Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Vyhľadávanie vzoru na obraze pomocou rýchlej Fourierovej transformácie

Bakalárska práca

2018
Daniel Kyselica

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Vyhľadávanie vzoru na obraze pomocou rýchlej Fourierovej transformácie

Bakalárska práca

Študijný program:	aplikovaná informatika
Študijný odbor:	aplikovaná informatika
Školiace pracovisko:	Katedra informatiky
Školiteľ:	RNDr. Andrej Lúčny, PhD.

Bratislava, 2018
Daniel Kyselica

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Daniel Kyselica
Študijný program: aplikovaná informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: aplikovaná informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Vyhľadávanie vzoru na obraze pomocou rýchlej Fourierovej transformácie
Template matching on images via fast Fourier transform

Anotácia: kompilačná a reimplementačná práca z oblasti počítačového videnia

Cieľ: Kompilačná práca pojednáva o vyriešenom probléme efektívneho vyhľadávania vzoru na obraze, za ktorým stojí cyklická konvolúcia realizovaná pomocou Fourierovej transformácie. Ide o pomerne komplikovaný matematický aparát. Cieľom je na príkladoch vysvetliť čitateľovi ako, prečo a kedy táto metóda funguje.
Súčasťou práce je reimplementácia metódy v OpenCV (Pyhton alebo C++)

Literatúra: Matematická analýza III.
dokumentácia k OpenCV
články o template matching

Kľúčové slová: počítačové videnie, template matching

Vedúci: RNDr. Andrej Lúčny, PhD.
Katedra: FMFI.KAI - Katedra aplikovanej informatiky
Vedúci katedry: prof. Ing. Igor Farkaš, Dr.

Spôsob sprístupnenia elektronickej verzie práce:
bez obmedzenia

Dátum zadania: 05.10.2018

Dátum schválenia: 10.10.2018 doc. RNDr. Damas Gruska, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Poďakovanie: Ďakujem môjmu školiteľovi doktorovi Andrejovi Lúčnemu za jeho odborné vedenie a cenné rady, ako aj mojim rodičom za podporu pri vypracovaní bakalárskej práce.

Abstrakt

Cieľom tejto práce je vysvetlenie princípu fungovania algoritmov hľadajúcich objekt na obraze pomocou nami vytvorených vizualizácií a jednoduchých príkladov v Pythone. Pri zhotovovaní algoritmov a ilustračných funkcií sa používajú externé knižnice NumPy a OpenCV. Práca vysvetľuje princíp diskrétnej Fourierovej transformácie pre jednorozmerné a dvojrozmerné dáta. Tá je základom algoritmov rozpoznávajúcich objekt na obraze. Efektivitu algoritmov oproti klasickým metódam zaručuje algoritmus Fast Fourier transform (rýchla Fourierova transformácia), ktorý je vysvetlený a implementovaný v prílohe. V práci sú uvedené tri algoritmy hľadajúce objekt na obraze. Algoritmus využívajúci cirkulárnu konvolúciu je schopný nájsť posunutý objekt na obraze. Druhý algoritmus na báze fázovej korelácie je odolný voči zmene kontrastu. V závere je popísaný algoritmus, ktorý pomocou geometrických transformácií a fázovej korelácie nájde na obraze otočený, škálovaný a posunutý objekt.

Kľúčové slová: Fourierova transformácia, cirkulárna konvolúcia, fázová korelácia, spracovanie obrazu, vyhľadávanie vzoru na obraze

Abstract

The aim of this bachelor thesis is an explanation of the image registration algorithms by visualisations and working examples implemented by us in Python. External libraries NumPy and OpenCV are used in examples and implementations of algorithms. Fourier transformation for one and two dimensional data is explained by thesis. This transformation is the basis of image registration algorithms. Fast Fourier transform algorithm, described in appendix, guarantees effectiveness of these algorithms compared to classical methods. There are three algorithm described in thesis. The first algorithm based on circular convolution is able to find a shifted replica of an object in a picture. Another algorithm uses phase correlation. In comparison with the first algorithm it is invariant to contrast. Geometrical transformations and phase correlation are used in the last algorithm, which can find a rotated, scaled and shifted replica of an object in a picture.

Keywords: Fourier transformation, circular convolution, phase correlation, image processing, image registration

Obsah

Zoznam obrázkov

Úvod

Práca sa zaoberá princípom fungovania algoritmu, ktorý hľadá vzor na obraze s použitím fázovej korelácie. Ďalej pomocou vizualizácií a pomocou jednoduchých príkladov v Pythone vysvetľuje princíp fungovania Fourierovej transformácie a algoritmov využívajúcich ju na vyhľadávanie vzoru na obraze.

Prvú verziu tohto algoritmu publikovali v roku 1975 C.D.Kuglin and D.C.Hines. Ako prvý ukázali využitie fázovej korelácie pri vyhľadávaní vzoru na obraze. Tento algoritmus vedel efektívne vyhľadať rôzne posunuté objekty na obraze. Ďalšia práca sa sústredila na rozšírenie tohto konceptu o otáčanie a škálovanie, ktorých rôzne riešenia publikovali: B. Srinivasa Reddy a B. N. Chatterji v roku 1996, Georgios Tzimiropoulos a Tania Stathaki v roku 2009.

Táto práca sa venuje vysvetľovaniu diskrétnej Fourierovej transformácie, nakoľko študentom často nie je jasné čo sa pri tejto transformácií deje, ako aj algoritmom ju využívajúcim. V dnešnej dobe je v knižniciach podobných OpenCV implementovaných množstvo funkcií, ktoré vedia vyhľadať vzor na obraze. Každý z nich má svoje výhody a nevýhody, a preto je vhodný pri určitých situáciách. Práca je zameraná na jeden konkrétny algoritmus využívajúci fázovú koreláciu na vyhľadávanie vzoru na obraze. Na vysvetlenie a ukážky je použitá knižnica NumPy v Pythone, ktorá obsahuje nástroje na prácu s maticami, vektormi a podobne. Vďaka nej sa dá písať v Pythone podobným štýlom ako v Matlabe.

Fourierova transformácia je veľmi používaná pri práci so signálmi. Jej použitie pri práci s obrazom nie je na prvý pohľad zrejmé. Obraz sa však dá považovať za určitý druh signálu. Vďaka algoritmu rýchlej Fourierovej transformácie je možné spracovať obraz rýchlejšie ako klasickými algoritmami, ktoré nevyužívajú Fourierovu transformáciu, pričom dosiahneme rovnaký výsledok.

Práca pozostáva z piatich kapitol. Prvá kapitola je venovaná nástrojom v programovacom jazyku Python, ktoré sú potrebné pre prácu s komplexnými číslami, prácu s obrazom a zobrazovanie grafov. Druhá kapitola obsahuje podrobné odvodenie Fourierovej transfor-

mácie. Transformácie je vysvetlená pomocou obrázkov, ktoré pomáhajú čitateľovi lepšie pochopiť princíp jej fungovania. Taktiež obsahuje aj krátky program a vysvetľuje rozdiel v rôznych implementáciách Fourierovej transformácie. V tretej kapitole je vysvetlený a popísaný matematický aparát cirkulárnej konvolúcie, ktorý je základom výsledného algoritmu. Štvrtá kapitola ukazuje na príkladoch algoritmus hľadajúci vzor na šedo-tónovom obraze ak je vzor posunutý a má zmenený kontrast. Záverečná kapitola je venovaná cieľovému algoritmu, ktorý je schopný nájsť vzor na obraze, ak je vzor otočený, posunutý, škálovaný a so zmeneným kontrastom. Tiež sú v nej popísané rôzne geometrické transformácie obrazu potrebné pre fungovanie algoritmu. Práca obsahuje prílohu o algoritme rýchlej Fourierovej transformácie. Efektívnosť tohto algoritmu je základom efektivity algoritmov v práci.

Kapitola 1

Predpoklady

1.1 Knižnice v Pythone

Pri programovaní v Pythone jeho základná funkcionalita často nie je dostatočná [9]. Žiadanú funkcionalitu nám ponúkajú externé knižnice. Väčšinu knižníc v Pythone je možné nainštalovať pomocou príkazu

```
> pip install <nazov_kniznice>
```

ktorý prichádza s inštaláciou Pythonu. My budeme používať tieto tri knižnice

NumPy

Knižnica NumPy je základným balíčkom funkcií pre vedecké výpočty. Obsahuje nástroje a základné funkcie lineárnej algebry, Fourierovu transformáciu, náhodné premenné [4]. Knižnica implementuje matice, ktoré sa využívajú pri reprezentácií obrazov, a prácu s nimi. Názov knižnice pre inštaláciu : *NumPy*

Matplotlib

Matplotlib je knižnica na vytváranie grafov rôznych formátov v interaktívnych prostrediach pre rôzne platformy [5]. Knižnicu budeme využívať na prácu s obrazom, na načítanie, uloženie, zobrazenie grafov a podobne. Názov knižnice pre inštaláciu : *MatPlotLib*

OpenCV

OpenCV je open source knižnica pre prácu s obrazom. Obsahuje nástroje na načítanie, uloženie a prácu s obrazom ako aj algoritmy k tomu potrebné, napríklad detekcia hrán, sledovanie objektu, vyhľadávanie vzoru a podobne. Je implementovaná vo viacerých jazykoch ako Python a C++ [3]. Názov knižnice pre inštaláciu : *opencv-python*

1.2 Komplexné čísla v Pythone

Komplexné čísla reprezentujú body v rovine. Na rozdiel od reálnych čísel obsahujú okrem veľkosti aj informáciu o uhle, keďže neležia na jednej súradnicovej osi ako reálne čísla. Preto sú vhodné na prácu so signálmi, lebo signál reprezentuje amplitúda a fáza. Napríklad bod $[3, 5]$ karteziánskej súradnicovej sústavy reprezentuje komplexné číslo $3 + 5 \cdot i$ v algebraickom zápise [6]. V Pythone sú komplexné čísla priamo implementované. Imaginárna jednotka sa zapisuje pomocou kľúčového písmena j. Imaginárnu časť čísla sa zapisuje ako jeden reťazec ktorý má na konci znak 'j'. [4]

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

1.2.1 Reprezentácie

Algebraický tvar - Algebrický

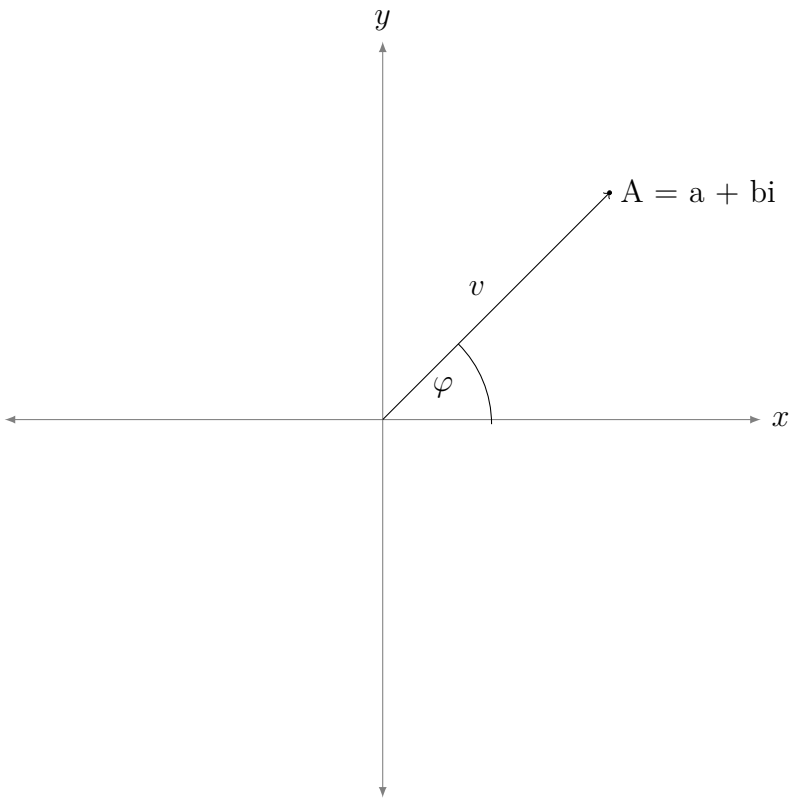
Algebraický tvar je v Pythone priamo podporovaný. Reálna a imaginárna zložka komplexného čísla v tomto zápise zodpovedá euklidovským súradniciam bodu v rovine. Na zistenie reálnej a imaginárnej časti čísla sú v knižnici Numpy implementované funkcie.

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

Všetky čísla v Pythone sú objektami. Komplexné číslo je objekt triedy *complex*. Funkcia *np.real(a)* vracia veľkosť reálnej zložky komplexného čísla a, funkcia *np.imag(a)* vracia veľkosť imaginárnej zložky.

Goniometrický - polárny tvar

V goniometrickej reprezentácií číslo $a \in \mathbb{C}$ predstavuje vektor zo stredu súradnicovej osi. Vektor je jednoznačne určený jeho veľkosťou v a uhlom φ ktorý zviera tento vektor s osou x, viď Obr. 1.1. Veľkosť vektora v sa tiež nazýva veľkosť komplexného čísla.



Obr. 1.1: Geometrická interpretácia komplexného čísla

V Pythone nie je možnosť priamo pracovať s číslami v goniometrickom tvare. Knižnica NumPy obsahuje funkcie na získanie veľkosti uhla φ ako aj veľkosti vektora 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 nepovinný argument *deg*. Ak nie je určený návratová hodnota určuje veľkosť uhla v radiánoch. Ak *deg=True* návratová hodnota určuje veľkosť uhla v stupňoch.

Exponenciálny tvar

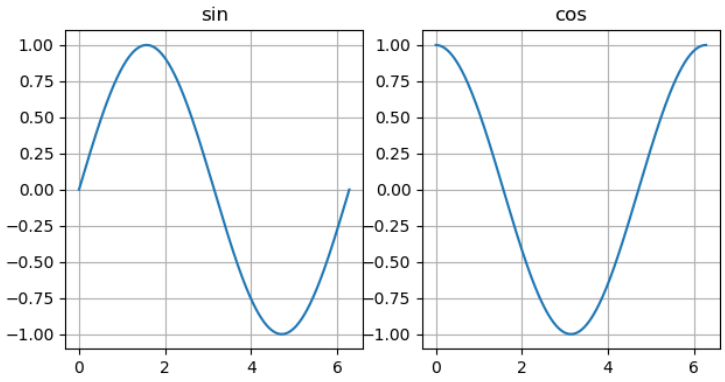
Komplexné číslo $a \in \mathbb{C}$ v exponenciálnom tvare je jednoznačne určené nasledujúcim vzťahom

$$a = ve^{i\varphi}$$

kde v je veľkosť čísla a φ jeho uhol. Knižnica NumPy má funkciu *exp*, ktorá akceptuje vstup v komplexných číslach. Preto sa dá využívať na konverziu čísla z exponenciálneho tvaru na algebraický.

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

V príklade $np.abs(a) \neq 3$, čo je spôsobené reprezentovaním desatinných čísel v Pythone. Pre číslo a platí $a = ve^{i\varphi} = v(\cos\varphi + i\sin\varphi)$. Pre $v = 1$ reálna zložka čísla obsahuje kosínus uhla φ a imaginárna jeho sínus. Vďaka tejto vlastnosti je možné použiť exponenciálny tvar na výpočet sínusu a kosínusu ľubovoľného uhlu.



