a+bi$，这里 $i=\sqrt{-1}$。在平面直角坐标系中，每一个 z 在几何上表示为沿实（水平）轴长度为 a，沿虚（垂直）轴长度为 b 的二维向量。数 $z=a+bi$ 的复度量定义为 $|z|=\sqrt{a^2+b^2}$，恰好等于在复平面上从原点到这个复数的距离。复数 $z=a+bi$ 的复共轭是 $\overline{z}=a-bi$。

复算术中著名的 Euler 公式是指 $e^{i\theta}=\cos\theta+i\sin\theta$。$z=e^{i\theta}$ 的复度量是 1，因此，这种形式的复数落在复平面的单位圆上。可以把任意复数 a + bi 表示成极坐标 $z=a+bi=re^{i\theta}$，这里 r 的复度量是 $|z|=\sqrt{a^2+b^2}$，

$$\theta=\arctan\frac{b}{a}$$

复平面中的单位圆与度量 r=1 的复数相对应。为了把单位圆的两个数 $e^{i\theta}$ 和 $e^{i\gamma}$ 相乘，我们可以先转换成三角函数，然后再相乘：

$$e^{i\theta}e^{i\gamma}=(\cos\theta+i\sin\theta)(\cos\gamma+i\sin\gamma)$$

$$=\cos\theta\cos\gamma-\sin\theta\sin\gamma+i(\sin\theta\cos\gamma+\cos\theta\sin\gamma)$$

回忆正余弦加法公式，可以把它写成

$$\cos(\theta+\gamma)+i\sin(\theta+\gamma)=e^{i(\theta+\gamma)}$$

恰好是指数相加$e^{i\theta}e^{i\gamma}=e^{i(\theta+\gamma)}$，该式表明单位圆上两个数的乘积给出了单位圆上一个新的点，它的角度是这两个数的角度之和。Euler 公式隐藏了像正弦和余弦加法公式这样的三角学细节，而且使记法更简单。尽管它完全可以在实数内完成，但是 Euler 公式有深刻的简化效果。

我们挑选出度量为 1 的复数的一类特殊子集。如果$z^n=1$，那么复数 z 是 n 次单位根。实数轴上仅有两个单位根，-1 和 1；然而在复平面上却有许多个。对任意的 k<n，如果 n 次单位根不是 k 次单位根，那么这个 n 次单位根称为本原的。容易验证，对任意整数 n，复数$w_n=e^{-2\pi i/n}$是一个本原 n 次单位根。数$e^{-2\pi i/n}$也是一个本原 n 次单位根，但是我们将按通常的习惯用前者作为 Fourier 变换的基。

这里有一个关键的恒等式，我们以后在简化离散 Fourier 变换的计算中将需要它，当 n>1 时，用 w 表示 n 次单位根$w_n=e^{-2\pi i/n}$，则有

$$1+w+w^2+w^3+\ldots+w^{n-1}=0$$

这个等式可通过下式证得

$$(1-w)(1+w+w^2+w^3+\ldots+w^{n-1})=1-w^n=0$$

因为左式第一项不是 0，所以第二项必须是 0。

接下来引出离散 Fourier 变换(DFT)。

设 **x**=$[x_0,x_1,x_2,\ldots,x_{n-1}]^T$是一个(实值)n 维向量，记为$w=e^{-2\pi i/n}$；x 的离散 Fourier(DFT)变换是 n 维向量 **y**=$[y_0,y_1,y_2,\ldots,y_{n-1}]$，

$$y_k=\frac{1}{\sqrt{n}}\sum_{j=0}^{n-1}x_jw^{jk}$$

该引理表明，**x**=$[1,1,\ldots,1]^T$的 DFT 是 **y**=$[\sqrt{n},0,0,\ldots,0]$。用矩阵表示：

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \cdots \\ y_{n-1} \end{bmatrix}=\begin{bmatrix} a_0+ib_0 \\ a_1+ib_1 \\ a_2+ib_2 \\ \cdots \\ a_{n-1}+ib_{n-1} \end{bmatrix}=\frac{1}{\sqrt{n}}\begin{bmatrix} w^0 & w^0 & w^0 & \ldots & w^0 \\ w^0 & w^1 & w^2 & \ldots & w^{n-1} \\ w^0 & w^2 & w^4 & \ldots & w^{2(n-1)} \\ w^0 & w^3 & w^6 & \ldots & w^{3(n-1)} \\ & & \cdots & & \\ w^0 & w^{n-1} & w^{2(n-1)} & \ldots & w^{(n-1)^2} \end{bmatrix}\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \cdots \\ x_{n-1} \end{bmatrix}$$

