# Zpracování digitální obrazové informace pro stegoanalýzu pomocí neuronových sítí

## Processing digital image information for the steganalysis using neural networks

Bc. Jiří Sedlák

Diplomová práce
2010



Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

*** nascannované zadání str. 1 ***

*** nascannované zadání str. 2 ***

## ABSTRAKT

Cílem této diplomové práce bylo vytvořit multiplatformní JPEG dekodér, který bude jako výstup vracet histogram Huffmanova kódování, sloužící pro účely stegoanalýzy pomocí neuronových sítí. V práci je popsána obecně problematika steganografie, dále se zabývá dekódováním obrazového formátu JPEG a strukturou jeho uložení do JFIF. Praktická část obsahuje příklad dekódování souboru a popis vytvořeného dekodéru.



Klíčová slova: JPEG, JFIF, Python, Huffmanovo kódování



## ABSTRACT

The aim of master thesis was to create cross-platform JPEG decoder, which will return histogram of Huffman code. This histogram will be used for purpose of steganalysis by neural network. In theoretical part, stegoanography is generally described and also the thesis deals with graphic JPEG decoding and its export to JFIF. Practical part contains file decode example and description of created decoder.



Keywords: JPEG, JFIF, Python, Huffman encoding

Poděkování, motto

Chtěl bych velmi poděkovat vedoucí mé diplomové práce Ing. Zuzaně Oplatkové PhD. a Ing. Jiřímu Hološkovi za jejich nesmírnou trpělivost a užitečné rady.

**Prohlašuji, že**

- beru na vědomí, že odevzdáním diplomové/bakalářské práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb. o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších právních předpisů, bez ohledu na výsledek obhajoby;
- beru na vědomí, že diplomová/bakalářská práce bude uložena v elektronické podobě v univerzitním informačním systému dostupná k prezenčnímu nahlédnutí, že jeden výtisk diplomové/bakalářské práce bude uložen v příruční knihovně Fakulty aplikované informatiky Univerzity Tomáše Bati ve Zlíně a jeden výtisk bude uložen u vedoucího práce;
- byl/a jsem seznámen/a s tím, že na moji diplomovou/bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- beru na vědomí, že podle § 60 odst. 1 autorského zákona má UTB ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- beru na vědomí, že podle § 60 odst. 2 a 3 autorského zákona mohu užít své dílo – diplomovou/bakalářskou práci nebo poskytnout licenci k jejímu využití jen s předchozím písemným souhlasem Univerzity Tomáše Bati ve Zlíně, která je oprávněna v takovém případě ode mne požadovat přiměřený příspěvek na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše);
- beru na vědomí, že pokud bylo k vypracování diplomové/bakalářské práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými
  subjekty pouze ke studijním a výzkumným účelům (tedy pouze k nekomerčnímu
  využití), nelze výsledky diplomové/bakalářské práce využít ke komerčním účelům;
- beru na vědomí, že pokud je výstupem diplomové/bakalářské práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá. Neodevzdání této součásti může být důvodem k neobhájení práce.

**Prohlašuji,**

- že jsem na diplomové práci pracoval samostatně a použitou literaturu jsem citoval. V případě publikace výsledků budu uveden jako spoluautor.
- že odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou totožné.

Ve Zlíně

…….
podpis diplomanta

# OBSAH

# ÚVOD

Na Univerzitě Tomáše Bati ve Zlíně Fakultě Aplikované informatiky vyvinuli novou steganoanalytickou metodu, která jako vstup pro klasifikaci JPEG obrázků využívá histogram Huffmanova kódování. V souvislosti s prací na této metodě vznikl problém získání zmíněného histogramu, jelikož většina programů nebo funkčních knihoven nemá funkci pro získání histogramu implementovánu. Proto vznikl požadavek od Ing. Jiřího Hološky, na zpracování multiplatformního JPEG dekodéru, který je schopný tento histogram vyextrahovat.

V teoretické části budu nejprve popisovat steganografie, jak z hlediska historického, tak i moderní digitální steganografie. V rámci digitální steganografie budu věnována pozornost hlavně obrázkům jako nosiči skrytých dat a metodě nejméně významného bitu pro jejich vkládání. Podrobně se budu věnovat standardu JPEG, nejprve obecnému popisu, dále popisu principu ztrátové komprese v sekvenčním módu. V poslední kapitola teoretické části se zaměřím na uložení zakódovaných dat do souboru. Podrobně popíši význam a obsah jednotlivých značek, které se mohou vyskytovat v sekvenčním módu.

V praktické části popíši detailně dekódování ukázkového obrázku. V druhé části už se budu věnovat modul dekodéru, který je napsán v programovacím jazyku Python verze 2.6, s použitím optimalizačního modulu Psyco. Celý dekodér se skládá z jedné třídy jpegDecoder, která má dvě veřejné funkce, konstruktor __init__ a funkci decode, zahajující dekódovací proces a jako návratovou hodnotu vrací histogram Huffmanova kódování.

# I. TEORETICKÁ ČÁST

# 1 STEGANOGRAFIE

Slovo steganografie pochází z řeckého steganós - schovaný a gráphein – psát. Steganografie je věda zabývající se utajením komunikace, ukrýváním zpráv a zatajováním probíhající komunikace.[1]

Někdo by se mohl z výše zmíněné definice domnívat, že steganografie je to samé co kryptografie. Ovšem to je omyl! Steganografie by se spíše dala nazvat jako příbuznou kryptografie. Kryptografie se zabývá znemožněním nějakou zprávu číst. Tudíž zašifrovaná zpráva je na první pohled rozeznatelná od ostatní komunikace. Takže v případě zachycení zprávy stojí mezi odhalením obsahu zprávy a útočníkem pouze síla šifry. Kdežto steganografie se zabývá ukrytím zprávy. U takto ošetřené zprávy není a nemá být na první pohled patrné, že obsahuje nějakou důvěrnou zprávu. Ovšem pokud útočník odhalí, jak je zpráva schována, tak mezi odhalením zprávy a útočníkem nestojí nic. Proto se steganografie spíše používá pro přenos zašifrovaných zpráv.

## 1.1 Historie

Zápisky o použití steganografie jsou již ze starého Řecka. V 5 století před naším letopočtem Demaratus poslal do Řecka varování o přípravě Peršanů na invazi pomocí voskové psací destičky. Nechal z dřevěné destičky rozpustit všechen vosk, poté na destičku vyškrábal své varování a nechal destičku opět zalít voskem. Tak propašoval zprávu až do Řecka. Kde opět rozpustili vosk a varování si přečetli.[1][2][4]

Dalším příkladem z antiky je doručení poselství miltskému vůdci Aristagorovi, aby povstal proti Peršanům a pomohl tak Řekům. Byl použit otrok, kterému byla oholena hlava a na ni vytetováno poselství, pak stačilo počkat než mu hlava opět zaroste vlasy. Otrok poselství bez povšimnutí Peršanů doručil.[1][2][4]

Velice zajímavý způsob doručování zpráv vymysleli Číňané. Napsali zprávu na hedvábí, které zmuchlali do kuličky a zalili voskem, posel potom kuličku spolkl a doručil, ovšem nikde se nepíše jak kuličku se zprávou dostali z posla ven.

V průběhů let se ovšem využívaly mnohem prostší způsoby doručování zpráv, jako byly duté berle, protézy, kupy hnoje nebo fekální vozy. Prakticky jako každý kontraband. Popřípadě i jinými mnohem kurióznější způsoby. Kapitolou samo o sobě jsou neviditelné inkousty ty jednodušší třeba z mléčné šťávy pampelišek, citrónové šťávy, které se zviditelnily po zahřáti, nebo inkousty z mnohem složitějších syntetických sloučenin, které jde vidět pod speciálním světlem nebo po kontaktu s jinou chemickou látkou.[1]

Je také nutné uvést jméno mnicha Johannese Trithemiem, zabýval se steganografií a kryptografií, jeho nejznámější knihou je Steganographia. Je to kniha o třech svazcích o které se až do nedávna myslelo, že je o černé magii, ale byly to stegotexty s velkým významem.[3]

Steganografie se také hojně využívala během druhé světové války. Příkladem může být rádiové vysíláni spojenců z Británie, kdy špióni či odbojové organizace poslouchali v určenou hodinu například, přání rodinám nebo počasí atd. Jednotlivé slova, fráze, popřípadě věty znamenaly pokyny k zahájení akce nebo kdy a kde bude špión vyzvednut. Jako další metoda se používaly mikrotečky. Šlo o miniaturizaci textu či grafiky do velikosti tečky vytvořené psacím strojem. Tyto mikrotečky se poté lepily v dopisech na interpunkční znaménka a následně překrývaly matným materiálem.[1][4]

## 1.2 Digitální steganografie

S rozmachem výpočetní techniky v osmdesátých letech a masovým rozšířením e-mailové komunikace přišel další milník, už žádní poslové s tetováním na hlavě, ale zprávy přenášené v řádu sekund. To sebou ovšem nese i nové požadavky na ukrývání citlivých zpráv. A proto také vznikla digitální steganografie. Dnes se tajné zprávy ukrývají do digitálních obrázků, zvukových záznamů, skrytých partition, spustitelných aplikací nebo přímo do přenosových protokolů např. TCP/IP.

## 1.3 Rozdělení steganografie

Podle používaných metod v digitální steganografii je možné rozdělit steganografické techniky do tří skupin.[4]

### 1.3.1 Injekční steganografie

Injekční steganografie, metoda vytváření steganogramu vyplývá již z názvu, steganogram se tvoří vložením skryté informace do již existujícího média, kterým může být text, obrázek nebo audio soubor. Vkládáním se zvětší objem nosného média, celý proces vkládání musí být proveden tak aby program, který by měl číst původní data (textový editor, prohlížeč obrázků, hudební přehrávač) neodhalil přítomnost přidaných dat. Většinu multimediálních souborů je možné použít jako nosné médium, potom se takovému souboru říká steganogram.

### 1.3.2 Substituční steganografie

Substituční steganografie jedná se o metodu která využívá nahrazování nejméně významných bitů, kdy se nahrazují části nosného média tak aby ovšem byl čitelný "původním" programem pro který tyto data byly určené. Substituce je možná i u spustitelných souborů a to v modulech nebo částech kódu které jsou využívány jen velmi zřídka nebo vůbec ne. Znakem takto upravených souborů je vizuální degradace nosného souboru týká se jak ve video tak i statických obrazových souborů, nebo výskyt šumu u audio souborů.

### 1.3.3 Propagační steganografie

Propagační steganografie je založena na mechanizmu kdy při předání vstupní zprávy tento mechanizmus (generátor) produkuje výstup jenž se stává nosnou složkou. Tento výstup, je často označován slovem "mimic", může být realizován neobvyklým obrázkem, audio souborem,textem bez hlubšího významu, fraktálem nebo jinak. Při opakovaném kódování stejné zprávy bude výstup vykazovat jen drobné odchylky výsledného stagnogramu, při kódování se nevyužívá žádného dalšího nosného média jako u předchozích metod.

## 1.4 Steganografie v obrázcích

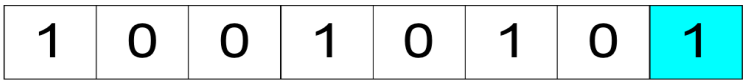
Obrázky jsou jako krycí soubor naprosto ideální. A to ze dvou důvodů. Na internetu se s nimi potkáváme všude, můžou byt i přiloženy v e-mailech popřípadě posílány přes IM. Díky různým ztrátovým kompresi jsou obrázky přítomny v různých kvalitách, což znamená, že pokud je obraz lehce zdegenerován kvůli vložené zprávě nebudeme tomu přikládat valný význam. A navíc přesně vystihují filozofii steganografie, být na očích, ale nebýt vidět.[5]

Níže se budu zabývat tou nejběžnější a zároveň nejjednodušší metodou vkládání zprávy do krycího objektu a tou je metoda nejméně významných bitů.

### 1.4.1 Metoda nejméně významných bitů.

V rastrové grafice se obraz reprezentuje pomocí matice obrazových bodů nebo-li pixelů. U barevných obrázku z 24 bitovou hloubkou je jeden pixel reprezentován 3 kanály základními barvami a to R – červená, G – zelená a B – modrá každá tato barva je vyjádřena jedním bajtem. Pro ukládání zpráv se používají jednotlivé kanály.

To bylo něco o rastrové grafice teď co je to vlastně ten nejméně významný bit. Jak jistě každý ví, tak informace se reprezentují po bajtech a jeden bajt se skládá z 8 bitů. Nejméně významný bit je bit úplně vpravo na Obr. 1., matematicky lze vyjádřit jako $2^0$.




*Obr. 1: Jeden bajt rozdělený na bity s označeným LSB[5]*

Nejméně významný bit můžeme také označovat pomocí anglické zkratky LSB -Least Significant Bit. Tento bit se používá proto, že jeho změna výslednou hodnotu bajtu změní minimálně. Dekadicky vyjádřená hodnota bajtu na Obr.1. je 149, pokud změníme LSB na 0 tak se výsledná hodnota změní na 148 což není tak velký rozdíl. Pokud by jsme, ale změnili bit úplně vlevo na 0, tak dostaneme hodnotu 21 to je rozdíl 128 oproti původní hodnotě. Proto se používají nejméně významné bity.

Malá ukázka jak uložíme písmeno A do obrazových dat. Písmeno A si převedeme do ASCII kódu ten je 65 a jeho binární reprezentace je 01000001. Teď potřebujeme nosič informací a tím budou 2 pixely bílé barvy a ze třetího jen červený a zelený kanál. Bílá barvy je v RGB reprezentována samými jedničkami. Takže náš nosič bude vypadat takto: [6]

pixel 1 11111111 11111111 11111111

pixel 2 11111111 11111111 11111111

pixel 3 11111111 11111111

Teď budeme měnit LSB v jednotlivých bajtech podle binárního reprezentace písmene A.

pixel 1 1111111**0** 1111111**1** 1111111**0**

pixel 2 1111111**0** 1111111**0** 1111111**0**

pixel 3 1111111**0** 1111111**1**

Jak vidíme na schématu výše písmeno A je momentálně schováno v LSB bitech. Toto je ovšem ten nejjednodušší příklad takzvaného naivního algoritmu. Reálný steganografický program nahrazuje bity s určitou náhodností, proto jsou stego-obrázky poměrně těžce detekovatelné.[6]

Příklad uložených zpráv v obrazcích. Byl použit krycí obrázek ve formátu 24 bitový BMP v rozměrech 585x275 Obr.2. Stego-obrázek , kde je zakódováno zpráva o 2000 náhodných znacích a pro uchování informace je použit nejméně významný bit můžeme vidět na

Obr.3. Na Obr.4. stego-obrázek kde je uloženo 250000 znaků a jsou použity 6 nejméně významných bitu.

