介绍

周期函数的复指数形式展开

记:

$$E_k(x) = e^{ikx}, \quad 0, \pm 1, \cdots$$

利用 $E_k(x)$ 可以构造复数集上 $L_2[0,2\pi]$ 空间的标准正交系, 其中 $L_2[0,2\pi]$上的内积如下定义:

$$(f,g) = \frac{1}{2\pi}\int_0^{2\pi} f(x)\overline{g(x)}dx$$

即意味着:

$$(E_k, E_m) = 0 \quad k \neq m \quad ; \quad (E_k, E_k) = 1$$

对于离散形式, 考虑:

$$(f,g)_N = \frac{1}{N}\sum_{j=0}^{N-1} f(x_j)\overline{g(x_j)}$$

其中:

$$x_j = \frac{2\pi j}{N}, \quad 0 \le j \le N-1$$

注意, 上述 $(\cdot,\cdot)_N$ 不是内积, 不满足正定性: $(f,f)_N = 0 \Longleftrightarrow f = 0$

f只需要在所有节点上为0 即有 $(f,f)_N = 0$

引理

$\forall N \ge 1:$

$$(E_k, E_m)_N = \begin{cases} 1 & N|k-m \\ 0 & otherwise \end{cases}$$

假设 $f(x)$ 是以 2π 为周期的周期函数, 称 $f(x)$ 为次数不超过 $N-1$ 的指数多项式如果 $f(x)$ 有如下形式:

$$f(x) = \sum_{k=0}^{N-1} c_k e^{ikx} = \sum_{k=0}^{N-1} c_k E_k(x)$$

为了确定c_k , 两边分别利用 E_m 作离散内积得:

$$(f(x), E_m)_N = \sum_{k=0}^{N-1} c_k (E_k, E_m)_N, \quad 0 \le m \le N-1$$

将两边展开并且利用上面引理立得:

$$c_m = \frac{1}{N}\sum_{j=0}^{N-1} f(x_j)e^{-imx_j}, \quad 0 \le m \le N-1$$

其中 $x_j = \dfrac{2\pi j}{N}$, 在实际情况下, 通常需要通过 $\{f(x_j)\}$ 来确定 c_k ; 反之, 或者通过 c_k 来确定 $f(x_j)$, 显然, 直接对上式进行操作, 复杂度将会是 $O(N^2). Cooley, Tukey$ 提出了计算 c_k 的高效算法将计算 $p(x_j)$的复杂度降低为 $O(Nlog_2N)$ 这个方法即 FFT

矩阵形式

经过观察,很容易发现,实际上记 $c = (c_0, c_1, \cdots, c_{N-1})^T$ 并且注意 $e^{ix_j} = \omega_N^j$ 其中: ω_N 表示N次单位根 则可以将上面的关系写为矩阵形式如下:

$$\begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{N-1} \end{pmatrix} = \frac{1}{N}\begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \overline{\omega}_N & \overline{\omega}_N^2 & \cdots & \overline{\omega}_N^{N-1} \\ 1 & \overline{\omega}_N^2 & \overline{\omega}_N^4 & \cdots & \overline{\omega}_N^{2\cdot(N-1)} \\ & & \cdots & \cdots & \\ 1 & \overline{\omega}_N^{N-1} & \overline{\omega}_N^{2(N-1)} & \cdots & \overline{\omega}_N^{(N-1)\cdot(N-1)} \end{pmatrix}\begin{pmatrix} f_0 \\ f_1 \\ f_2 \\ \vdots \\ f_{N-1} \end{pmatrix}$$

其中, $\omega_N = e^{\frac{2\pi i}{N}}$,称该由 $f_0, f_1, \cdots, f_{N-1}$ 求解出$c_0, c_1, \cdots, c_{N-1}$的过程为离散傅里叶变换

$$\begin{pmatrix} f_0 \\ f_1 \\ f_2 \\ \vdots \\ f_{N-1} \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \omega_N & \omega_N^2 & \cdots & \omega_N^{N-1} \\ 1 & \omega_N^2 & \omega_N^4 & \cdots & \omega_N^{2\cdot(N-1)} \\ & & \cdots & \cdots & \\ 1 & \omega_N^{N-1} & \omega_N^{2(N-1)} & \cdots & \omega_N^{(N-1)\cdot(N-1)} \end{pmatrix}\begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ \vdots \\ c_{N-1} \end{pmatrix}$$

$\omega_N = e^{\frac{2\pi i}{N}}$,称该由$c_0, c_1, \cdots, c_{N-1}$ 求出 $f_0, f_1, \cdots, f_{N-1}$的过程为离散傅里叶逆变换