每一个$y_k=a_k+ib_k$是一个复数。

n×n 矩阵

$$\mathbf{F}_n=\frac{1}{\sqrt{n}}\begin{bmatrix} w^0 & w^0 & w^0 & \dots & w^0 \\ w^0 & w^1 & w^2 & \dots & w^{n-1} \\ w^0 & w^2 & w^4 & \dots & w^{2(n-1)} \\ w^0 & w^3 & w^6 & \dots & w^{3(n-1)} \\ & & \dots & & \\ w^0 & w^{n-1} & w^{2(n-1)} & \dots & w^{(n-1)^2} \end{bmatrix}$$

叫做 Fourier 矩阵。除了第一行外，Fourier 矩阵的每一行加起来都等于 0。

使用离散 Fourier 变换(DFT)只是用 n×n 矩阵$\mathbf{F}_n$相乘的问题, 因此需要 O(n^2)次运算[明确地讲，是n^2次乘法和 n(n-1)次加法]。我们建立了可以明显减少运算次数的 DFT 形式，称为快速 Fourier 变换。

下面介绍快速 Fourier 变换(FFT)。

以传统的方式对 n 维向量用离散 Fourier 变换需要 O(n^2) 次运算, Cooley 和 Tukey 找到一种只需要 O（nlogn）次运算来完成 DFT 的方法，这种算法称为快速 Fourier 变换（FFT）。这一成就促进了 Fourier 变换方法的广泛应用。与问题本身的大小“几乎线性地”成比例的方法是很重要的。例如，对于实时数据就有使用它的可能性，这是因为分析大致能够出现在需要数据的同一时间尺度处。不久以后人们用特殊的电路来实现 FFT，现在则是用 DSP 芯片来表示 FFT 这种芯片广泛应用于分析和控制电子系统中。信号处理领域由于使用该算法将原先的模拟信号转化为数字信号。我们将解释这种方法并且通过运算次数来说明它相对于原本的 DFT 的优越性。

可以把 DFT $\mathbf{F}_n\mathbf{x}$ 写成

$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \\ \dots \\ y_{n-1} \end{bmatrix}=\frac{1}{\sqrt{n}}\mathbf{M}_n\begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \dots \\ x_{n-1} \end{bmatrix}$$

这里

$$\mathbf{M}_n=\begin{bmatrix} w^0 & w^0 & w^0 & \dots & w^0 \\ w^0 & w^1 & w^2 & \dots & w^{n-1} \\ w^0 & w^2 & w^4 & \dots & w^{2(n-1)} \\ w^0 & w^3 & w^6 & \dots & w^{3(n-1)} \\ & & \dots & & \\ w^0 & w^{n-1} & w^{2(n-1)} & \dots & w^{(n-1)^2} \end{bmatrix}$$

我们将说明如何递推地计算 $\mathbf{z}=\mathbf{M}_n\mathbf{x}$.要完成 DFT，只需要除以$\sqrt{n}$，或者 $y=\mathbf{F}_n\mathbf{x}=\mathbf{z}/\sqrt{n}$。

$$y_0 = w^0x_0 + w^0x_2 + w^0x_4 + \ldots + w^0x_{n-2} + w^0(w^0x_1 + w^0x_3 + w^0x_5 + \ldots + w^0x_{n-1})$$

$$y_1 = w^0x_0 + w^2x_2 + w^4x_4 + \ldots + w^{n-2}x_{n-2} + w^1(w^0x_1 + w^2x_3 + w^4x_5 + \ldots + w^{n-2}x_{n-1})$$

$$y_2 = w^0x_0 + w^4x_2 + w^8x_4 + \ldots + w^{2(n-2)}x_{n-2} + w^2(w^0x_1 + w^4x_3 + w^8x_5 + \ldots + w^{2(n-1)-2}x_{n-1})$$