*Obr. 2: Krycí obrázek*

*Obr. 3: Stego-obrázek se zprávou o 2000 znaků*

*Obr. 4: Stego-obrázek se zprávou o 250000 znacích*

V takto rozmanitém krycím obrázku není krátká zpráva vůbec patrná. Na obrázku Obr.4 kde se používají i jiné bity než LSB a zvedl se i objem dat, můžeme vidět patrné deformace způsobené vloženou zprávou. Stego-obrázky byly vytvořeny pomocí softwaru OpenStego, který umožňuje nastavení metody vkládání a bity, které se mají k ukládání zprávy použít.

## 1.5 Steganografický software

Na internetu můžeme najít velké množství volně dostupného steganografického softwaru, který umí pracovat s obrázky, ale i s jinými krycími soubory. Vyberu z této celé škály pouze dva Steghide a OpenStego. Steghide je jeden ze známějších steganografických nástrojů pro příkazovou řádku, který dokáže pracovat ze soubory formátu JPG, BMP, WAV a AU. OpenStego je pro změnu s grafickým rozhraním napsáný v programovacím jazyku Java, jako výstup umí pouze obrázky ve formátu PNG.

# 2 JPEG

V dnešní době jedena z nejpoužívanějších kompresních metod vůbec. Metoda pracuje na základě selektivního zanedbávání určité informace, díky čemuž dosahuje vynikajících kompresních poměrů. Jeho největší síla spočívá v kompresi fotografií v této doméně dosahuje vynikajících výsledků, s minimální ztrátou kvality. Pokud chceme komprimovat obrázky, které mají spoustu ostrých přechodů, či obsahují velké množství textu je lepší využít jiných metod.

Název JPEG je zkratkou pro organizaci „Joint Photographic Experts Group.“, tato organizace vytvořila standard pro kompresi obrazů, pod záštitou ISO.

V původním standardu je určen i formát, ve kterém se zkomprimovaná data mají přenášet. Má zkratku JIF „JPEG Interchange Format“. Určuje jak mají být části uloženy, ale nijak nespecifikuje barevný model a má poměrně volně definované některé části, takže může docházet k nekompatibilitě mezí jednotlivými aplikacemi či zařízeními. Proto byl uveden obrazový formát JFIF „JPEG File Interchange Format“, který specifikuje používaný barevny model a určuje přesný formát uložených části.

## 2.1 Kompresní Módy

Originální JPEG standard definuje 4 kompresní módy hierarchický, progresivní,sekvenční a bezztrátový. V dodatcích, standard definuje další kódovací postupy pro jednotlivé módy. Vztahy mezi jednotlivými módy a kódovacími postupy jsou zobrazeny v tabulce Tab.1. Běžně se však setkáváme pouze s několika módy a jejich režimy. Především se sekvenčním, méně už z progresivním módem, a jejich režimem používajícím Huffmanovo kódování a 8 bitové vzorky dat.[7]

*Tab. 1: Módy JPEG*

<table>
<tr><td rowspan="8">JPEG</td><td rowspan="4">Sekvenční</td><td rowspan="2">Huffmanovo</td><td>8-bit</td></tr>
<tr><td>12-bit</td></tr>
<tr><td rowspan="2">Aritmetické</td><td>8-bit</td></tr>
<tr><td>12-bit</td></tr>
<tr><td rowspan="4">Progresivní</td><td rowspan="2">Huffmanovo</td><td>8-bit</td></tr>
<tr><td>12-bit</td></tr>
<tr><td rowspan="2">Aritmetické</td><td>8-bit</td></tr>
<tr><td>12-bit</td></tr>
<tr><td rowspan="2">Bezztrátový</td><td colspan="2">Originál bezztrátový</td></tr>
<tr><td colspan="2">JPEG-LS</td></tr>
<tr><td colspan="3">Hierarchický</td></tr>
</table>

### 2.1.1 Sekvenční

Sekvenční mód je nejednodušší JPEG mód. Jak už jméno naznačuje v sekvenčním módu je obrázek kódován z vrchu dolů. Sekvenční mód podporuje vzorky data s 8 a 12 bitovou přesností.

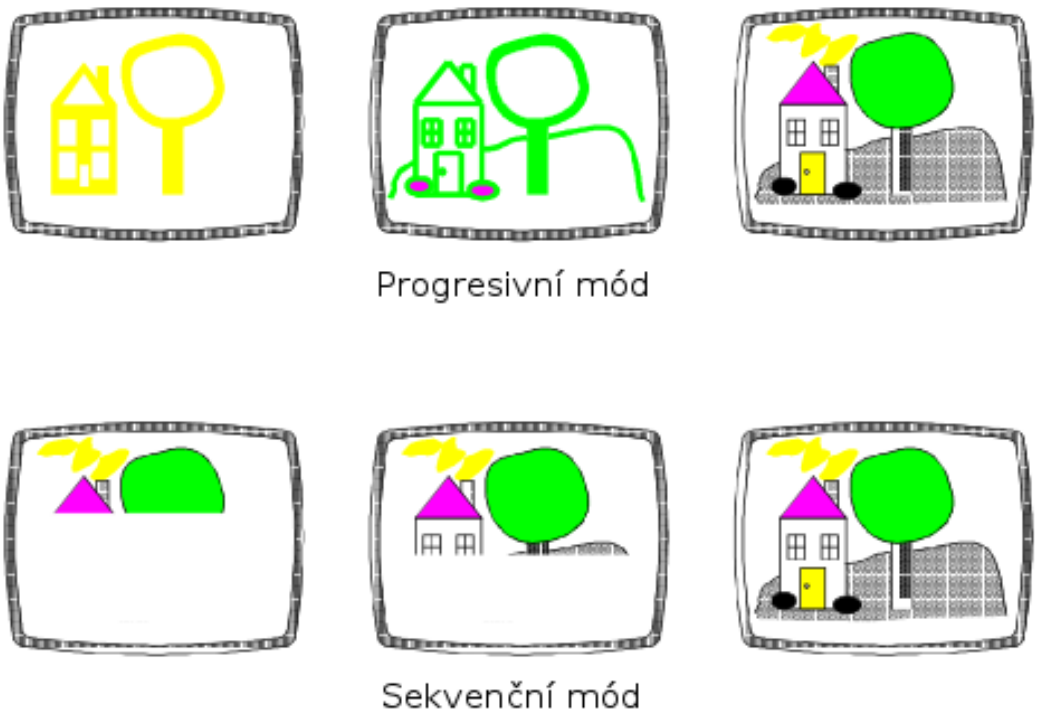
Obrázek se kóduje jedním průchodem celých dat a je uložen v jediném bloku komprimovaných dat, který se nazývá scan a obsahuje všechny barevné komponenty. Ve většině formátů je celý komprimovaný pixel uložený v jedné oblasti dat souboru. V JPEG standardu, je každý průchod obrázkem uložený v odděleném bloku dat scanu

V rámci sekvenčního módu jsou definovány dvě alternativy pro entropické bezztrátové komprimační metody a to Huffmanovo kódování a aritmetické kódování.

### 2.1.2 Progresivní

V progresivním módu, jsou komponenty zakódovány ve více blocích. Komprimovaná data jsou uložena ve 2 až 896 blocích, většinou je jejich počet na dolní hranici tohoto rozmezí. První blok vytvoří hrubou verzi obrázku a každý následující blok ji vylepší viz Obr.1. Proto po prvním dekódovaném bloku, můžeme vidět jak bude obrázek vypadat. Bylo to užitečné zejména při přenosu obrázku po pomalé síti, protože po přenesení malého množství dat, jsme mohli vidět, jak bude zhruba obrázek vypadat.[7]

Hlavní nevýhodou sekvenčního módu je ta, že je na dekódování mnohem složitější než sekvenční. Progresivní mód byl určen hlavně pro přenos po síti, kdy rychlost sítě je menší než relativní výkon procesoru. Obecně platí, že velikost obrázku u progresivního módu je téměř stejná jak u sekvenčního.

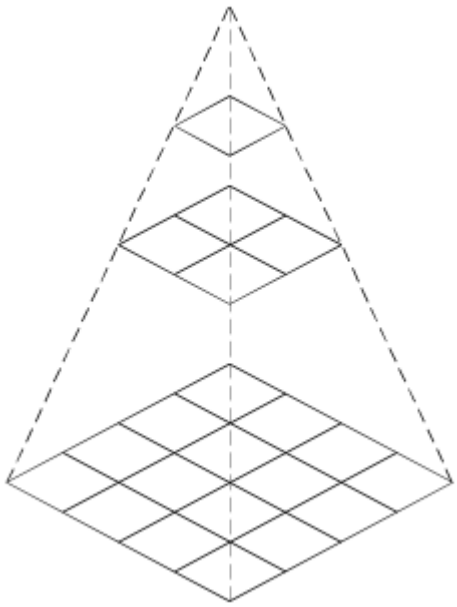



*Obr. 5: Rozdíl mezi sekvenčním a progresivním módem[8]*

### 2.1.3 Hierarchický

Je znám také jako super progresivní mód. Kde obrázek je „rozbit“ na několik pod-obrázku zvaných frames. Každý frame je kolekcí jednoho či více scanů. V hierarchickém módu první frame vytvoří verzi obrázku v nízkém rozlišení. Další framy postupně vylepšují obrázek, pomocí zvětšování rozlišení.

Nevýhodou hierarchického módu je jeho komplexnost. Je složitější na zpracování a množství framů zvyšuje objem dat, který je třeba přenést. Jeho výhodou je pouze to, že pokud potřebujeme obrázek v nižším rozlišení tak, nám stačí přenést pouze část dat a poté přenos zastavit. S tímto formátem se běžně nesetkáme.



*Obr. 6: Hierarchický mód[8]*

### 2.1.4 Bezztrátový

Původní JPEG standard definoval bezztrátový kompresní režim, který jak už název napovídá, zachovával nezměněný původní obrázek. Bezztrátový mód nikdy neměl tak dobou kompresi jako ztrátový. Navíc v době vzniku standartu už byly jiné metody, které uměly bezztrátovou kompresi lépe. Byla vydána i nová bezztrátová metoda označovaná jako JPEG-LS, proto je původní verze bezztrátové komprese zastaralá, avšak ani nová verze se příliš neujala a běžně se s ní prakticky nesetkáme.

## 2.2 Komprese v sekvenčním módu

Nyní si popíšeme základní operace při kompresi obrázku v sekvenčním módu. U komprese se postupuje podle těchto kroků:

1. Transformace barev do barevného modelu YCbCr
2. Podvzorkování barvonosného složky
3. Rozdělení na bloky 8x8 a dopředná diskrétní kosinová transformace
4. Kvantování matic
5. Bezztrátová komprese pomocí Huffmanova nebo aritmetického kódování
6. Uložení dat do formátu JFIF nebo JIF

### 2.2.1 Transformace barev

Existuje mnoho způsobů, jak reprezentovat barvy numericky. Systém pro reprezentaci barev se nazývá barevný model. Barevné modely jsou obvykle navrženy tak, aby využívaly přednosti jejich reprezentace na příslušném zařízení či metody, která s nimi pracuje.

Nejběžnějším modelem se kterým se můžeme setkat je model RGB, kde každá komponenta barevného modelu vyjadřuje intenzitu jedné ze základních barev červené, zelené a modré. Informace které jednotlivé složky vyjadřují je pro lidské oko více méně „rovnoměrná“. Což znamená, že RGB barevný model je nevhodný pro podvzorkování, které provádí JPEG, proto se převádí na model YCbCr.

Model YCbCr se skládá z Y – jasu a Cb,Cr jsou modrý a červený chrominační komponent. Pro lidské oko je nejvýznamnější složka jasová Y, protože na tu je oko nejvíce citlivé, ostatní dvě složky nejsou tak důležité. Tento model je proto vhodný pro podvzorkování.

Barvy se ve výpočetní technice nereprezentují čísly od 0 do 1, ale v bitech. Počet bitů vyjadřuje délku slova, což nám dává konečný počet hodnot, které je systém schopný reprezentovat. Běžná délka slova je 8 bitů tedy 256 možných hodnot.

Následující dva vzorce udávají přepočtový vztah mezi modelem RGB a YCbCr v 8-bitové reprezentaci.

RGB → YCbCr:

$$Y = 0.299\ R + 0.587\ G + 0.114\ B$$

$$Cb = -\ 0.1687\ R - 0.3313\ G\ + 0.5\ B + 128$$

$$Cr = 0.5\ R - 0.4187\ G - 0.0813\ B + 128$$

YCbCr → RGB:

$$R = Y + 1.402\ (Cr-128)$$

$$G = Y - 0.34414\ (Cb-128) - 0.71414\ (Cr-128)$$

$$B = Y + 1.772\ (Cb-128)$$

### 2.2.2 Podvzorkování signálu

Podvzorkování signálu je jedna část ztrátové komprese. Jak už bylo zmíněno výše lidské oko je citlivé na jasovou složku signálu, čemuž odpovídá barevný model YCbCr. Proto si můžeme dovolit ze vstupních dat redukovat (podvzorkovat) některé složky Cb a Cr, čímž dosáhneme jisté úspory dat.

Před podvzorkováním je zvolen tzv. vzorkovací faktor, který nám udává jakým poměrem bude sníženo rozlišení komponent v jednotlivých směrech. Vzorkovací faktor jsou tři dvojice čísel, každá dvojice je určena jedné složce barevného modelu, jedna udává horizontální směr a druhá vertikální.. Pokud použijeme vzorkovací faktor 2x2,1x1,1x1, Tak to znamená, že v horizontálním směru jsou dva vzorky jasové složky na jeden vzorek barvonosných složek a v horizontálním směru totéž. Když vynásobíme obě čísla dostaneme kolik bytu budeme potřebovat k popisu 2x2 pixelu. Normálně potřebujeme k takovémuto popisu 12 bajtů 4 bajty na složku,ale po výše uvedeném podvzorkování to bude pouze 6 bajtů 4 pro komponentu Y, 1 pro Cb a 1 pro Cr. To nám zredukuje data o 50%.

JPEG standard neurčuje že se musí podvzorkovávat pouze barvonosné složky, ale je logické, že budeme spíše redukovat barvonosné složky než jas u kterého by mohlo být podvzorkování patrné . JPEG standart též udává tři možné frekvence a to 1 až 4, problém by byl z hodnotou 3, kdy by mohlo docházet k nestejnoměrnému podvzorkování, proto se toto nastavení v programech téměř nevyskytuje. Běžně se setkáváme s vzorkovacími faktory 2x2 1x1 1x1, 2x1(1x2) 1x1 1x1 a posledním je 1x1 1x1 1x1 kdy vlastně nedochází k žádnému podvzorkování.

