1 離散畳み込み (Discrete Convolution)

1.1 離散フーリエ変換 (DFT)

離散畳み込みを理解するための、第一歩として、離散フーリエ変換を説明する。

定義 1 $b = (b_0, \dots, b_{2M-2}) \in \mathbb{C}^{2M-1}$ に対して、$a = \mathcal{F}(b) \in \mathbb{C}^{2M-1}$ を

$$a_k = \mathcal{F}(b) := \sum_{j=0}^{2M-2} b_j e^{-2\pi \mathrm{i}\left(\frac{jk}{2M-1}\right)}, \quad |k| < M$$

とし、これを**離散フーリエ変換 *(DFT)*** と呼ぶ。

1.2 逆離散フーリエ変換 (IDFT)

定義 2 $a = (a_k)_{|k|<M} = (a_{-M+1}, \dots, a_{M-1}) \in \mathbb{C}^{2M-1}$ に対して、$b = \mathcal{F}^{-1}(a) \in \mathbb{C}^{2M-1}$ を

$$\begin{aligned} b_j &= \mathcal{F}^{-1}(a) \\ &:= \sum_{k=-M+1}^{M-1} a_k e^{2\pi \mathrm{i}\left(\frac{jk}{2M-1}\right)} \quad j = 0, \dots, 2M-2 \end{aligned}$$

とし、**逆離散フーリエ変換 *(IDFT)*** と呼ぶ。

注意 一般的な DFT/IDFT はスケーリング係数をつけた形で定義されることが多い。この点で上の定義は一般的な定義と違う。

1.3 離散畳み込みのアルゴリズム

u_1, u_2 を周期 L 、変数 t に関する周期関数とし、$\omega = \frac{2\pi}{L}$ とする。このとき、u_1, u_2 をフーリエ級数展開すると、

$$\begin{aligned} u_1(t) &= \sum_{k\in\mathbb{Z}} a_k^{(1)} e^{\mathrm{i}k\omega t}, \quad a^{(1)} = (a_k^{(1)})_{k\in\mathbb{Z}} \\ u_2(t) &= \sum_{k\in\mathbb{Z}} a_k^{(2)} e^{\mathrm{i}k\omega t}, \quad a^{(2)} = (a_k^{(2)})_{k\in\mathbb{Z}}. \end{aligned}$$

そして、これらの周期関数の積は、

$$u_1(t)u_2(t) = \sum_{k\in\mathbb{Z}} (a^{(1)} * a^{(2)})_k e^{\mathrm{i}k\omega t}$$

と表される。ここで $(a^{(1)} * a^{(2)})_k$ を**離散畳み込み**といい、

$$(a^{(1)} * a^{(2)})_k = \sum_{\substack{k_1+k_2=k \\ k_1,k_2\in\mathbb{Z}}} a_{k_1}^{(1)} a_{k_2}^{(2)}, \quad k \in \mathbb{Z}$$

と表される。

さらに、数値計算への応用を意識すると、u_1, u_2 のような（有限モードのフーリエ級数で表される）周期関数が p 個 $(p \in \mathbb{N})$ あったとき、

$$\begin{aligned} u_i(t) &= \sum_{|k|<M} a_k^{(i)} e^{\mathrm{i}k\omega t}, \\ a^{(i)} &= (a_k^{(i)})_{|k|<M} \quad i = 1, \cdots, p \quad M \in \mathbb{Z}. \end{aligned}$$

離散畳み込みはこれらの周期関数の積

$$u_1(t) \cdots u_p(t) = \sum_{|k|\le p(M-1)} (a^{(1)} * \cdots * a^{(p)})_k e^{\mathrm{i}k\omega t}$$

を表す事になる。ここで

$$(a^{(1)} * \cdots * a^{(p)})_k = \sum_{\substack{k_1+\cdots+k_p=k, \\ |k|\le p(M-1), \\ |k_1|,\cdots,|k_p|<M}} a_{k_1}^{(1)} \cdots a_{k_p}^{(p)}$$