...

$$y_{n-1} = w^0x_0 + w^{2(n-1)}x_2 + w^{4(n-1)}x_4 + \ldots + w^{(n-2)(n-1)}x_{n-2} + w^{n-1}(w^0x_1 + w^{2(n-1)}x_3 + w^{4(n-1)}x_5 + \ldots + w^{(n-1)^2-(n-1)}x_{n-1})$$

又$w^n = 1$，y_0~$y_{n/2-1}$不变，改写$y_{n/2}$~y_{n-1}

$$y_{n/2} = w^0x_0 + w^0x_2 + w^0x_4 + \ldots + w^0x_{n-2} + w^{n/2}(w^0x_1 + w^0x_3 + w^0x_5 + \ldots + w^0x_{n-1})$$

$$y_{n/2+1} = w^0x_0 + w^2x_2 + w^4x_4 + \ldots + w^{n-2}x_{n-2} + w^{n/2+1}(w^0x_1 + w^2x_3 + w^4x_5 + \ldots + w^{n-2}x_{n-1})$$

$$y_{n/2+2} = w^0x_0 + w^4x_2 + w^8x_4 + \ldots + w^{2(n-2)}x_{n-2} + w^{n/2+2}(w^0x_1 + w^4x_3 + w^8x_5 + \ldots + w^{2(n-1)-2}x_{n-1})$$

...

$$y_{n-1} = w^0x_0 + w^{2(n/2-1)}x_2 + w^{4(n/2-1)}x_4 + \ldots + w^{(n-2)(n/2-1)}x_{n-2} + w^{n-1}(w^0x_1 + w^{2(n/2-1)}x_3 + w^{4(n/2-1)}x_5 + \ldots + w^{(n-2)(n/2-1)}x_{n-1})$$

定义 $\mathbf{u}=(u_0, u_1, u_2, \ldots, u_{n/2-1})^T=\boldsymbol{M}_{n/2}(x_0, x_2, x_4, \ldots, x_{n-2})^T$

$\mathbf{v}=(v_0, v_1, v_2, \ldots, v_{n/2-1})^T=\boldsymbol{M}_{n/2}(x_1, x_3, x_5, \ldots, x_{n-1})^T$

所以$\mathbf{M}_n\mathbf{x}$ 可写为

$$
\begin{gathered}
y_0 = u_0 + w^0 v_0 \\
y_1 = u_1 + w^1 v_1 \\
y_2 = u_2 + w^2 v_2 \\
\cdots \\
y_{n/2-1} = u_{n/2-1} + w^{n/2-1} v_{n/2-1} \\
y_{n/2} = u_0 + w^{n/2} v_0 \\
y_{n/2+1} = u_1 + w^{n/2+1} v_1 \\
y_{n/2+2} = u_2 + w^{n/2+2} v_2 \\
\cdots \\
y_{n-1} = u_{n/2-1} + w^{n-1} v_{n/2-1}
\end{gathered}
$$

总之，DFT（n）的计算已被缩减为一组 DFT（n/2）的计算再加上某些额外乘法和加法。

暂时不考虑$\frac{1}{\sqrt{n}}$，DFT（n）能够被缩减为计算两个 DFT（n/2）再加上 2n-1 次额外运算（n-1 次乘法及 n 次加法）。

接下来，证明 FFT 运算的总次数。

设 n 是 2 的幂。规格为 n 的快速 Fourier 变换可以用 n(2$\log_2 n$-1)＋1 次加法和乘法以及一次除法（除以$\sqrt{n}$）来完成。

证　不考虑在最后要用到的平方根。定理的结果等价于：DFT（2^m）能够用2^m（2m-1）＋1 次加法和乘法来完成。事实上，当 n 是偶数时，我们从上面看到如何把 DFT（n）化成一对 DFT（n/2）。如果 n 是 2 的幂（譬如说 n=2^m），那么可以递推地分解这个问题直到我们得到 DFT（1），这是用 1×1 单位矩阵相乘，用了零次运算。从最低开始，DFT（1）无须进行运算，而 DFT（2）需要两次加法和一次乘法：$y_0 = u_0 + 1v_0$，$y_1 = u_0 + wv_0$，这里u_0和v_0是 DFT（1）（即$u_0 = y_0$，$v_0 = y_1$）。