### 2.2.3 Rozdělení do bloků a diskrétní kosinová transformace

Před samotnou DCT se musí podvzorkovaný obrázek rozdělit do MCU (Minimum Coded Unit). Je to blok který popisuje určitou část obrázku. V jednom MCU jsou zastoupeny všechny složky v blocích dat o rozměru 8x8 bodů. Pokud bude vzorkovací faktor 1x1 1x1 1x1 tak v MCU bude každá složka zastoupena právě jednou což znamená 3 bloky. Při vzorkovacím faktoru 2x2 1x1 1x1 zde bude 6 bloků. Čtyři pro složku Y a po jednom pro složky Cb a Cr.

Pokud obrázek nemá rozměry v násobku MCU, bude jeho velikost dorovnána na nejbližší možný násobek v obou směrech. To se provádí opakováním posledních hodnot. Při

dekompresi se tyto body musejí oříznout. Bloky se zpracovávají zleva doprava a z shora dolů. První se bere složka Y potom Cb a nakonec Cr.

Ještě před zpracováním 8x8 bloků se upraví tak, že se od každé položky odečte konstanta 2n-1 kde n je počet bitů na jednu hodnotu. Pro 8 bitů to tedy bude číslo 128 a tím se hodnoty vzorků transformují do intervalu <-128,127>. Poté se provede dvoudimenzionální diskrétní kosinová transformace(DCT) podle vzorce na obrázku níže

$$X_{k_1,k_2} = \sum_{n1=0}^{N_1-1} \sum_{n2=0}^{N_2-1} x_{n_1,n_2} \cos\left[\frac{\pi}{N_1}\left(n_1+\frac{1}{2}\right)k_1\right] \cos\left[\frac{\pi}{N_2}\left(n_2+\frac{1}{2}\right)k_2\right.$$

*Obr. 7: Vzorec diskrétní Kosinovy transformace DCT*

Tím převedeme jednotlivé prvky z časové oblasti do oblasti frekvenční. Nízké frekvence budou v levém horním rohu a postupně rostou směrem k pravému dolnímu rohu. Důležité pro nás je, že vyšší frekvence nám o obrázku neposkytují tak důležité data jako nízké frekvence.

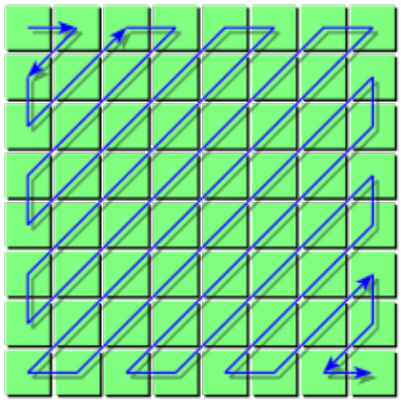
### 2.2.4 Kvantování koeficientů

Po DCT se provede kvantování koeficientů. Kvantování je jenom jiný název pro celočíselné dělení. Kdy se jednotlivé hodnoty vydělí hodnotami v kvantizační tabulce. Kvantizační tabulka je opět blok dat 8x8. Hodnoty jsou stanoveny tak, aby se co nejvíce zachovaly nízké frekvence a vypadly nám vysoké frekvence. Tím získáme v nízkých frekvencích nuly.

Příklad kvantované tabulky:

```
-40 10  6 -4  0  0  0  0
  0  2  2 -3  0  0  0  0
  2 -6  2  0  0  0  0  0
  0  3  0  0  0  0  0  0
  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0
  0  0  0  0  0  0  0  0
```

Z takto upravenými hodnotami se provede linearizace bloku pomocí metody Cik-Cak. To znamená že se matice 8x8 převede na lineární posloupnost 64 koeficientů. První hodnota bude v levém horním poli[0,0], druhá je na pozici [0,1], třetí [1,0], ….., poslední bude v pravém dolním rohu na pozici [8,8] viz obrázek.



*Obr. 8: Převod na Cik-Cak[10]*

Pokud si převedeme blok dat po kvantizaci, který je uveden výše tak dostaneme posloupnost koeficientů: -40,10,0,2,2,6,-4,2,-6,0,0,3,2,-3 a 51 nul. Což je velice výhodné pro další zpracování.

### 2.2.5 Kódování koeficientů

Koeficienty po kosinově transformaci se dají rozdělit na DC(zkratka z anglického názvu pro stejnosměrný proud) stejnosměrné a AC(pro střídavý proud) střídavé. Stejnosměrný DC koeficient je jediný a to v matici 8x8 ten vlevo nahoře, všechny ostatní jsou AC. Pro oba druhy koeficientů se používá jiné kódování.

#### *2.2.5.1 DC koeficienty*

Ještě před zakódováním koeficientu se provede odečtení předchozí hodnoty koeficientu. Pokud je to první hodnota tak se samozřejmě nic neodečte. Díky tomu se minimalizuje velikost hodnoty kterou musíme zapsat, protože hodnoty se od sebe příliš v sousedních blocích neliší.

Hodnoty se neukládají přímo, ale v zakódované hodnotě. Kdy počet bitů určuje kódovaný rozsah a samotná ukládaná hodnota je posun ve stanoveném rozsahu hodnot viz Tabulka Tab2. Takže například hodnotu -40 by jsme uložili jako 6,23.

*Tab. 2: Tabulka pro kódování DC koeficientů*

| Počet bitů | Rozsah |
|---|---|
| 0 | {0} |
| 1 | { -1}, {1} |
| 2 | {-3, -2}, { 2, 3} |
| 3 | {-7 . . -4}, { 4 . . 7 } |
| 4 | {-15 . . -8}, { 8 . . 15 } |
| 5 | {-31 . . -16}, { 16 . . 31} |
| 6 | {-63 . . -32}, { 32 . . 63 } |
| 7 | {127 . . -64},{ 64 . . 127} |
| 8 | {-255 . . -128}, { 128 . . 255 } |
| 9 | {-511 . . -256}, { 256 . . 511 } |
| 10 | {-1023 . . -512}, { 512 . . 1023} |
| 11 | {-2047 . . -1024}, { 1024 . . 2047} |

#### *2.2.5.2 AC koeficienty*

U AC koeficientů se první opět určí počet bitů určující rozsah, ale to je pouze polovina toho co potřebujeme. Rozsah určují dolní čtyři bity a horní čtyři bity určují kolik nulových koeficientů je před tímto koeficientem. Pokud bude hodnota v hexadecimální soustavě 0x45, tak nám to říká, že před tímto koeficientem jsou 4 nulové a příslušný koeficient je v rozsahu pěti bitovém, za kódem bude 5bitová hodnota určující posun v rozsahu.

Jsou dvě speciální hodnoty pro kódování AC koeficientů a to hodnota 0x00 která říká, že zbytek koeficientů je nulových (také označovaná jako konec bloku) a hodnota 0xF0, která říká, že je následujících 16 koeficientů nulových.

Níže je tabulka, pro kódování AC koeficientů, můžeme vidět že zde není hodnota 0 a to proto, že se používá metoda ukládání koeficientů popsaná výše. Chybí také poslední rozsah pro 11 bitů a to proto, že u AC koeficientů se nedělá diference, takže hodnota nikdy nemůže překročit rozsah 10 bitů.

*Tab. 3: Tabulka pro kódování AC koeficientů*

| Počet bitů | Rozsah |
|---|---|
| 1 | { -1}, {1} |
| 2 | {-3, -2}, { 2, 3} |
| 3 | {-7 . . -4}, { 4 . . 7 } |
| 4 | {-15 . . -8}, { 8 . . 15 } |
| 5 | {-31 . . -16}, { 16 . . 31} |
| 6 | {-63 . . -32}, { 32 . . 63 } |
| 7 | {127 . . -64},{ 64 . . 127} |
| 8 | {-255 . . -128}, { 128 . . 255 } |
| 9 | {-511 . . -256}, { 256 . . 511 } |
| 10 | {-1023 . . -512}, { 512 . . 1023} |

Příklad kódování:

zdrojová data: -40,10,0,2,2,6,-4,2,-6,0,0,3,2,-3 a 51 nul.

Zakódovaný výsledek:

(0x06,0x17), (0x04,0x8), (0x12,0x03), (0x02,0x03), (0x03,0x06), (0x03,0x03), (0x02,0x03), (0x03,0x01), (0x22,0x04), (0x02,0x03), (0x02,0x00), (0x00)

Horní data jsou v dekadické soustavě spodní data jsou v hexadecimální soustavě. Z příkladu je jasně vidět že místo 64 koeficientů, každý z délkou až 11bitů, jsme nuceni uložit pouze několik koeficientů proměnlivé délky.

### 2.2.6 Bezztrátová komprese

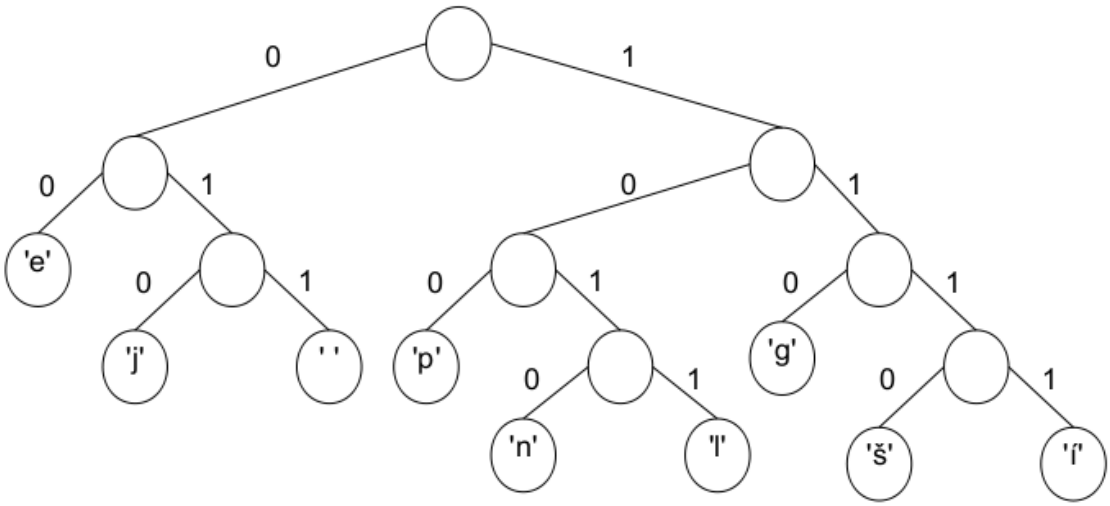
JPEG používá dva druhy bezztrátové komprese založených na pravděpodobnosti výskytů znaků v souboru. Huffmanovo a aritmetické kódování. Ve většině případech se setkáme pouze s Huffmanovým kódováním, protože při vydání standardu bylo aritmetické kódování zatíženo patentem.

#### *2.2.6.1 Huffmanovo kódování*

Huffmanovo kódování je algoritmus využívaný pro bezeztrátovou kompresi dat. Konvertuje znaky vstupního souboru do bitových řetězců různé délky. Znaky, které se ve vstupním souboru vyskytují s nejvyšší pravděpodobností, jsou konvertovány do bitových řetězců s nejkratší délkou (nejfrekventovanější znak tak může být zakódován do jediného bitu, znaky, které se vyskytují velmi zřídka, jsou zakódovány do delších řetězců.

Ukázka vytvoření Huffmanova kódování. Vezmeme si větu „jpeg je nejlepší“. Spočítáme si četnosti výskytů jednotlivých písmen ve větě:

'e' – 4, 'j' – 3, ' ' - 2, 'p' – 2, 'n' – 1, 'l' – 1, 'g' – 1, 'š' – 1, 'í' – 1,




*Obr. 9: Huffmanovo kódování*

Z vytvořeného stromu si můžeme odvodit binární řetězce, které nám budou reprezentovat písmena tabulka Tab. 4.

*Tab. 4: Vygenerované Huffmanovy kódy*

| Písmeno | Binární řetězec |
|---|---|
| 'e' | 00 |
| 'j' | 010 |
| ' ' | 011 |
| 'p' | 100 |
| 'n' | 1010 |
| 'l' | 1011 |
| 'g' | 110 |
| 'š' | 1110 |
| 'í' | 1111 |

Ve standardu JPEG je délka možných binárních řetězců omezena na maximálně 16 bitů. Jako hodnoty pro generování koeficientů jsou brány četnosti výskytu jednotlivých zakódovaných koeficientů lépe řečeno pro DC hodnoty je to počet bitů určujících rozsah, pro AC koeficienty je to hodnota určující rozsah s počtem nul.

V sekvenčním módu jsou standardně generovány 4 tabulky Huffmanových kódů. Pro jasovou složku jsou generovány zvlášť AC a DC. Pro barvonosné složky jsou také zvlášť generovány AC a DC. Některá zařízení a aplikace vždy uloží tabulky ze všemi možnými kombinacemi kódů, i když nejsou v obrázku použity.

#### *2.2.6.2 Aritmetické kódování*

Aritmetické kódování je opět bezztrátová kompresní metoda založená na četnosti výskytu znaku v souboru.

Algoritmus aritmetického kódování nejprve vyhodnotí statistické zastoupení jednotlivých znaků v textu. Poté vezme interval (0,1>, a ten rozdělí v poměru jednotlivých pravděpodobností. Intervalům takto vzniklým přiřadí zastupovaný znak. Ze vstupního souboru přečte první znak a uloží interval, do kterého patří. Ten pak opět rozdělí v podle statistického zastoupení a takto postupuje až do konce souboru. Z koncového intervalu vybere libovolné číslo a to pak uloží spolu se statistikou do komprimovaného souboru.[11

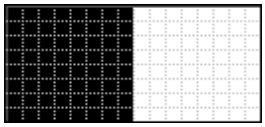
# 3 JFIF

JPEG sice definuje rámcově jak by měl být uložen, ale přesnou specifikaci určuje až JFIF. Bohužel ne každé zařízení a aplikace přesně dodržují tento standart.

## 3.1 Ukázkový soubor

Ukázkový soubor je obrázek o rozměrech 16x8 pixelů a nemá žádné podvzorkování. Takže jeho vzorkovací faktor je 1x1 1x1 1x1. Není to zrovna příklad obrázku, který by měl být uložen v JPEG, ale je natolik jednoduchý, že se hodí pro ukázku. Podle výše uvedených dat můžeme spočítat počet MCU jednotek na 2, obrázek navíc nemá žádné vyšší frekvence takže jeho dekódování bude poměrně snadné. Na obrázku 6 vidíme originál a na obrázku 7 je jeho zvětšenina s ohraničenými pixely.