FFT的基本引理

引理1

假设 $p(x), q(x)$ 为 $N-1$ 阶的指数多项式, 并且使得: 对 $y_j = \frac{\pi j}{N}$ 成立:

$$p(y_{2j}) = f(y_{2j}), \quad q(y_{2j}) = f(y_{2j+1}), \quad 0 \le j \le N-1$$

则f满足上述条件的阶数小于 $2N-1$的指数插值多项式存在,并且可以由下式给出:

$$P(x) = \frac{1}{2}(1 + e^{iNx})p(x) + \frac{1}{2}(1 - e^{iNx})q(x - \frac{\pi}{N})$$

引理2

假设 引理1 中的多项式 $p(x), q(x), P(x)$ 分别由下式给出:

$$p(x)=\sum_{j=0}^{N-1}\alpha_j E_j(x) \quad q(x)=\sum_{j=0}^{N-1}\beta_j E_j(x) \quad P(x)=\sum_{j=0}^{2N-1}\gamma_j E_j(x)$$

则 $\forall 0\le j\le N-1$:

$$\gamma_j=\frac{1}{2}\alpha_j+\frac{1}{2}e^{\frac{-ij\pi}{N}}\beta_j \quad \gamma_{j+N}=\frac{1}{2}\alpha_j-\frac{1}{2}e^{\frac{-ij\pi}{N}}\beta_j$$

FFT 算法以及时间复杂度

直接推导

总结前面, 已经有了如下三个式子:

- $$f(x)=\sum_{k=0}^{N}c_k E_k(x)$$
  $$c_k=\frac{1}{N}\sum_{j=0}^{N-1}f(x_j)e^{-ikxj}$$
- 假设 p(x), q(x) 为次数等于$N-1$ 的指数多项式, $y_j=\frac{\pi j}{N}$, f 满足: $\forall 0\le j\le N-1$
  $$p(y_{2j})=f(y_{2j}), \quad q(y_{2j})=f（y_{2j+1}）$$

则存在 f(x) 在节点 y_j处 次数不超过 $2N-1$ 的插值多项式$P(x)$满足:

$$P(x)=\frac{1}{2}(1+e^{iNx})p(x)+\frac{1}{2}(1-e^{iNx})q(x-\frac{\pi}{N})$$

- 对上述的 $p(x), q(x), P(x)$ 并且满足:
  $$p(x)=\sum_{j=0}^{N-1}\alpha_j E_j(x), \quad q(x)=\sum_{j=0}^{N-1}\beta_j E_j(x), \quad P(x)=\sum_{j=0}^{2N-1}\gamma_j E_j(x)$$

则系数 α,β,γ有以下关系:

对 $0\le j\le N-1$

$$\gamma_j=\frac{1}{2}\alpha_j+\frac{1}{2}e^{\frac{-ij\pi}{N}}\beta_j$$
$$\gamma_{j+N}=\frac{1}{2}\alpha_j-\frac{1}{2}e^{\frac{-ij\pi}{N}}\beta_j$$

现在, 考虑对 $f(x)=\sum\limits_{k=0}^{2N-1}c_k E_k(x)$

$$c_k=\frac{1}{2N}\sum_{j=0}^{2N-1}f(x_j)e^{-ikxj} \quad x_j=\frac{\pi j}{N}$$

利用上式, 有:

$$\begin{cases}\gamma_k=\frac{1}{2N}\sum\limits_{j=0}^{2N-1}f(x_j)e^{-\frac{ikj\pi}{N}}\\ \alpha_k=\gamma_k+\gamma_{k+N}=\frac{1}{N}\sum\limits_{j=0}^{N-1}f(x_{2j})e^{-\frac{2ikj\pi}{N}} \quad 0\le k\le N-1\\ \beta_k=(\gamma_k-\gamma_{k+N})e^{i\frac{k\pi}{N}}=\frac{1}{N}\sum\limits_{j=0}^{N-1}f(x_{2j+1})e^{-\frac{2ikj\pi}{N}} \quad 0\le k\le N-1\end{cases}$$

$$\begin{cases}\alpha_k=\frac{1}{N}\sum\limits_{j=0}^{N-1}f(x_{2j})e^{-\frac{2ikj\pi}{N}}\\ \beta_k=\frac{1}{N}\sum\limits_{j=0}^{N-1}f(x_{2j+1})e^{-\frac{2ikj\pi}{N}}\\ \gamma_k=\frac{1}{2N}\sum\limits_{j=0}^{2N-1}f(x_j)e^{-\frac{ikj\pi}{N}}\end{cases}$$

