快速傅里叶变换

2025 年 7 月 9 日

目录

1 前言

一般来说，FFT是和信号处理密不可分的东西，在学习FFT之前需要学习大量的数字信号处理的前置知识。它是用来计算离散傅里叶变换（DFT）的工具，DFT是FFT的基础，但这不在本文的讨论范围内，我们将从代数的角度出发，来看一下FFT与众不同的东西。

2 FFT解决什么问题

为了回答这个问题，不妨考虑两个2次多项式$A(x)$和$B(x)$，并定义$A(x) = x^2+4$和$B(x) = 3x^2+x+1$。要计算$C(x) = A(x)\cdot B(x)$ 正常的方法是把每个项都乘起来，也就是：

$$\begin{aligned}C(x) =&A(x)B(x)\\ =&(x^2+4)(3x^2+x+1)\\ =&(x^2+4)3x^2+(x^2+4)x+(x^2+4)\\ =&3x^4+x^3+13x^2+4x+4\end{aligned}$$

如果要写成代码，则上述算式给出将A(x)的每个项都和B(x)的每个项配对，这至少需要2个for循环，因此它的时间复杂度为$O(n^2)$

在这有一种更快速的算法FFT，它的时间复杂度是$O(N\log N)$，它是用来解决多项式乘积问题的，它的思路如下：

一个n次多项式函数可以由$n+1$个点决定，那么那么我们任取几个点$1,2,3,4,5$，$A(x)$可以由坐标$(1,A(1)),(2,A(2)),(3,A(3))\cdots$决定，同样的$B(x)$也是。那么我们给出表格：

x:	1	2	3	4	5
A(x)	5	8	13	20	29
B(x)	5	15	31	53	81
C(x)	25	120	403	1060	2349

那么$C(x)$也可以用坐标的形式去决定，但不同的是，$C(x)$是次数为4的多项式，他需要5个点表示，即在$A(x)$和$B(x)$取得5个点，因为$C=A\cdot B$，所以它的坐标就是$(x,C(x))=(x,A(x)B(x))$ 。即

$$(1,25),(2,120),(3,403),(4,1060),(5,2349)$$

这是通过$(1,5*5),(2,8*15)$这样的形式得到的。

可以简单的验算这是成立的。因此，在这种方法下，它的计算量只有$O(n)$。

上述的这种通过坐标计算多项式乘积方式叫值表示法，我们现在知道了这种表示法比直接计算多项式乘积要快，那么我们有一些计划

1 将多项式转化为坐标表示

2 取一些点

3 做乘法得到新多项式的坐标

4 想办法把值多项式还原为多项式。

在这些步骤中，我们缺少了两个工具，1是如何把系数变成值来表示。以及反过来，如何把值表示转为多项式的系数。

在讲述之前，我们必须解决一些理论性的问题。

3 数学上的证明

3.1 多项式插值的唯一性

给定一个n次多项式$P(x)$和经过n次多项式上的$n+1$个点，则$P(x)$这个多项式的系数是唯一确定的。

证明： 令$P(x)$是n次多项式

$$P(x)=p_0+p_1x+\cdots+p_nx^n$$

取$n+1$个经过$P(x)$的点

$$\{(x_0,P(x_0)),(x_1,P(x_1)),\cdots,(x_n,P(x_n))\}$$

那么这$n+1$个点还能这样表示：

$$P=M*p\Rightarrow\begin{bmatrix}P\left(x_0\right)\\P\left(x_1\right)\\\vdots\\P\left(x_n\right)\end{bmatrix}=\begin{bmatrix}1&x_0&x_0^2&\cdots&x_0^n\\1&x_1&x_1^2&\cdots&x_1^n\\\vdots&\vdots&\vdots&\ddots&\vdots\\1&x_n&x_n^2&\cdots&x_n^n\end{bmatrix}\begin{bmatrix}p_0\\p_1\\\vdots\\p_n\end{bmatrix}$$

