Bakalárska práca

Daniel Kyselica

7. januára 2019

1 Komplexné čísla v Pythone

Komplexné čísla reprezentujú body v rovine. Napríklad bod $[3, 5]$ reprezentuje číslo $3+5 \cdot i$ v algebraickom zápise. V Pythone sú komplexné čísla priamo implementované. Imaginárna jednotka sa zapiseju pomocou kľúčového písmena j.

```
>>> a = 3 + 5j
```

1.1 Knižnica NumPy

Knižnica NumPy je základným balíčkom funkcií pre vedecké výpočty v Pythone. Obsahuje nástroje a základné funkcie lineárnej algebry, Fourierovú tranformáciu, náhodné púremenné.

1.2 Reprezentácie

1.2.1 Polárna tvar - Algebrický

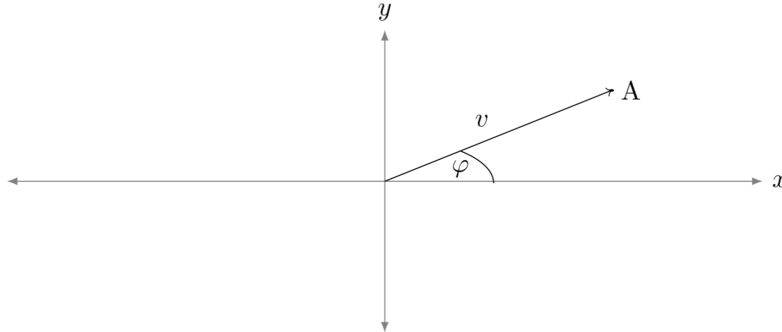
Algebraický tvar je v Pythone priamo podporovaný. Rozdiele je iba v označení imaginárnej jednotky. Číslo $a = 5 + 3i$ zapíšeme

```
>>> a = 5 + 3j
>>> a
(5+3j)
>>> a.real
5.0
>>> a.imag
3.0
>>> type(a)
<class 'complex'>
```

Imaginárna časť čísla sa zapisuje ako jeden reťazec s klúčovým písmenom j na konci. Všetky čísla v Pythone sú objektami. Komplexné číslo je objekt triedy *complex* a má atribúty *real* hovoriaci o jeho reálnej zložke a atribút *imag* hovoriaci o veľkosťi jeho imaginárnej zložky.

1.2.2 Goniometrický tvar

V goniometrickej reprezentácií je číslo $a \in \mathbb{C}$ reprezentovné dvojicou: veľkosť komplexného čísla - veľkosť vektora v zo stredu súradnicovej sústavy do bodu a a uhol φ ktorý zviera tento vektor s osou x.



Obr. 1: Bod A. vektor v,uhol φ

V Pythone nie je možnosť priamo zapisovať čísla v goniometrickom tvare. V knižnica NumPy sú implementované funkcie na získanie veľkosti uhla φ ako aj veľkosti komplexného čísla.

```
>>> import numpy as np
>>> a
(5+3j)
>>> np.angle(a)
0.5404195002705842
>>> np.angle(a, deg=True)
30.96375653207352
>>> np.abs(a)
5.8309518948453
```

Funkcia *angle* má nepovínný argument *deg*. Ak nie je určený výsledok bude v radiánoch. Po nastavení *deg=True* funkcia vráti výsledok v stupňoch.

1.2.3 Exponenciálny tvar

Exponenciálny tvar reprezentuje číslo $x \in \mathbb{C}$

$$x = ze^{i\varphi}$$

Knižnica NumPy má funkciu *exp*, ktorá akceptuje vstup v komplexných číslach. Preto sa dá využívať na konverziu čísla z exponencionálneho tvaru na polárny.

```
>>> a = 3 * np.exp(2j)
>>> a
(-1.2484405096414273+2.727892280477045j)
>>> np.angle(a)
2.0
>>> np.abs(a)
2.9999999999999996
```