*Obr. 10: Ukázkový obrázek*



*Obr. 11: Ukázkový obrázek zvětšený*

Ukázkový soubor převedený do hexadecimálního vyjádření, jednotlivé značky a jejich obsah je barevně oddělen:

```
FF D8 FF E0 00 10 4A 46 49 46 00 01 02 00 00 64 00 64 00 00 FF DB 00 84
00 02 02 02 02 02 02 02 02 02 02 03 02 02 02 03 04 03 02 02 03 04 05 04
04 04 04 04 05 06 05 05 05 05 05 05 06 06 07 07 08 07 07 06 09 09 0A 0A
09 09 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 01 03 03 03 05 04 05
09 06 06 09 0D 0B 09 0B 0D 0F 0E 0E 0E 0E 0F 0F 0C 0C 0C 0C 0C 0F 0F 0C
0C 0C 0C 0C 0C 0F 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C
0C 0C 0C 0C 0C 0C 0C 0C 0C 0C FF C0 00 11 08 00 08 00 10 03 01 11 00 02
11 01 03 11 01 FF C4 01 A2 00 00 00 07 01 01 01 01 01 00 00 00 00 00 00
00 00 04 05 03 02 06 01 00 07 08 09 0A 0B 01 00 02 02 03 01 01 01 01 01
00 00 00 00 00 00 00 01 00 02 03 04 05 06 07 08 09 0A 0B 10 00 02 01 03
03 02 04 02 06 07 03 04 02 06 02 73 01 02 03 11 04 00 05 21 12 31 41 51
06 13 61 22 71 81 14 32 91 A1 07 15 B1 42 23 C1 52 D1 E1 33 16 62 F0 24
72 82 F1 25 43 34 53 92 A2 B2 63 73 C2 35 44 27 93 A3 B3 36 17 54 64 74
C3 D2 E2 08 26 83 09 0A 18 19 84 94 45 46 A4 B4 56 D3 55 28 1A F2 E3 F3
C4 D4 E4 F4 65 75 85 95 A5 B5 C5 D5 E5 F5 66 76 86 96 A6 B6 C6 D6 E6 F6
37 47 57 67 77 87 97 A7 B7 C7 D7 E7 F7 38 48 58 68 78 88 98 A8 B8 C8 D8
E8 F8 29 39 49 59 69 79 89 99 A9 B9 C9 D9 E9 F9 2A 3A 4A 5A 6A 7A 8A 9A
AA BA CA DA EA FA 11 00 02 02 01 02 03 05 05 04 05 06 04 08 03 03 6D 01
00 02 11 03 04 21 12 31 41 05 51 13 61 22 06 71 81 91 32 A1 B1 F0 14 C1
D1 E1 23 42 15 52 62 72 F1 33 24 34 43 82 16 92 53 25 A2 63 B2 C2 07 73
D2 35 E2 44 83 17 54 93 08 09 0A 18 19 26 36 45 1A 27 64 74 55 37 F2 A3
B3 C3 28 29 D3 E3 F3 84 94 A4 B4 C4 D4 E4 F4 65 75 85 95 A5 B5 C5 D5 E5
F5 46 56 66 76 86 96 A6 B6 C6 D6 E6 F6 47 57 67 77 87 97 A7 B7 C7 D7 E7
F7 38 48 58 68 78 88 98 A8 B8 C8 D8 E8 F8 39 49 59 69 79 89 99 A9 B9 C9
D9 E9 F9 2A 3A 4A 5A 6A 7A 8A 9A AA BA CA DA EA FA FF DA 00 0C 03 01 00
02 11 03 11 00 3F 00 FC FF 00 E2 AF EF F3 15 7F FF D9
```

## 3.2 Značky

V souboru se vyskytují tzv. Markers česky značky. Ty udávají co je v následujícím segmentu dat uloženo. Každá značka začíná hexadecimální hodnotou 0xFF, která je následována jejím identifikátorem viz Tabulka. Nejsou to všechny značky, které se mohou vyskytovat v souboru, ale jsou to všechny značky přípustné pro sekvenční mód, který využívá Huffmanovo kódování. V souboru komprimovaných dat se ovšem může vyskytovat hodnota 0xFF to se řeší tak, že se použije identifikátor 0x00, tak můžeme tuto hodnotu naprosto přesně identifikovat při čtení proudu dat identifikátor 0x00 ignorovat.

*Tab. 5: Značky vyskytující se v JFIF souboru pouze pro sekvenční mód*

| Značka | Plný název | Hodnota | Význam |
|---|---|---|---|
| SOI | Start Of Image | 0×FFD8 | Začátek obrázku |
| EOI | End Of Image | 0×FFD9 | Konec obrázku |
| RSTi | Restart Marker i | 0×FFD0-7 | Restartující značky |
| SOFn | Start Of Frame n | 0×FFC0-4 | Specifikace tvaru obrazových dat |
| SOS | Start Of Scan | 0×FFDA | Začátek kódovaných obrazových dat |
| APPn | Application Marker | 0×FFE0-F | Značka specifických dat |
| COM | Comment | 0×FFFE | Komentář se zadanou délkou |
| DRI | Define Restart Interval | 0×FFDD | Interval výskytu restretujících značek RSTi |
| DQT | Define Quantization Table | 0×FFDB | Definice tabulky kvantizačních koeficientů |
| DHT | Define Huffman Table | 0×FFC4 | Definice Huffmanových tabulek |

### 3.2.1 SOI a EOI

Jsou značky které neobsahují žádná další data. Mezi těmito dvěmi značkami jsou uložena všechny ostatní značky. Znamenají začátek a konec obrázku. JFIF udává že značka SOI je na začátku souboru následovaná značkou 0xFFE0 APP0 a EOI je vždy na konci souboru.

### 3.2.2 APPn

Jsou v nich uloženy specifická data aplikací, které buďto soubor vytvořily nebo s ním nějak manipulovaly. Slouží k ukládání informací nad rámec specifikace JPEG. Formát těchto dat není striktně určen jediné co musí obsahovat za touto značkou je délku dat, které tato značka obsahuje. Takže aplikace zpracovávající tento obrázek může data, která nerozpozná prostě přeskočit.

Značka typu APPn může být uložena kdekoliv v souboru. Po uložení délky dat může následovat identifikační řetězec ukončený nůlovým znakem, ale není povinný.

JFIF formát používá značku APP0, která je popsána v tabulce

```
FF E0 00 10 4A 46 49 46 00 01 02 01 00 64 00 64 00 00
```

*Tab. 6: Obsah značky APP0*

| velikost | hodnota | popis |
|---|---|---|
| 2 | 0xFFE0 | značka APP0 |
| 2 | 0x0010 | délka segmentu APP0 |
| 5 | 0x4A46494600 | nulou ukončený řetězec „JFIF“ |
| 1 | 0x01 | číslo verze je |
| 1 | 0x02 | číslo subverze je také |
| 1 | 0x01 | rozlišení obrázku (následující čtyři bajty) v jednotkách:<br>0 – není definováno, 1 - DPI (dots per inch),<br>2-DPC (dot per cm) |
| 2 | 0x0064 | horizontální rozlišení |
| 2 | 0x0064 | vertikální rozlišení |
| 1 | 0x00 | šířka náhledového obrázku |
| 1 | 0x00 | výška náhledového obrázku |

### 3.2.3 COM

Poznámková značka. Nemá formální strukturu, musí pouze obsahovat velikost obsažených dat, která je uložena v prvních dvou bajtech za značkou. Měl by obsahovat text, nebo specifická data aplikací stejně jako APPn. Může být ignorován.

### 3.2.4 DQT

Obsahuje definice kvantizačních tabulek použitých v obrázku. Může definovat více kvantizačních tabulek a to až 4. Za značkou je informace o velikosti dat v rozsahu dvou bajtů. Každá tabulka obsahuje jako první informační bajt. Pokud jsou první 4 bity 0 tak každý koeficient je definovaný jedním bajtem a celá tabulka včetně informačního bitu dlouhá 65 bajtů, pokud je hodnota 1 každý koeficient je vyjádřen 2 bajty a celková délka tabulky je 129 bajtů. Dvoubajtové tabulky používají pouze obrázky z 12-bitovou hloubkou. Spodní 4 bity jsou numerický identifikátor tabulky, ten může nabývat hodnot 0 až 3. Poté následuje 64 kvantizačních koeficientů uloženy Cik-Cak.

Ještě je rozdíl mezi ukládáním tabulek JFIF verze 1.2 a ostatními. Na příkladu dat níže můžeme vidět uloženy dvě tabulky za sebou, které jsou od sebe odděleny barvou. To je verze 1.2. Verze 1.1 ukládá každou do samostatné značky.

```
FF DB 00 84 00 02 02 02 02 02 02 02 02 02 02 03 02 02 02 03 04 03 02 02
03 04 05 04 04 04 04 04 05 06 05 05 05 05 05 05 06 06 07 07 08 07 07 06
09 09 0A 0A 09 09 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 01 03 03
03 05 04 05 09 06 06 09 0D 0B 09 0B 0D 0F 0E 0E 0E 0E 0F 0F 0C 0C 0C 0C
0C 0F 0F 0C 0C 0C 0C 0C 0C 0F 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C
0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C 0C
```

*Tab. 7: Obsah značky DQT*

| velikost | hodnota | popis |
|---|---|---|
| 2 | FFDB | značka |
| 2 | 84 | Velikost segmentu |
| 1 | 0 | 4 horní bity definují velikost koeficientů: 0 – 1 bajt, 1 – 2bajty<br>4 dolní bity jsou identifikátor 0 - 3 |
| 64/128 | 0202.... | Koeficienty kvantizační tabulky |

### 3.2.5 DHT

Obsahuje definice Huffmanových tabulek použitých v obrázku. Může definovat více Huffmanových tabulek a to až 4 pro každou ze dvou tříd DC a AC. Za značkou je informace o velikosti dat v rozsahu dvou bajtů. Každá tabulka obsahuje jako první 17 bajtů. První bajt je informační. Pokud jsou první 4 bity 0 tak se jedná o tabulku třídy DC, pokud je hodnota 1 tak následující tabulka patří do třídy AC. Spodní 4 bity jsou numerický identifikátor tabulky, ten může nabývat hodnot 0 až 3. Pole 16 bytu udává počet Huffmanových kódu pro jednotlivé bitové délky (1-16). Pak už jsou jednotlivé koeficienty, jejich počet je proměnlivý. Zjistíme jej tak, že sečteme předchozích 16 bajtů, které udávaly počty Huffmanových kódů. Každý koeficient je vyjádřen jedním bajtem. Na základě této informace si můžeme sestavit počítadlo podle kterého načteme všechny koeficienty.

Opět je rozdíl mezi ukládáním tabulek JFIF verze 1.2 a ostatními. Na příkladu dat níže můžeme vidět uloženy čtyři tabulky za sebou, které jsou od sebe odděleny barvou. To je verze 1.2. Verze 1.1 ukládá každou do samostatné značky.

```
FF C4 01 A2 00 00 00 07 01 01 01 01 01 00 00 00 00 00 00 00 04 05 03
02 06 01 00 07 08 09 0A 0B 01 00 02 02 03 01 01 01 01 01 00 00 00 00 00
00 00 01 00 02 03 04 05 06 07 08 09 0A 0B 10 00 02 01 03 03 02 04 02 06
07 03 04 02 06 02 73 01 02 03 11 04 00 05 21 12 31 41 51 06 13 61 22 71
81 14 32 91 A1 07 15 B1 42 23 C1 52 D1 E1 33 16 62 F0 24 72 82 F1 25 43
34 53 92 A2 B2 63 73 C2 35 44 27 93 A3 B3 36 17 54 64 74 C3 D2 E2 08 26
83 09 0A 18 19 84 94 45 46 A4 B4 56 D3 55 28 1A F2 E3 F3 C4 D4 E4 F4 65
75 85 95 A5 B5 C5 D5 E5 F5 66 76 86 96 A6 B6 C6 D6 E6 F6 37 47 57 67 77
87 97 A7 B7 C7 D7 E7 F7 38 48 58 68 78 88 98 A8 B8 C8 D8 E8 F8 29 39 49
59 69 79 89 99 A9 B9 C9 D9 E9 F9 2A 3A 4A 5A 6A 7A 8A 9A AA BA CA DA EA
FA 11 00 02 02 01 02 03 05 05 04 05 06 04 08 03 03 6D 01 00 02 11 03 04
21 12 31 41 05 51 13 61 22 06 71 81 91 32 A1 B1 F0 14 C1 D1 E1 23 42 15
52 62 72 F1 33 24 34 43 82 16 92 53 25 A2 63 B2 C2 07 73 D2 35 E2 44 83
17 54 93 08 09 0A 18 19 26 36 45 1A 27 64 74 55 37 F2 A3 B3 C3 28 29 D3
E3 F3 84 94 A4 B4 C4 D4 E4 F4 65 75 85 95 A5 B5 C5 D5 E5 F5 46 56 66 76
86 96 A6 B6 C6 D6 E6 F6 47 57 67 77 87 97 A7 B7 C7 D7 E7 F7 38 48 58 68
78 88 98 A8 B8 C8 D8 E8 F8 39 49 59 69 79 89 99 A9 B9 C9 D9 E9 F9 2A 3A
4A 5A 6A 7A 8A 9A AA BA CA DA EA FA
```

*Tab. 8: Obsah značky DHT*

| velikost | hodnota | popis |
|---|---|---|
| 2 | FFC4 | značka |
| 2 | 01A2 | Velikost segmentu |
| 1 | 00 | 4 horní bity definují velikost třídu tabulky 0 – DC, 1 – AC<br>4dolní bity jsou identifikátor 0 - 3 |
| 16 | 000007 | Počet Huffmanových kódu pro délky řetězce od 1 do 16 |
| proměnlivá | 0405 | Hodnoty koeficientů |

### 3.2.6 DRI

Definuje počet MCU jednotek mezi jednotlivými restartujícími značkami v komprimovaných datech. Obsahuje dva bajty udávající délku dat ve značce. Následovaná jediným údajem, dvěma bajty, které určují restartující interval. Pokud je restartující interval nastaven a 0, tak resrtartující značky jsou nastaveny na 0.

### 3.2.7 RSTn

Rstartující značka udává blok dat, které se mohou dekódovat nezávisle. Stojí vždy samostatně a jsou rozmístěny v komprimovaných datech přesně podle definovaného intervalu ve značce DRI.

### 3.2.8 SOF

Začátek framu definuje obsah framu. U sekvenčního a progresivního módu je pouze jeden frame, více freamů je pouze v hierarchickém módu. Značka se skládá ze dvou částí a to fixní části a strukturou pro každou komponentu vyskytující se ve framu.

Po samotné značce a definici velikosti segmentu je fixní část. První bajtu udává velikost vzorku dat v bitech může být hodnota 8 nebo 12. Poté jsou dva dvou-bajty udávající výšku a šířku obrázku, jako poslední fixní část je bajt udávající počet komponent. JPEG dovoluje počet komponent 0 až 255 JFIF omezuje hodnoty na 3

Struktura, která je pro každou komponentu vyskytující se ve framu stejná. V ukázkovém obrázku jsou 3 tyto struktury, každá jinou barvou. Struktura obsahuje identifikátor u JFIF v přesném pořadí (Y, Cb, Cr) jejich identifikátory jsou(1,2,3). Další baj je rozdělen na dvě 4 bitové hodnoty. Horní 4 bajty znamenají horizontální vzorkovací frekvenci a dolní 4 bity udávají vertikální vzorkovací frekvenci. Vzorkovací frekvence mohou nabývat hodnot 1-4 často však aplikace nepodporují 3, protože může docházet ke konfliktům při špatně nastaveném poměru vertikální ku horizontální frekvenci u komponent. Poslední je identifikátor kvantizační tabulky.


