Ministerul Educatiei din R. Moldova

Universitatea de Stat din Moldova

Facultate de Matematica si Informatica

Dare de seama la tema:

Transformare discreta Fourier (DFT)

si Transformarea rapida Fourier (FFT)

Profesor: N. Objelean

Masterand: Stinca Erast

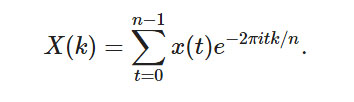
Chisinau 2017

Introducere

Transforamrea discreta Fourier (DFT) este un algoritm de baza si foarte versatil pentru procesarea semnalelor digitale (DSP). DFT este o operație care se aplică unei funcții complexe și produce o altă funcție complexă care conține aceeași informație ca funcția originală, dar reorganizată după [frecvențele](https://ro.wikipedia.org/wiki/Frecven%C8%9B%C4%83) componente. De exemplu, dacă funcția inițială este un semnal dependent de [timp](https://ro.wikipedia.org/wiki/Timp), transformata sa Fourier descompune semnalul după frecvență și produce un spectru al acestuia. Același efect se obține dacă funcția inițială are ca argument poziția într-un [spațiu](https://ro.wikipedia.org/wiki/Spa%C8%9Biu) uni- sau [multidimensional](https://ro.wikipedia.org/wiki/Dimensiune), caz în care transformata Fourier relevă spectrul uni- sau multidimensional al frecvențelor spațiale care alcătuiesc funcția de intrare.

Vom implementa acest algoritm de la zero. Pentru aceasta vom utiliza limbajul de programare Java.

Functia DFT mapeaza vectorul n de numere complexe la alt vector n de numere complexe. Utilizand indexarea 0-based, **x(t)** reprezinta elemental **t** a vectorului de intrare si **X(k)** reprezinta elementul **k** al vectorului de iesire. Atunci formula de baza DFT este:



Interpretarea este ca vectorul ***x*** reprezinta nivelul semnalului la diverse intervale de timp, si vectorul ***X*** reprezinta nivelul semnalului la diverse frecvente. Ce spune formula este ca nivelul semnalului la frecventa ***k*** este egala cu suma {la nivelul semnalului la orice timp t inmultita cu un exponential complex}.

Prezentarea codului

Cu cunostintele de baza a notatiei sigma, numere complexe si limbajul de programare Java, putem converti descrierea de mai sus in cod sursa. Scopul nostru va fi ghidarea prin procesul de translare pas cu pas.

Prima parte a descrierii spune ca DFT are ca parametri un vector de intrare de numere complexe ***n*** si calculeaza vectorul de iesire ***n*** de numere complexe. Vom simula numerele complexe cu o pereche de numere reale. Vectorul este o secventa de numere ce poate fi reprezentat de un array. In loc sa returnam tablourile, le vom transmite prin referinta ca argumente.

void dft(double[] inreal, double[] inimag, double[] outreal, double[] outimag) {

// Presupunem ca toate tablourile au aceiasi dimensiune

int n = inreal.length;

// Incomplet: Corpul metodei

}

Vom crea un ciclu extern in functia noastra pentru atribuirea fiecarui element de iesire.

for (int k = 0; k < n; k++) { // Pentru fiecare element de iesire

outreal[k] = ?; // Incomplet

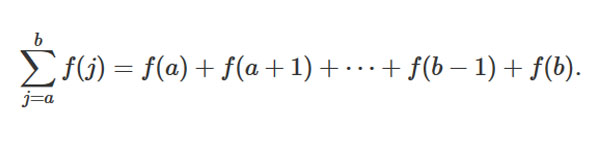
outimag[k] = ?; // Incomplet

}

Sumarea

Notatia sigma, in timp c ear putea arata initimidant, in realitate este usor de inteles.

Forma generala a unei sume finite:



In cod imperative codul arata in urmatorul fel:

double sum = 0;

for (int j = a; j <= b; j++) {

sum += f(j);

}

// valoarea sumei ‘sum’ are rezultatul dorit

Aritmetica compexa

Adaugarea de numere complexe este simpla:

(*a* + *bi*) + (*c* + *di*) = (*a* + *c*) + (*b* + *d*)*i*.

Inmultirea numerelor complexe este putin mai dificila, utilizand legea distributiei si identitatea

*i*2 = −1 obtinem:

(*a* + *bi*)(*c* + *di*)=*ac* + *adi* + *bci* – *bd* = (*ac* − *bd*) + (*ad* + *bc*)*i*.