Obr. 1.2: Graf sínusu a kosínusu

Na obr. 1.2 je graf sínusu a kosínusu s amplitúdou jedna bez fázového posunu. Obrázok je vyobrazený pomocou knižnice *Matplotlib*.

1.2.2 Operácie nad $\mathbb{C}$ v Pythone

Práca s komplexnými číslami v Pythone je jednoduchá. Operátory *, -, +, / sú preťažené a je ich možné použiť pri práci s komplexnými číslami. [4]

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

Vďaka tomu je práca s komplexnými číslami v Pythone veľmi jednoduchá.

Kapitola 2

Fourierova transformácia

Fourierova transformácia je jedným zo základných aparátov pre prácu so signálom. Obraz sa dá reprezentovať ako dvojrozmerný signál. Preto sa Fourierova transformácia využíva aj na spracovanie obrazu [10] [7].

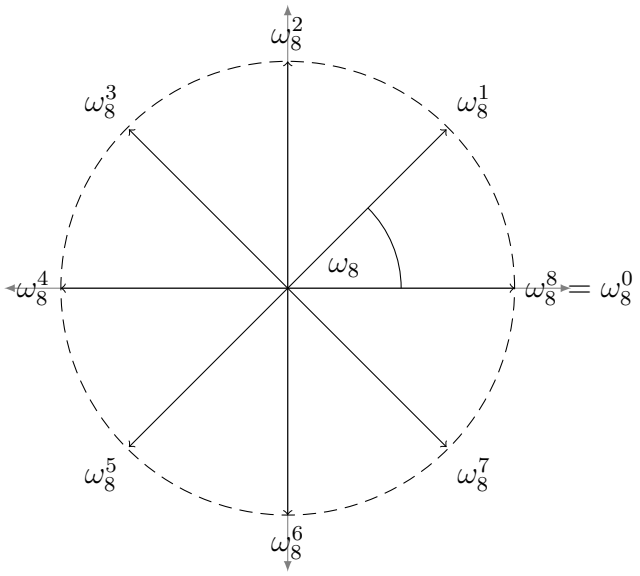
2.1 Definícia

Diskrétna Fourierova transformácia (DFT) je matematický aparát - funkcia $F : f \to g$. Transformuje diskrétnu funkciu z časového spektra do frekvenčného. Dnešné počítače sú digitálne a teda fungujú diskrétne, preto sa v práci pracujeme s diskrétnou Fourierovou transformáciou. [6]

2.2 Odvodenie

2.2.1 Komplexná odmocnina jednotky

Nech $\omega_n = \sqrt[n]{1}, \omega \in \mathbb{C}$. ω_n, nazývaná aj komplexná odmocnina jednotky, je v grafickom znázornení uhol výseku jednotkovej kružnice, ak by sme ju rozdelili na n rovnakých častí. Potom ω_n^k predstavuje súčet k takýchto uhlov, viď obr. 2.1.



Obr. 2.1: Číslo ω_8 a jeho mocniny v $\mathbb{C}$

Z obrázku možno zapísať rovnosť:

$$|\omega_n^k| = 1, \omega_n^k = 1 \cdot e^{i\varphi \cdot k} \tag{2.1}$$

φ_n je uhol výseku kružnice zodpovedajúceho ω_n a teda:

$$\varphi_n = \frac{2\pi}{n} \tag{2.2}$$

Následne vidieť súvislosť medzi mocninami ω_n^k a jeho uhlom φ_{n^k}. Veľkosť tohto uhla pre ω_n^k je:

$$\varphi_{n^k} = \frac{2\pi \cdot k}{n} \tag{2.3}$$

Zo vzťahu 2.3 a Obr. 2.1 prirodzene na základe skladania vektorov platia nasledovné rovnosti:

$$\omega_n^n = \omega_n^0 = 1 \tag{2.4}$$

$$\omega_n^k \cdot \omega_n^{n-k} = \omega_n^0 = 1 \tag{2.5}$$

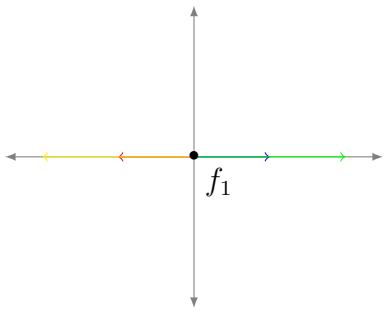
$$\sum_{k=1}^{n} \omega_n^k = 0 \tag{2.6}$$

2.2.2 Intuitívna definícia

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

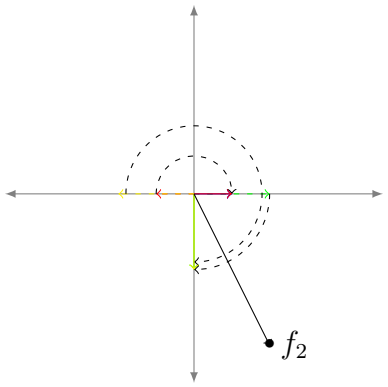
Čísla rotujú v smere hodinových ručičiek okolo bodu [0,0] súradnicovej sústavy reprezentujúcej $\mathbb{C}$.

1. V prvom kroku každé číslo rotuje, resp každé číslo je násobené nultou mocninou príslušnej ω_4^k. Súčtom rotovaných vektorov je číslo $f_1 = 0$ viď Obr.2.2 .



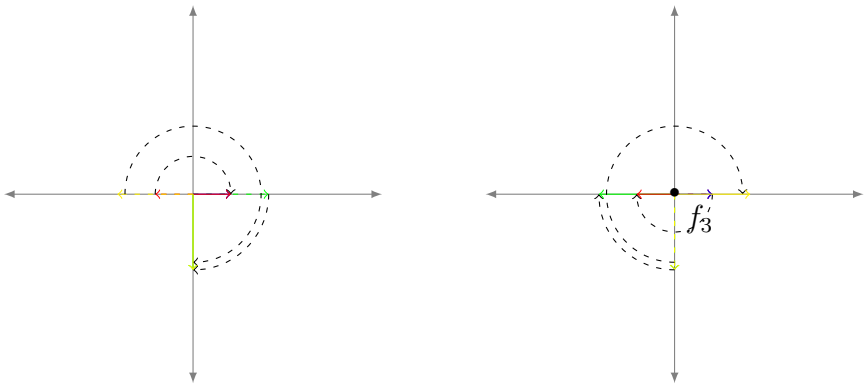
Obr. 2.2: Rotácia - $(\omega_4^k)^0 \cdot x$

2. V druhom kroku je každé číslo násobené prvou mocninou príslušnej ω_4^k. Súčtom rotovaných vektorov je číslo $f_2 = 2 - 4i$ viď Obr. 2.3 .



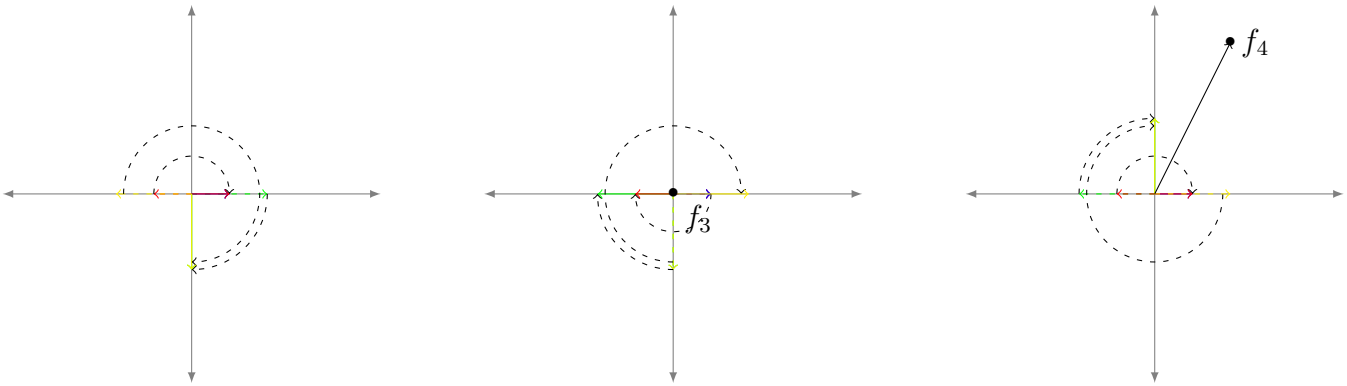
Obr. 2.3: Rotácia - $(\omega_4^k)^1 \cdot x$

3. V treťom kroku je každé číslo násobené druhou mocninou príslušnej ω_4^k. Súčtom rotovaných vektorov je číslo $f_3 = 0$ viď Obr.2.4. Proces na grafoch znázorňuje násobenie čísel. Druhá mocnina spôsobí, že číslo sa dvakrát po sebe vynásobí príslušnou ω_4^k,a teda dvakrát sa otočí o uhol φ_{4^k}.



Obr. 2.4: Rotácia - $(\omega_4^k)^2 \cdot x$

4. V štvrtom kroku je každé číslo násobené treťou mocninou príslušnej ω_4^k. Súčtom rotovaných vektorov je číslo $f_4 = 2 + 4i$ viď Obr.2.5. Proces na grafoch znázorňuje násobenie čísel. Druhá mocnina spôsobí, že číslo sa trikrát po sebe vynásobí príslušnou ω_4^k.



Obr. 2.5: Rotácia - $(\omega_4^k)^3 \cdot x$

Čísla f_1, f_2, f_3, f_4 sú hodnoty Fourierovej transformácie pre funkciu reprezentovanú hodnotami $1, 2, -1, -2$. f_1 sa označuje ako $F(1)$

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

Tento proces je obsiahnutý v nasledujúcej formule - vzorec diskrétnej Fourierovej transformácie $F : f \to g$. Vstupné čísla predstavujú funkciu f s predpisom $f(x) = 1+2x-x^2-2x^3$.

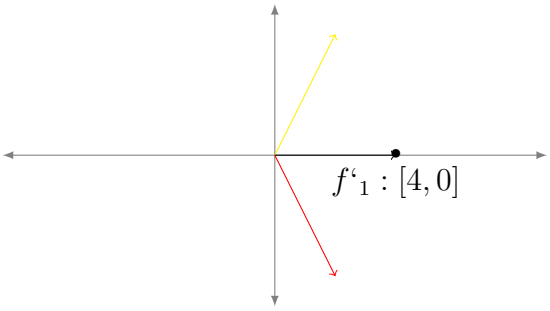
Transformovaná funkcia g má predpis :

$$g(k) = \sum_{m=0}^{n-1} f(m) \cdot \omega_n^{m \cdot k} = \sum_{m=0}^{n-1} f(m) \cdot e^{-i \cdot \frac{2\pi k}{n} m} \tag{2.7}$$

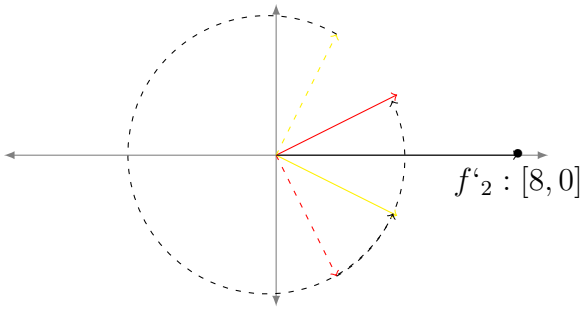
kde n je veľkosť definičného oboru funkcie f.

2.3 Inverzná diskrétna Fourierova transformácia

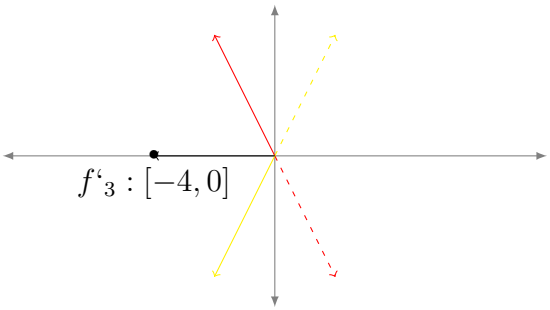
Inverzná diskrétna Fourierova transformácia (IDFT) je funkcia $F' : g \to f$. DFT bola demonštrovaná pomocou rotácie vektorov funkcie f. IDFT funguje a dá sa znázorniť podobne, ale ide o rotáciu opačným smerom ako pri DFT.



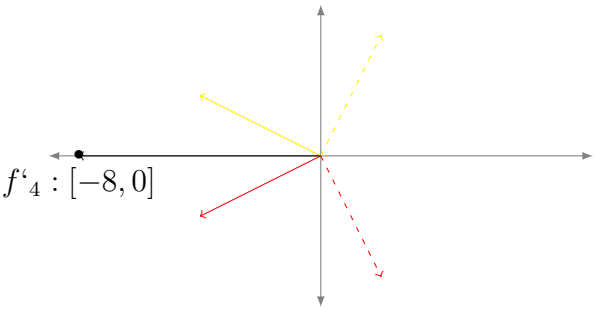
Obr. 2.6: Rotácia o $-(\omega_4^k)^0$



Obr. 2.7: Rotácia o $-(\omega_4^k)^1$



Obr. 2.8: Rotácia o $-(\omega_4^k)^2$



Obr. 2.9: Rotácia o $-(\omega_4^k)^3$

Čiarkované čiary na obrázkoch 2.6, 2.7, 2.8 a 2.9 predstavujú vektory pred rotáciou, plné čiary po rotácií. Čiernou farbou je znázornení výsledný vektor súčtu. Výsledné hodnotu sa nezhodujú s pôvodnými hodnotami funkcie f. Existuje medzi nimi závislosť. Naše hodnotu sú n-krát väčšie ako pôvodné hodnoty, kde n je počet vstupných vektorov. Preto výsledné čísla vydelíme číslom n. Výsledná funkcia f má predpis:

$$f(k) = \frac{1}{n} \sum_{m=0}^{n-1} g(m) e^{i \cdot \frac{2\pi k}{n} m} \tag{2.8}$$

2.4 Priamočiará implementácia v Pythone

Ak priamočiaro prepíšeme vzorce diskrétnej Fourierovej transformácie 2.7 a jej inverznej transformácie 2.8 v Python kóde, vytvoríme nasledujúce funkcie:

DFT - diskrétna Fourierova transformácia

```
def dft(inputs : List) -> List:
    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
```

IDFT - inverzná diskrétna Fourierova transformácia

```
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(2jpikm/n)
        g_k /= n
        res.append(g_k)
    return res
```

Tieto funkcie majú pri vstupe veľkosti n časovú zložitosť $\Theta(n^2)$. Ich výhoda je v prehľadnosti a porozumitelnosti kódu na základe vzorca.

2.4.1 Rýchla Fourierova transformácia

Rýchla Fourierova transformácia alebo Fast Fourier Transform, **FFT**, je algoritmus počítajúci diskrétnu Fourierovu transformáciu rovnako ako predchádzajúca priamočiara implementácia, ale efektívnejšie. v časovej zložitosti $O(n \log(n))$). FFT je implementovaná v knižnici NumPy spolu s IFFT. Pracuje na princípe *rozdeľuj a panuj*, ktorý umožní menšiu zložitosť ako pri priamočiarej implementácií [4]

```
>>> 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])
```

	10	100	1000	10000	100000	1000000
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

Hodnota nekonečno v tabuľke predstavuje čas, ktorý je prakticky nemerateľný. Na jeho presné odmeranie by bolo potrebné, aby program bežal niekoľko hodín až dní. Ďalej v práci bude používaná FFT implementovaná v knižnici NumPy kvôli efektivite.