我们的目的是证明$p_0,\cdots,p_n$是唯一的，只需要计算矩阵M的行列式$|M|$是否为0即可。注意到这其实是范德蒙德行列式的转置，从而$|M|=|M^T|$。$|M|=0$当且仅当M中元素$(x_0,x_1,\cdots,x_n)$有一对是相同的，但这不可能。所以p向量组是唯一确定的，也就是系数是被唯一确定的。

3.2 如何为多项式选择方便的x用来计算

这是第二个问题，我们在第二节描述的算法选用的x比较简单，也就是从1到5。当点数较多的时候，这样的方法不一定有用，因为这还是让计算变得复杂，那是否有这么一些选择，当计算出某个点的坐标时，通过某种关系可以立马得到另一个坐标的值。一个简单的例子，对函数$f(x)=x^2$，我们当然可以如上随便取点，但要是对于特定的点$-2,-1,1,2$，注意到它们的坐标是$(-2,4),(-1,1),(1,1),(2,4)$，利用这种对称，我们可以立马得到跟1，2相反的值。这其中的奥秘非常简单，因为$f(x)$是偶函数，对于$f(x)$有$f(x)=f(-x)$，那么还有一种可能，若$f(x)$是奇函数，同样的，$f(x)$是奇函数则有$f(-x)=-f(x)$，我们依然可以快速得到一些想要的值。只是在前面需要加一个负号而已。藉此，对于奇偶函数，我们并不需要去计算所有的点，只需要计算一半的点即可。

另一个问题是，这对多项式也有效吗？现在给定5次多项式$P(x)$，那么可以这样做分解

$$\begin{aligned}P(x)=&a_5x^5+a_4x^4+a_3x^3+a_2x^2+a_1x+a_0\\=&(a_4(x^2)^2+a_2x^2+a_0)+x(a_5(x^2)^2+a_3x^2+a_1)\\=&P_e(x^2)+xP_o(x^2)\end{aligned}$$

其中,P_e是偶次项组成的多项式，xP_o是奇次项组成的多项式，在这种情况下，我们对多项式取点只需要取一半的值就行了。这很容易推广到一般情况下。

就像这样，对$P_e(x^2)$，展开之后可以有

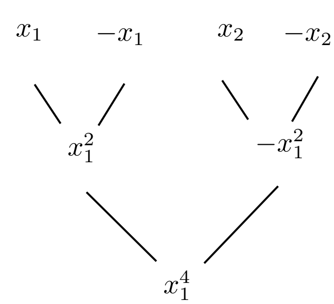
$$
\begin{aligned}
P_e(x^2) =& a_4(x^2)^2 + a_2x^2 + a_0 \\
=& a_4y^2 + a_2y + a_0 \qquad \text{用y代替}x^2\text{做视觉上的简化} \\
=& (a_4y^2 + a_0) + (ya_2) \\
=& (a_4(x^2)^2 + a_0) + (x^2)a_2 \qquad \text{变回原来的样子} \\
=& P_{e_1}(x^2) + x^2P_{o_1}(x^2)
\end{aligned}
$$

我们每次都能把子式重新做分割，很容易联想到递归，因此，这种递归式的数量不超过$\log n$个子式，因为每一层都递归$n/2^k, k \in 1, 2, \cdots$。所以计算一个式子花费$O(\log n)$的复杂度，加上$n$个点就是$O(n \log n)$。[1]

3.3 选择什么样的根？

上述式子有个漏洞，我相信聪明的朋友已经发现了，采用相对的根是好的选择，但分解为子式之后，实际上计算的是它们的平方，这个时候所有负的根也会变成正的了。计算不再方便。所以，我们希望有一种选择可以满足平方之后依然是相对的状态。一个自然的想法是复数i，注意$i^2 = -1$。那么我们如何选择?