1.3 Operácie nad $\mathbb{C}$

Práca s komplexnými číslami v Pythone je jednoduchá, operátory *, -, +, / sú implementované.

```
>>> a = 3 + 5j
>>> b = 8 - 12j
>>> a + b
(11-7j)
>>> a - b
(-5+17j)
>>> a * b
(84+4j)
>>> a / b
(-0.1730769230769231+0.3653846153846154j)
```

2 Fourierová transformácia

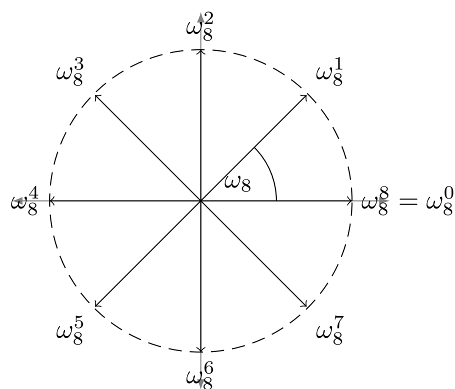
2.1 Definícia

Diskrétna Fourierova transformácia (DFT) je matematický aparát - funkcia $F : f \to g$. Rozkladá funkciu z časového spektra do frekvenčného. V informatike neexistujú skutočne spojité hodnoty, preto sa sa práca zaoberá diskrétnou fourierovou transformácou.

2.2 Odvodenie

2.2.1 ω_n - n-tá odnocnina jednotky v $\mathbb{C}$

Nech $\omega_n = \sqrt[n]{1}, \omega \in \mathbb{C}$. ω je v grafickom znázornení uloh výseku jednotkovej kružnice, ak by sme ju rozdelili na n rovnakých častí.



Obr. 2: Číslo ω_8 a jeho mocniny v $\mathbb{C}$

Z grafu možno zapísať platnosť rovnosti:

$$|\omega_n^k| = 1, \omega_n^k = 1 \cdot e^{i\varphi \cdot k} \tag{1}$$

φ je uhol výseku kružnice zodovedajúceho ω_n a teda:

$$\varphi_n = \frac{2 \cdot \pi}{n} \tag{2}$$

Následne vidieť súvislost medzi mocninami ω_n^k a jeho uhlom φ. Veľkosť tohto uhla pre ω_n^k je:

$$\varphi_{n^k} = \frac{2 \cdot \pi \cdot k}{n} \tag{3}$$

Zo vzťahu 3 a Obr. 2 prirodzene na základe skladania vektorov platia naslednovné rovnosti:

$$\omega_n^n = 1 + 0 \cdot i \tag{4}$$

$$\omega_n^k \cdot \omega_n^{n-k} = 1 + 0 \cdot i \tag{5}$$

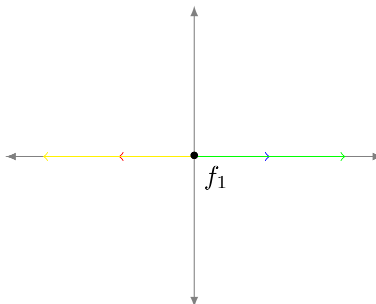
$$\sum_{k=1}^{n} \omega_n^k = 0 + 0 \cdot i \tag{6}$$

2.2.2 Intuitívna definícia

Dané sú čísla $1, 2, -1, -2$ z $\mathbb{C}$. Čísla sú zoradené v poradí. Ku každému číslu je priradená ω_4^k podľa príslušnej pozície nasledovne : $\omega_4^0 \rightarrow 1, \omega_4^1 \rightarrow 2, \omega_4^2 \rightarrow -1, \omega_4^3 \rightarrow -2$.

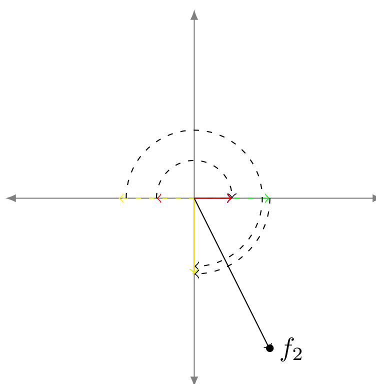
Čísla rotujú v smere hodinových ručiciek okolo bodu [0,0] súradnicovej sústavy reprezentujúcej $\mathbb{C}$.