DFT（4）需要两次 DFT（2）加上 2×4-1=7 次进一步的运算，总共需要 2（3）＋7=2^m（2m-1）＋1 次运算，这里 m=2。我们通过归纳继续：假设对给定的 m，这个公式是正确的，那么 DFT（2^{m+1}）需要用两个 DFT（2^m），它要进行 2[2^m（2m-1）＋1]次运算，加上 2×2^{m+1}-1 次额外的运算，总共是

2[2^m(2m-1)＋1]＋2^{m+2}-1=2^{m+1}(2m-1＋2)＋2-1=2^{m+1}[(2m＋1)-1]＋1.

因此，对于 DFT（2^m）的快速形式证得运算次数的公式2^m（2m-1）+ 1. 由此，得到结论[1]。

最后，我们使用 python 的 numpy 库实现上述算法[2]。下面定义了一个递归函数 fft 来实现算法(暂时不考虑$\frac{1}{\sqrt{n}}$)。

```
#!/usr/bin/env python
# coding: utf-8

# In[19]:

import numpy as np
x=np.array([1,2,3,4,5,6,7,8]) #输入 x 向量,个数应为 2 的幂
def fft(x):
    w=complex(np.cos(-2*np.pi/x.shape[0]),np.sin(-2*np.pi/x.shape[0]))
    if x.shape[0]==1:
        return x
    else:
     a=np.zeros(0)
     for i in range(0,x.shape[0],2):
        a=np.append(a,[x[i]])
     u=fft(a)
     a=np.zeros(0)
     for i in range(1,x.shape[0],2):
        a=np.append(a,[x[i]])
     v=fft(a)

     y=np.zeros(0)
     for i in range(0,x.shape[0]//2):
        y=np.append(y,[u[i]+w**i*v[i]])
     for i in range(x.shape[0]//2,x.shape[0]):
        y=np.append(y,[u[i-x.shape[0]//2]+w**i*v[i-x.shape[0]//2]])

     return y
```

fft(x)

In[20]:

np.fft.fft(x)

与 numpy.fft 库里的 fft 函数作比较，如图，结果一致，程序正确。

In [19]:
```python
import numpy as np

x=np.array([1,2,3,4,5,6,7,8]) #输入x向量，个数应为2的幂


def fft(x):
   w=complex(np.cos(-2*np.pi/x.shape[0]),np.sin(-2*np.pi/x.shape[0]))
   if x.shape[0]==1:
        return x
   else:
    a=np.zeros(0)
    for i in range(0,x.shape[0],2):
        a=np.append(a,[x[i]])
    u=fft(a)
    a=np.zeros(0)
    for i in range(1,x.shape[0],2):
        a=np.append(a,[x[i]])
    v=fft(a)

    y=np.zeros(0)
    for i in range(0,x.shape[0]//2):
        y=np.append(y,[u[i]+w**i*v[i]])
    for i in range(x.shape[0]//2,x.shape[0]):
        y=np.append(y,[u[i-x.shape[0]//2]+w**i*v[i-x.shape[0]//2]])

    return y


fft(x)
```

Out[19]:
```
array([36.+0.j        , -4.+9.65685425j, -4.+4.j        , -4.+1.65685425j,
       -4.+0.j        , -4.-1.65685425j, -4.-4.j        , -4.-9.65685425j])
```

In [20]:
```python
np.fft.fft(a)
```

Out[20]:
```
array([36.+0.j        , -4.+9.65685425j, -4.+4.j        , -4.+1.65685425j,
       -4.+0.j        , -4.-1.65685425j, -4.-4.j        , -4.-9.65685425j])
```

参考文献

[1] SAUER T. 数值分析 [M]. 数值分析, 2010.
[2] 张若愚. Python 科学计算 [M]. Python 科学计算, 2016.