*Tab. 9: Obsah značky SOF fixní část*

| velikost | hodnota | popis |
|---|---|---|
| 2 | FFC0 | značka |
| 2 | 0011 | Velikost segmentu |
| 1 | 08 | Definuje bitovou přesnost vzorku 8bit nebo 12bit |
| 2 | 0008 | Výška obrázku v pixelech |
| 2 | 0010 | Šířka obrázku v pixelech |
| 1 | 03 | Počet komponent vyskytujících se ve framu |

*Tab. 10: Obsah značky SOF struktura popisující komponentu*

| velikost | hodnota | popis |
|---|---|---|
| 1 | 01 | Identifikátor komponenty v JFIF 1(Y), 2(Cb), 3(Cr) |
| 1 | 11 | 4 horní bity horizontální vzorkovací frekvence 1-4<br>4dolní bity vertikální vzorkovací frekvence 1-4 |
| 1 | 00 | Šířka obrázku v pixelech |

### 3.2.9 SOS

Tato značka znamená začátek komprimovaných dat u sekvenčního je pouze jedna u progresivního jich je 2 a více. První bajt popisuje počet komponent, které jsou v komprimovaných datech použity. Další dva bajty vždy popisují komponentu. První bajt je identifikátor komponenty. Další baj je rozdělen na dvě 4-bitové hodnoty. Horní 4 bity obsahují identifikátor DC Huffmanovy tabulky a dolní 4 bity obsahují identifikátor AC Huffmanovy tabulky.

Další 3 jednobajtové hodnoty se používají pro progresivní mód, počáteční hodnota spektrální selekce, konečná hodnota spektrální selekce a úspěšnost aproximace. Tyto tři hodnoty jsou u sekvenčního módu vždy nastaveny na 0, 63 a 0. Za těmito informacemi následují komprimovaná data. Komprimovaná data se ovšem nezaznamenávají do délky této značky. Před touto značkou musí být uloženy všechny informace nutné k dekompresi dat.


*Tab. 11: Obsah značky SOS*

| velikost | hodnota | popis |
|---|---|---|
| 2 | FFC0 | značka |
| 2 | 000C | Velikost segmentu |
| 1 | 03 | Počet komponent |
| 2 | 0100 | Popis komponenty |
| 1 | 00 | Počáteční hodnota spektrální selekce |
| 1 | 3F | Konečná hodnota spektrální selekce |
| 1 | 00 | Úspěšnost aproximace |

# 4 PROGRAMOVÉ PROSTŘEDKY

## 4.1 Python

Byl vytvořen v roce 1990 Guido van Rossumem na Stichting Mathematisch Centrum v Nizozemsku jako nástupce jazyku zvaného ABC. Python je interpretovaný, interaktivní, objektově orientovaný programovací jazyk. [13]

Interpretr včetně základních knihoven je dostupný pro více operačních systémů, Windows, Linux, Mac OS a můžeme se s ním setkat i v některých mobilních telefonech a zařízeních. Klasický Python je implementován v jazyce C, někdy je také nazýván CPython. Ten umožňuje používat rozšíření pomocí C, C++ modulů, pro vyšší výkon aplikací. V době psaní této práce je poslední stabilní verzí Python 3.1 a Python 2.6, který slouží pro přechod na vyšší verzi.

Dalšími implementacemi Pythonu je Jython. Kód napsaný Jythonu je možné spustit v Java Virtual Machine (interpretr Javy) a může využívat knihovny prostředí Java.[wiki Python]. Nevýhodou je zaostávání za verzemi CPythonu.

Poslední implementací je určená pro prostředí .NET/Mono A její název je IronPython.

Základní knihovna Pythonu obsahuje nepřebernou množství modulů. Například pro práci z textovými řetězci, soubory, …, sítí, obsahuje i modul pro tvorbu jednoduchého grafického rozhraní Tk nebo můžeme využít i jiné knihovny než standartní jako wxPython a PyQt.

Jako interpretovaný jazyk dokáže ušetřit spoustu času, protože kód už nemusíme kompilovat a díky interaktivní konzoli můžeme i malé části kódu okamžitě vyzkoušet. Soubory spustitelné na interpretru Pythonu mají příponu py. Pro urychlení spuštění můžeme moduly překompilovat do bytekódu, se kterým interpretr nativně pracuje. Tyto soubory budou mít příponu pyc. Nelze je převést zpět na čitelnou podobu, takže Python může sloužit i pro tvorbu proprietárního softwaru.

## 4.2 Psyco

Je to zkratka pro Python Specializing Compiler, vyvinotý Arminem Rigo[14]. Python modul, který dokáže zrychlit vykonávání programu 3x-10x v extrémních případech až 100x. Psyco funguje zjednodušeně tak, že pracuje nad interpretrem Pythonu a před samotným vykonáváním kódu provádí za běhu jeho optimalizaci.

Výhodou použití Psyca pro urychlení vykonávání programu je to, že kód nemusíme nijak upravovat, stačí pouze naimportovat modul a použít jej.

## 4.3 Eclipse

Projekt Eclipse vytvořila firma IBM a nyní jej spravuje společnost Eclipse Foundation.

Eclipse je vývojové prostředí IDE. Jedná se o open source vývojovou platformu[15].

Eclipse je napsán v Javě, proto je nutné pro jeho zprovoznění mít nainstalován interpretr Javy JVM. Původně slouží pro psaní kódu v Javě, ale díky snadné integraci rozšiřujících modulů a zaštítě mnoha významných softwarových a hardwarových firem se vytvořila celá škála rozšíření pro tvorbu programů i v jiných jazycích.

Pydev je rozšíření Eclipse pro tvorbu softwaru v jazyce Python. Po nainstalování Pydev a nastavení interpretru, můžeme v Eclipse psát a testovat kód napsaný v Pythonu. Jako další nástroje pro usnadnění vývoje nabízí automaticky zvýrazňující syntaxi, nápovědu použitých proměnných a funkci během psaní, atd.

# II. PRAKTICKÁ ČÁST

# 5 DEKÓDOVÁNÍ

Pro příklad získaní histogramu Huffmanova kódování použijeme obrázek Obr. 10 z kapitoly 3.1. Obrázek je na dekódování velice jednoduchý a přímočarý.

## 5.1 Načtení dat

Před samotným dekódováním obrázku musíme načíst všechny potřebné značky, kterými jsou DHT, SOF a SOS.

Protože obrázek obsahuje Huffmanovy tabulky, které uchovávají všechny přípustné kombinace kódů. Tak níže v tabulce uvedu pouze kódy nutné k dekódování obrázku. Celý obsahu Huffmanových tabulek naleznete v příloze PII.

*Tab. 12:Ořezané Huffmanovy tabulky*

| Tabulka | Binární řetězec | kód |
|---|---|---|
| DC 0 | 1111110 | 0x0A |
| DC 1 | 1100 | 0x00 |
| AC0 | 01 | 0x00 |
| AC 1 | 01 | 0x00 |

Ještě potřebujeme informace ze značky SOF a SOS. Abychom mohli spočítat počet MCU jednotek, jejich velikost a přiřadit jednotlivým komponentům Huffmanovy tabulky.

Vzorkovací faktor - 1x1 1x1 1x1

Počet MCU jednotek – 2

*Tab. 13: Přiřazené tabulky jednotlivým komponentám*

| Komponenta | Kvantizační tabulka | Huffmanova tabulka | |
|---|---|---|---|
| | | DC | AC |
| Y | 0 | 0 | 0 |
| Cb | 1 | 1 | 1 |
| Cr | 1 | 1 | 1 |

Nyní můžeme přistoupit k dekódování dat. Zkomprimované data v hexadecimální podobě:

```
FC FF 00 E2 AF EF F3 15 7F
```

Převedeme si řetězec do binární formy a vynecháme při tom třetí bajt 0x00, protože je to identifikátor značky, který se při čtení komprimovaných dat vynechává. Binární reprezentace dat:

1111 1100 1111 1111 1110 0010 1010 1111 1110 1111 1111 0011 0001 0101 0111 1111

## 5.2 Dekódování MCU 1

**Y komponenta DC:**

V binární reprezentaci najdeme řetězec nacházející se v Huffmanově tabulce pro DC hodnotu s identifikátorem 0

1111 1100 1111 1111 1110 0010 1010 1111 1110 1111 1111 0011 0001 0101 0111 1111

Kód je 0x0A což nám udává že musíme načíst 10 bitů

1111 1100 1111 1111 1110 0010 1010 1111 1110 1111 1111 0011 0001 0101 0111 1111

Tato hodnota reprezentuje číslo -512

**Y komponenta AC:**

Najdeme řetěze z Huffmanovy tabulky AC 0

1111 1100 1111 1111 1110 0010 1010 1111 1110 1111 1111 0011 0001 0101 0111 1111

Kód reprezentuje hodnotu 0x00 což znamená konec bloku. Další hodnoty už se nebudou načítat protože konec bloku říká, že všechny následující komponenty jsou 0.

**Cb komponenta DC:**

Najdeme řetěze z Huffmanovy tabulky DC 1

1111 1100 1111 1111 1110 0010 1010 1111 1110 1111 1111 0011 0001 0101 0111 1111

Kód reprezentuje hodnotu 0x00, žádná další data nejsou pro DC hodnotu načítána.

**Cb komponenta AC:**

Najdeme řetěze z Huffmanovy tabulky AC 1

1111 1100 1111 1111 1110 0010 1010 1111 1110 1111 1111 0011 0001 0101 0111 1111

Kód reprezentuje hodnotu 0x00, všechny další hodnoty tohoto bloku jsou 0.

**Cr komponenta DC:**

Najdeme řetěze z Huffmanovy tabulky DC 1

1111 1100 1111 1111 1110 0010 1010 1111 1110 1111 1111 0011 0001 0101 0111 1111

Kód reprezentuje hodnotu 0x00, žádná další data nejsou pro DC hodnotu načítána.

**Cr komponenta AC:**

Najdeme řetěze z huffmanovy tabulky AC 1

1111 1100 1111 1111 1110 0010 1010 1111 1110 1111 1111 0011 0001 0101 0111 1111

Kód reprezentuje hodnotu 0x00, všechny další hodnoty tohoto bloku jsou 0.

## 5.3 Dekódování MCU 2

**Y komponenta DC:**

1111 1100 1111 1111 1110 0010 1010 1111 1110 1111 1111 0011 0001 0101 0111 1111

získaný kód 0x0A

1111 1100 1111 1111 1110 0010 1010 1111 1110 1111 1111 0011 0001 0101 0111 1111

dekódovaná hodnota 1020

**Y komponenta AC:**

1111 1100 1111 1111 1110 0010 1010 1111 1110 1111 1111 0011 0001 0101 0111 1111

získaný kód 0x00 všechny AC koeficienty jsou 0

**Cb komponenta DC:**

1111 1100 1111 1111 1110 0010 1010 1111 1110 1111 1111 0011 0001 0101 0111 1111

Kód 0x00

**Cb komponenta AC:**

1111 1100 1111 1111 1110 0010 1010 1111 1110 1111 1111 0011 0001 0101 0111 1111

Kód 0x00

**Cr komponenta DC:**

1111 1100 1111 1111 1110 0010 1010 1111 1110 1111 1111 0011 0001 0101 0111 1111

Kód 0x00

**Cr komponenta AC:**

1111 1100 1111 1111 1110 0010 1010 1111 1110 1111 1111 0011 0001 0101 0111 1111

Kód 0x00

## 5.4 Zbytek

Konec komprimovaných dat nebo konec MCU jednotky před restartovací značkou RST musí být dorovnán do celého bajtu. Takže tento zbytek jednoduše zanedbáme.

1111 1100 1111 1111 1110 0010 1010 1111 1110 1111 1111 0011 0001 0101 0111 1111

## 5.5 Výsledný histogram Huffmanova kódování

Z dekódovaných Huffmanových koeficientů můžeme sestavit histogram, který je nutný pro stegoanalýzu pomocí neuronové sítě. Výsledný histogram je v tabulce Tab. 14.

Tab. 14: Histogram Huffmanova kódování

| Bitová délka | DC | | AC | |
|---|---|---|---|---|
| | 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 0 | 0 |
| 2 | 0 | 4 | 0 | 4 |
| 3 | 0 | 0 | 0 | 0 |
| 4 | 0 | 0 | 2 | 0 |
| 5 | 0 | 0 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 |
| 7 | 2 | 0 | 0 | 0 |
| 8 | 0 | 0 | 0 | 0 |
| 9 | 0 | 0 | 0 | 0 |
| 10 | 0 | 0 | 0 | 0 |
| 11 | 0 | 0 | 0 | 0 |
| 12 | 0 | 0 | 0 | 0 |
| 13 | 0 | 0 | 0 | 0 |
| 14 | 0 | 0 | 0 | 0 |
| 15 | 0 | 0 | 0 | 0 |
| 16 | 0 | 0 | 0 | 0 |

# 6 DEKODÉR

Dekodér je napsán v programovacím jazyku Python verze 2.6, s použitím optimalizačního modulu Psyco. Celý dekodér se skládá z jedné třídy jpegDecoder, která má dvě veřejné funkce, konstruktor __init__ a funkci decode, která zahájí dekódovací proces a jako návratovou hodnotu vrací histogram huffmanova kódování.

## 6.1 Použité moduly

V práci jsou použity dva standardní moduly sys a struct. A optimalizační modul psyco, který není součástí standardní knihovny a musí se instalovat zvlášť. Je nepovinný a pokud se při jeho zavedení vyvolá výjimka tak se zachytí a program pokračuje se dál. Problém bude akorát ten, že kód se bude vykonávat 4x pomaleji než s modulem.

Volání funkce psyco.full() zajistí optimalizaci celého modulu, jsou i další možnosti volání optimalizace pomocí Psyco, ale tato se ukázala jako nejlepší.

```
import sys
import struct
try:
    import psyco
    psyco.full()
except ImportError:
    pass
```

Instalace Psyco modulu je v Linuxu je velice jednoduchá. Ve většině distribucích je obsažena už v repositářích nebo pokud distribuce umí pracovat s distribučními balíčky debianu, tak je můžeme stáhnou ze stránky packages.debian.org, jako poslední možnost se nabízí instalace přímo ze zdrojového kódu dostupného z sourceforge.net.