1. V prvom kroku každé číslo rotuje, resp každé číslo je násobené nultou mocninou príslušnej ω_4^k. Súčtom vzniknutých čísel je číslo $f_1 = 0$ viď Obr.3 .



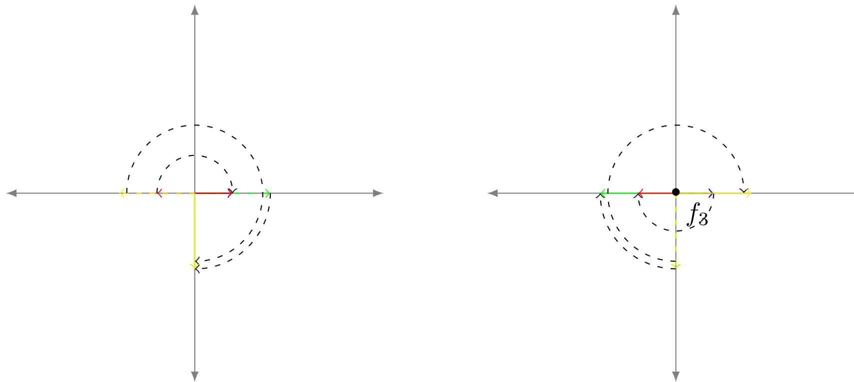
Obr. 3: Rotácia - $(\omega_4^k)^0 \cdot x$

2. V druhom kroku každé číslo je násobené prvou mocninou príslušnej ω_4^k. Súčtom vzniknutých čísel je číslo $f_2 = 2 - 4i$ viď Obr.2 .



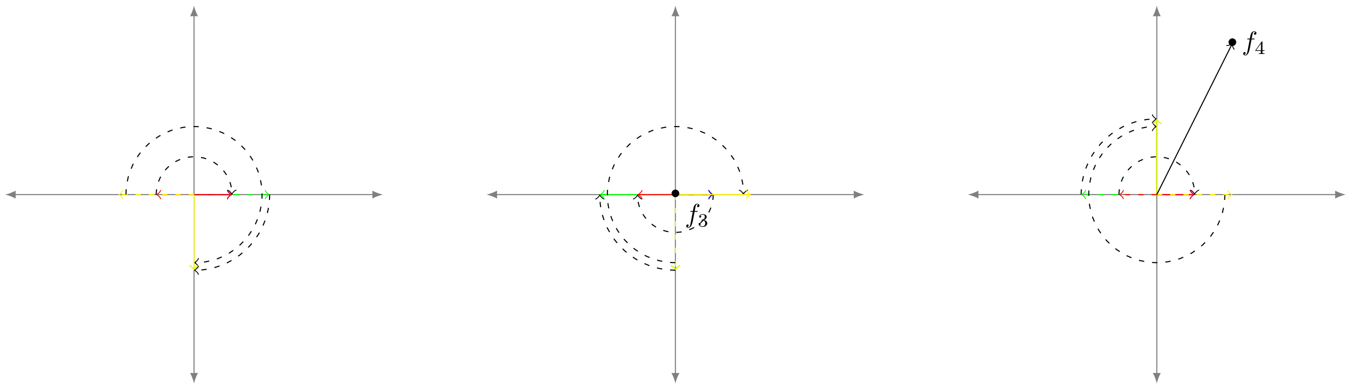
Obr. 4: Rotácia - $(\omega_4^k)^1 \cdot x$

3. V treťom kroku každé číslo je násobené druhou mocninou príslušnej ω_4^k. Súčtom vzniknutých čísel je číslo $f_3 = 0$ viď Obr.3 . Proces na grafoch znázornuje násobenie čísel. Druhá mocnina spôsobý, že číslo sa dvakrát po sebe vynásobí príslušnou ω_4^k.



Obr. 5: Rotácia - $(\omega_4^k)^2 \cdot x$

4. V štvrtom kroku každé číslo je násobené treťou mocninou príslušnej ω_4^k. Súčtom vzniknutých čísel je číslo $f_4 = 2{+}4i$ viď Obr.4. Proces na grafoch znázornuje násobenie čísel. Druhá mocnina spôsobý, že číslo sa trikrát po sebe vynásobí príslušnou ω_4^k.