と表される。

1.3.1 畳み込みの定理

畳み込みを離散フーリエ変換したものは、それぞれのフーリエ係数の離散フーリエ変換の積になる。

$$\begin{aligned} \mathcal{F}(a^{(1)} * \cdots * a^{(p)}) &= \mathcal{F}(a^{(1)}) \hat{*} \cdots \hat{*} \mathcal{F}(a^{(p)}) \\ &= b^{(1)} \hat{*} \cdots \hat{*} b^{(p)}. \end{aligned}$$

ここで $b^{(1)} \hat{*} \cdots \hat{*} b^{(p)}$ におけるベクトル同士の積は、要素毎の積を表す。

1.3.2 離散フーリエ変換を使った畳み込みの計算方法 (FFT アルゴリズム)

実際の畳み込みの計算方法について説明する。

周期 L、変数 t の周期関数 $u_i(t)$ が有限項のフーリエ級数

$$u_i(t) = \sum_{|k|<M} a_k^{(i)} e^{\mathrm{i}k\omega t}, \quad a^{(i)} = (a_k^{(i)})_{|k|<M}$$

で表されているとする。ここで $\omega = \frac{2\pi}{L}$ とする。このとき、p 個の関数の積

$$u_1(t)\cdots u_p(t) = \sum_{|k|\le p(M-1)} c_k e^{\mathrm{i}k\omega t}$$

を表現するフーリエ係数 $(c_k)_{|k|\le p(M-1)}$ を以下の計算方法により求める。

入力: $a^{(i)} = (a_k^{(i)})_{|k|<M} \in \mathbb{C}^{2M-1} \quad (i = 1, \cdots, p)$

step1: エイリアシングエラーを防ぐために、入力された値 $a^{(i)}$ の両脇に $(p-1)M$ 個の 0 を付け加える。これを $\tilde{a}^{(i)}$ と書く。

$$\tilde{a}^{(i)} = (\underbrace{0,\cdots,0}_{(p-1)M\text{ 個}}, \underbrace{a_{-M+1}^{(i)},\cdots,a_{M-1}^{(i)}}_{2M-1\text{ 個}}, \underbrace{0,\cdots,0}_{(p-1)M\text{ 個}}) \in \mathbb{C}^{2pM-1}$$

step2: step1 で得た値 $\tilde{a}^{(i)}$ に対して逆離散フーリエ変換を行う。変換した後の値を $\tilde{b}^{(i)}$ と置く。

$$\tilde{b}^{(i)} = \mathcal{F}^{-1}(\tilde{a}^{(i)}) \in \mathbb{C}^{2pM-1}$$

step3: $(\tilde{b}^{(1)} \hat{*} \cdots \hat{*} \tilde{b}^{(p)})$ を計算する。上記の畳み込みの定理と同じく、このベクトル同士の積は、要素毎の積を表す。

$$(\tilde{b}^{(1)} \hat{*} \cdots \hat{*} \tilde{b}^{(p)})_j = \tilde{b}_j^{(1)} \cdots \tilde{b}_j^{(p)}, \quad j = 0, \cdots, 2pM-2$$

step4: step3 で求めた $(\tilde{b}^{(1)} \hat{*} \cdots \hat{*} \tilde{b}^{(p)})$ に対して、離散フーリエ変換を行い、得た値を $2pM-1$ で割る。

$$c_k = \frac{1}{2pM-1}\mathcal{F}_k(\tilde{b}^{(i)} \tilde{*} \cdots \tilde{*} \tilde{b}^{(p)}) \quad |k| \le p(M-1)$$

求めた c_k のうち、実際に必要なのは両脇の $p-1$ 個を取り除いた $|k| \le p(M-1)$ 個である。

$$c = (\underbrace{0,\cdots,0}_{(p-1)M\text{ 個}}, \underbrace{a_{-M+1}^{(i)},\cdots,a_{M-1}^{(i)}}_{2M-1\text{ 個}}, \underbrace{0,\cdots,0}_{(p-1)M\text{ 個}}) \in \mathbb{C}^{2pM-1}$$