Kapitola 3

Konvolúcia a Cirkulárna konvolúcia s využitím DFT

3.1 Konvolúcia

Konvolúciou je matematický operátor spracovávajúci dve funkcie. Označuje sa znakom $*$. Konvolúcia dvoch polynomiálnych funkcií je súčin polynómov. Nech existujú dva polynómy A a B oba $(N-1)$ rádu. Konvolúciou týchto polynómov je polynóm C $(2N-2)$ rádu.

$$A = a_0 + a_1x + a_2x^2 + a_3x^3 + \ ... \ + a_{N-1}x^{N-1}$$
$$B = b_0 + b_1x + b_2x^2 + b_3x^3 + \ ... \ + b_{N-1}x^{N-1}$$
$$C = c_0 + c_1x + c_2x^2 + c_3x^3 + \ ... \ + c_{2N-2}x^{2N-2}$$

Násobenie dvoch polynómov dĺžky N má časovú zložitosti $O(N^2)$ (každý člen prvého polynómu je násobený z každým členom druhého, spolu N^2 súčinov). Jednotlivé koeficienty výsledného polynómu C sú týmto postupom vypočítavané nasledovne

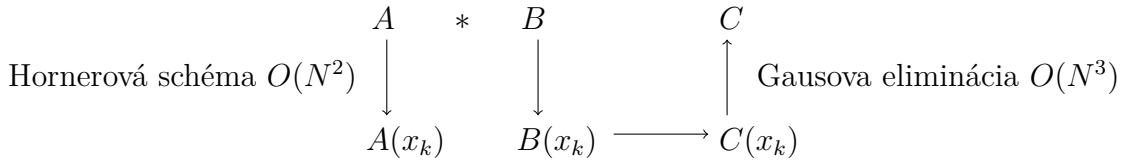
$$c_0 = a_0 \cdot b_0$$
$$c_1 = a_0 \cdot b_1 + a_1 \cdot b_0$$
$$c_2 = a_0 \cdot b_2 + a_1 \cdot b_1 + a_2 \cdot b_0$$
$$...$$
$$c_{N-1} = a_0 \cdot b_{N-1} + a_1 \cdot + b_{N-2} + a_2 \cdot + b_{N-3} + \ ... \ + a_{N-1} \cdot + b_0$$
$$...$$
$$c_{N-2} = a_{N-1} \cdot b_{N-1}$$

Počet sčítaní potrebných na výpočet jedného koeficientu c_k pre $0 \leq k < 2N-1$ sa pre $0 \leq k < N$ zväčšuje pre $k \leq N$ sa ich počet zmenšuje. Všeobecne zapísané

$$c_k = \sum_{m=0}^{k} a_m b_{k-m} \tag{3.1}$$

3.1.1 Reprezentácia polynómov bodmi

Polynómy sa dajú reprezentovať aj pomocou bodov v rovine. Na jednoznačné určenie polynómu A $(N-1)$ rádu je potrebné poznať hodnoty $A(x)$ aspoň v N rôznych bodoch $A(x_k) = h_k$ pre $k \leq N$. Tieto hodnoty je možné vypočítať pomocou Hornerovej schémy v čase $O(n^2)$. Súčin polynómov sa dá potom vypočítať pomocou nasledujúceho postupu na obr. 3.1



Obr. 3.1: Konvolúcia

Pomocou Hornerovej schémy vypočítame body $A(x_k)$ a $B(x_k)$ pre všetky $k \leq 2N-2$. Je potrebné zväčšiť počet bodov pôvodných polynómov, nakoľko výsledný polynóm C je stupňa $2N-2$. Potom

$$C(x_k) = A(x_k)B(x_k) \tag{3.2}$$

Pomocou Gausovej eliminácie prevedieme polynóm naspäť z bodovej reprezentácia. Časová zložitosť tohto algoritmu je časová zložitosť Gausovej eliminácie $O(N^3)$.

3.1.2 Použitie Fourierovej transformácie

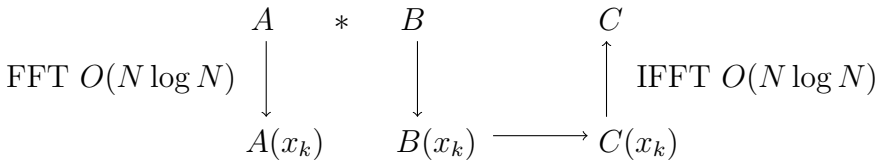
Využijeme postup násobenia pri bodovej reprezentácií polynómov v špeciálne vybraných bodoch x_k. Pri vhodnej voľbe x_k je možné dosiahnuť efektívnejší algoritmus. Za hodnoty x_k pre $k \leq 2N-2$ zvolíme komplexné odmocniny z jednotky.

$$x_k = \omega_{2N-2}^{k} \tag{3.3}$$

V dôsledku použitia komplexných odmocnín jednotky dochádza k ich cirkulácií (2.1), lebo platí pre $0 \leq i < n$, $j > n$, $j \bmod n = i$:

$$\omega_n^i = \omega_n^j \tag{3.4}$$

potom je možné vypočítať hodnoty $A(x_k)$ a $B(x_k)$ pomocou **FFT** v čase $O(NlogN)$ viď obr. 3.2. Pôvodné polynómy rozšírime nulovými koeficientmi na veľkosť $2N-2$, aby výsledný polynóm C bol správneho rádu. Ak by sme tak neurobili išlo by o cirkulárnu konvolúciu, tej sa budem sa venovať v nasledujúcej časti.



Obr. 3.2: Konvolúcia

Jednotlivé koeficienty c_k je možné vypočítame nasledovne

$$c_k = \sum_{m=0}^{2N-2} a_m b_{(k-m)mod(2N-2)} \tag{3.5}$$

Rozdiel oproti vzorcu, kde sme počítali koeficienty pomocou násobenia polynómov je vo vlastnostiach komplexnej odmocniny jednotky.

$$\omega_{2N-2}^k = \omega_{2N-2}^2 N - 2 + k = \omega_{2N-2}^x (2N-2) + k$$

Vďaka tejto vlastnosti pre výpočet ľubovoľného koeficientu c_k je potrebný rovnaký počet súčinov. Tento počet nerastie a neklesá ako pri predchádzajúcom spôsobe ale je konštantnej veľkosti $2N-2$.

3.1.3 Implementácia v Pythone

Násobenie polynómov implementujeme funkciou `nasob(p,q)`. Jej návratová hodnota je polynóm, ktorý je výsledkom násobenia polynómov p a q.

```
>>> A = [1, 5, -6, 18, 2]
>>> B = [6, 2, 8, 1, 4]
>>> nasob(A,B)
array([  6.,  32., -18., 137.,   9., 162.,  10.,  74.,   8.])
```

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

    f_p = np.fft.fft(p2) # aplikovanie fft na vstupne polynomi
    fq = np.fft.fft(q2)

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

    # aplikovanie ifft, pre vysledok staci realna
    return np.fft.ifft(f_res).real  zlozka
```

3.2 Cirkulárna konvolúcia

Cirkulárna konvolúcia sa označuje znakom $\circledast$.

3.2.1 Princíp

Ide o matematický aparát podobný násobeniu polynómov s využitím **FFT**. Avšak pri cirkulárnej konvolúcií nerozširujeme veľkosť vstupných polynómov, vďaka čomu výsledný polynóm je rovnakého rádu ako oba vstupné. Nech na vstupe sú polynómy A a B rádu $N-1$ a výsledný polynóm $C = A \circledast B$ tiež rádu $N-1$.

$$A = a_0 + a_1x + a_2x^2 + a_3x^3 + \ ... \ + a_{N-1}x^{N-1}$$
$$B = b_0 + b_1x + b_2x^2 + b_3x^3 + \ ... \ + b_{N-1}x^{N-1}$$
$$C = c_0 + c_1x + c_2x^2 + c_3x^3 + \ ... \ + c_{N-1}x^{N-1}$$

Koeficienty výsledného polynómu c_k pre $0 \leq k < N$ sú vyjadrené nasledovne

$$c_0 = a_0b_0 + a_1b_{N-1} + a_2b_{N-2} + ... + a_{N-2}b_2 + a_{N-1}b_1$$
$$c_1 = a_0b_1 + a_1b_0 + a_2b_{N-1} + ... + a_{N-2}b_3 + a_{N-1}b_2$$
$$c_2 = a_0b_2 + a_1b_1 + a_2b_0 + ... + a_{N-2}b_4 + a_{N-1}b_3$$
$$...$$

Ako je možné pozorovať na príklade, indexy pri koeficientoch b_x sa cyklicky posúvajú v závislosti od prislúchajúceho indexu pre koeficient c_k a a_y. Súhrnne zapísané

$$c_k = a_0 b_{(k-0)modN} + a_1 b_{(k-1)modN} + ... + a_{N-1} b_{(k-(N-1))modN} \tag{3.6}$$

$$c_k = \sum_{m=0}^{N-1} a_m b_{(k-m)modN} \tag{3.7}$$

3.2.2 Implementácia v Pythone

Funkcia cirkulárnej konvolúcie je veľmi podobná funkcii `nasob`. Funkcia `circular_convolution` iba nedoplní vstupné polynómy nulami.

```
import numpy as np

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

    f_res = np.multiply(fp, fq)

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

3.3 Výpočet posunu dátovej vzorky

Existuje jednorozmerné pole P prirodzených čísel dĺžky N a druhé pole Q, ktoré je cyklickým posunutím poľa P o K prvkov doprava. Nech polia P a Q majú dĺžku $N = 5$ a Q je cyklicky posunuté o $K = 3$ miest. Nasledujúca postupnosť príkazov generuje takéto polia s náhodnými hodnotami od 0 do 100.

```
>>> import numpy as np
>>> from random import randint as ri
>>> N = 5
>>> K = 3
>>> P = np.array([ri(1,100) for i in range(N)])[np.newaxis]
>>> P
array([[99, 40, 80, 15, 56]])
>>> Q = np.roll(P, K)
>>> Q
array([[80, 15, 56, 99, 40]])
```

Problém je zostaviť efektívny algoritmus, ktorý nájde hodnotu cyklického posunu K z polí P a Q.

3.3.1 Funkcia chyby (Error function)

Definujeme si funkciu E veľkosti chyby pre posun k poľa Q k poľu P veľkosti N pre všetky hodnoty $0 \leq k < N$. Funkcia pre posun k vráti súčet druhých mocnín rozdielov prvkov s rovnakým indexom v poliach. Vďaka druhej mocnine $E(k)$ je vždy kladná hodnota a väčšie rozdiely dvojíc prvkov majú väčšiu váhu vo veľkosti chyby.

$$E(k) = \sum_{i=0}^{N-1} (P_i - Q_{(i+k)modN})^2 \tag{3.8}$$

Rovnicu môžeme ďalej rozpísať nasledovne:

$$\begin{aligned} E(k) &= \sum_{i=0}^{N-1} (P_i - Q_{(i+k)modN})^2 && (3.9)\\ &= \sum_{i=0}^{N-1} (P_i^2 - 2P_iQ_{(i+k)modN} + Q_{(i+k)modN}^2) && (3.10)\\ &= \sum_{i=0}^{N-1} P_i^2 - 2\sum_{i=0}^{N-1} P_iQ_{(i+k)modN} + \sum_{i=0}^{N-1} Q_{(i+k)modN}^2 && (3.11) \end{aligned}$$

Skutočný posun K je potom k s najmenšou hodnotou $E(k)$

$$K = argmin(E(k)) \tag{3.12}$$

Pri hľadaní minimálnej hodnoty výsledok neovplyvnia členy rovnaké pre každé k. V upravenej rovnici funkcie E iba stredný člen závisí od hodnoty k a nie je konštantný pre všetky hodnoty k. Potom posun k vyjadríme

$$K = argmin(-2\sum_{i=0}^{N-1} P_iQ_{(i+k)modN}) \tag{3.13}$$

Konštanta 2 rovnako neovplyvňuje výslednú hodnotu k, preto ju môžme zanedbať. Znamienko mínus odstránime a otočíme funkciu $argmin$ na $argmax$, čo nezmení výslednú hodnotu, ale odstráni prácu so zápornými číslami.

$$K = argmin_{(0\leq k<N)}(-2\sum_{i=0}^{N-1} P_iQ_{(i+k)modN}) \tag{3.14}$$

$$= argmin_{(0\leq k<N)}(-\sum_{i=0}^{N-1} P_iQ_{(i+k)modN}) \text{ odstránenie konštanty} \tag{3.15}$$

$$= argmax_{(0\leq k<N)}(\sum_{i=0}^{N-1} P_iQ_{(i+k)modN}) \text{ odstránenie mínus} \tag{3.16}$$

3.3.2 Triviálny algoritmus

Na výpočet hodnoty K z polí P a Q musíme vypočítať sumu $\sum_{i=0}^{N-1} P_iQ_{(i+k)modN}$ pre všetky možné posuny $0 \leq k < N$. Výpočet sumy má časovú zložitosť $O(N)$. Takýchto súm je potrebné vypočítať N. Preto celková zložitosť algoritmu je $O(N^2)$.

```
def find_shift(P,Q):
    N = len(P)
    max_sum = -inf
    k = 0

    for s in range(N):
        P_2 = np.roll(P, -s)
        sum = 0
        for i in range(N):
            sum += P_2[i]*Q[i]

        if sum < max_sum:
            max_sum = sum
            k = s
    return k
```

3.3.3 Algoritmus s využitím cirkulárnej konvolúcie

Výsledkom cirkulárnej konvolúcie pre polia P a Q veľkosti N je pole R. Pre prvok poľa R s indexom x platí

$$R_k = \sum_{i=0}^{N-1} P_i Q_{(x-i)mod(N)} \tag{3.17}$$

tento vzťah je veľmi podobný sume na výpočet skutočného posunu K

$$K = argmax_{(0 \leq k < N)}(\sum_{i=0}^{N-1} P_i Q_{(i+k)modN}) \tag{3.18}$$

Úpravou vstupných parametrov cirkulárnej konvolúcie sa pokúsime docieliť výpočet sumy potrebnej pre zistenie posunu. R_x vyjadríme pomocou jednotlivých súčtov

$$\begin{aligned}
R_0 &= P_0Q_0 + P_1Q_{N-1} + P_2Q_{N-2} + ... + P_{N-2}Q_2 + P_{N-1}Q_1 \\
R_1 &= P_0Q_1 + P_1Q_0 + P_2Q_{N-1} + ... + P_{N-2}Q_3 + P_{N-1}Q_2 \\
R_2 &= P_0Q_2 + P_1Q_1 + P_2Q_0 + ... + P_{N-2}Q_4 + P_{N-1}Q_3 \\
&...
\end{aligned}$$

je potrebné transformáciou π zmeniť pole Q na Q' tak aby platilo

$$\begin{aligned}
R_0 &= P_0Q'_0 + P_1Q'_1 + P_2Q'_2 + ... + P_{N-2}Q'_{N-2} + P_{N-1}Q'_{N-1} \\
R_1 &= P_0Q'_1 + P_1Q'_2 + P_2Q'_3 + ... + P_{N-2}Q'_{N-1} + P_{N-1}Q'_0 \\
R_2 &= P_0Q'_2 + P_1Q'_3 + P_2Q'_4 + ... + P_{N-2}Q'_0 + P_{N-1}Q'_1 \\
&...
\end{aligned}$$

toto je docielené transformovaním Q na Q' nasledovným spôsobom

$$\begin{aligned}
Q'_0 &= Q_0 \\
Q'_1 &= Q_{N-1} \\
Q'_2 &= Q_{N-2} \\
&... \\
Q'_j &= Q_{(N-j)modN} \text{ pre } 0 \leq j < N
\end{aligned}$$

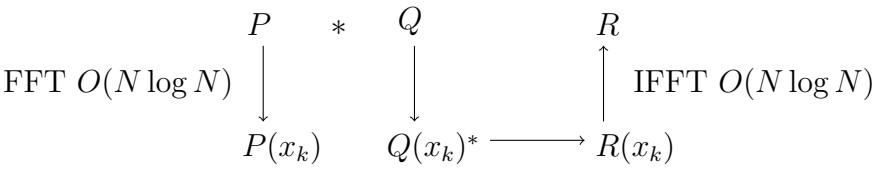
vďaka tejto transformácií z polí P a Q' je možné použitím cirkulárnej konvolúcie vypočítať

$$K = argmax_{(0 \leq k < N)}(\sum_{i=0}^{N-1} P_i Q'_{(i+k)modN}) \tag{3.19}$$

výsledný posun K je index prvku s najväčšou hodnotou v poli $R = P \circledast Q'$. Transformovať môžeme pole pred aj po aplikovaní Fourierovej transformácie pred násobením po bodoch

```
>>> Q = np.array([[80, 15, 56, 99, 40]])
>>> Q_r = Q[::-1]
>>> np.roll(Q_r,1)
array([[40, 80, 15, 56, 99]])
```

Rovnaký výsledok ako s použitím vyššie uvedenej transformácie je možné docieliť úpravou algoritmu cirkulárnej konvolúcie, kde pred uskutočnením násobenia po bodoch nepoužijeme vektor Fourierovej transformácie poľa Q ale jeho komplexne združený vektor.