Obr. 6: Rotácia - $(\omega_4^k)^3 \cdot x$

Čísla f_1, f_2, f_3, f_4 hotnoty furierovej transformácie pre dané čísla:

$$F(1) = f_1 = 0, F(2) = f_2 = 2 - 4i, F(3) = f_3 = 0, F(4) = f_4 = 2 + 4i$$

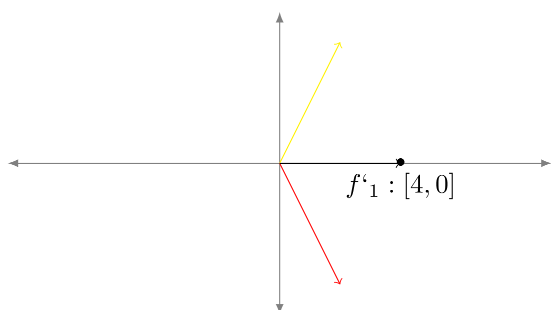
Tento proces je obsiahnutý v nasledujúcej formule - vzorec diskrétnej fourierovej transformácie $F : f \to g$. Funkcia f má predpis $f(x) = 1 + 2x - x^2 - 2x^3$. Výsledná fukcia g má predpis :

$$g(k) = \sum_{m=0}^{n-1} f(m) \cdot \omega_n^{m \cdot k} = \sum_{m=0}^{n-1} f(m) \cdot e^{-i \cdot \frac{2\pi k m}{n}} \tag{7}$$

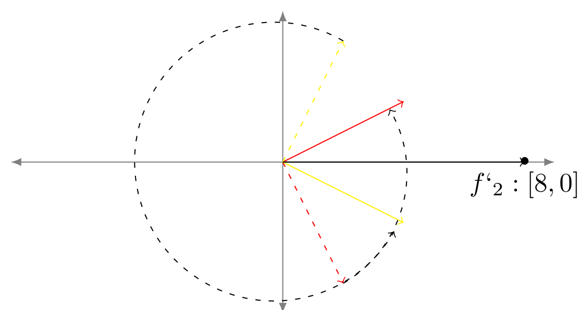
kde n je veľkosť definičného oboru funkcie f.

2.3 Inverzná diskrétna fourierová transformácia

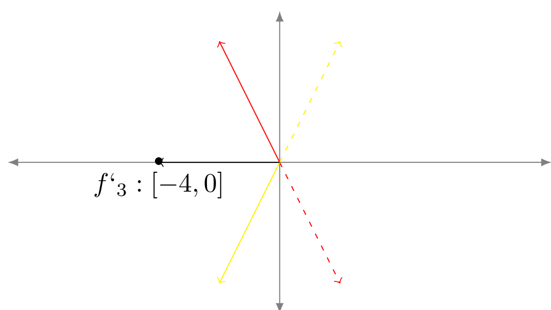
Inverzná diskrétna fourierová transformácia (IDFT) je funkcia $F‘ : g \to f$. DFT sme ukazovali pomocou rotácie vektorov funkcie f. IDFT funguje na podobnom princípe, ale rotuje opačným smerom.



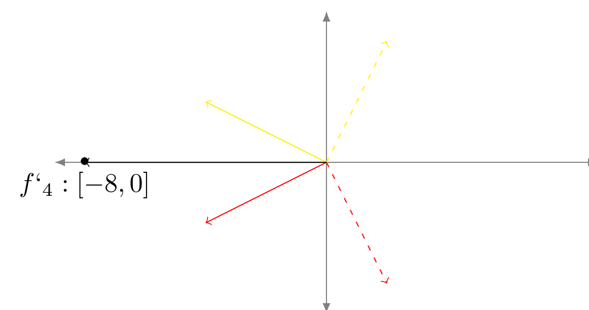
Obr. 7: Rotácia o $-(\omega_4^k)^0$



Obr. 8: Rotácia o $-(\omega_4^k)^1$



Obr. 9: Rotácia o $-(\omega_4^k)^2$



Obr. 10: Rotácia o $-(\omega_4^k)^3$

Výsledné hodnotu sa nezhodujú s pôvodnými hodnotami funkcie f. Existuje medzi nimi závisloť. Naše hodnotu sú n-krát väčšie ako pôvodné hodnoty. Výsledná funkcia f má predpis:

$$f(k) = \frac{1}{n} \sum_{m=0}^{n-1} g(m) e^{i \cdot \frac{2\pi}{n} \cdot m \cdot k} \tag{8}$$