Instalace z repositáře Ubuntu:

*sudo apt-get install python-psyco*

Instalace ze zdroje Ubuntu :

*sudo python setup.py install*

U Windows je ovšem situace trochu komplikovanější. Zde musíme instalovat jedině ze zdroje, protože na stránce sourceforge.net není k dispozici přeložený binární soubor pro Python verze 2.6. Abychom mohli instalovat ze zdroje v operačním systému Windows musíme splnit dvě podmínky a těmi jsou instalovaný C/C++ překladač a příkaz python do systémové proměnné *Path*.

Přidat Python do systémové proměnné *Path* se dá jednoduše přes nabídku *Ovládací panely → Systém → Upřesnit → Proměnné prostředí*. V nabídce Systémové proměnné najdeme proměnou *Path* a klikneme na tlačítko upravit. Na konec proměnné přidáme cestu k adresáři, kde máme Python nainstalovaný a nesmíme před něj zapomenout dát středník.

*;C:\Program Files\Python26*

Pro otestování zda jsme vše udělali správně stačí otevřít Příkazovou řádku systému Windows a napsat *python*, pokud se nám otevřela konzola Pythonu, vše je v pořádku.

Nejdříve si stáhneme překladač, já jsem zvolil MinGW, ten je k dispozici na stránce sourceforge.net. Překladač nainstalujeme. Po úspěšné instalaci zaregistrujeme složku překladače \MinGW\Bin do systémové proměnné *Path* stejně jako předtím Python:

*;C:\Program Files\MinGW\Bin*

Po instalaci překladače ho musíme zaregistrovat v Pythonu. Ve složce C:\Program Files\Python26\lib\distutils\. Vytvoříme textový soubor *distutils.cfg*. Jeho obsahem bude:

[build]

compiler = mingw32

Po všech úspěšně vykonaných krocích můžeme modul nainstalovat. V příkazové řádce si otevřeme složku, kde jsme uložili zdrojové kódy Psyco a spustíme příkaz:

*python setup.py install*

## 6.2 Třída jpegDecoder

Je to jediná velká třída v celém modulu. Má jediný konstruktor, který jako vstupní argument potřebuje objekt typu file. Konstruktor si ověří zda je vstupní argument opravdu typu file, poté si načte první dva bajty a ověří jestli se shodují ze značkou SOI. Nakonec se vytvoří datová struktura pro uchování Huffmanových tabulek, která obsahuje dvourozměrný seznam 4 * 16 a v každé položce je slovník.

Někdo může namítat, že konstrukce pomocí binárního stromu by byla rychlejší, než vyhledávání v jednotlivých slovnících. Toto tvrzení by platilo v jazyku C, ale Python rychleji vyhledává ve slovníku, který je optimalizovaný a napsaný jako nízko úrovňová třída.

```
Class jpegDecoder:

    def __init__(self, inputfile):
```

```
        if isinstance(inputfile,file):
            self.fin = inputfile
        else:
            raise Exception('Except file as input to jpegDecoder')
        data = struct.unpack('>H',self.fin.read(2))

        if data[0]!=0xffd8:
            raise Exception('File is not JPEG image')

        self.huffTables=[]
        for i in xrange(4):
            self.huffTables.append([])
            for j in xrange(16):
                self.huffTables[i].append({})
```

### 6.2.1 Decode

Zavoláním funkce decode se spustí dekódovací proces, který pokud nenastane žádná chyba skončí navrácením histogramu Huffmanova kódováni v dvourozměrném listu o rozměrech 4 x 16. Funkce postupně prochází soubor a hledá znak s hexadecimální hodnotou 0xFF. Pokud najde shodný znak, tak načte další znak ze souboru a použije ho jako klíč ke slovníku crossway. Ve slovníku jsou klíči znaky odpovídající druhým bajtům značek, hodnoty jsou odkazy na funkce třídy jpegDecoder, které zpracovávají příslušné značky.

Když se klíč vyskytuje ve slovníku zavolá se příslušná funkce. Pokud ne vyvolá se výjimka KeyError, kterou zachytíme a zjistíme jestli je znak roven hodnotě 0xF9 což znamená konec souboru.

```
def decode(self):
    crossway={chr(0xC0):self.__readSOF,chr(0xC4):self.__readDHT,
              chr(0xDA:self.__decodeSOS,chr(0xDB):self.__skip,
              chr(0xE0):self.__readAPP0,chr(0xDD):self.__skip,
              chr(0xE1):self.__skip,chr(0xE2):self.__skip,
              chr(0xE3):self.__skip,chr(0xE4):self.__skip,
              chr(0xE5):self.__skip,chr(0xE6):self.__skip,
              chr(0xE7):self.__skip,chr(0xE8):self.__skip,
              chr(0xE9):self.__skip,chr(0xEA):self.__skip,
              chr(0xEB):self.__skip,chr(0xEC):self.__skip,
              chr(0xED):self.__skip,chr(0xEE):self.__skip,
              chr(0xEF):self.__skip,chr(0xFE):self.__skip}

    while 1:
        while '' !=self.fin.read(1) != '\xff':
            pass
        try:
```

```
            pomchar = self.fin.read(1)
            crossway[pomchar]()
        except KeyError:
            if pomchar==chr(0xD9):
                break
            else:
                raise Exception('Unknow Marker0xFF{0}'.format
                                        hex(ord(pomchar)) [2:]))
    self.fin.close()
    return self.huffStats
```

### 6.2.2 Funkce skip

Všechny značky, které nestojí samostatně mají údaj o své velikosti. Jsou to první dva bajty za značkou. Funkce _skip se používá vždy, když nepotřebujeme číst obsah značky.

```
def __skip(self):
   self.fin.seek(self.fin.tell()+struct.unpack('>H',self.fin.read(2))[0])
```

### 6.2.3 Načtení APP0

Funkce načte značku APP0, která by měla sloužit pro uchování informací o JFIF. Načítá se pouze prvních 9 bajtů. Pomocí modulu struct a jeho funkce unpack. Ověří se identifikační řetězec, pokud neodpovídá vyvolá se výjimka. Zbytek nepřečtené značky se ignoruje a přeskočí se na její konec.

```
def __readAPP0(self):
    dataAPP0 = struct.unpack('>H5sBB',self.fin.read(9))
    if dataAPP0[1]=='JFIF\0':
        self.version=dataAPP0[7:9]
    elif dataAPP0[1]=='JFXX\0':
        self.verzion=[0,0]
    else:
        raise Exception('Unknow JFIF identifer'+dataAPP0[1])
    self.fin.seek(self.fin.tell() + dataAPP0[0]-9)
```

### 6.2.4 Načtení značky SOF

Tato funkce načte celý obsah značky SOF do n-tice markerSOF. Navíc rozdělí horizontální a vertikální vzorkovací frekvenci do samostatných proměnných. Obsah značky SOF viz kapitola Chyba: zdroj odkazu nenalezen. A ověří jestli délka značky odpovídá pozici v souboru, pokud ne vyvolá vyjímku.

```
def __readSOF(self):
    pomlength = self.fin.tell()
    length=struct.unpack('>H',self.fin.read(2))[0]
    self.markerSOF=struct.unpack('>BHHB',self.fin.read(6))
```

```
    for i in xrange(self.markerSOF[3]):
        pomstruct=struct.unpack('>BBB',self.fin.read(3))
        self.markerSOF+=(pomstruct[0],pomstruct[1] >> 4,pomstruct[1] &
         0b00001111,pomstruct[2])
    if pomlength + length != self.fin.tell():
     raise Exception('Unexcepted SOF marker end at'+hex(self.fin.tell()))
```

### 6.2.5 Načtení značky SOS

Funkce načte celý obsah značky SOS do n-tice markerSOS. Rozdělí bajt, který identifikuje jaké používá Huffmanovy tabulky pro DC a AC koeficienty. Na dva samostatné pomocí maskování a bitového posunu. Stejně tak rozdělí bajt úspěšné aproximace na dva samostatné bajty. Obsah značky SOS viz kapitola Chyba: zdroj odkazu nenalezen. A ověří jestli délka značky odpovídá pozici v souboru, pokud ne vyvolá vyjímku.

```
def __readSOSheader(self):
    pomlength = self.fin.tell()
    self.markerSOS=struct.unpack('>HB',self.fin.read(3))
    for i in xrange(self.markerSOS[1]):
        pomstruct=struct.unpack('>BB',self.fin.read(2))
        self.markerSOS+=(pomstruct[0],pomstruct[1] & 0b11110000 >> 4,pomstruct[1] & 0b1111)
    pomstruct=struct.unpack('>BBB',self.fin.read(3))
    self.markerSOS+=(pomstruct[0],pomstruct[1],pomstruct[2] & 0b11110000
    >> 4,pomstruct[2] & 0b1111)
    if  pomlength + self.markerSOS[0]!= self.fin.tell():
     raise Exception('Unexcepted SOS marker end at'+hex(self.fin.tell()))
```

### 6.2.6 Načtení Huffmanových tabulek

Na začátku funkce si ze souboru načteme délku značky DHT. Zjistíme aktuální pozici v souboru a přičteme ji k délce značky mínus dva, to nám dá pozici v souboru na které by jsme měli dočíst tuto značku a hodnotu uložíme do proměnné EndPos.

Funkce v cyklech načítá jednu tabulku za druhou dokud je pozice v souboru menší než obsah proměnné EndPos. V cyklu se první načte identifikátor tabulky, po úpravě nám bude dělat ukazatel do proměnné huffTables uložíme jej do proměnné pom. Ze souboru načteme do proměnné pomTab n-tici 16 8-bitových hodnot, které nám udávají počet kódů v různých bitových délkách. Ještě potřebujeme inicializovat na 0 celočíselnou proměnnou lastCode, pomocí které budeme generovat Huffmanův kód.

První cyklus for určuje délku kódu lastCode se posune bitovým posunem o 1 bit doleva, vnořený cyklus se opakuje tolikrát, kolik kódů má byt vytvořeno v příslušné bitové délce. Postupně přidáváme do vybraného slovníku přidáváme Huffmanovy kódy generované z

proměnné lastCode jako klíče a načítáme vždy jednu celočíselnou hodnotu ze souboru, která udává zakódovanou hodnotu koeficientu.

Nakonec funkce se ověří jestli pozice v souboru se shoduje s proměnnou EndPos, pokud ne soubor je poškozen a nelze zrekonstruovat Huffmanovy tabulky.

```
def __readDHT(self):
    length = struct.unpack('>H',self.fin.read(2))[0]
    EndPos = self.fin.tell() + length -2

    while self.fin.tell() < EndPos:
        pomchar = self.fin.read(1)
        pom = (ord(pomchar)>>4) * 2 + ord(pomchar) & 0b00001111
        pomTab = struct.unpack('>BBBBBBBBBBBBBBBB',self.fin.read(16))
        lastCode = 0

        for j in xrange(1,17):
            pomFormat = '{0:0'+str(j)+'b}'
            lastCode <<= 1
            for k in xrange(pomTab[j-1]):
               self.huffTables[pom][j-1][pomFormat.format(lastCode)]=ord(self.fin.read (1))
               lastCode=lastCode+1

    if self.fin.tell() != EndPos:
        raise IndexError('DHT Bad')
```

### 6.2.7 Dekódování dat

Jako první se volá funkce pro načtení značky SOS. Pak se vytvoří seznam všech komponent v obrázku. Pro každou komponentu se uloží informace v tomto pořadí identifikátor komponenty, identifikátor Huffmanovy tabulky DC, AC, vertikální vzorkovací frekvence, horizontální vzorkovací frekvence a identifikátor kvantizační tabulky.

Na základě vzorkovacích frekvencí dojde se vypočítá kolik MCU jednotek je v celých obrazových datech přítomno. Ještě před zahájením dekódování jednotlivých komponent, musíme vytvořit proměnou huffStats, do které se bude ukládat statistika výskytu Huffmanových kódu v obrazových datech a zavolat funkci pro naplnění binárního bufferu.

Funkce v cyklu prochází MCU jednotky jednu za druhou a volá funkci pro dekódování komponenty, která se opakuje tolikrát, kolik je v obrazových datech komponent. Po každé MCU jednotce se ověří jestli za ní nenásledoval restartovací značka, pokud ano tak se posune index ukazující pozici v binárním bufferu bitOffset. Funkce nemá žádnou návratovou hodnotu.

```
def __decodeSOS(self):
    self.__readSOSheader()
    components=[]
    for i in xrange(self.markerSOS[1]):
        components.append(list(self.markerSOS[2+3*i:5+3*i]))
        for j in xrange(self.markerSOF[3]):
            if components[i][0]==self.markerSOF[4+4*j]:
                components[i].append(self.markerSOF[5+4*j])
                components[i].append(self.markerSOF[6+4*j])
                components[i].append(self.markerSOF[7+4*j])

    xFrequency=[]
    yFrequency=[]
    for i in xrange(len(components)):
        xFrequency.append(components[i][3])
    for i in xrange(len(components)):
        yFrequency.append(components[i][4])

    MCUx=(self.markerSOF[2]+8*max(xFrequency)-1)/(max(xFrequency)*8)
    MCUy=(self.markerSOF[1]+8*max(yFrequency)-1)/(max(yFrequency)*8)

    self.huffStats = []
    for i in xrange(4):
        self.huffStats.append([0]*16)

    buffer = self.__fillBuffer(self.fin)
    self.bitOffset = 0

    restartMarkers = self.rstMarkers

    for i in range(MCUx*MCUy):
        for j in range(len(components)):
            for k in range(components[j][4]*components[j][3]):
                self.__huffDecode(components[j],buffer)

        if self.bitOffset > restartMarkers[0]:
            self.bitOffset = restartMarkers.pop(0) + 8
```

### 6.2.8 Funkce pro naplnění Bufferu

Pro dekódování, musíme převést data reprezentované textem na binární reprezentaci. K tomuto účelu slouží funkce __fillBuffer. Jako svůj vstupní argument funkce požaduje objekt typu file. Proměnná byteTObits je n-tice naplněná 256 8-bitovými binárními řetězci, reprezentující všechny možné kombinace. Self.Markers slouží k zachycení resetovacích značek. Funkce postupně převádí vstupní soubor z textu na binární řetězce. V

případě zachycení znaku 0xff načte další znak zjistí jestli je to restartovací značka, zástupná značka pro znak 0xff nebo jiná značka, kterou celý převod končí. Funkce jako návratovou hodnotu vrací jeden velký binární řetězec.

```
def __fillBuffer(self,input):
    buffer = []
    counter = 0
    self.rstMarkers = []
    byteTObits = tuple(map('{0:08b}'.format, xrange(256)))
    while 1:
        pomchar = input.read(1)
        if pomchar=='\xff':
            pomchar = input.read(1)
            if pomchar =='\x00':
                buffer.append(byteTObits[255])
                counter += 1
            elif 208 <= ord(pomchar) <= 215:
                self.rstMarkers.append(counter * 8 - 8)
            else:
                input.seek(input.tell()-2)
                break
        else:
            buffer.append(byteTObits[ord(pomchar)])
            counter += 1
    self.rstMarkers.append(counter*8+8)
    return ''.join(buffer)
```

### 6.2.9 Dekódování komponenty

Poslední funkcí je __huffDecode. Tady dochází k dekódování komponenty za použití Huffmanova kódování. Vstupní argumenty funkce jsou seznam obsahující vlastnosti komponenty a binární řetězec souboru dat. První si musíme nastavit proměnnou numOfCoef, která nám udává počet dekódovaných koeficientů jedné komponenty. Nastavíme ji na 1, protože DC koeficient se dekóduje vždy. Jako další si přepíšeme globální proměnné na lokální a zavedeme si proměnnou i, která nám bude ukazovat délku právě dekódovaného slova, tzn. příslušný slovník.