Formula Euler ne spune ca eix = cos(x) + *i* sin(x), pentru orice numare real x. In plus functia cosinus este para, deci cos(-x) = cos(x) si functia sinus este impara, prin urmare sin(-x) = -sin(x). Prin substitutie obtinem:



Fie Re(x) partea real pentru x si Im(x) este partea imaginara pentru x. Dupa definitie,

x = Re(x) + *i* Im(x). Prin urmare:



Extindem multiplicare complexa si obtinem:



Deci fiecare termen din sigma are urmatorul cod pentru partea reala si cea imaginara:

double angle = 2 \* Math.PI \* t \* k / n;

double real = inreal[t] \* Math.cos(angle) + inimag[t] \* Math.sin(angle);

double imag = -inreal[t] \* Math.sin(angle) + inimag[t] \* Math.cos(angle);

Metoda finala de transformare Fourier va fi urmatoarea:

static void dft(double[] inreal, double[] inimag, double[] outreal, double[] outimag) {

int n = inreal.length;

for (int k = 0; k < n; k++) { // Pentru fiecare element de iesire

double sumreal = 0;

double sumimag = 0;

for (int t = 0; t < n; t++) { // Pentru fiecare element de intrare

double angle = 2 \* Math.PI \* t \* k / n;

sumreal += inreal[t] \* Math.cos(angle) + inimag[t] \* Math.sin(angle);

sumimag += -inreal[t] \* Math.sin(angle) + inimag[t] \* Math.cos(angle);

}

outreal[k] = sumreal;

outimag[k] = sumimag;

}

}

Transformarea rapida Fourier (FFT)

(Fast Fourier Transdform) FFT este o versiune mai rapida a (Discrete Fourier Transform) DFT. FFT utilizearea algoritmi inteligenti ce ne ofera un calcul mult mai rapid decat DFT.

public final class Fft {

/\*

\* Calculeaza transformarea reapida Fourier (DFT) pentru vectorul complex dat, stocand rezultatul inapoi in vector.

\* Vectorul poate avea orice lungime.

\*/

public static void transform(double[] real, double[] imag) {

if (real.length != imag.length)

throw new IllegalArgumentException("Mismatched lengths");

int n = real.length;

if (n == 0)

return;

else if ((n & (n - 1)) == 0) // Is power of 2

transformRadix2(real, imag);

else // Malgoritmi mai complicati pentru marime arbitrara

transformBluestein(real, imag);

}

/\*

\* Calculeaza trasformarea discreta Fourier inversa (IDFT) pentru vectorul complex dat, stocand rezultatul inapoi in vector.

\* Vectorul poate avea orice lungime. Aceasta transformare nu executa nici o sacalare, deci inversa nu este o inversa reala.

\*/

public static void inverseTransform(double[] real, double[] imag) {

transform(imag, real);

}

/\*

\* Calculeaza transformarea discreta Fourier (DFT) a vectorului complex dat, stocand rezultatul inapoi in vector.

\* Lungimea vectorului trebuie sa fie puterea lui 2. Utilizarea algoritmului radix-2 decimarii in timp Cooley-Tukey.

\*/

public static void transformRadix2(double[] real, double[] imag) {

// Initializarea

if (real.length != imag.length)

throw new IllegalArgumentException("Mismatched lengths");

int n = real.length;

int levels = 31 - Integer.numberOfLeadingZeros(n); // Egal cu floor(log2(n))

if (1 << levels != n)

throw new IllegalArgumentException("Lungimea nu este puterea lui 2");

double[] cosTable = new double[n / 2];

double[] sinTable = new double[n / 2];

for (int i = 0; i < n / 2; i++) {

cosTable[i] = Math.cos(2 \* Math.PI \* i / n);

sinTable[i] = Math.sin(2 \* Math.PI \* i / n);

}

// Adresare permutata inversa de bit

for (int i = 0; i < n; i++) {

int j = Integer.reverse(i) >>> (32 - levels);

if (j > i) {

double temp = real[i];

real[i] = real[j];

real[j] = temp;

temp = imag[i];

imag[i] = imag[j];

imag[j] = temp;