假设有一个3次多项式，那么我们需要4个点来求值，并且这些点应该是正负成对出现的，即$x_1, -x_1, x_2, -x_2$，那么递归一层之后，要计算的便是x_1^2, x_2^2，那它也应该满足是正负成对的，即$x_1^2 = -x_2^2$，最后，再递归一层之后，将剩下x_1^4。



我们当然可以假设$x_1 = 1$，那么另一个就是$-x_1 = -1$。递归后得到的$x_1^2 = 1$，此时为了满足正负成对的要求，则$-x_1^2 = -1$。之后，就有$x_1^4 =$

[1]可以使用换底公式得到，但这点差距其实无所谓

1，那么剩下的x_2为了满足$x_2^2=-1$，则可以考虑的值只有$x_2=i$，因此我们需要的4个点就是$1,-1,i,-i$

因此，若存在一个n-1次多项式，则需要n个根，满足$n=2^k,k\in\mathbb{Z}$。这个选择恰好就是单位根

3.3.1 定义：单位根

我们说ω_n是n次单位根，若它满足$\omega_n^n=1$

它有如下性质：

$$\omega_n^k=e^{(2\pi ik)/2n}=(\omega^{(2\pi i/n)})^k,\omega^n=1$$

其中$\omega_n^{2\pi i/n}$是n次单位根，利用欧拉公式，我们还知道

$$\omega=e^{2\pi i/n}=\cos(2\pi/n)+i\sin(2\pi/n)$$

即

$$\omega^k=\cos(2\pi k/n)+i\sin(2\pi k/n)$$

那么我们又可以推出如下4个引理：

1：$\omega_{dn}^{dk}=\omega_n^k$

2：$\omega_n^{n/2}=\omega_2=-1$

3：$(\omega_n^{k+n/2})^2=(\omega_n^k)^2=\omega_{n/2}^k$

4：$\sum_{i=0}^{n-1}(\omega_n^k)^i=0$

证明1：

$$\omega_{dn}^{dk}=e^{\frac{2\pi i(dk)}{dn}}=(e^{\frac{2\pi i}{n}})^k=\omega_n^k$$

证明2： $\omega_n^{n/2}=(e^{2\pi i/n*(n/2)})=e^{\pi i}=-1$

证明3： 首先，$\omega_n^{k+n/2}=\omega_n^k\omega_n^{n/2}=-\omega_n^k$，接着$(\omega_n^k)^2=(e^{2\pi i2/n})^k=e^{2\pi i/(n/2)*k}=\omega_{n/2}^k$

证明：4 $\sum_{i=0}^{n-1}(\omega_n^k)^i=\frac{(\omega_n^k)^n-1}{\omega_n^k-1}=\frac{1^k-1}{\omega_n^k-1}=0$

现在，我们开始编写代码

4 代码编写

4.1 程序伪代码

一般来说，我们可以通过递归的方式来解决大问题中的子问题。首先是停止条件，即拆分多项式到次数为1的时候理应停止继续拆分。我们给出一个长度为n的向量a，其中n是2的幂，我们使用$A_{[0]_k}$表示偶子式的第k层多项式。$A_{[1]_k}$是第k层的奇子式

Algorithm 1 FFT(a)

Ensure: $n = a.length$, $n = 2^k$

1: **if** n == 1 **then**
2:     **return** a
3: **end if**
4: $\omega_n = e^{2\pi i/n}$
5: $\omega = 1$
6: $a0 = (a_0, a_2, \cdots, a_{n-2})$
7: $a1 = (a_1, a_3, \cdots, a_{n-1})$
8: $y0 = FFT(a0)$
9: $y1 = FFT(a1)$
10: **for** $k = 0$ **to** n/2-1
11:     $y_k = y_{[0]_k} + \omega y_{[1]_k}$
12:     $y_{k+(n/2)} = y_{[0]_k} - \omega y_{[1]_k}$
13:     $\omega = \omega\omega_n$
14: **return** y

我们的fft执行流程如下：第2-3行表示处理基本情况，一个元素的DFt是其自身，因为一次多项式是$y_o - a_0\omega_1^0 = a_0 \cdot 1 = a_0$