Obr. 3.3: Zistenie posunu

Keďže časová zložitosť $argmax$ je $O(N)$ celková zložitosť algoritmu je rovnaká ako pri cirkulárnej konvolúcií: $O(N) + O(Nlog(N)) = O(Nlog(N))$.

3.3.4 Implementácia a demonštrácia

```
import numpy as np

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

    fp_reversed = fp[::-1]  # otocenie
    fp = np.roll(fp_reversed,1)  # posun

    f_res = np.multiply(fp, fq)

    r = np.fft.ifft(f_res)

    return np.argmax(r) # vrati index najvacsieho prvku
```

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

    fp = np.conj(fp) # conj - komplexne zdruzeny vektor

    f_res = np.multiply(fp, fq)

    r = np.fft.ifft(f_res)
    return np.argmax(r) # vrati index najvacsieho prvku
```

Správne fungovanie funkcie demonštrujeme na príklade nájdenia cyklického posunu poľa $Q = [80, 15, 56, 99, 40]$ k poľu $P = [99, 40, 80, 15, 56]$.

```
>>> P = np.array([[99, 40, 80, 15, 56]])
>>> Q = np.array([[80, 15, 56, 99, 40]])
>>> find_shift_conj(P,Q)
3
>>> find_shift_otacanie(P,Q)
3
```

3.3.5 Hľadanie podpostupnosti dátovej vzorky

Náš postup sa dá veľmi ľahko upraviť na vyhľadávanie podpostupnosti vzorky v postupnosti - poli. Podobne ako pri násobení polynómov doplnením podpostupnosti nulami dosiahne rovnaký rozmer ako pole. Algoritmus potom nájde posun doplnenej podpostupnosti oproti pôvodnému poľu, čo predstavuje začiatočný index podpostupnosti v hľadanom poli.

```
>>> P = [1,2,3,4,5,6]
>>> Q = [2,3]
>>> Q2 = Q + [0]*(len(P)-len(Q))
>>> Q2
[4, 5, 0, 0, 0, 0]
>>> find_shift(Q2, P)
4     # zlý výsledok skutočný posun je 1
```

Pri otestovaní na dátach s ľubovoľnými nezápornými hodnotami tento algoritmus nefunguje. Náš algoritmus pre každý možný posun vypočíta súčet súčinov prislúchajúcich prvkov polí. Z týchto vypočítaných hodnôt hľadáme maximum. Je veľmi ľahko dokázateľné prečo v našom prípade dostaneme maximum pre posun 4 a nie pre posun 1.

```
>>> P = [1,2,3,4,5,6]
>>> Q = [0,2,3,0,0,0]
>>> sum(np.multiply(P,Q))
13
>>> Q = [0,0,0,0,2,3]
>>> sum(np.multiply(P,Q))
28
```

Pri hľadaní posunu dvoch polí, rovnaké hodnoty v poliach nám garantovali, že súčet súčinov prislúchajúcich prvkov bude najväčší ak prvky súčinov sú rovnaké, a teda ide o druhé mocniny týchto hodnôt. Doplnením núl táto garancia neplatí. Pri násobení po prvkoch všetky dvojice obsahujúce nulu majú nulový výsledok a neovplyvňujú súčet. Nenulové prvky sa násobia s väčšími číslami ako sú ony. Výsledok môže byť v tomto prípade väčší ako pri správnom posune.

Avšak pre binárne polia táto garancia stále platí, nakoľko pri hodnotách 0 a 1 nemôže byť jednotka násobená väčším číslom ako jedna. Preto najväčší súčet súčinov nastane iba v prípade že jednotky v posunutej podpostupnosti odpovedajú jednotkám v poli.

```
>>> Q = [1,1,0,0]
>>> P = [0,0,1,1,0,0,0,1,0,1]
>>> Q2 = Q + [0]*(len(P)-len(Q)) #doplnenie vzorky
>>> find_shift(Q, P)
2  # táto hodnota odpovedá skutočnému začiatku vzorky v poli
```

Tento algoritmus pre hľadanie podpostupnosti nie je najefektívnejší. Knuth–Morris–Pratt algoritmus má časovú zložitosť $O(n)$, nie je však možné rozšíriť ho na vyhľadávanie v 2D.

Kapitola 4

Vyhľadávanie vzoru na obraze bez rotácie a škálovania

4.1 2D diskrétna Fourierova transformácia - DFT2

Fourierova transformácia sa dá ľahko rozšíriť na dvojrozmerné dáta. V jednorozmerných dátach sú výstupom hodnoty vstupnej funkcie $\mathbb{R} \to \mathbb{R}$ v bodoch komplexnej odmocniny jednotky. Výstupom 2D Fourierovej transformácie sú hodnoty funkcie $\mathbb{R}^2 \to \mathbb{R}$ v bodoch $[x, y]$, kde x, y sú komplexné odmocniny jednotky.

$$
\begin{aligned}
F[x,y] &= \sum_{m=0}^{M-1}\sum_{n=0}^{N-1} f[m,n]e^{-i2\pi(\frac{mx}{M}+\frac{ny}{N})} && (4.1)\\
&= \sum_{m=0}^{M-1}\sum_{n=0}^{N-1} f[m,n]\omega_N^{yn}\omega_M^{xm} && (4.2)\\
&= \sum_{m=0}^{M-1}\sum_{n=0}^{N-1} f[m,n]e^{-i2\pi\frac{mxN+nyM}{MN}} && (4.3)\\
&= \sum_{m=0}^{M-1}\sum_{n=0}^{N-1} f[m,n]\omega_{MN}^{ynM+xmN} && (4.4)
\end{aligned}
$$

Z rovnice pre 2D Fourierovu transformáciu je možné zistiť na akom princípe funguje. Vnútorná suma je jednorozmernou Fourierovou transformáciou pre riadok. Potom tieto hodnoty prejdu druhou Fourierovou transformáciou pre stĺpce. Algoritmus počítajúci **FFT** sa dá preto aplikovať pre 2D dáta. Použitím **FFT** je časová zložitosť **FFT2** algoritmu (Fast Fourier transform 2D) $O(N^2 logN)$ pre dáta veľkosti $N \times N$. Tento algoritmus je implementovaný funkciou `np.fft.fft2` v knižnici **NumPy**.

2D inverznú Fourierovu transformácia potom vypočítame nasledovne:

$$f[x,y] = \frac{1}{MN} \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} F[m,n] e^{i2\pi(\frac{mx}{M} + \frac{ny}{N})} \tag{4.5}$$

taktiež je implementovaná funkciou `np.fft.ifft2` v knižnici **NumPy**.

4.1.1 Obrazové zobrazenie

Ak použijeme DFT2 transformáciu na 2D obraz, výsledná Fourierova transformácia F obsahuje veľkosti jednotlivých frekvencií prítomných v obraze. Vďaka rovnici

$$\omega_N^y \omega_M^x = \omega_{MN}^{yM+xN} \tag{4.6}$$

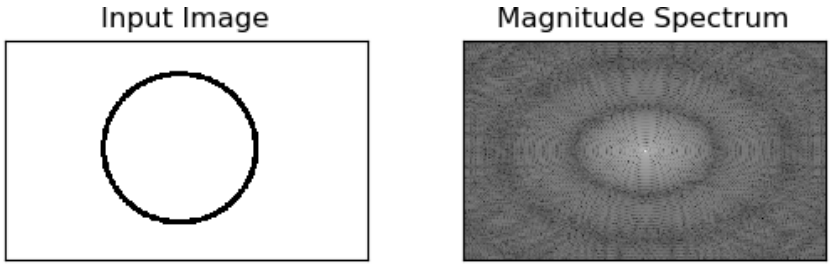
vypočítame frekvencie (hodnoty komplexných odmocnín jednotky) pre 2D pole veľkosti 4×4

	ω_4^0	ω_4^1	ω_4^2	ω_4^3
ω_4^0	ω_{16}^0	ω_{16}^4	ω_{16}^8	ω_{16}^{12}
ω_4^1	ω_{16}^4	ω_{16}^8	ω_{16}^{12}	ω_{16}^{16}
ω_4^2	ω_{16}^8	ω_{16}^{12}	ω_{16}^{16}	ω_{16}^{20}
ω_4^3	ω_{16}^{12}	ω_{16}^{16}	ω_{16}^{20}	ω_{16}^{24}

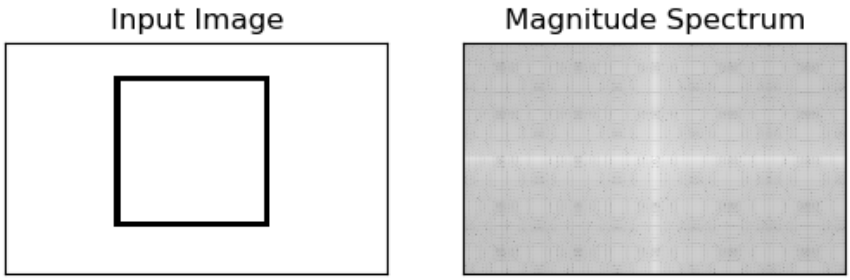
Tabuľka 4.1: Tabuľka frekvencií pre $N = 4$, $M = 4$

transformácia F, viď tabuľka 4.1, bude obsahovať hodnoty týchto frekvencií. Pre lepšie zobrazenie sa zvykne zmeniť poradie v tabuľke. Nulové frekvencie sa presunú do stredu nasledovným spôsobom. F rozdelíme na 4 kvadranty. Ľavý horný vymeníme s pravým dolným a pravý horný vymeníme z ľavým dolným kvadrantom. Funkcia *np.fft.fftshift* robí túto výmenu. Vďaka tomu môžeme zobraziť amplitúdové spektrum Fourierovej transformácie obrázka. V transformácií môžeme zreteľne vidieť rôzne geometrické tvary.

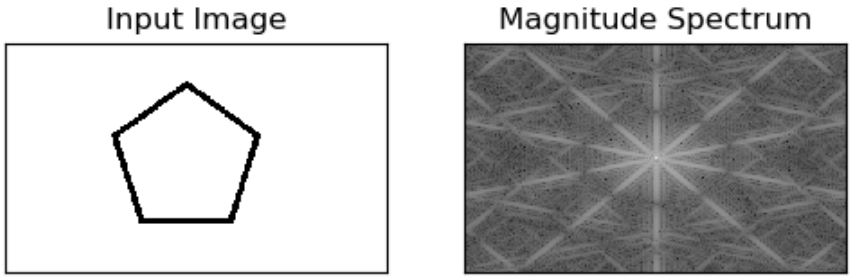
```
import cv2
import numpy as np
from matplotlib import pyplot as plt

def zobraz(name):
    img = cv2.imread(f'{name}.png',0)
    f = np.fft.fft2(img)
    fshift = np.fft.fftshift(f)
    magnitude_spectrum = np.abs(fshift)
    magnitude_spectrum = np.log(magnitude_spectrum) #log pre normalizáciu hodnôt

    plt.subplot(121),plt.imshow(img, cmap = 'gray')
    plt.title('Input Image'), plt.xticks([]), plt.yticks([])
    plt.subplot(122),plt.imshow(magnitude_spectrum, cmap = 'gray')
    plt.title('Magnitude Spectrum'), plt.xticks([]), plt.yticks([])
    plt.show()