}

}

// radix-2 decimarea in timp Cooley-Tukey FFT

for (int size = 2; size <= n; size \*= 2) {

int halfsize = size / 2;

int tablestep = n / size;

for (int i = 0; i < n; i += size) {

for (int j = i, k = 0; j < i + halfsize; j++, k += tablestep) {

double tpre = real[j+halfsize] \* cosTable[k] + imag[j+halfsize] \* sinTable[k];

double tpim = -real[j+halfsize] \* sinTable[k] + imag[j+halfsize] \* cosTable[k];

real[j + halfsize] = real[j] - tpre;

imag[j + halfsize] = imag[j] - tpim;

real[j] += tpre;

imag[j] += tpim;

}

}

if (size == n) // prevenirea overflow in 'size \*= 2'

break;

}

}

/\*

\* Calculeaza transformarea discreta Fourier (DFT) a vectorului complex dat, stocand rezultatul inapoi in vector.

\* Vectorul poate avea orice lungime. Aceasta necesta functia de convultie, care necesita functia radix-2 FFT.

\* utilizarea algoritmului z-transformarii Bluestein's chirp.

\*/

public static void transformBluestein(double[] real, double[] imag) {

// Find a power-of-2 convolution length m such that m >= n \* 2 + 1

if (real.length != imag.length)

throw new IllegalArgumentException("Mismatched lengths");

int n = real.length;

if (n >= 0x20000000)

throw new IllegalArgumentException("Array too large");

int m = Integer.highestOneBit(n \* 2 + 1) << 1;

// Trignometric tables

double[] cosTable = new double[n];

double[] sinTable = new double[n];

for (int i = 0; i < n; i++) {

int j = (int)((long)i \* i % (n \* 2)); // O acuratete mai mare de cat j = i \* i

cosTable[i] = Math.cos(Math.PI \* j / n);

sinTable[i] = Math.sin(Math.PI \* j / n);

}

// TempVectori temporari si preprocesare

double[] areal = new double[m];

double[] aimag = new double[m];

for (int i = 0; i < n; i++) {

areal[i] = real[i] \* cosTable[i] + imag[i] \* sinTable[i];

aimag[i] = -real[i] \* sinTable[i] + imag[i] \* cosTable[i];

}

double[] breal = new double[m];

double[] bimag = new double[m];

breal[0] = cosTable[0];

bimag[0] = sinTable[0];

for (int i = 1; i < n; i++) {

breal[i] = breal[m - i] = cosTable[i];

bimag[i] = bimag[m - i] = sinTable[i];

}

// Convolutie

double[] creal = new double[m];

double[] cimag = new double[m];

convolve(areal, aimag, breal, bimag, creal, cimag);

// Postprocesarea

for (int i = 0; i < n; i++) {

real[i] = creal[i] \* cosTable[i] + cimag[i] \* sinTable[i];

imag[i] = -creal[i] \* sinTable[i] + cimag[i] \* cosTable[i];

}

}

/\*

\* Calculeaza convolutia circulara pentu vectorul real dat. Lungimea fiecarui vector trebuie sa fie aceeasi.

\*/

public static void convolve(double[] x, double[] y, double[] out) {

if (x.length != y.length || x.length != out.length)

throw new IllegalArgumentException("Mismatched lengths");

int n = x.length;

convolve(x, new double[n], y, new double[n], out, new double[n]);

}

/\*

\* Calculeaza convolutia circulara pentu vectorul real dat. Lungimea fiecarui vector trebuie sa fie aceeasi.

\*/

public static void convolve(double[] xreal, double[] ximag, double[] yreal, double[] yimag, double[] outreal, double[] outimag) {

if (xreal.length != ximag.length || xreal.length != yreal.length || yreal.length != yimag.length || xreal.length != outreal.length || outreal.length != outimag.length)

throw new IllegalArgumentException("Mismatched lengths");

int n = xreal.length;

xreal = xreal.clone();

ximag = ximag.clone();

yreal = yreal.clone();

yimag = yimag.clone();

transform(xreal, ximag);

transform(yreal, yimag);

for (int i = 0; i < n; i++) {

double temp = xreal[i] \* yreal[i] - ximag[i] \* yimag[i];

ximag[i] = ximag[i] \* yreal[i] + xreal[i] \* yimag[i];

xreal[i] = temp;

}

inverseTransform(xreal, ximag);

for (int i = 0; i < n; i++) { // Scalarea (pentru ca implementarea FFT omite aceasta)

outreal[i] = xreal[i] / n;

outimag[i] = ximag[i] / n;

}

}

}