2.4 Pseudokód - Python (pomalá implementácia)

2.4.1 DFT

```
def dft(inputs : List) -> List:
''' inputs je obor hodnôt vstupnej funkcie f'''

    res = [0]*len(inputs)
    n = len(inputs)

    for k in range(n):
        g_k = complex(0)
        for m, vector in enumerate(inputs):
            g_k += vector * cmath.exp(-1j *2 *pi *k *m /n)

        res[k] = g_k

    return res
```

2.4.2 IDFT

```
def idft(FmList : List) -> List:

    n = len(FmList)
    res = []

    for k in range(n):
        g_k = 0
        for m in range(n):
            g_k += FmList[m]*cmath.exp(2j*pi*k*m/n)
        g_k /= n
        res.append(g_k)

    return res
```

2.5 Fast Fourier Transform

Fast Fourier Transform alebo rýchla fourierova transformácia je algoritmus počítajúci diskrétnu fourierovú transformácou ale v menšej časovej zlozitosti. FFT je implementovaná v knižnici NumPy spolu s IFFT.

```
>>> a = np.array([1,2,-1,-2])
>>> b = np.fft.fft(a)
>>> b
array([0.+0.j, 2.-4.j, 0.+0.j, 2.+4.j])
>>> np.fft.ifft(b)
array([ 1.+0.j,  2.+0.j, -1.+0.j, -2.+0.j])
```

Ďalej v práci budepoužívaná FFT kvôli efektivite.

	10	100	1000	10000	100000	1000000	10000000
DFT	0.000999927 s	0.100939 s	10.7133 s	$\sim$ 100 s	∞	∞	∞
FFT	0.0 s	0.0 s	0.0009996 s	0.00199 s	0.02098 s	0.23285436630 s	2.60239 s

3 Cirkulárna konvolúcia

Konvolúciou signálov je matematický operátor spracovávajúci dve funkcie. Označuje sa znakom $*$.

3.1 Súčin polynómov

Nech existujú dva signály aproximované pomocou taylorových polynómov na polynómy:

$$A = a_0x^0 + a_1x^1 + a_2x^2 + a_3x^3 + \ ... \ + a_{n-1}x^{n-1}$$
$$B = b_0x^0 + b_1x^1 + b_2x^2 + b_3x^3 + \ ... \ + b_{n-1}x^{n-1}$$

Súčinom týchto dvoch polynómov je $A * B = C$. Výsledný polynóm C má tvar:

$$C = c_1x^0 + c_2x^1 + b_3x^2 + c_4x^3 + \ ... \ + c_{2n-2}x^{2n-2}$$

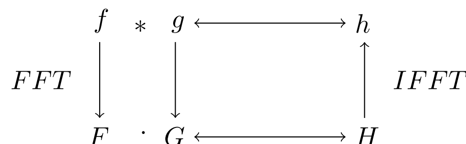
Násobenie polynómov má zložitosť $\theta(n^2)$, kde n je rád vstupných polynómov. Je potrebné vynásobiť každý člen prvého polynómu, každým členom druhého, čo je rádovo n^2 krokov. Výsledné súčiny s rovnakou mocninou x sú sčítané to koeficientu pre danú mocninnu polynómu C.

$$c_0 = a_0 \cdot b_0$$
$$c_1 = a_0 \cdot b_1 + a_1 \cdot b_0$$
$$c_2 = a_0 \cdot b_2 + a_1 \cdot b_1 + a_2 \cdot b_0$$
$$...$$
$$...$$
$$...$$
$$c_{n-1} = a_0 \cdot b_{n-1} + a_1 \cdot + b_{n-2} + a_2 \cdot + b_{n-3} + \ ... \ + a_{n-1} \cdot + b_0$$
$$...$$
$$...$$
$$...$$
$$c_{n-2} = a_{n-1} \cdot b_{n-1}$$

Počet súčinov koeficientov pôvodných polynómov potrebných na určenie koeficientu výsledného polynómu sa mení. Najviac členov treba pre stredný koeficient, najmenší pre prvý a posledný.