>>> zobraz('kruznica')
```



Obr. 4.1: FFT2 kružnica



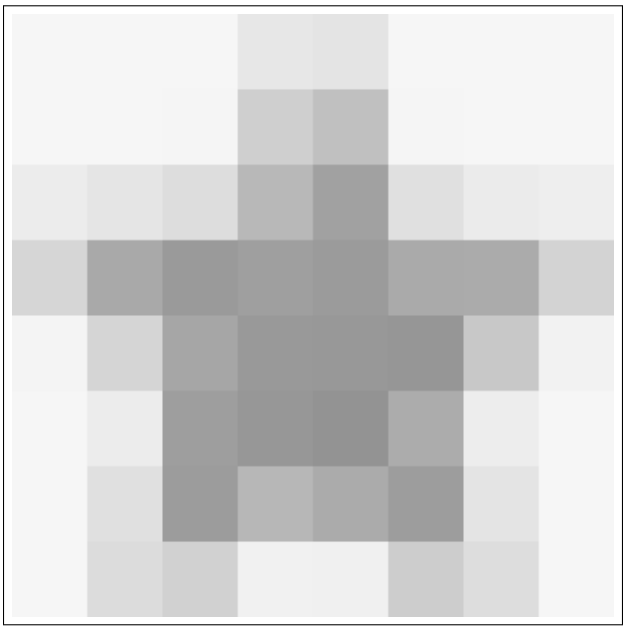
Obr. 4.2: FFT2 štvorec



Obr. 4.3: FFT2 päťuholník

4.1.2 Grafické demonštrácia inverznej Fourierovej transformácie 2D

Inverzná Fourierova transformácia na 2D dátach vyzerá veľmi zložito a je ťažké si predstaviť ako sa z hodnôt DFT (diskrétna Fourierova transformácia) vytvorí naspäť pôvodný obraz. Máme šedo-tónový obraz hviezdy rozmerov 8×8 pixlov (obr. 4.4).



Obr. 4.4: Hviezda 8x8 v odtieňoch sivej

Väčšinou sa pri vizualizácií DFT používa iba magnitúdové spektrum, pričom prichádzame o fázovú zložku. Tento problém je spôsobený tým, že DFT obsahuje komplexné hodnoty skladajúce sa z dvoch častí - reálnej a imaginárnej. Jednému pixlu neodpovedá iba jedna hodnota ale dve hodnoty pre reálnu a imaginárnu zložku. Aby som v obrázku zachoval túto informáciu, hodnotu reálnej zložky bude *modrá* zložka v RGB farbe a imaginárna zložka bude *zelená*.

Vzorec IDFT (inverzná diskrétna Fourierova tranformácia) si rozložíme na menšie časti.

$$f[x,y] = \frac{1}{MN}\sum_{m=0}^{M-1}\sum_{n=0}^{N-1} F[m,n]e^{i2\pi(\frac{mx}{M}+\frac{ny}{N})} \tag{4.7}$$

Zlomok $\frac{1}{MN}$ je normalizačná zložka rovnaká pre všetky $f[x,y]$ to môžeme zatiaľ v našom znázornení zanedbať. Ostane nám

$$\sum_{m=0}^{M-1}\sum_{n=0}^{N-1} F[m,n]\omega_M^{mx}\omega_N^{ny} \tag{4.8}$$

Každému prvku $F[m,n]$ odpovedá jeden prvok $\omega_M^{mx}\omega_N^{ny}$. Keďže F je matica 8×8 a my prejdeme všetkými jej prvkami prvky $\omega_M^{mx}\omega_N^{ny}$ musia tvoriť rovnako veľkú prislúchajúcu maticu pre x a y. Táto matica vznikne vynásobením dvoch vektorov komplexných odmocnín jednotky - jeden vektor pre ω_M^{mx} a druhý pre ω_N^{ny}

$$\begin{bmatrix} \omega_N^0 \\ \omega_N^y \\ \omega_N^{2y} \\ \dots \\ \omega_N^{(N-1)y} \end{bmatrix} \begin{bmatrix} \omega_M^0 & \omega_M^x & \omega_M^{2x} & \dots & \omega_M^{(M-1)x} \end{bmatrix} = \begin{bmatrix} \omega_M^0\omega_N^0 & \omega_M^0\omega_N^y & \dots & \omega_M^0\omega_N^{(N-1)y} \\ \omega_M^x\omega_N^0 & \omega_M^x\omega_N^y & \dots & \omega_M^x\omega_N^{(N-1)y} \\ \omega_M^{2x}\omega_N^0 & \omega_M^{2x}\omega_N^y & \dots & \omega_M^{2x}\omega_N^{(N-1)y} \\ \dots & \dots & \dots & \dots \\ \omega_M^{(M-1)x}\omega_N^0 & \omega_M^{(M-1)x}\omega_N^y & \dots & \omega_M^{(M-1)x}\omega_N^{(N-1)y} \end{bmatrix} \tag{4.9}$$

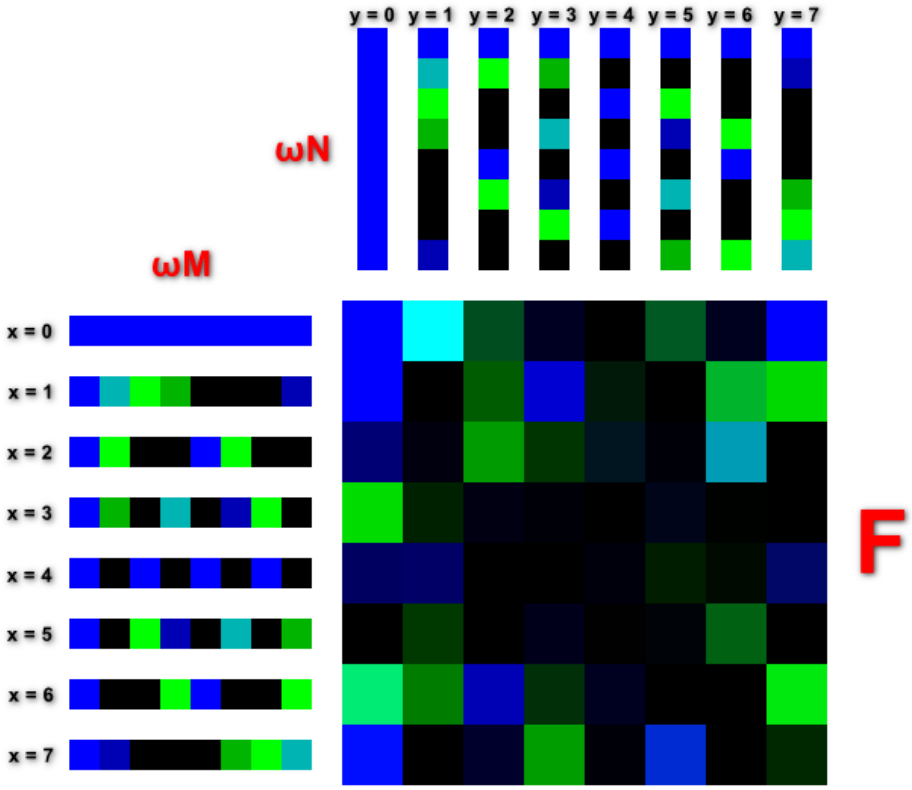
$$\omega M = \begin{bmatrix} \omega_M^0 & \omega_M^x & \omega_M^{2x} & \dots & \omega_M^{(M-1)x} \end{bmatrix} \tag{4.10}$$

$$\omega N = \begin{bmatrix} \omega_N^0 \\ \omega_N^y \\ \omega_N^{2y} \\ \dots \\ \omega_N^{(N-1)y} \end{bmatrix} \tag{4.11}$$

Potom sa dá IDFT zapísať ako násobenie matíc

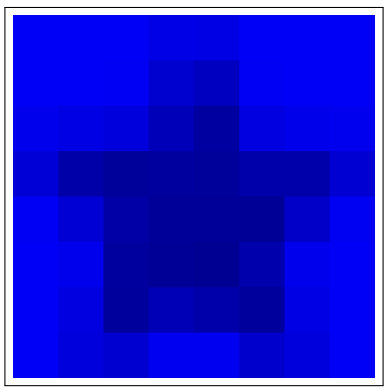
$$f = \frac{1}{MN}\omega N\omega M F \tag{4.12}$$

V obrázku 4.5 sú zakreslené tieto vektory a matica Fourierovej transformácie vo forme modro-zelených obrázkov. Modrá farba reprezentuje reálnu časť a zelená imaginárnu časť čísla.



Obr. 4.5: IDFT 2D

Vynásobením týchto komponentov získame napäť náš pôvodný obrázok v odtieňoch modrej 4.6 keďže sa nám komplexné zložky vynulujú a ostanú znova iba reálne hodnoty zhodné s pôvodným obrazom.



Obr. 4.6: Hviezda 8x8 v odtieňoch modrej

4.2 Výpočet posunu v 2D dátach

Nech existuje dvojrozmerné pole P a Q s hodnotami $x_ij \in$
R, pričom Q je cyklicky posunuté o K_1 riadkov smerom nadol a o K_2 stĺpcov smerom doprava.

```
>>>P = np.array([\
     [1,2,3,4,5],\
     [1,3,5,7,9],\
     [0,2,4,6,8],\
     [5,4,3,2,1],\
     [3,3,3,2,2]])
>>> Q = np.roll(P,(2,3),axis=(1,0))
>>> Q
array([[6, 8, 0, 2, 4],
       [2, 1, 5, 4, 3],
       [2, 2, 3, 3, 3],
       [4, 5, 1, 2, 3],
       [7, 9, 1, 3, 5]])
```

Primitívny algoritmus by skúsil posúvať pole P o všetky možné posuny (N^2 možností) a pre každý posun vypočítal súčet súčinov vzniknutého poľa s polom Q po prvkoch. Najväčšia hodnota súčtu by zodpovedala skutočnému posunu poľa. Tento algoritmus nie je veľmi efektívny, jeho časová zložitosť pre vstup veľkosti $N \times N$ je $O(N^4)$.
Algoritmus na výpočet posunu pre jednorozmerné polia s využitím **FFT** mal zložitosť $O(NlogN)$. Ak v tomto algoritme nahradíme **FFT** za **FFT2** vznikne algoritmus schopný vypočítať posun dvoch dvojrozmerných polí.

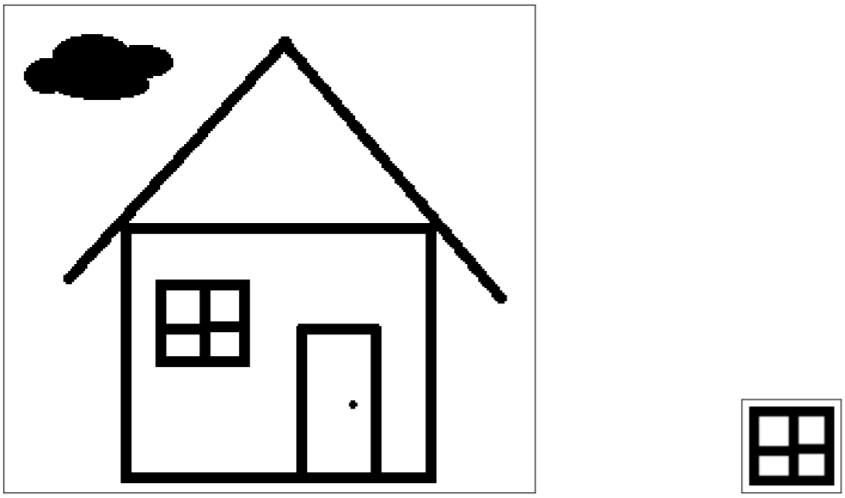
```
>>> fP = np.fft.fft2(P)
>>> fQ = np.fft.fft2(Q)
>>> fR = np.multiply(fP, np.conj(fQ))
>>> R = np.fft.ifft2(fR)
# np.argmax vracia int aj pre 2D dáta, np.unravel_index vypočíta súradnice
>>> shift = np.unravel_index(np.argmax(np.real(R)),R.shape)
>>> shift
(2, 3)
```

Časová zložitosť tohto algoritmu je rovnaká ako zložitosť **FFT2**: $O(N^2 \log N)$

4.3 Vyhľadávanie vzoru na binárnom obraze

V predchádzajúcej časti sú uvedené algoritmy hľadajúce podpostupnosť v poli. Z týchto algoritmov najefektívnejší z možnosťou využitia pre 2D polia bol algoritmus s využitím **FFT**. 2D binárne polia môžu predstavovať dvojfarebné obrázky. Pôvodný algoritmus upravíme podobne ako pri výpočte posunu. Náhrada **FFT** za **FFT2** umožní vyhľadávať podobraz - vzorku na obraze. Časová zložitosť algoritmu pri jednorozmerných dátach veľkosti N je $O(NlogN)$, čo je časová zložitosť **FFT**. Podobne časová zložitosť algoritmu pri dvojrozmerných dátach veľkosti $N \times N$ je $O(N^2logN)$ a je rovnaká ako zložitosť **FFT2**

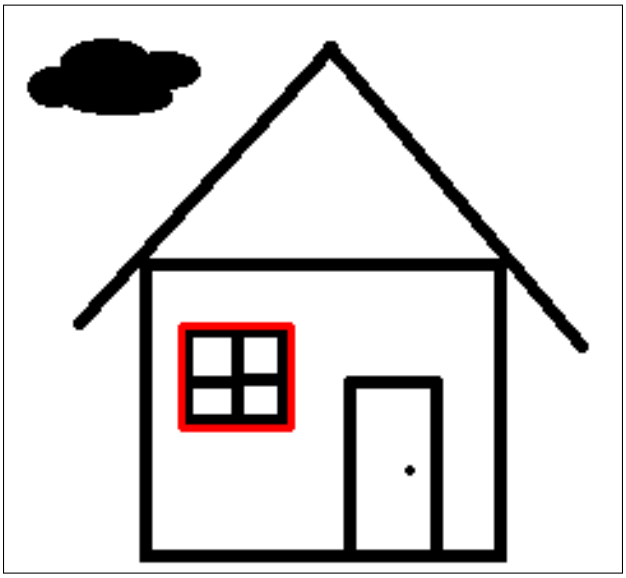
Príklad



Obr. 4.7: Binárny obraz - Dom a okno

Funkčnosť algoritmu demonštrujeme na príklade vyhľadania obrázku **okna** na obraze **domu** 4.7. Okno je identické z oknom na dome. Skutočná pozícia ľavého horného rohu okna na dome je $[66, 122]$

```
def find_shift(image, patern):
img1 = image.astype(np.float32)
    img1 = 1 - img1 / 255 # in opencv 255 is white this conver black to 1
                          # and white to 0
    img2 = np.zeros(img1.shape, dtype=np.float32)
    # add zero padding to obj
    img2[0:patern.shape[0],0:patern.shape[1]] = 1 - patern / 255

    f1 = np.fft.fft2(img1)
    f2 = np.fft.fft2(img2)
    f_res = np.multiply(f1, np.conj(f2))
    res = np.fft.ifft2(f_res)

    y,x =  np.unravel_index(np.argmax(np.real(res)),res.shape)
    return x, y

>>> dom = cv2.imread('bin_images/dom.png',0)
>>> okno = cv2.imread('bin_images/okno.png',0)
>>> find_shift(dom, okno)
(66, 122)
```



Obr. 4.8: Nájdené okno

Algoritmus správne vypočítal súradnice okna. Nájdené okno je vyznačené na obrázku 4.8

4.4 Vyhľadávanie vzoru na šedo-tónovom obraze

Podobne, ako pri hľadaní podreťazca v binárnych dátach algoritmus pre obrazy s hodnotami od 0 do 255 nemusí vrátiť správny výsledok. Príčinou je počítanie súčtu súčinov vzorky a prislúchajúcej časti obrazu ako bolo vysvetlené v časti o hľadaní podreťazca.

12	10	0
56	128	255
32	150	200

vzorka

255	255	255
255	255	255
255	255	255

prislúchajúca časť obrazu

Obr. 4.9: Vzorka porovnávaná s nezodpovedajúcou časťou obrazu

Ak nastane situácia ako na obr. 4.9 algoritmus vráti túto časť obrazu ako vzor. Ide o chybný výsledok.

4.4.1 Hľadanie najbližšieho súčtu

Priamym využitím algoritmu na vyhľadávanie v binárnom obraze nevieme s určitosťou nájsť posun vzorky v šedo-tónovom obraze, pri hľadaní maxima. Avšak je možné ľahko určiť súčet súčinov pixlov vzorky a časti obrazu pri ktorom časť obrazu odpovedá vzorke.

12	10	0
56	128	255
32	150	200

vzorka

12	10	0
56	128	255
32	150	200

prislúchajúca časť obrazu

12^2	10^2	0^2
56^2	128^2	255^2
32^2	150^2	200^2

súčin po pixeloch

148313

súčet súčinov

Obr. 4.10: Súčet súčinov pri dobrom posune vzorky

Ak pozmeníme algoritmus využívajúci cirkulárnu konvolúciu aby nehľadal maximum ale hľadal hodnotu súčtu druhých mocnín pixlov vo vzorke (4.10), resp. číslo mu blízke, dokážeme zistiť presne pozíciu (posun) vzorky v obraze.