Dekódování DC koeficientu probíhá v jednom cyklu, kdy si ověřujeme jestli řetězec o délce i z proměnné binBuffer je klíčem ve slovníku pomDcHuff, pokud není zvýšíme i o 1 a znovu ověřujeme. Když nalezneme klíč, který se nachází ve slovníku, načteme jeho hodnotu. Ta udává kolik následujících bitů musíme načíst, abychom dostali zakódovaný koeficient viz. kapitola Chyba: zdroj odkazu nenalezen. Zvýšíme hodnotu v proměnné huffStat odpovídající i o 1 a navýšíme proměnou pomBitOffset o hodnotu i + index.

Obdobně postupujeme i u AC koeficientů, ale zde musíme proce opakovat dokud nebude proměnná numOfCoef větší nebo rovna 64. Ještě je zde jeden rozdíl a to ten že při načtení zakódované hodnoty koeficientu z tabulky, jej musíme rozdělit na horní a dolní čtyři bity. Horní udává počet nul o ten navýšíme proměnou numOfCoef a dolní čtyři bity udávají počet bitů, které musíme načíst, abychom dostali zakódovanou hodnotu viz. kapitola Chyba: zdroj odkazu nenalezen

```
def __huffDecode(self,compProperty,binBuffer):
    dcHuff = compProperty[1]
    acHuff = compProperty[2]+2
    numOfCoef = 1

    pomDcHuff = self.huffTables[dcHuff]
    pomAcHuff = self.huffTables[acHuff]
    pomAcStats = self.huffStats[acHuff]
    pomDcStats = self.huffStats[dcHuff]

    pombitOffset = self.bitOffset
    i = 0
    #Prace s DC dohnotou
    while i < 16:
        if binBuffer[pombitOffset:pombitOffset + i + 1] in pomDcHuff[i]:
            index=pomDcHuff[i][binBuffer[pombitOffset:pombitOffset+i+1]]
            pombitOffset = pombitOffset + index + i +1
            pomDcStats[i] += 1
            break
        else:
            i = i + 1
    #Prace s AC hodnotami
    while numOfCoef <64:
        i = 0
        while i < 16:
            if binBuffer[pombitOffset:pombitOffset+i+1] in pomAcHuff[i]:
              index=pomAcHuff[i[binBuffer[pombitOffset:pombitOffset+i+1]]
                if index == 0:
                    numOfCoef = 64
                    pomAcStats[i] += 1
                    pombitOffset = pombitOffset + i +1
                    break
                elif index == 0xf0:
                    numOfCoef = numOfCoef + 16
                    pomAcStats[i] += 1
                    pombitOffset = pombitOffset + i +1
                    break
                numOfCoef = numOfCoef + (index >> 4) + 1
```

```
                pombitOffset = pombitOffset + (index & 0b00001111) + i +1
                pomAcStats[i] += 1
                break
            else:
                i = i + 1
    self.bitOffset = pombitOffset
```

## 6.3 Použití modulu

Pokud chceme použít modul máme dvě možnosti. Buď jej můžeme spustit z příkazové řádky jako skript, nebo jej importovat jako modul do programu.

Použití v programu je velice jednoduché stačí modul naimportovat, pozor modul musí být ve složce společně se souborem ve kterém ho chceme použít nebo umístěn do adresáře Pythonu. V operačním systému Windows musíme otevřít soubor pro čtení v binární režimu jinak dekodér nebude fungovat správně. Příklad:

```
import jpegDecoder
soubor = open('/home/uzivatel/obrazky/obrazek.jpg','rb')
dekoder = jpegDecoder.jpegDecoder(soubor)
vysledek = dekoder.decode()
print vysledek
```

Modul neobsahuje jen třídu, ale i kód, který umožní modul použít i z příkazové řádky. Když se modul spustíme jako skript z příkazové řádky proměnná Pythonu *__name__* bude mít hodnotu *'__main__'* a pomocný kód se vykoná. Jinak má proměnná stejný název jako modul v našem případě *'jpegDecoder'*.

Skrip umožňuje zpracovat dva argumenty v pořadí vstupní soubor, výstupní soubor. Ve skutečnosti má každý skript ještě jeden argument a to cestu k souboru ze kterého byl spuštěn, ten vyplňuje automaticky Python.

```
if __name__ == '__main__':
    if len(sys.argv)==2:
        try:
            fin = open(sys.argv[1],'rb')
            sys.stdin = fin
        except:
            print 'Error: Soubor "{0}", nelze precist'.format(sys.argv[1])
            sys.exit(0)
    elif len(sys.argv)==3:
        try:
            fin = open(sys.argv[1],'rb')
            sys.stdin = fin
        except:
```

```
            print 'Error: Soubor "{0}", nelze precist'.format(sys.argv[1])
            sys.exit(0)
        try:
            fout=open(sys.argv[2],'a')
            sys.stdout=fout
        except:
            print 'Error: Spatny vystupni sounbor "{0}"'.format(sys.argv[2])
            sys.exit(0)

try:
    instanc=jpegDecoder(sys.stdin)
    result=instanc.decode()
except Exception,e:
    print 'Error: Chyba programu u souboru "{0}". {1}'.format(sys.stdin.name,e.args)
else:
    print str(result).translate(None,'[] ')

if sys.stdout!=sys.__stdout__:
    fout.close()
    sys.stdout=sys.__stdout__

if sys.stdin!=sys.__stdin__:
    fin.close()
    sys.stdin=sys.__stdin__
```

Pokud není zadán ani jeden argument, skript čte ze standardního vstupu, to je výhodné pro přesměrování standardního vstupu už v příkazové řádce. Výstup se chová úplně stejně, když nezadáme žádný nebo jenom jeden argument program vypisuje výsledek na standardní výstup. V případě zadání obou argumentů, výstup nebude přepisovat výstupní soubor, ale bude připisovat na konec.

Příklad spuštění skriptu s přesměrovaným vstupem a výstupem:

*python jpegDecoder.py < /home/uzivatel/obrazek.jpg >/home/uzivatel/vysledek.txt*

Příklad se zadaným vstupem:

*python jpegDecoder.py /home/uzivatel/obrazek.jpg*

Výsledek bude vypsaný v konzole příkazového řádku:

*0,3,1,0,0,0,0,0,0,0,0,0,0,0,0,0,2,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0 ,18,6,4,16,3,1,0,0,0,0,0,0,0,0,0*

Výstupem je histogram Huffmanova kódování řazený za sebou v pořadí tabulek DC0, DC1, AC0, AC1.

# ZÁVĚR

Cílem této diplomové práce bylo vytvořit multiplatformní JPEG dekodér, který bude jako výstup vracet histogram Huffmanova kódování, sloužící pro účely stegoanalýzy pomocí neuronových sítí.

V teoretické části byla nejprve popsána steganografie, jak z hlediska historického, tak i moderní digitální steganografie. V rámci digitální steganografie byla věnována pozornost hlavně obrázkům jako nosiči skrytých dat a metodě nejméně významného bitu pro jejich vkládání. Další kapitola se zabývala standardem JPEG. Nejprve obecnému popisu standardu, dále byl popsán princip ztrátové komprese v sekvenčním módu. V poslední kapitola teoretické části byla zaměřena na uložení zakódovaných dat do souboru. Podrobně byl popsán význam a obsah jednotlivých značek, které se mohou vyskytovat v sekvenčním módu.

V praktické části bylo popsáno detailně dekódování ukázkového obrázku z kapitoly 3.1. V druhé části je popsán modul dekodéru, který byl napsán v programovacím jazyku Python verze 2.6, s použitím optimalizačního modulu Psyco. Celý dekodér se skládá z jedné třídy jpegDecoder, která má dvě veřejné funkce, konstruktor __init__ a funkci decode, která zahájí dekódovací proces a jako návratovou hodnotu vrací histogram Huffmanova kódování.

Největší problém při tvorbě dekodéru v jazyce Python, byla jeho rychlost. Po analýze celého kódu a úpravy kritických části dekodéru se podařilo zvýšit jeho výkon i když byl stále nedostatečný oproti očekáváním. Opravdu zásadní zrychlení přineslo, až použití Python modulu Psyco, který zrychlil vykonávání kódu až 4x.

Dekodér byl úspěšně otestován na platformách Windows, Linux a Mac OS X. U všech testovacích obrázků program vracel očekávané hodnoty histogramu Huffmanova kódování.

# ZÁVĚR V ANGLIČTINĚ

The aim of master thesis was to create cross-platform JPEG decoder, which will return histogram of Huffman code. This histogram will be used for purpose of steganalysis by neural network.

In theoretical part, steganography was described both historical and modern digital view. Digital view was aimed at pictures as a carrier of hidden data and less-important bit method for its inserting. Next chapter deals with JPEG standard. General account of this standard was described and then method of lossy compression in the sequential mode. The last chapter of theoretical part was aimed at export of encoded data to file. Relevance and content of markers, which can occur in sequential mode, were described in detail.

In practical part, decoding od picture from chapter 3.1 was described. The second part deals with decoder module, which was programmed in Python 2.6 language, including Psyco module. Whole decoder consists one jpegDecode class, which has two public functions, __init__ constructor and decode function, which starts decoding process and returns histogram of Huffman code.

During the decoder programming, the biggest problem was the decoding process rate. After analysis of the whole code and modification of critical parts decoding rate was increased, but short of expectations. Radical increase of decoding rate was achieved by using Python module Psyco, which increased speed by 4 times.

Decoder was succesfully tested on Windows, Linux and Mac OS X. Application returned expected values of Huffman code histogram.

# SEZNAM POUŽITÉ LITERATURY

[1] KAJÍNEK, Milan. Tajemství šifer - po stopách kryptografie a steganografie [online]. 2008 , 12.6.2008 [cit. 2010-5-26]. Dostupný z WWW: <http://www.velkaepocha.sk/200806125316/Tajemstvi-sifer-po-stopach-kryptografie-a-steganografie.html>.

[2] Wikipedia : Steganography [online]. 2009 , 18.5.2010 [cit. 2010-5-26]. Anglicky. Dostupný z WWW: <http://en.wikipedia.org/wiki/Steganography>.

[3] Johannes Trithemius [online]. 2010 [cit. 2010-05-26]. Dostupný z WWW: <http://cs.wikipedia.org/wiki/Johannes_Trithemius>.

[4] Owens, M. A Discussion of Covert Channels and Steganography. [paper] SANS Institute, 2002. URL <http://www.gray-world.net/papers/adiscussionofcc.pdf>

[5] Wikipedia Least significant bit [online]. 2009 [cit. 2009-11-30]. Anglicky. Dostupný z WWW: <http://en.wikipedia.org/wiki/Least_significant_bit>.

[6] Crypto-World. Mgr. Pavel Vondruška. 1999- , roč. devátý, č. 3/2007- . Dostupný z WWW: <http://crypto-world.info/>. ISSN ISSN 1801-214

[7] MIANO, John. Compressed image file formats : JPEG, PNG, GIF, XBM, BMP. 1st edition. [s.l.] : Addison Wesley Longman, Inc., 1999. 264 s. ISBN 0-201-60443-4.

[8] ITU-T Rec. T.81. INFORMATION TECHNOLOGY DIGITAL COMPRESSION AND CODING OF CONTINUOUS-TONE STILL IMAGES REQUIREMENTS AND GUIDELINES . [s.l.] : INTERNATIONAL TELECOMMUNICATION UNION, září 1992. 186 s. Dostupné z WWW: <www.w3.org/Graphics/JPEG/itu-t81.pdf>.

[9] TIŠNOVSKÝ, Pavel. JPEG : král rastrových grafických formátů? [online]. 2006 [cit. 2010-05-28]. Dostupný z WWW: <http://www.root.cz/clanky/jpeg-kral-rastrovych-grafickych-formatu/>.

[10] JPEG. In Wikipedia : the free encyclopedia [online]. St. Petersburg (Florida) : Wikipedia Foundation, 1.9.2001, last modified on 24.5.2010 [cit. 2010-25-05]. Dostupné z WWW: <http://en.wikipedia.org/wiki/JPEG#Entropy_coding>.

[11] Gimliho stránky [online]. 2006 [cit. 2010-05-28]. Komprese a kompresní algoritmy. Dostupné z WWW: <http://gimli.mysteria.cz/komprese/algoritmy.html>.

[12] JPEG File Interchange Format. Version 1.02. [s.l.] : C-Cube Microsystems, 1.9.1992. 9 s. Dostupné z WWW: <http://www.w3.org/Graphics/JPEG/jfif3.pdf>.

[13] Python Programming Language [online]. c1990-2010 [cit. 2010-05-28]. Dostupné z WWW: <http://www.python.org/>.

[14] RIGO, Armin. Theory of Psyco [online]. [s.l.] : [s.n.], 2006 [cit. 2010-06-04]. Dostupné z WWW: <psyco.sourceforge.net/theory_psyco.ps.gz>.

[15] Eclipse [online]. c2010 [cit. 2010-06-02]. Dostupné z WWW: <http://www.eclipse.org/>.

# SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

| | |
|---|---|
| JPEG | Joint Photographic Experts Group – Grafický rastrový formát |
| JFIF | JPEG File Interchange Format |
| TCP/IP | Transmission Control Protocol/ Internet Protocol - Síťové protokoly |
| IM | Instant Messaging |
| LSB | Least significant bit – Nejméně významný bit |
| BMP | Bitmap - Grafický rastrový formát |
| WAV | Waveform Audio File Format – Audio formát |
| AU | Audio file format – Audio formát |
| PNG | Portable Network Graphics - Grafický rastrový formát |
| JIF | JPEG Interchange Format |
| YCbCr | Barevný model YCbCr |
| RGB | Barevny mode RGB |
| DCT | Discrete Cosine Transformation – Diskrétní Kosinova transformace |
| MCU | Minimum coded unit – Minimální kódovací jednotka |
| DC | Direct Coefficient - Stejnosměrdný koeficient |
| AC | Alternately Coefficients - Střídavé koeficienty |
| SOI | Start Of Image – Začátek obrázku |
| EOI | End Of Image – Konec Obrázku |
| RSTi | Restart Marker i – Restartovací značka |
| SOFn | Start Of Frame - Specifikace tvaru obrazových dat |
| SOS | Start Of Scan - Začátek kódovaných obrazových dat |
| APPn | Application Marker - Značka specifických dat |
| COM | Comment - Komentář |
| DRI | Define Restart Interval – Značka intervalu vyskytujících restretujících značek |
| DQT | Define Quantization Table - Definice tabulky kvantizačních koeficientů |
| DHT | Define Huffman Table - Definice Huffmanových tabulek |
| IDE | Integrated Development Environment - Integrované vývojové prostředí |

# SEZNAM OBRÁZKŮ

# SEZNAM TABULEK

# SEZNAM PŘÍLOH

# PŘÍLOHA P I: NÁZEV PŘÍLOHY

Přiložené CD obsahuje tuto práci v elektronické podobě (formát PDF) a zdrojové kódy dekodéru.