3.2 Použitie Fourierovej transformácie

Cieľom je s využitím fourierovej transformácie docieliť efektívnejšie násobenie dvoch polynómov. Fourierová transformácia pracuje s mocninami ω_n. Vstupné polynómy teda budú pracovať namiesto mocnín x s mocninami ω_n. Ide o proces spájania dvoch signálov (polynómov) do jedného.



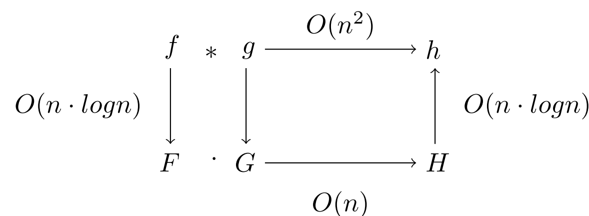
Obr. 11: Cirkulárna konvolúcia

Vďaka vlastnostiam Fourierovej transformácie po aplikovaní transformácie na obe funkcie f a g, reprezentované koeficientami vektorov, vzniknú funkcie F, G. Jednotlivé prvky vektorov týchto funkcií reprezentujú bod $F(x^k)$, $G(x^k)$, kde $x = \omega_n$. $F(x^k)$, $G(x^k)$ určujú bod pre konkrétne x^k. Ich súčin určuje bod $H(k) = F(x^k) \cdot G(x^k)$. Výsledný vektor H je výsledkom násobenia po prvkoch vstupných vektorov:

$$H(k) == \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} F(i) \cdot G(i) \cdot \omega_n^{i+j} \tag{9}$$

$$H(k) = \sum_{i=0}^{2n-2} c_i \cdot \omega_n^i \tag{10}$$

Na výslednú funkciu sa aplikuje inverzná transformá, ktorej výsledkom je konvolúcia pôvodných dvoch polynómov. Časová zložitosť tohto algoritmu je $O(n \cdot logn)$



Obr. 12: Časová zložitosť

Tento postup je známy ako **cirkulárna konvolúcia**. Aby tento algoritmus fungoval aj na násobenie polynómov je potrebné doplniť v stupných vektoroch nuly tak aby mali dĺžku $d =$ *veľkosť prvého polýnómu* $+$ *veľkosť druhého polýnómu* -2. Tým zabezpečíme veľkosť výsledného polynómu $2n - 2$

3.3 Implementácia v Pythone

```
def circular_convolution(p, q):
    fp = np.fft.fft(p2) # aplikovanie fft na vstupne polynomi
    fq = np.fft.fft(q2)

    f_res = np.multiply(fp, fq) # sucin po prvkoch

    return np.fft.ifft(f_res) # aplikovanie ifft


def nasob(p, q):

    p2 = np.append(p, [0]*(len(q)-1)) # doplnenie vstupnych polynomov nulami
    q2 = np.append(q, [0]*(len(p)-1))

    return circular_convolution(p2, q2)
```

4 Zistenie shiftu - posunu v jednorozmernom poli

Nech existe konečná postupnosť calých čísel a druhá postupnosť, ktorá je oproti prvej posunutá cirklicky doprava o k miest. Próblém je nasledovný - zistiť posun druhého signálu oproti prvéhu tzv. nájsť shift.