```
def find(a, b):
    '''find b in a'''
    A = a
    B = np.zeros(A.shape)
    B[0:b.shape[0],0:b.shape[1]] = b

    FA = np.fft.fft2(A)
    FB = np.fft.fft2(B)
    D = np.fft.ifft2( np.multiply(A, np.conj(B)) )

    # najdenie majblizsieho cisla k b**2
    expected_sum = np.sum(b**2)
    error = np.abs(D - expected_sum)

    return np.unravel_index(np.argmin(error), D.shape)
```

Algoritmus v tejto podobe očakáva na vstupe obraz a vzor, v ktorých číslo 0 označuje bielu a číslo 255 čiernu (prípadne číslo 1 čiernu farbu). Algoritmus otestujeme na obraze 4.11

Obr. 4.11: Dom a okno číslo 2

Na obrázku 4.12 je je výsledok hľadania okna na obraze domu pre dva vyššie spomenuté algoritmy. Červeným rámom je označený výsledok algoritmu hľadajúci maximum, v zelenom ráme je výsledok algoritmu hľadajúci hodnotu najbližšiu k očakávanému súčtu.

Obr. 4.12: Nájdené vzory na obraze

4.5 Fázová korelácia

V reálnom svete sú málokedy k dispozícií vyhľadávané vzory úplne rovnaké ako na obraze. Môžu sa líšiť v kontraste, jase, rozmeroch, natočení a podobne. Z týchto zmien sa budeme venovať zmene kontrastu. Pri zmene kontrastu je nová hodnota pixlu násobkom pôvodnej hodnoty pixlu a konštanty, rovnakej pre celý obraz. Zmena kontrastu nastáva napríklad pri zmene svetelných podmienok.

Hodnota Fourierovej transformácie pôvodného obrazu pre jednotlivé pixly sa skladá z amplitúdy A a fázy φ.

$$a_{i,j} \rightarrow_{FFT} F_1[i,j] = Ae^{\varphi} \tag{4.13}$$

Po zmene kontrastu, vynásobeniu obrazu konštantou c, hodnota Fourierovej transformácie nového obrazu pre jednotlivé pixly je

$$ca_{i,j} \rightarrow_{FFT} F_2[i,j] = cAe^{\varphi} \tag{4.14}$$

Z týchto rovníc vyplýva, že aj pred aj po vynásobení obrazu konštantou veľkosť fázy DFT je rovnaká.

Táto vlastnosť sa dá využiť pri hľadaní vzoru na obraze. Aplikujeme DFT na obraz a všetky amplitúdy $F[i,j]$ upravíme na veľkosť jedna ale zachováme fázu - pre nulové amplitúdy nastavíme na amplitúdu $A = 1$ a uhol $\varphi = 0$, teda $F[i,j] = 1$. Prvky s nenulovou amplitúdou vydelíme veľkosťou amplitúdu $F[i,j] = F[i,j]/|F[i,j]|$. DFT obrazu bude teraz obsahovať iba informáciu o fáze, amplitúdu zanedbáva ako sa uvádza v článku [1].

4.5.1 Algoritmus fázovej korelácie

Analogicky ako pri predchádzajúcich algoritmoch postup je nasledovný. Hľadaný vzor doplníme nulami (pre obraz kde 0 je biela a 255 je čierna) do veľkosti obrazu, na ktorom vyhľadávame. Vytvoríme Fourierovu transformáciu obrazu F_a a doplneného vzoru F_b. Nájdeme komplexne združený obraz k F_b, označíme ho F_b^*, namiesto transformovania obrazu pred aplikovaním DFT (viď Algoritmus s využitím cirkulárnej konvolúcie). Eliminujeme amplitúdu F_a a F_b^*. Následne postupujeme rovnako ako pri algoritme pre hľadanie vzorky v binárnom obraze. Po aplikovaní IDFT na F_a a F_b^* nájdeme maximálny prvok, ktorého súradnice predstavujú súradnice vzoru na obraze. Ak zapíšeme úpravu amplitúd a násobenie po prvkoch dostaneme [1]

$$\frac{F_a}{|F_a|} \cdot \frac{F_b^*}{|F_b^*|} = \frac{F_a c F_b}{|F_a||F_b^*|} \tag{4.15}$$

Táto vlastnosť sa nazýva korelácia. Nakoľko výsledok závisí iba na fáze ide o **fázovú koreláciu** [1].

```
def phase(x):
    n = np.abs(x)
    n[n==0] = 1 #pre tie prvky x kde je amplitúda nulová
    return x/n

def find(a, b):
    A = a
    B = np.zeros(a.shape)
    B[0:b.shape[0],0:b.shape[1]] = b

    A = phase(A)
    B = phase(B)
    FA = np.fft.fft2(A)
    FB = np.fft.fft2(B)
    FD = np.multiply(A, np.conj(B))

    D = np.fft.ifft2(FD)
    return np.unravel_index(np.argmin(D), D.shape)
```

Obr. 4.13: Dom a okno so zmeneným kontrastom

Algoritmus s využitím fázovej korelácie spolu s ostatnými algoritmami otestujeme na obraze domu a okna 4.13, pričom tentokrát okno oproti pôvodnému má zmenený kontrast. Vo výslednom obraze 4.14 modrou farbou je označený výsledok algoritmu s využitím fázovej korelácie, zelenou algoritmus hľadajúci najbližší súčet a červenou algoritmus hľadajúci maximum.

Obr. 4.14: Nájdené vzory na obraze použitím rôznych algoritmov:
- modrá: algoritmus využívajúci fázovú koreláciu
- zelená: algoritmus najbližšieho súčtu
- červená: algoritmus hľadajúci maximum

Z výsledku je možné konštatovať, že algoritmus využívajúci fázovú koreláciu je oproti ostatným pri zmene kontrastu presnejší.

Kapitola 5

Vyhľadávanie vzoru na obraze s rotáciou a škálou

Táto kapitola bude venovaná vysvetleniu algoritmu, ktorý nájde vzor na obraze ak je vzor na obraze posunutý, otočený a zväčšený alebo zmenšený zároveň. Algoritmus bude vysvetlený podľa článku [2]. Nakoľko ide o zložité matematické operácie, niektoré časti budú vysvetlené zjednodušene bez detailného odvodenia. Rovnako pre algoritmy jednotlivých komponentov hlavného algoritmu budú popísané princípy fungovania ale nie sú uvedené vlastné implementácie, nakoľko sú implementované v knižnici OpenCV.

5.1 Posun - rekapitulácia

Nájdenie pozície vzoru na obraze, ak sa vzor na obraze nachádza v rovnakej veľkosti a bez rotácie. V predchádzajúcej kapitole je vysvetlený algoritmus využívajúci fázovú koreláciu, ktorý rieši tento problém.
Závislosť vzoru a obrazu sa dá matematicky vyjadriť. Uvažujme, že vzor je cyklicky posunutý obraz . Potom ich vzťah je možné vyjadriť nasledovne:

$$f_{vzor}(x, y) = f_{obraz}(x - x_0, y - y_0) \tag{5.1}$$

$$F_{vzor}(\xi, \eta) = F_{obraz}(\xi, \eta) \cdot e^{-2\pi i(\xi x_0 + \eta y_0)} \tag{5.2}$$

po aplikovaní Fourierovej transformácie na obraz dochádza k osamostatneniu člena $e^{-2\pi i(\xi x_0+\eta y_0)}$ nakoľko táto hodnota ostáva konštantná pre ξ a η. ξ a η reprezentujú súradnice Fourierovej transformácie, aby boli odlíšené od x a y v pôvodnom obraze. Posun sa dá teraz ľahko nájsť s využitím fázovej korelácie.

5.2 Rotácia a posun

Na nájdenie posunu rotovaného obrazu priamočiare aplikovanie fázovej korelácie nebude fungovať, lebo nikdy sa nám nepodarí iba posúvaním vzoru nájsť jeho polohu na obraze. Na nájdenie veľkosti rotácie zatiaľ nemáme žiadny spôsob. Vzťah vzoru a obrazu s otočením Θ_0 a posunom x_0, y_0 sa dá vyjadriť nasledovne:

$$f_{vzor}(x,y) = f_{obraz}(x\cos\Theta_0 + y\sin\Theta_0 - x_0, -x\sin\Theta_0 + y\cos\Theta_0 - y_0) \tag{5.3}$$

$$F_{vzor}(\xi,\eta) = F_{obraz}(\xi\cos\Theta_0 + \eta\sin\Theta_0, -\xi\sin\Theta_0 + \eta\cos\Theta_0)\cdot e^{-2\pi i(\xi x_0+\eta y_0)} \tag{5.4}$$

Podobne ako iba pri posune sa nám osamostatnil člen $e^{-2\pi i(\xi x_0+\eta y_0)}$. V tejto podobe nevieme žiadnym postupom zistiť uhol Θ_0 alebo posun (x_0, y_0). Vytvoríme magnitúdové spektrum, čím sa zbavíme člena $e^{-2\pi i(xx_0+yy_0)}$. Tento člen predstavuje vektor na jednotkovej kružnici, a teda jeho amplitúda je vždy rovná jednej, preto nemá vplyv na celkovú veľkosť magnitúdy.

$$M_{vzor}(\xi,\eta) = M_{obraz}(\xi\cos\Theta_0 + \eta\sin\Theta_0, -\xi\sin\Theta_0 + \eta\cos\Theta_0) \tag{5.5}$$

Vidíme, že magnitúdové spektrá vzoru a obrazu sú rovnaké, iba jedno je rotovanou kópiou druhého. Teraz M_{vzor} aj M_{obraz} sú v algebraickom tvare. Ak by sme ich vedeli transformovať do polárneho tvaru priamo by sme získali informáciu o uhle rotácie. Vzťah by vyzeral nasledovne:

$$M_{vzor}(\rho,\Theta) = M_{obraz}(\rho,\Theta-\Theta_0) \tag{5.6}$$

Toto sa dá dosiahnuť aplikovaním polárnej transformácie na pôvodný obraz, ale je dokázané, že rovnaký výsledok dostaneme ak je transformácia aplikovaná na Fourierovu transformáciu obrazu.

Aplikovaním transformácie sa informácia o rotácií zmenila na informáciu o posune v polárnych súradniciach, ktorý je možné pomocou fázovej korelácie vypočítať. Problémom je však transformácia na polárne súradnice - vysvetlenie v nasledujúcej časti. Zo vzťahu vyplýva, že veľkosť hľadaného uhlu, v polárnej reprezentácií posunu, závisí iba od druhej súradnice. Preto keď pomocou fázovej korelácie nájdeme maximum o veľkosti rotácie hovorí iba číslo riadka v ktorom sa prvok s maximálnou hodnotou nachádza.

5.2.1 Polárna transformácia

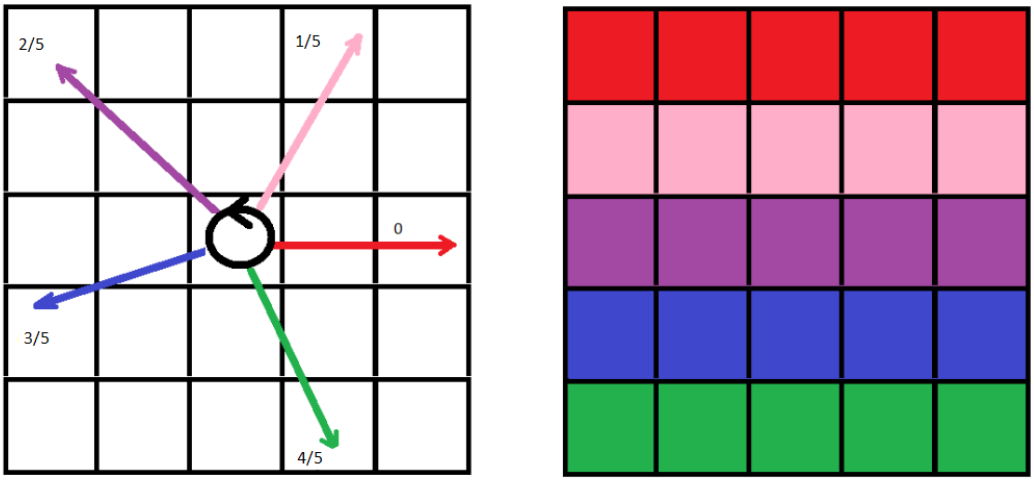
Potrebujeme zistiť spôsob transformácie obrazu, z algebraickej reprezentácie na polárnu. Určíme stred rotácie transformácie, bod od ktorého sa počíta veľkosť uhla. Následne sa pre všetky uhly zistia hodnoty pixlov obrazu ležiacich na priamke zvierajúcej daný uhol s priamkou reprezentujúcou uhol 0π. Tieto hodnoty sa namapujú do nového obrazu v poradí podľa veľkosti uhlu a vzdialenosti od stredu rotácie.

$$f(x, y) \rightarrow f(\rho, \Theta) \tag{5.7}$$

$$\rho = \sqrt{x^2 + y^2} \tag{5.8}$$

$$\Theta = tan^{-1}\frac{y}{x} \tag{5.9}$$

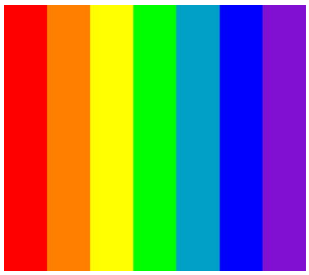
Grafická reprezentácia tejto transformácie vyzerá nasledovne 5.1:



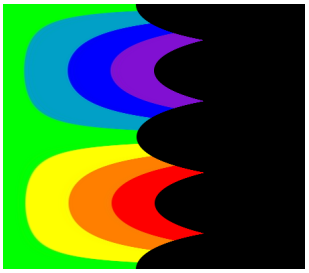
Obr. 5.1: Polárna transformácia - grafická reprezentácia

Polárnu transformáciu je možné implementovať použitím OpenCV funkcie `cv2.remap`. Ukážka na reálnom obraze 5.2 5.3.

```
map_x = np.zeros((rows, cols))
map_y = np.zeros((rows, cols))

theta_d = 2*np.pi / rows
theta = 0

for i in range(rows):
    for j in range(cols):
        map_x[i,j] = np.sin(theta)*j + centre[1]
        map_y[i,j] = np.cos(theta)*j + centre[0]
    theta += theta_d

map_x = map_x.astype(np.float32)
map_y = map_y.astype(np.float32)

pol = cv2.remap(img, map_x, map_y, 1)
```



Obr. 5.2: Obraz - farebné pásy



Obr. 5.3: Polárna transformácia obrazu

Polárna transformácia je taktiež implementovaná v knižnici OpenCV metódou `cv2.linearPolar(obr., stred, polomer, flags)`.