# PŘÍLOHA P II: DEKÓDOVANÉ HUFFMANOVY TABULKY

**Huffmanova tabulka:**

**Trida: DC**

**Identifikator: 0**

**delka kod**

==========

1 bitu:

2 bitu:

3 bitu: 03 02 05 04 00 06 01

4 bitu: 07

5 bitu: 08

6 bitu: 09

7 bitu: 0a

8 bitu: 0b

9 bitu:

10 bitu:

11 bitu:

12 bitu:

13 bitu:

14 bitu:

15 bitu:

16 bitu:

Expandovna DHT tabulka

==== 1bitove:====

==== 2bitove:====

==== 3 itove klice:====

000 : 04

001 : 05

010 : 03

011 : 02

100 : 06

101 : 01

110 : 00

==== 4bitove:====

1110 : 07

==== 5 bitove:====

11110 : 08

==== 6 bitove:====

111110 : 09

==== 7 bitove:====

1111110 : 0a

==== 8 bitove:====

11111110 : 0b

==== 9 bitove:====

====10 bitove:====

====11 bitove:====

====12 bitove:====

====13 bitove:====

====14 bitove:====

====15 bitove:====

====16 bitove:====

**Huffmanova tabulka:**

**Trida: DC**

**Identifikator: 1**

**delka kod**

==========

1 bitu:

2 bitu: 01 00

3 bitu: 02 03

4 bitu: 06 04 05

5 bitu: 07

6 bitu: 08

7 bitu: 09

8 bitu: 0a

9 bitu: 0b

10 bitu:

11 bitu:

12 bitu:

13 bitu:

14 bitu:

15 bitu:

16 bitu:

Expandovna DHT tabulka

==== 1 bitove:====

==== 2 bitove:====

00 : 01

01 : 00

==== 3 bitove:====

100 : 02

101 : 03

==== 4 bitove:====

1100 : 04

1101 : 05

1110 : 06

==== 5 bitove:====

11110 : 07

==== 6 bitove:====

111110 : 08

==== 7 bitove:====

1111110 : 09

==== 8 bitove:====

11111110 : 0a

==== 9 bitove:====

111111110 : 0b

====10 bitove:====

====11 bitove:====

====12 bitove:====

====13 bitove:====

====14 bitove:====

====15 bitove:====

====16 bitove:====

**Huffmanova tabulka:**

**Trida: AC**

**Identifikator: 0**

**delka kod**

==========

1 bitu:

2 bitu: 01 02

3 bitu: 03

4 bitu: 11 04 00

5 bitu: 12 05 21

```
 6 bitu:  31 41
 7 bitu:  13 61 06 51
 8 bitu:  71 22
 9 bitu:  07 a1 14 81 32 91
10 bitu:  d1 b1 15 42 23 52 c1
11 bitu:  16 e1 33
12 bitu:  72 24 f0 62
13 bitu:  f1 82
14 bitu:  43 25 34 53 a2 92
15 bitu:  63 b2
16 bitu:  58 68 45 94 28 1a f5 66 4a 3a 27 93 f4 65 f8 29 88 78 ba ca e3 f2 e5 d5 b3 a3 85 75 e8 d8 aa 9a d2 e2 0a 18 c7 d7 47 57 c3 74 35 44 84 19 b7 a7 95 a5 d9 e9 39 49 37 f6 76 86 c2 73 c5 b5 c9 b9 7a 8a 69 59 da ea fa 08 26 b8 c8 a6 96 e7 f7 d3 55 6a 5a 54 64 d4 e4 c4 f3 09 83 a8 98 48 38 79 89 56 b4 d6 e6 17 36 67 77 46 a4 a9 99 f9 2a c6 b6 97 87

Expandovna DHT tabulka

==== 1 bitove:====

==== 2 bitove:====

00 : 01

01 : 02

==== 3 bitove:====

100 : 03

==== 4 bitove:====

1010 : 11

1011 : 04

1100 : 00

==== 5 bitove:====

11010 : 05

11011 : 21

11100 : 12

==== 6 bitove:====

111010 : 31

111011 : 41

==== 7 bitove:====

1111000 : 51

1111001 : 06

1111010 : 13

1111011 : 61

==== 8 bitove:====

11111000 : 22

11111001 : 71

==== 9 bitove:====

111110100 : 81

111110101 : 14

111110110 : 32

111110111 : 91

111111000 : a1

111111001 : 07

====10 bitove:====

1111110100 : 15

1111110101 : b1

1111110110 : 42

1111110111 : 23

1111111000 : c1

1111111001 : 52

1111111010 : d1

====11 bitove:====

11111110110 : e1

11111110111 : 33

11111111000 : 16

====12 bitove:====

111111110010 : 62

111111110011 : f0

111111110100 : 24

111111110101 : 72

====13 bitove:====

1111111101100 : 82

1111111101101 : f1

====14 bitove:====

11111111011100 : 25

11111111011101 : 43

11111111011110 : 34

11111111011111 : 53

11111111100000 : 92

11111111100001 : a2

====15 bitove:====

111111111000100 : b2

111111111000101 : 63

====16 bitove:====

1111111110001100 : 73

1111111110001101 : c2

1111111110001110 : 35

1111111110001111 : 44

1111111110010000 : 27

1111111110010001 : 93

1111111110010010 : a3

1111111110010011 : b3

1111111110010100 : 36

1111111110010101 : 17

1111111110010110 : 54

1111111110010111 : 64

1111111110011000 : 74

1111111110011001 : c3

1111111110011010 : d2

1111111110011011 : e2

1111111110011100 : 08

1111111110011101 : 26

1111111110011110 : 83

1111111110011111 : 09

1111111110100000 : 0a

1111111110100001 : 18

1111111110100010 : 19

1111111110100011 : 84

1111111110100100 : 94

1111111110100101 : 45

1111111110100110 : 46

1111111110100111 : a4

1111111110101000 : b4

1111111110101001 : 56

1111111110101010 : d3
```

1111111110101011 : 55
1111111110101100 : 28
1111111110101101 : 1a
1111111110101110 : f2
1111111110101111 : e3
1111111110110000 : f3
1111111110110001 : c4
1111111110110010 : d4
1111111110110011 : e4
1111111110110100 : f4
1111111110110101 : 65
1111111110110110 : 75
1111111110110111 : 85
1111111110111000 : 95
1111111110111001 : a5
1111111110111010 : b5
1111111110111011 : c5
1111111110111100 : d5
1111111110111101 : e5
1111111110111110 : f5
1111111110111111 : 66
1111111111000000 : 76
1111111111000001 : 86
1111111111000010 : 96
1111111111000011 : a6
1111111111000100 : b6
1111111111000101 : c6
1111111111000110 : d6
1111111111000111 : e6
1111111111001000 : f6
1111111111001001 : 37
1111111111001010 : 47
1111111111001011 : 57
1111111111001100 : 67
1111111111001101 : 77
1111111111001110 : 87
1111111111001111 : 97
1111111111010000 : a7
1111111111010001 : b7
1111111111010010 : c7
1111111111010011 : d7
1111111111010100 : e7
1111111111010101 : f7
1111111111010110 : 38
1111111111010111 : 48
1111111111011000 : 58
1111111111011001 : 68
1111111111011010 : 78
1111111111011011 : 88
1111111111011100 : 98
1111111111011101 : a8
1111111111011110 : b8
1111111111011111 : c8
1111111111100000 : d8
1111111111100001 : e8
1111111111100010 : f8
1111111111100011 : 29
1111111111100100 : 39
1111111111100101 : 49
1111111111100110 : 59
1111111111100111 : 69
1111111111101000 : 79
1111111111101001 : 89
1111111111101010 : 99
1111111111101011 : a9
1111111111101100 : b9
1111111111101101 : c9
1111111111101110 : d9
1111111111101111 : e9
1111111111110000 : f9
1111111111110001 : 2a
1111111111110010 : 3a
1111111111110011 : 4a
1111111111110100 : 5a
1111111111110101 : 6a
1111111111110110 : 7a
1111111111110111 : 8a
1111111111111000 : 9a
1111111111111001 : aa
1111111111111010 : ba
1111111111111011 : ca
1111111111111100 : da
1111111111111101 : ea
1111111111111110 : fa

**Huffmanova tabulka:**

**Trida: AC**

**Identifikator: 1**

**delka kod**

==========

1 bitu:

2 bitu: 01 00

3 bitu: 02 11

4 bitu: 03

5 bitu: 04 21

6 bitu: 31 12 41

7 bitu: 22 61 13 05 51

8 bitu: 06 71 32 91 81

9 bitu: 14 f0 a1 b1

10 bitu: 42 d1 c1 e1 23

11 bitu: 52 15 72 62 f1 33

12 bitu: 24 34 82 43

13 bitu: 25 53 16 92 b2 c2 63 a2

14 bitu: 73 07 d2

15 bitu: 44 35 e2

16 bitu: 48 58 a3 f2 f3 84 f5 46 4a 3a f4 65 e8 f8 78 68 ba ca a4 94 e5 d5 17 83 85 75 d8 c8 aa 9a 19 26 64 74 b7 c7 f6 47 18 0a 37 55 a7 97 95 a5 d9 e9 39 49 e6 d6 56 66 c5 b5 c9 b9 7a 8a 69 59 da ea fa 36 45 a8 b8 86 76 d7 e7 d3 e3 6a 5a 08 09 d4 e4 c4 b4 27 1a 98 88 38 f7 79 89 29 28 b6 c6 93 54 57 67 b3 c3 a9 99 f9 2a a6 96 87 77

Expandovna DHT tabulka

==== 1 bitove:====
==== 2 bitove:====
00 : 01
01 : 00
==== 3 bitove:====
100 : 02
101 : 11
==== 4 bitove:====
1100 : 03
==== 5 bitove:====
11010 : 04
11011 : 21
==== 6 bitove:====
111000 : 12
111001 : 31
111010 : 41
==== 7 bitove:====
1110110 : 05
1110111 : 51
1111000 : 13
1111001 : 61
1111010 : 22
==== 8 bitove:====
11110110 : 06
11110111 : 71
11111000 : 81
11111001 : 91
11111010 : 32
==== 9 bitove:====
111110110 : a1
111110111 : b1
111111000 : f0
111111001 : 14
====10 bitove:====
1111110100 : c1
1111110101 : d1
1111110110 : e1
1111110111 : 23
1111111000 : 42

====11 bitove:====
11111110010 : 15
11111110011 : 52
11111110100 : 62
11111110101 : 72
11111110110 : f1
11111110111 : 33
====12 bitove:====
111111110000 : 24
111111110001 : 34
111111110010 : 43
111111110011 : 82
====13 bitove:====
1111111101000 : 16
1111111101001 : 92
1111111101010 : 53
1111111101011 : 25
1111111101100 : a2
1111111101101 : 63
1111111101110 : b2
1111111101111 : c2
====14 bitove:====
11111111100000 : 07
11111111100001 : 73
11111111100010 : d2
====15 bitove:====
111111111000110 : 35
111111111000111 : e2
111111111001000 : 44
====16 bitove:====
1111111110010010 : 83
1111111110010011 : 17
1111111110010100 : 54
1111111110010101 : 93
1111111110010110 : 08
1111111110010111 : 09
1111111110011000 : 0a
1111111110011001 : 18
1111111110011010 : 19

1111111110011011 : 26
1111111110011100 : 36
1111111110011101 : 45
1111111110011110 : 1a
1111111110011111 : 27
1111111110100000 : 64
1111111110100001 : 74
1111111110100010 : 55
1111111110100011 : 37
1111111110100100 : f2
1111111110100101 : a3
1111111110100110 : b3
1111111110100111 : c3
1111111110101000 : 28
1111111110101001 : 29
1111111110101010 : d3
1111111110101011 : e3
1111111110101100 : f3
1111111110101101 : 84
1111111110101110 : 94
1111111110101111 : a4
1111111110110000 : b4
1111111110110001 : c4
1111111110110010 : d4
1111111110110011 : e4
1111111110110100 : f4
1111111110110101 : 65
1111111110110110 : 75
1111111110110111 : 85
1111111110111000 : 95
1111111110111001 : a5
1111111110111010 : b5
1111111110111011 : c5
1111111110111100 : d5
1111111110111101 : e5
1111111110111110 : f5
1111111110111111 : 46
1111111111000000 : 56
1111111111000001 : 66

1111111111000010 : 76
1111111111000011 : 86
1111111111000100 : 96
1111111111000101 : a6
1111111111000110 : b6
1111111111000111 : c6
1111111111001000 : d6
1111111111001001 : e6
1111111111001010 : f6
1111111111001011 : 47
1111111111001100 : 57
1111111111001101 : 67
1111111111001110 : 77
1111111111001111 : 87
1111111111010000 : 97
1111111111010001 : a7
1111111111010010 : b7
1111111111010011 : c7
1111111111010100 : d7
1111111111010101 : e7
1111111111010110 : f7
1111111111010111 : 38
1111111111011000 : 48
1111111111011001 : 58
1111111111011010 : 68
1111111111011011 : 78
1111111111011100 : 88
1111111111011101 : 98
1111111111011110 : a8
1111111111011111 : b8
1111111111100000 : c8
1111111111100001 : d8
1111111111100010 : e8
1111111111100011 : f8
1111111111100100 : 39
1111111111100101 : 49
1111111111100110 : 59
1111111111100111 : 69
1111111111101000 : 79
1111111111101001 : 89
1111111111101010 : 99
1111111111101011 : a9
1111111111101100 : b9
1111111111101101 : c9
1111111111101110 : d9
1111111111101111 : e9
1111111111110000 : f9
1111111111110001 : 2a
1111111111110010 : 3a
1111111111110011 : 4a
1111111111110100 : 5a
1111111111110101 : 6a
1111111111110110 : 7a
1111111111110111 : 8a
1111111111111000 : 9a
1111111111111001 : aa
1111111111111010 : ba
1111111111111011 : ca
1111111111111100 : da
1111111111111101 : ea
1111111111111110 : fa