接着，第6-7行定义偶数和奇数次多项式A_0, A_1的系数向量a_0, a_1，接着，8-9行递归的执行DFT，对于$k = 0, 1, \cdots, n/2 - 1$有

$$y_{[0]_k} = A_{[0]}(\omega_{n/2}^k)$$
$$y_{[1]_k} = A_{[1]}(\omega_{n/2}^k)$$

当然，也可以由定义3.3.1中的引理3推出$\omega_{n/2}^{k}=\omega_{n}^{2k}$，就有

$$
\begin{aligned}
y_{[0]_k} &= A_{[0]}(\omega_n^{2k}) \\
y_{[1]_k} &= A_{[1]}(\omega_n^{2k})
\end{aligned}
$$

对11-12行，我们综合了递归$\mathrm{DFT}_{n/2}$的结果，对$y_0, y_1, \cdots, y_{n/2-1}$，第11行给出

$$
\begin{aligned}
y_k =& y_{[0]_k} + \omega_n^k y_{[1]_k} \\
=& A_{[0]}(\omega_n^{2k}) + \omega_n^k A_{[1]}(\omega_n^{2k}) \\
=& A(\omega_n^k)
\end{aligned}
$$

其次，对$y_{n/2}, y_{n/2+1}, \cdots, y_{n-1}$，令$k=0,1,\cdots,n/2-1$。利用单位根性质，$\omega^{k+(n/2)_n}=-\omega_n^k$，就有

$$
\begin{aligned}
y =& y_{[0]_k} - \omega_n^k y_{[1]_k} \\
=& y_{[0]_k} + \omega_n^{k+(n/2)} y_{[1]_k} \\
=& A_{[0]}(\omega_n^{2k+n}) + \omega_n^{k+(n/2)} A_{[1]}(\omega_n^{2k+n}) \\
=& A(\omega_n^{k+(n/2)})
\end{aligned}
$$

4.2 实际代码编写

Listing 1: FFT(x)

```
void fft(complex *x, int n){
 int(n <= 1) return;
 complex *even = (complex *)malloc(n/2 * sizeof(complex));
 complex *odd = (complex *)malloc(n/2 * sizeof(complex));

 int j = 0;
 for(int i = 0; i< n;i+=2 ){
  even[j] = x[i];
  odd[j]  = x[i+1];
  j++
 }
 fft(even, n/2);
 fft(odd,n/2);
 for(int k = 0; k < n/2;k++){
  complex t - cexp(-2 * PI * I * k/n) * odd[k];
  x[k] = even[k] + t;
  x[k+n/2] = even[k] - t;
 }
 free(even);
 free(odd);
}
```

5 附录

5.1 O(n^2)的多项式乘法的代码：

原型：

Listing 2: 多项式乘法原型

```
int *PolyMult(const int* A_coeff, int A_size, const int*
    B_coeff,int B_size){
 int result_size = A_size + B_size -1;
 int* result = (int*)calloc(*result_size, sizeof(int)); //
     初始化为0
 for(int i; i< A_size;i++){
  for(int j; j< B_size;b++){
   result[i+j] = A_coeff[i] * B_coeff[j];
  }
 }
 return result;
}
```