```
>>> img = cv2.imread('rainbow2.jpg')
>>> centre = img.shape[0]//2, img.shape[1]//2
>>> radius = (centre[0]**2 + centre[1]**2)**0.5
>>> res = cv2.linearPolar(img, centre, radius, 8)
>>> cv2.imshow('polar',res)
```

Rozdiel oproti funkcii `cv2.linearPolar` a vlastnej implementácií je v začiatočnom uhle. Linear Polar začína otočená na sever. Vo vlastnej implementácií je možné si určiť začiatok rotácie.

Problémom podobných transformácií je strata informácie obrazu. Ako vidieť na grafickej reprezentácií je potrebné niektoré hodnoty spriemerovať a dopočítavať aby sme dostali výsledný obraz. Niektoré uhly prechádzajú hranicou pixlov alebo mapujú väčší počet pixlov z pôvodnej reprezentácie na menší počet v polárnej reprezentácií. Tým dochádza k strate informácie. Pri analógových obrazoch nie je takýto problém nakoľko nás nelimituje počet pixlov ale obraz sa dá rozoberať prakticky na ľubovoľne malé časti.

5.3 Zmena škály

Zmenou škály alebo škálovaním obrazu rozumieme transformáciu, keď škálovaný obraz sa líšia od seba iba v rozmeroch. Rozmery škálovaného obrazu sú násobkom rozmerov pôvodného obrazu. Veľkosť škály udávajú dve čísla (a, b), kde a je koeficient zväčšenia v smere osi y a b je koeficient zväčšenia v smere osi x.
Nech f_{vzor} je škálovaná replika f_{obraz}, pričom (a, b) sú koeficienty škály. Potom ich Fourierove transformácie sú v nasledujúcom vzťahu:

$$F_{vzor}(\xi, \eta) = \frac{1}{ab} F_{obraz}(\xi/a, \eta/b) \tag{5.10}$$

Podobne ako pri rotácií nie je možné z tohto vzťahu známym algoritmom priamo zistiť škálu. Znova však existuje transformácia, ktorá delenie a násobenie pretransformuje na rozdiel a súčet. Logaritmus má žiadanú vlastnosť, ktorá vie zmeniť podiel vnútri logaritmu na rozdiel logaritmov. Ak aplikujeme **logaritmickú** transformáciu dostaneme

$$\begin{aligned} F_{vzor}(\log \xi, \log \eta) &= F_{obraz}(\log \xi/a, \log \eta/b) && (5.11) \\ &= F_{obraz}(\log \xi - \log a, \log \eta - \log b) && (5.12) \end{aligned}$$

Člen $\frac{1}{ab}$ môžeme zanedbať, lebo je stále konštantný. Logaritmické výrazy nahradia tieto premenné

$$x = \log \xi, y = \log \eta \tag{5.13}$$

$$c = \log a, d = \log b \tag{5.14}$$

Teraz vzťah transformácií vyzerá nasledovne.

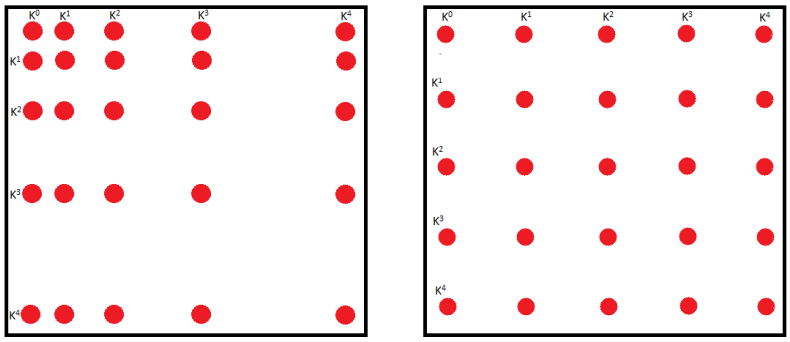
$$F_{vzor}(x, y) \quad = \quad F_{obraz}(x - c, y - d) \tag{5.15}$$

Teraz je už zreteľné, že znova ide o výpočet posunu vzoru voči obrazu. Tento problém vyriešime algoritmom využívajúcim fázovú koreláciu. Po zistení premenných c a d, umocnením základu logaritmu na tieto premenné získame hodnoty pôvodných koeficientov škály. Pre prirodzený logaritmus sú hodnoty koeficientov $a = e^c$ a $b = e^d$.

5.3.1 Logaritmická transformácia

Logaritmická transformácia funguje nasledovne. V x-ovej aj y-ovej osi vyberieme referenčné pixel ktoré budú od seba rovnako vzdialené na transformovanom obraze. Tieto pixel vyberieme na základe exponenciálnej postupnosti základu logaritmu k, podľa ktorého je transformácia realizovaná. Z prvého riadku vyberieme k^0, k^1, k^2... -tý pixel a rozmiestnime ich do prvého riadku nového obrazu tak, aby boli medzi nimi rovnaké medzery. Medzery sa potom vyplnia hodnotami ktoré získame z pixlov medzi referenčnými pixlami. Rovnako postupujeme aj pri spracovaní stĺpcov.

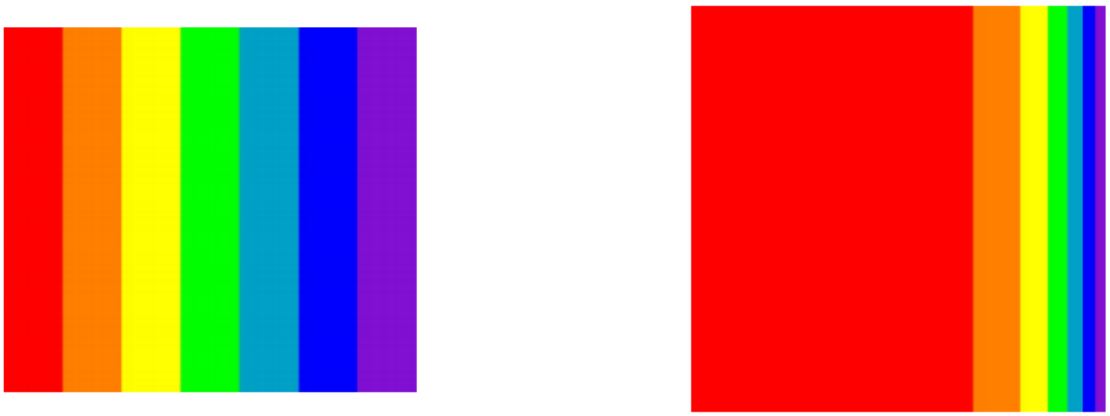
Grafická reprezentácia tejto transformácie vyzerá nasledovne 5.4:



Obr. 5.4: Grafická reprezentácia logaritmickej transformácie

V openCV nie je táto transformácia priamo implementovaná. Dá sa však pomocou metódy `remap` ľahko implementovať. Základ logaritmu môže mať rôzne hodnoty. Veľké hodnoty základu spôsobia , že výsledný obraz bude veľmi malý, preto v ukážke je zvolená hodnota 1.017, obr. 5.5.

```
img = cv2.imread('rainbow2.jpg')
map_x = np.zeros(img.shape[:-1])
map_y = np.zeros(img.shape[:-1])
cols = img.shape[1]
rows = img.shape[0]
LOG_BASE = 1.017

for i in range(rows):
    for j in range(cols):
        map_x[i,j], map_y[i,j] = LOG_BASE**j, LOG_BASE**i
map_x = map_x.astype(np.float32)
map_y = map_y.astype(np.float32)
log = cv2.remap(img, map_x, map_y, 1);
```



Obr. 5.5: Obraz a jeho logaritmická transformácia so základom 1.017

5.4 Zmena škály a rotácia

Nech existuje vzor f_{vzor}, ktorý sa nachádza zväčšený , koeficientom a pre oba smery, otočený o uloh Θ_0 a posunutý o x_0 a y_0 na obraze f_{obraz}. Aby sme zistili otočenie aj koeficient škály po aplikovaní Fourierovej transformácie aplikumeje polárnu a nakoniec logaritmickú transformáciu. Ak (x, y) je škálované na $(x/a, y/a)$, potom polárna transformácia vyzerá nasledovne

$$\rho_1 = \sqrt{x^2 + y^2} \tag{5.16}$$

$$\Theta_1 = tan^{-1}\frac{y}{x} \tag{5.17}$$

$$\rho_2 = \sqrt{(x/a)^2 + (y/a)^2} = \frac{1}{a}\sqrt{x^2 + y^2} = \frac{\rho_1}{a} \tag{5.18}$$

$$\Theta_2 = tan^{-1}\frac{y/a}{x/a} = tan^{-1}\frac{y}{x} = \Theta_1 \tag{5.19}$$

Magnitúdové spektrum v polárnej reprezentácií obrazu a vzoru je

$$M_1(\rho, \Theta) = M_2(\rho/a, \Theta - \Theta_0) \tag{5.20}$$

Teraz sa aplikuje logaritmická transformácia.

$$M_1(\log\rho, \Theta) = M_2(\log\rho - \log a, \Theta - \Theta_0) \tag{5.21}$$

$$\psi = \log\rho \tag{5.22}$$

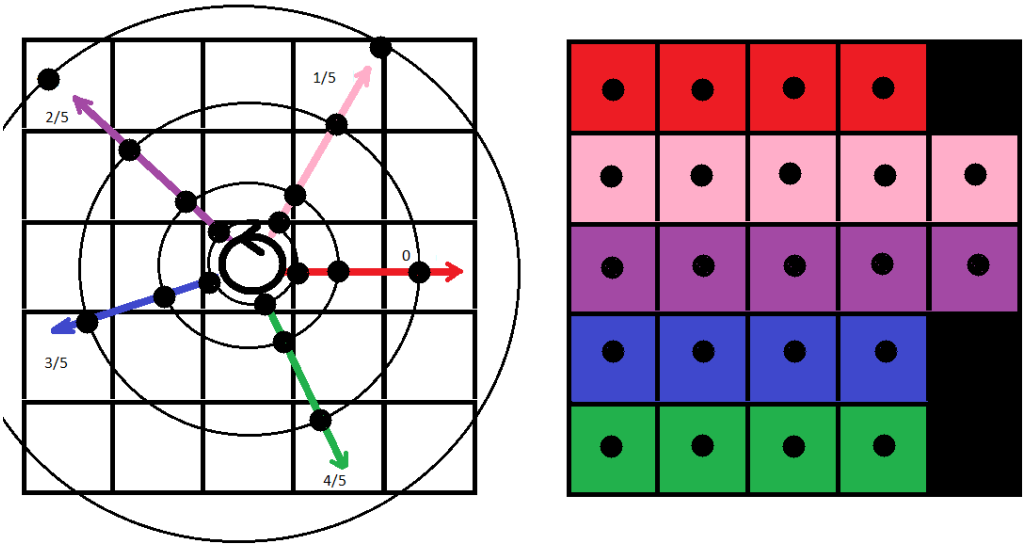
$$d = \log a \tag{5.23}$$

$$M_1(\psi, \Theta) = M_2(\psi - d, \Theta - \Theta_0) \tag{5.24}$$

Z tohto vzorca použitím fázovej korelácie viem zistiť veľkosť otočenia Θ_0 a koeficient škály a. Aplikovanie polárnej a potom logaritmickej transformácie sa dá nahradiť aplikovaním jednej **logaritmicko-polárnej** (logPolar) transformácie.

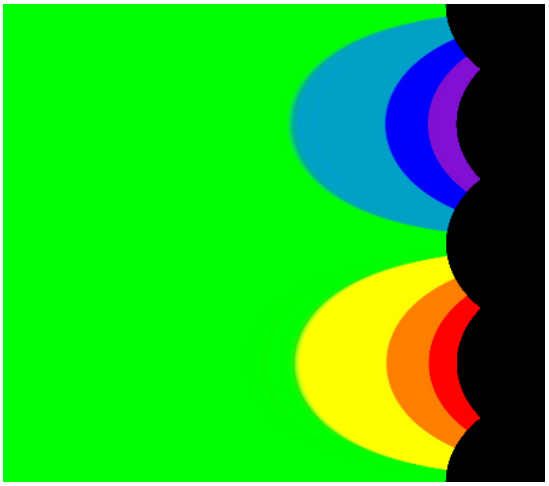
5.4.1 Logaritmicko-polárna transformácia

S využitím poznatkov o polárnej a logaritmickej transformácií je ľahké odvodiť ako vyzerá Logaritmicko-polárna transformácia. Riadky transformovaného obrazu rovnako ako pri polárnej transformácií zodpovedajú pixlom nachádzajúcim sa na priamke zvierajúcej určitý uhol s počiatočnou pozíciou. Rovnako ako pri logaritmickej transformácií, pixly v týchto riadkoch sú distribuované podľa logaritmickej postupnosti, obr. 5.6.



Obr. 5.6: Logaritmicko-polárna transformácia

V openCV nie je táto transformácia priamo implementovaná - `cv2.logPolar`. Pri použití `cv2.remap` je však zreteľne vidieť ako transformácia funguje, obr. 5.7.



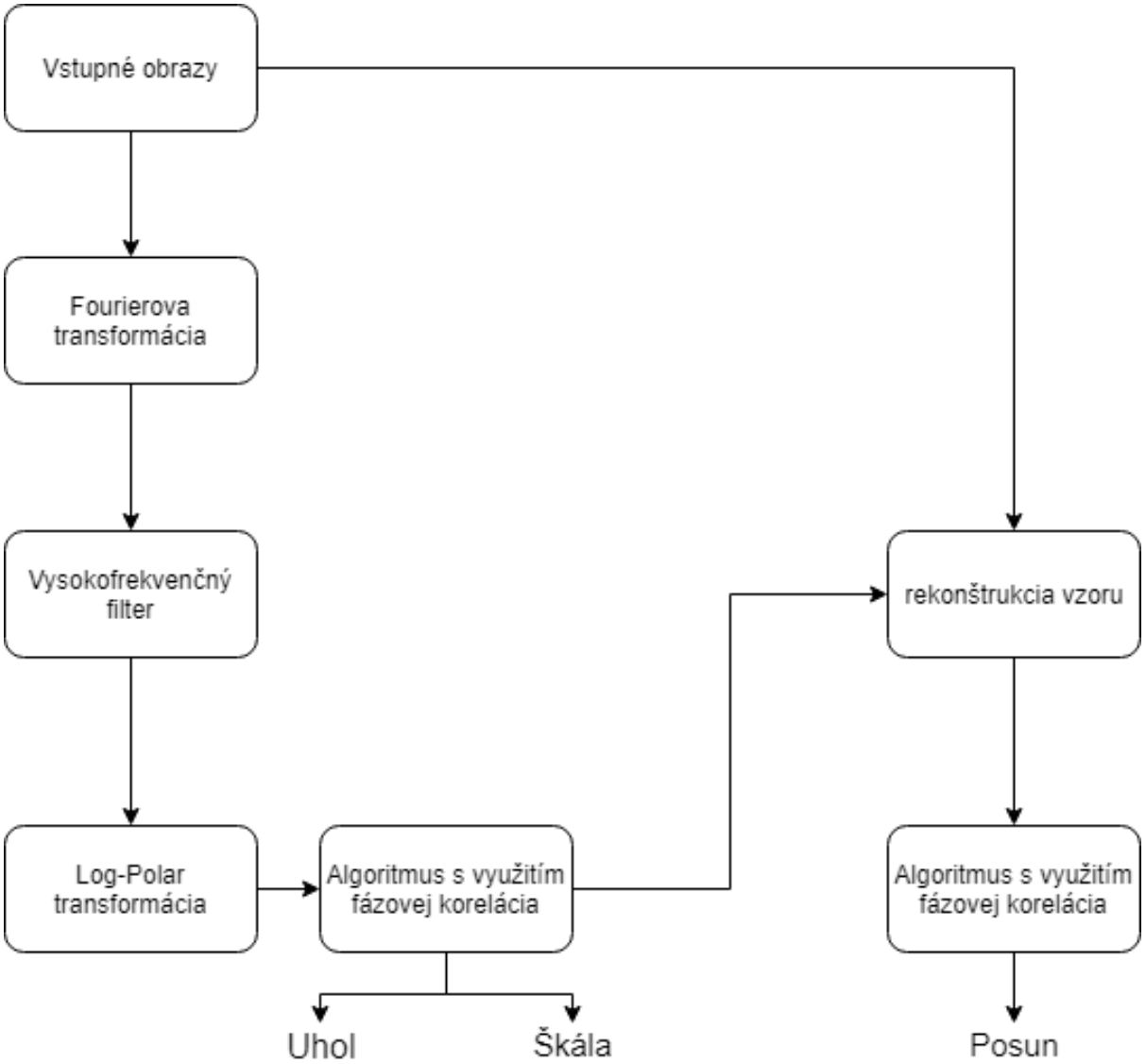
Obr. 5.7: Príklad logaritmicko-polárnej transformácie