FFT 的矩阵推导

由前面可以看出, 离散傅里叶变换实际上即通过如下矩阵乘法实现 $f_0, f_1,\cdots,f_{N-1}$ 到 $c_0,c_1,\cdots,c_{N-1}$的转换

$$\begin{pmatrix}c_0\\c_1\\c_2\\\vdots\\c_{N-1}\end{pmatrix}=\frac{1}{N}\begin{pmatrix}1&1&1&\cdots&1\\1&\overline{\omega}_N&\overline{\omega}_N^2&\cdots&\overline{\omega}_N^{N-1}\\1&\overline{\omega}_N^2&\overline{\omega}_N^4&\cdots&\overline{\omega}_N^{2\cdot(N-1)}\\&&\cdots&\cdots&\\1&\overline{\omega}_N^{N-1}&\overline{\omega}_N^{2(N-1)}&\cdots&\overline{\omega}_N^{(N-1)\cdot(N-1)}\end{pmatrix}\begin{pmatrix}f_0\\f_1\\f_2\\\vdots\\f_{N-1}\end{pmatrix}$$

下面将通过四维的例子来在推导快速计算上面矩阵乘法的思路

由:

$$\begin{pmatrix}c_0\\c_1\\c_2\\c_3\end{pmatrix}=\frac{1}{4}\begin{pmatrix}1&1&1&1\\1&\overline{\omega}_4&\overline{\omega}_4^2&\overline{\omega}_4^3\\1&\overline{\omega}_4^2&\overline{\omega}_4^4&\overline{\omega}_4^6\\1&\overline{\omega}_4^3&\overline{\omega}_4^6&\overline{\omega}_4^9\end{pmatrix}\begin{pmatrix}f_0\\f_1\\f_2\\f_3\end{pmatrix}$$

下面交换右端矩阵的奇数列和偶数列, 将上述右端改写为如下等价形式:

$$\begin{pmatrix}c_0\\c_1\\c_2\\c_3\end{pmatrix}=\frac{1}{4}\begin{pmatrix}1&1&1&1\\1&\overline{\omega}_4^2&\overline{\omega}_4&\overline{\omega}_4^3\\1&\overline{\omega}_4^4&\overline{\omega}_4^2&\overline{\omega}_4^6\\1&\overline{\omega}_4^6&\overline{\omega}_4^3&\overline{\omega}_4^9\end{pmatrix}\begin{pmatrix}f_0\\f_2\\f_1\\f_3\end{pmatrix}$$

记

$$F_4=\begin{pmatrix}1&1&1&1\\1&\overline{\omega}_4&\overline{\omega}_4^2&\overline{\omega}_4^3\\1&\overline{\omega}_4^2&\overline{\omega}_4^4&\overline{\omega}_4^6\\1&\overline{\omega}_4^3&\overline{\omega}_4^6&\overline{\omega}_4^9\end{pmatrix}, \quad F_2=\begin{pmatrix}1&1\\1&\overline{\omega}_4^2\end{pmatrix}=\begin{pmatrix}1&1\\1&\overline{\omega}_2\end{pmatrix} \quad D_2=\begin{pmatrix}1&0\\0&\overline{\omega}_4\end{pmatrix}$$

有:

$$F_4 f = \begin{pmatrix} F_2 & D_2 F_2 \\ F_2 & -D_2 F_2 \end{pmatrix} \begin{pmatrix} f_{even} \\ f_{odd} \end{pmatrix}$$

从而对于一般的情况 (假设N为偶数), 容易知道, 仍有:

$$F_N f = \begin{pmatrix} F_{\frac{N}{2}} & D_{\frac{N}{2}} F_{\frac{N}{2}} \\ F_{\frac{N}{2}} & -D_{\frac{N}{2}} F_{\frac{N}{2}} \end{pmatrix} \begin{pmatrix} f_{even} \\ f_{odd} \end{pmatrix} = \begin{pmatrix} I & D_{\frac{N}{2}} \\ I & -D_{\frac{N}{2}} \end{pmatrix} \begin{pmatrix} F_{\frac{N}{2}} f_{even} \\ F_{\frac{N}{2}} f_{odd} \end{pmatrix}$$