该函数有4个参数，多项式A的系数，系数的个数，还有B的系数和系数个数，并返回多项式$A(x) \cdot B(x)$的系数

5.2 简单的FFT示例+应用

Listing 3: Full FFT

```
#include <stdio.h>
#include <stdlib.h>
#include <complex.h>
#include <math.h>
#include <string.h>

#define PI 3.1415926


void fft(complex double *x, int n) {
 if (n <= 1) return;

 complex double *even = (complex double *)malloc(n / 2 *
     sizeof(complex double));
 complex double *odd = (complex double *)malloc(n / 2 *
     sizeof(complex double));

 int j = 0;
 for (int i = 0; i < n; i += 2) {
  even[j] = x[i];
  odd[j] = x[i + 1];
  j++;
 }

 fft(even, n / 2);
 fft(odd, n / 2);

 for (int k = 0; k < n / 2; k++) {
  complex double t = cexp(2 * PI * I * k / n) * odd[k];
  x[k] = even[k] + t;
  x[k + n / 2] = even[k] - t;
 }

 free(even);
 free(odd);
}
```

```


char* complex_to_string(complex double z) {
 static char buffer[100];
 double real = creal(z);
 double imag = cimag(z);

 if (imag == 0) {
  sprintf(buffer, "%.4f", real);
 } else if (real == 0) {
  if (imag == 1) {
   strcpy(buffer, "i");
  } else if (imag == -1) {
   strcpy(buffer, "-i");
  } else {
   sprintf(buffer, "%.4fi", imag);
  }
 } else {
  if (imag > 0) {
   if (imag == 1) {
    sprintf(buffer, "%.4f + i", real);
   } else {
    sprintf(buffer, "%.4f + %.4fi", real, imag);
   }
  } else {
   if (imag == -1) {
    sprintf(buffer, "%.4f - i", real);
   } else {
    sprintf(buffer, "%.4f - %.4fi", real, fabs(imag));
   }
  }
 }

 return buffer;
}

void print_frequency_domain(complex double *data, int n) {
 printf("result:\n");
 for (int i = 0; i < n; i++) {
```

```c
    printf("X[%d] = %s\n", i, complex_to_string(data[i]));
  }
}


void print_polynomial(complex double *coeffs, int degree) {
  printf("poly: ");
  for (int i = 0; i <= degree; i++) {
    if (i > 0) printf(" + ");
    printf("%s x^%d", complex_to_string(coeffs[i]), i);
  }
  printf("\n");
}

int main() {
  //  A(x) = 1 + 2x + 3x^2
  // B(x) = 4 + 5x
  double A_coeffs[] = {1.0, 2.0, 3.0};
  double B_coeffs[] = {4.0, 5.0};
  int A_degree = sizeof(A_coeffs) / sizeof(A_coeffs[0]) - 1;
  int B_degree = sizeof(B_coeffs) / sizeof(B_coeffs[0]) - 1;


  int product_degree = A_degree + B_degree;
  int n = 1;
  while (n <= product_degree) n *= 2; //find numbers of nth roots

  complex double *A = (complex double *)calloc(n, sizeof(complex double));
  complex double *B = (complex double *)calloc(n, sizeof(complex double));

  for (int i = 0; i <= A_degree; i++) A[i] = A_coeffs[i];
  for (int i = 0; i <= B_degree; i++) B[i] = B_coeffs[i];

  printf("A: ");
  print_polynomial(A, A_degree);
  printf("B: ");
```

```c
    print_polynomial(B, B_degree);

    complex double *A_fft = (complex double *)malloc(n * sizeof
        (complex double));
    complex double *B_fft = (complex double *)malloc(n * sizeof
        (complex double));

    for (int i = 0; i < n; i++) {
     A_fft[i] = A[i];
     B_fft[i] = B[i];
    }

    fft(A_fft, n);
    fft(B_fft, n);

    printf("\nA_FFT:\n");
    print_frequency_domain(A_fft, n);

    printf("\nB_FFT:\n");
    print_frequency_domain(B_fft, n);

    complex double *C_fft = (complex double *)malloc(n * sizeof
        (complex double));
    for (int i = 0; i < n; i++) {
     C_fft[i] = A_fft[i] * B_fft[i];
    }

    printf("\nA*B_FFT:\n");
    print_frequency_domain(C_fft, n);

    free(A);
    free(B);
    free(A_fft);
    free(B_fft);
    free(C_fft);

    return 0;
  }
```

结果：

```
A: poly: 1.0000 x^0 + 2.0000 x^1 + 3.0000 x^2
B: poly: 4.0000 x^0 + 5.0000 x^1

A_FFT:
result:
X[0] = 6.0000
X[1] = -2.0000 + 2.0000i
X[2] = 2.0000
X[3] = -2.0000 - 2.0000i

B_FFT:
result:
X[0] = 9.0000
X[1] = 4.0000 + 5.0000i
X[2] = -1.0000
X[3] = 4.0000 - 5.0000i

A*B_FFT:
result:
X[0] = 54.0000
X[1] = -18.0000 - 2.0000i
X[2] = -2.0000
X[3] = -18.0000 + 2.0000i
```