```
import cv2
import numpy as np

img = cv2.imread('rainbow2.jpg')

map_x = np.zeros(img.shape[:-1])
map_y = np.zeros(img.shape[:-1])

d = (centre[0]**2 + centre[1]**2)**0.5
K = 10**(np.log10(d) / rows)

theta_d = 2*np.pi / rows
theta = 0

for i in range(rows):
    for j in range(cols):
        radius = (K**j)

        map_x[i,j] = np.sin(theta)*radius + centre[1]
        map_y[i,j] = np.cos(theta)*radius + centre[0]

    theta += theta_d

map_x = map_x.astype(np.float32)
map_y = map_y.astype(np.float32)

logPol = cv2.remap(img, map_x, map_y, 1)




cv2.imshow('logPolar', logPol)
cv2.waitKey(10000)
```

5.5 Algoritmus

V tejto časti je vysvetlený výsledný algoritmus, ktorý všetky vyššie spomenuté časti vyhľadávania spojí do jedného algoritmu. Týmto spôsobom sa dá vyhľadať na obraze vzor, ktorý môže byť otočený, posunutý, škálovaný so zmenou kontrastu. Veľkosť škály musí byť rovnaká pre oba smery z dôvodov, ktoré sú vysvetlené časti o výpočte rotácie a škály zároveň.

5.5.1 Diagram



Vysokofrekvenčný filter - detekcia hrán

Vysokofrekvenčný filter, alebo detektor hrán, je aplikovaný na obraz aby na obraze ostala iba dôležitá informácia. Keďže pri logaritmicko-polárnej transformácii dochádza k strate informácie, týmto filtrom zamedzíme strate informácie. Menej významná informácia nemá vplyv na výsledok algoritmu.

Rekonštrukcia vzoru

Rekonštrukcia vzoru sa dá aplikovať pomocou funkcií knižnice OpenCV
- Funkcia `cv2.getRotationMatrix2D(centre,angle,scale)` vráti maticu rotácie.
- Funkcia `cv2.warpAffine(img,rotationMatrix,shape)` vytvorí z rotačnej matice výsledný obraz.

Záver

Cieľom práce bolo na príkladoch vysvetliť čitateľovi metódu vyhľadávania vzoru na obraze s použitím Fourierovej transformácie a ukázať ako, kedy a prečo táto metód funguje.

V prvej kapitole bol čitateľ oboznámený s nástrojmi v programovacom jazyku Python, ktoré boli použité na vysvetlenie a vyjadrenie matematických súvislostí v ďalších častiach práce pomocou malých skriptov. V ďalšom bode bolo potrebné čitateľa dostatočne zrozumiteľne oboznámiť s Fourierovou transformáciou, vysvetliť na príkladoch o čo sa jedná a ako táto transformácia funguje, aby spoznal jej matematické pozadie. Tieto poznatky sú nevyhnutné pre porozumenie algoritmom využívajúcim algoritmus rýchlej Fourierovej transformácie. Cirkulárnej konvolúcií sme venovali samostatnú kapitolu, nakoľko je kľúčovým prvkom algoritmu fázovej korelácie. Čitateľ oboznámený s Fourierovou transformáciou a jej využitím vo fázovej korelácií sa v štvrtej kapitole dozvedá o jej využití pri vyhľadávaní vzoru na obraze. Pomocou týchto poznatkov je demonštrovaná funkcionalita algoritmu schopného rozpoznať vzor na šedo-tónovom obraze, ak je vzor posunutý a má zmenený kontrast. Postup aký je daný algoritmus vytvorený spolu s jeho zdrojovým kódom a vysvetlením jednotlivých častí pomôže čitateľovi lepšie pochopiť princíp jeho fungovania. Záverečná kapitola je venovaná algoritmu schopnému nájsť vzor na obraze, ak je vzor posunutý, otočený a škálovaný. Na porozumenie postupu akým algoritmus zistí uhol rotácie a veľkosť škály je potrebné ovládať tri nové geometrické transformácie obrazu a to polárnu transformáciu, logaritmickú transformácie a logaritmicko-polárnu transformáciu. Na príkladoch a obrázkoch je podrobne vysvetlená funkcionalita týchto transformácií. S nadobudnutými poznatkami je čitateľ schopný pomocou podrobnej schémy algoritmu implementovať svoju vlastnú verziu programu.

Prínosom práce je súhrnné zoskupenie poznatkov o tejto téme vyjadrených v dnešnej dobe populárnom skriptovacom jazyku Python a pomocou grafických reprezentácií inak zložitých matematických a geometrických javov. Študentom umožní pochopiť funkcionalitu algoritmov využívajúcich Fourierovu transformáciu pri práci s obrazom v im známom prostredí programovacieho jazyka. Osobným prínosom práce podrobné porozumenie už

zmienenej problematike.

Základný algoritmus, rozpoznávajúci posunutý objekt na obraze, je užitočný v priemysle, kde sú dobré predpoklady, že vyhľadávaný objekt nie je otočený. Kamera umiestnená na stálej pozícií zabezpečuje, že nedôjde k škálovaniu obrazu. Existuje celý rad praktických úloh, kde je potrebné zistiť pozíciu škálovaného a otočeného objektu na obraze. Na tento účel je vhodný záverečný algoritmu, ktorý je schopný zistiť rotáciu a škálovanie. Avšak jeho uplatnenie v praxi je diskutabilné kvôli povahe samotnej logaritmicko-polárnej transformácie aplikovanej na diskrétne dáta.

Výsledný algoritmus, schopný nájsť otočený, posunutý a škálovaný vzor na obraze naráža na hranicu dnešného poznania v oblasti vyhľadávania vzoru s využitím Fourierovej transformácie. Nie je možné na základe dnešných poznatkov zostrojiť algoritmus schopný nájsť vzor na obraze, ktorý je otočený, posunutý a škálovaný rôzne v dvoch smeroch. Táto oblasť spracovania obrazu ponúka veľký priestor pre ďalší výskum, vzhľadom na jej vysokú časovú efektívnosť, vďaka využitiu rýchlej Fourierovej transformácie.

Literatúra

[1] The phase correlation image aligment method - C.D.Kuglin and D.C.Hines, 1975

[2] An FFT-Based Technique for Translation, Rotation, and Scale-Invariant Image Registration - B. Srinivasa Reddy and B. N. Chatterji (IEEE TRANSACTIONS ON IMAGE PROCESSING, VOL. 5, NO. 8, AUGUST 1996)

[3] Learning OpenCV Gary - Bradski and Adrian Kaehler (O'Reilly Media, Inc. September 2008, ISBN: 978-0-596-51613-0)

[4] Documentation python numpy and scipy
<https://docs.scipy.org/doc/>

[5] Documentation python Matplotlib
<https://matplotlib.org/>

[6] Matematická analýza III. (skripta), FMFI UK - M.Barnovska a kol.

[7] Počítačové videnie a rozpoznávanie objektov - Elena Šikudová, Zuzana Černeková, Wanda Benešová, Zuzana Haladová, Júlia Kucerová

[8] Introduction to Algorithms, 2rd Edition (The MIT Press 2001, ISBN-978-0070131514) - Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein

[9] Learning Python, 5th Edition - Mark Lutz (O'Reilly Media, Inc. June 2013, ISBN: 9781449355739)

[10] The Fourier Transform And Its Applications - Ronald N. Bracewell (ISBN-0-07-303938-1)

[11] Robust FFT-Based Scale-Invariant Image Registration - Georgios Tzimiropoulos and Tania Stathaki, (4th SEAS DTC Technical Conference - Edinburgh 2009)

Príloha
Rýchla Fourierova transformácia

Pre efektívne používanie Fourierovej transformácie, je dôležité mať rýchly algoritmus na jej výpočet. Diskrétnu Fourierovu transformáciu popisuje vzorec pre vstup veľkosti N a $0 \leq k < N$ [**?**]

$$F(k) = \sum_{n=0}^{N-1} f(n) e^{-2i\pi \frac{n}{N} k} \tag{25}$$

Každá hodnota $F(k)$ je vypočitaná z N hodnôt vstupu. Priamočiarou implementáciou získame vzorec s časovou zložitosťou $O(N^2)$.

Cooley–Tukey algoritmus

Tento algoritmus rýchlej Fourierovej transformácie alebo fast Fourier transform, **FFT**, je najpoužívanejším algoritmom na výpočet Fourierovej transformácie [8]. Funguje na princípe *rozdeľuj a panuj*. Pôvodnú rovnicu upravíme tak, aby sme jej výpočet rozdelili na dva alebo viac menších častí.

$$\begin{aligned} F(k) &= \sum_{n=0}^{N-1} f(n) e^{-2i\pi \frac{n}{N} k} && (26) \\ &= \sum_{n=0}^{N-1} f(n) \omega_N^{nk} && (27) \end{aligned}$$

Sumu rozdelíme na dve. Prvá suma pre párne hosnoty n, $n = 2x$, a druhá suma pre nepárne n, teda pre $n = 2x + 1$.

$$\begin{aligned} &= \sum_{x=0}^{N/2-1} f(2x) e^{-2i\pi \frac{2x}{N} k} + \sum_{x=0}^{N/2-1} f(2x+1) e^{-2i\pi \frac{2x+1}{N} k} && (28) \\ &= \sum_{x=0}^{N/2-1} f(2x) e^{-2i\pi \frac{x}{N/2} k} + \sum_{x=0}^{N/2-1} f(2x+1) e^{-2i\pi \frac{1}{N} k} e^{-2i\pi \frac{x}{N/2} k} && (29) \end{aligned}$$

V indexe exponenta sme presunuly dvojku z čitateľa do menovateľa. Ďalej využijeme nasledujúcu vlastnosť.

$$e^{-2i\pi\frac{x}{N/2}k} = \omega_{N/2}^{x}k \tag{30}$$

Rovnica sa dá následne upraviť do tvaru

$$= \sum_{x=0}^{N/2-1} f(2x)\omega_{N/2}^{xk} + \sum_{x=0}^{N/2-1} f(2x+1)\omega_{N/2}^{k}\omega_{N/2}^{xk} \tag{31}$$

$$= \sum_{x=0}^{N/2-1} f(2x)\omega_{N/2}^{xk} + \omega_{N/2}^{k} \sum_{x=0}^{N/2-1} f(2x+1)\omega_{N/2}^{xk} \tag{32}$$

Po úprave nám vznikly dve sumy, pričom jasne vidieť, že ide o Fourierove transformácie pre párne a nepárne prvky vstupu. Tieto transformácie rovnakým spôsobom vypočítame z ich menších transformácií. Menšie transformácie pracujú s $0 \leq k < N/2$ avšak vo vzorci počítame aj pre hodnoty $N/2 \leq k < N$. Vďaka cyklickej vlastnosti ω_Y to nie je problém. Príklad pre $N/2 = 4$

$$\omega_4^{6x} = \omega_4^{2x} \tag{33}$$

Pre $N/2 \leq k < N$ sa použije už vypočítaná hodnota pre $0 \leq k < N/2$.

Sumarizácia algoritmu

Algoritmus funguje nasledovne:

1. Zavolá sa funkcia `FFT( parametre )` , kde parametre predstavujú postupnosť hodnôt vstupnej funkcie/signálu.
2. Parametre sa rozdelia podľa ich indexov na párne a nepárne. Ak je iba jeden vstupný parameter funkcia vráti jeho hodnotu.
3. Pre hodnoty na párnych indexoch sa zavolá funkcia `FFT(parne-parametre)`. Rovnako pre hodnoty na nepárnych indexoch sa zavolá `FFT(neparne-parametre)`. Tieto funkcie sú zavolané s polovičným počtom parametrov.
4. Vypočíta sa hodnota $\omega_{N/2}^{k}$.
5. Návratové hodnoty volaní `FFT(parne-parametre)` a `FFT(neparne-parametre)` sa rozšíria na veľkosť pôvodného počtu parametrov. Keďže návratové hodnoty majú polovičnú veľkosť oproti pôvodnej, duplikujeme ich.

6. Duplikované návratové hodnoty spojíme nasledovným vzorcom

$$parne + \omega_{N/2}^{k} neparne \tag{34}$$

7. Spojením vznikla nová postupnosť hodnôt, ktorá je výsledkom Fourierovej transformácie a funkcia ju vráti

Tento algoritmus však funguje len pre vstupy veľkosti $N = 2^a, a \in \mathbb{N}$. Miernymi úpravami sa dá docieliť rozdelenie v rôznych pomeroch ak $N = OP, O, P \in \mathbb{N}$. Pre ilustráciu demonštráciu myšlienky postačuje riešenie pre vstupy veľkosti $N = 2^a, a \in \mathbb{N}$.

Časová zložitosť

Čas trvania algoritmu pre vstup veľkosti N je

$$T(N) = 2T(N/2) + O(N) \tag{35}$$

Tento vzťah rozpíšeme ďalej podľa rekurzívnej formuly a vzťahu $T(1) = O(1)$

$$\begin{aligned}
T(N) &= 2T(N/2) + O(N) && (36)\\
&= 4T(N/4) + O(N) + 2O(N/2) && (37)\\
&= 8T(N/8) + O(N) + 2O(N/2) + 4O(N/4) && (38)\\
&= .... && (39)\\
&= 2^{\log N} T(1) + O(N) \log N && (40)\\
&= NO(1) + O(N) \log N && (41)\\
&= O(N) + O(N) \log N && (42)\\
&= O(N) \log N && (43)
\end{aligned}$$

Časová zložitosť algoritmu je $O(NlogN)$. Oproti priamočiarej implementácií je tento algoritmus $\frac{N}{logN}$-krát rýchlejší.

Implementácia v Pythone

Tu je uvedená ilustračná implementácia v Pythone. FFT je implementovaná vo väčšine matematických knižniciach ako napríklad Numpy, SciPy alebo knižniciach na spracovanie obrazu OpenCV.

```
def omega(base):
    power = np.linspace(0,base-1, base)
    return np.exp(-2*np.pi*1j*power/base)

def fft(x):

    N = len(x)

    if N == 1:
        return x

    even = fft(x[::2])
    even = np.append(even,even) # cirkulacia

    odd = fft(x[1::2])
    odd = np.append(odd,odd)

    return even + omega(N)* odd
```

Tieto implementácie sú efektívnejšie ako dole uvedená kvôli pomalej interpretácií kódu v Pythone oproti kompilovaným jazykom ako C a C++, v ktorých sú napísané ostatné implementácie.