注意, I 和 D_N 均为对角矩阵, 和向量作矩阵乘法每次只需要 N 次的计算量

算法时间复杂度

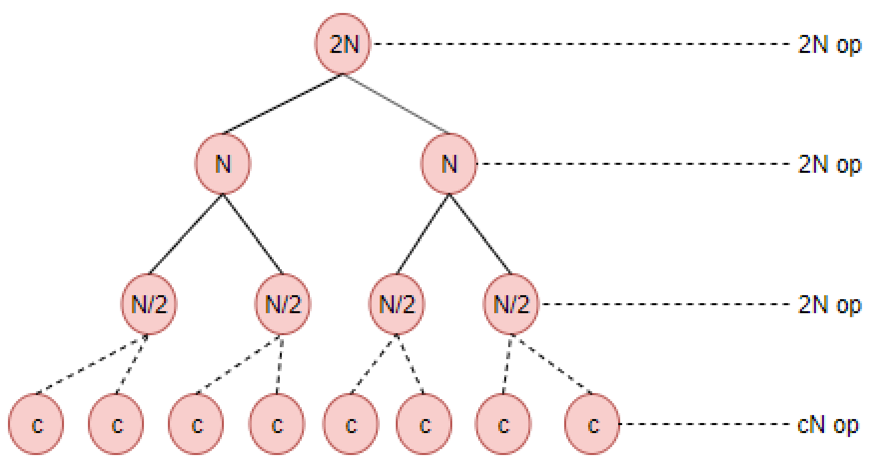
在计算时, 可以通过补 0 的方式将 N 凑为 2 的幂次方, 从而使得右边能够递归计算下去

上述算法的递归表达式如下:

$$T(N) = 2T(\frac{N}{2}) + 2N$$

从而 $T(N) = Nlog_2 N$

可以写为递归树理解如下:



上述表达式可利用递归树画出, 如图: 树的深度为 $log_2 N$ 每层需要进行行 $O(N)$ 的操作, 由此容易知道总的时间复杂度为 $O(Nlog_2 N)$

FFT 的代码实现

Numpy 模块下的复数运算

为了实现 FFT 需要使用复数操作

In [1]:
```
import numpy as np
```

In [2]:
```
# 创建复数元素数组

a = np.array([1 + 2j, 1 - 3j])
a
```

Out[2]:
```
array([1.+2.j, 1.-3.j])
```

In [3]:
```
print(type(a[0]))
```

```
<class 'numpy.complex128'>
```

In [4]:
```
# 快速创建复数元素的数组

b = np.zeros(3, dtype = np.complex128)
b
```

Out[4]:
```
array([0.+0.j, 0.+0.j, 0.+0.j])
```

In [5]:
```
# 作用在复数数组上的运算

c = np.sqrt(a)
c
```

Out[5]:
```
array([1.27201965+0.78615138j, 1.44261527-1.03977826j])
```

In [6]:
```
np.sqrt(-1+0j)
```

Out[6]:
```
1j
```

In [7]:
```
w = np.exp(complex(0, np.pi))
w
```

Out[7]:
```
(-1+1.2246467991473532e-16j)
```

In [8]:
```
print(w.real, w.imag)
```

```
-1.0 1.2246467991473532e-16
```

FFT 的递归实现

In [9]:
```python
def fft_rcs(a, m):
    if m == 1:
        return a
    a0 = a[::2]
    a1 = a[1::2]
    a0ft = fft_rcs(a0, m >> 1)
    a1ft = fft_rcs(a1, m >> 1)
    wn = np.complex(np.cos(2 * np.pi / m), -np.sin(2 * np.pi / m))
    res = np.zeros(m, dtype = np.complex128)                # 这里注意必须重新开辟内存, 不能直接对原数组 a 进行修改, python list 为可变对象
    for k in range(m >> 1):
        res[k] = a0ft[k] + (wn ** k) * a1ft[k]
        res[k + (m >> 1)] = a0ft[k] - (wn ** k) * a1ft[k]   # 这里位运算操作注意 位运算优先级要比普通运算低, 如果 m // 2可省略括号
    return res
```

In [10]:
```python
m = 8
f = lambda x: np.sin(5 * x + 1)
a = f(np.linspace(0, 2 * np.pi, m, dtype = np.complex128))
c = fft_rcs(a, m)
c
```

Out[10]:
```
array([0.84147098+0.j        , 0.8615517 +0.59157261j,
       1.1278941 +3.49511317j, 0.61694382-1.13487072j,
       0.67751766+0.j        , 0.61694382+1.13487072j,
       1.1278941 -3.49511317j, 0.8615517 -0.59157261j])
```

FFT 的高效实现

$numpy$下FFT的使用

In [11]:
```python
from numpy import fft
```

In [12]:
```python
fft.fft(a, m)
```

Out[12]:
```
array([0.84147098+0.j        , 0.8615517 +0.59157261j,
       1.1278941 +3.49511317j, 0.61694382-1.13487072j,
       0.67751766+0.j        , 0.61694382+1.13487072j,
       1.1278941 -3.49511317j, 0.8615517 -0.59157261j])
```

和上面的结果是一致的.