出力: $c = (c_k)_{|k|\le p(M-1)} \in \mathbb{C}^{2p(M-1)+1}$

下記のコードが離散畳み込みを実装した関数になる。また、本論文に記載されているコードは、2 段組の都合上、適宜改行されている。

```
function powerconvfourier(a::Vector{Complex{T}}
                                        ,p) where T
    M = Int((length(a)+1)/2)
    N = (p-1)*M

    # 1. Padding zeros: size(ta) = 2pM-1
    ta = [zeros(N,1);a;zeros(N,1)]

    # 2. IFFT of ta
    tb = ifft(ifftshift(ta))
    tb^p = tb.^p # 3. tb*^tb
    c^p = fftshift(fft(tb^p))*(2.0*p*M-1)^(p-1)

    # return (truncated, full) version
    return c^p[N+1:end-N], c^p[p:end-(p-1)]
end
```

1.4 離散畳み込みの精度保証付き数値計算

これから、離散畳み込みの精度保証を行う。離散畳み込みのアルゴリズムには、FFT が含まれるため、まず、FFT の精度保証を行うための関数を定義する。コードについては付録に記載する。

verifyfft は、要素数が 2 のべき乗の場合しか実行できないので、step1 の padding の部分で要素数を調整する。

$$\tilde{a} = (\underbrace{0,\cdots,0}_{L\text{ 個}}, \underbrace{0,\cdots,0}_{N=(p-1)M\text{ 個}}, \underbrace{a_{-M+1},\cdots,a_{M-1}}_{2M-1\text{ 個}}, \underbrace{0,\cdots,0}_{N\text{ 個}}, \underbrace{0,\cdots,0}_{L-1\text{ 個}}) \in \mathbb{C}^{2pM-2+2L}$$

$$c = (\underbrace{0,\cdots,0}_{L\text{ 個}}, \underbrace{0,\cdots,0}_{(p-1)\text{ 個}}, \underbrace{a_{-p(M-1)},\cdots,a_{p(M-1)}}_{2p(M-1)+1\text{ 個}}, \underbrace{0,\cdots,0}_{(p-1)\text{ 個}}, \underbrace{0,\cdots,0}_{L-1\text{ 個}}) \in \mathbb{C}^{2pM-2+2L}$$

下記のコードは、区間演算とベクトル、両方の型に対応できるように、多重ディスパッチを利用する離散畳み込みの関数である。

```
function powerconvfourier(a::Vector{Complex{Interval{T}}}
                                        ,p) where T
    M = Int((length(a)+1)/2) # length(a) = 2M-1
    N = (p-1)*M
    ia = map(Interval, a)

    length_ia = 2*p*M-1
    length_ia_ext = nextpow(2,length_ia)# 2pM-2+2L

    L = Int((length_ia_ext - length_ia + 1)/2)

    # step.1 : padding (p-1)M + L zeros for each sides
    ia_ext = map(Complex{Interval},zeros(length_ia_ext))
    ia_ext[L+N+1:end-L-N+1] = ia  #\tilda{a}

    # step.2 : inverse fft
    #sign = -1 : ifft
    ib_ext = verifyfft(ifftshift(ia_ext), -1)

    # step.3 : power p elementwisely
```

```
    ib_ext^p = ib_ext.^p

    # step.4 : fft with rescaling
    #sign = 1 : fft
    ic_ext^p = fftshift(verifyfft(ib_ext^p, 1))
                  * length_ia_ext^(p-1)

    # return ic_ext^p,ic_ext^p
    # return (truncated, full) version
    return ic_ext^p[L+N+1:end-N-L+1]
                          , ic_ext^p[L+p:end-(L+p-2)]
end
```