4.1 Bez použitia FFT

Pôvodný signál je posunutý o všetky možné hodnoty shift a po prvdok vynásobený s druhým signálom. Ak sa skutočné posunutie s novým zhoduje súčín po prvkoch je postupnosť druhých mocnín posunutého signálu. Suma takéhoto poľa je určite najvačšia oproti ostatným možným posunutiam nakoľko nikdy inokedy nevzdnikde pole druhých mocnín.

```
from random import randint as ri
from numpy import inf

a = np.array([ri(1,100) for i in range(N)])[np.newaxis]
shift = ri(0,N-1)
b = np.roll(a, shift) # cirklicky posunie o shift miest v

def find_shift(a,b):
    N = len(a)
    max_sum = -inf
    shift = 0

    for s in range(N):
        a_2 = np.roll(a, -s)
        sum = 0
        for i in range(N):
            sum += a_2[i]*b[i]

        if sum < max_sum:
            max_sum = sum
            shift = s

    return shift
```

Zložitosť tohto algoritmu je $\theta(n^2)$, kde n je veľkosť vstupného poľa. Algoritmus obsahuje dva vnorené for cykly, ktoré sa vžsy vykonajú n^2-krát.

4.2 S použitím FFT

Podobne ako násobenie polynómov aj tento problém sa dá algoritmicky vylepšiť použitím FFT. Cieľom je nájst také k pre ktoré sa minumaluzuje zasledovná funkcia:

$$E(k) = \sum_{i=0}^{N-1} (f(n) - g((n+k)modN))^2 \tag{11}$$

Nakoľko ide o ciklickú rotáciu argument g je $(n+k)modN$. Výsledný hľadaný shift K je:

$$K = argmmin(E(k)) \tag{12}$$

Pôvodná rovnica sa dá upraviť do tvaru:

$$\begin{aligned} E(k) &= \sum_{i=0}^{N-1} (f(n) - g((n+k)modN))^2 && (13) \\ &= \sum_{i=0}^{N-1} (f(n)^2 - 2 \cdot f(n) \cdot g((n+k)modN) + g((n+k)modN)^2) && (14) \\ &= \sum_{i=0}^{N-1} (f(n)^2) - 2 \sum_{i=0}^{N-1} (f(n) \cdot g((n+k)modN)) + \sum_{i=0}^{N-1} (g((n+k)modN)^2)) && (15) \end{aligned}$$

Prvý a posledný člen v rovnici sa pri meniacom sa k nikdy nemení, preto neovplyvnia výpočet výsleného K. Iba od stredného členu závysí výsledná hodnota $K = argmin(E(k))$.

$$K = argmin(-2\sum_{i=0}^{N-1}(f(n) \cdot g((n+k)modN))) \quad (16)$$

$$= argmin(-\sum_{i=0}^{N-1}(f(n) \cdot g((n+k)modN))) \text{ koštanta 2 je rovnaká pre všetky členy} \quad (17)$$

$$= argmax(\sum_{i=0}^{N-1}(f(n) \cdot g((n+k)modN))) \text{ minimum zo záporného čísla je maximum} \quad (18)$$

Toto sa veľmi podobá na násobenie polynómov. Ak použijeme FFT a cirkulárnu konvlúciu môžeme urýchliť tento výpočet, je potrebné ju trochu upraviť.

```
import numpy as np

def circular_convolution2(p, q):
    fp = np.fft.fft(p)
    fq = np.fft.fft(q)

    fp = np.conj(fp) # conj - komplexne zdruzene

    f_res = np.multiply(fp, fq)

    return np.fft.ifft(f_res)
```

Druhý signál po aplikovaní FFT zmeníme na jeho komplexne združený, čo nám umožní pri správnom shifte dostať druhú mocninu čísel, lebo pri komplexných číslach platí

$$a^2 = a \cdot a* \quad (19)$$

čím vzniknú najvačšie čísla pri správnom shifte. Z výsledného vektoru následne funkcia argmax zistí index s najvačšou hodnotou, teda $argmax(E(k))$ čo je hľadané K.

```
>>> data =    np.array([60, 17, 32, 97, 45, 13, 60, 20, 21,  8])
>>> shifted = np.array([45, 13, 60, 20, 21,  8, 60, 17, 32, 97])
>>> res = circular_convolution2(data, shifted)
>>> np.argmax(res)
6
```

Demonštrácia so shiftom 6.