KAUNO TECHNOLOGIJOS UNIVERSITETAS

FUNDAMENTALIŲJŲ MOKSLŲ FAKULTETAS

TAIKOMOSIOS MATEMATIKOS KATEDRA

Kristina Kupčiūnienė

DISKREČIŲJŲ BANGELIŲ TRANSFORMACIJOS TAIKYMO EFEKTYVUMO TYRIMAS SKAITMENINIŲ VAIZDŲ KODAVIME

Magistro darbas

Vadovas
prof. dr. J. Valantinas

KAUNAS, 2007

KAUNO TECHNOLOGIJOS UNIVERSITETAS

FUNDAMENTALIŲJŲ MOKSLŲ FAKULTETAS

TAIKOMOSIOS MATEMATIKOS KATEDRA

TVIRTINU
Katedros vedėjas
prof. dr. J.Rimas
2007 06 06

DISKREČIŲJŲ BANGELIŲ TRANSFORMACIJOS TAIKYMO EFEKTYVUMO TYRIMAS SKAITMENINIŲ VAIZDŲ KODAVIME

Taikomosios matematikos magistro baigiamasis darbas

Vadovas
( ) prof. dr. J. Valantinas
2007 06 03

Recenzentas
( ) doc. dr. P. Kanapeckas
2007 06 01

Atliko
FMMM 5 gr. stud.
( ) K. Kupčiūnienė
2007 05 25

KAUNAS, 2007

KVALIFIKACINĖ KOMISIJA

Pirmininkas: Leonas Saulis, habil. dr.,Vilniaus Gedimino technikos universiteto profesorius

Sekretorius: Eimutis Valakevičius, docentas

Nariai: Algimantas Jonas Aksomaitis, profesorius (KTU)

Arūnas Barauskas, dr.,UAB „Elsis“ generalinio direktoriaus pavaduotojas

Vytautas Janilionis, docentas (KTU)

Zenonas Navickas, profesorius (KTU)

Vidmantas Povilas Pekarskas, profesorius (KTU)

Rimantas Rudzkis, habil.dr., banko „NORD/LB“ vyriausiasis analitikas

Kupčiūnienė K. Analysis of discrete wavelet transform effectiveness for digital image coding: Master's work in applied mathematics / supervisor dr. assoc. prof. J. Valantinas; Department of Applied mathematics, Faculty of Fundamental Sciences, Kaunas University of Technology. – Kaunas, 2007. – 39 p.

SUMMARY

Digital images are widely used in computer applications. Uncompressed digital images require considerable storage capacity and transmission bandwidth. Efficient image compression solutions are becoming more critical with the recent growth of data intensive, multimedia-based applications.

This paper discusses specific properties of the discrete orthogonal wavelet transform, namely: relationships between the wavelet coefficients and image fragments, existence of zerotrees in the discrete wavelet spectrum of an image, etc. In parallels, the progressive image encoding idea and an efficient image encoding/decoding algorithm EZW (Embedded-Zerotree-Wavelet), based on the discrete wavelet (Haar, Daubechies) transform, are presented. Some preliminary experimental analysis results are discussed.

The EZW coder is one of the more efficient coding methods for wavelet coefficients. Multiple wavelet decompositions can be applied to the image as many times as possible to gain image compression improvements. Although, there is a limitation to multiple decompositions due to the fact that natural images do not have smooth color variations – fine details are often represented as sharp edges (high frequency variations) in between smooth (low frequency) variations. On a side note, EZW seems to be well suited for discrete signal noise reducing – it is possible to achieve noise removal by applying loss technique (removing too high and too low frequency coefficients). This is difficult to automate though – it's more reliable to apply this technique on visual criteria, as it is very specific to the type of signal/image data being processed.

TURINYS

KVALIFIKACINĖ KOMISIJA 2
ĮVADAS 7
1. BENDROJI DALIS 8
1.1. Trumpa efektyvaus vaizdų kodavimo metodų ir požiūrių apžvalga 8
1.1.1 Vaizdų kodavimo duomenų srityje ypatumai 8
1.1.2 Vaizdų kodavimas spektrinėje srityje 12
1.2. Diskrečiosios bangelių transformacijos – naujas efektyvus vaizdų kodavimo įrankis 16
2. TIRIAMOJI DALIS 23
2.1. Progresyviojo vaizdų kodavimo metodo (EZW) algoritmizavimas, realizacija ir tyrimas 23
2.1.1 Struktūrinė progresyviojo vaizdų kodavimo schema 23
2.1.2 Atskirų vaizdo kodavimo, taikant diskrečiąją bangelių transformaciją, etapų organizavimas ir algoritmavimas 23
2.2. Palyginamoji eksperimento rezultatų analizė 26
2.2.1 Vaizdų suspaudimas taikant skirtingas bangelių transformacijas 26
2.2.2 Rezultatų įvertinimas 27
3. PROGRAMINĖ REALIZACIJA IR INSTRUKCIJA VARTOTOJUI 31
DISKUSIJA 37
IŠVADOS 38
LITERATŪRA 39
1 PRIEDAS. PROGRAMOS TEKSTAS 40

Lentelių sąrašas

1.1 lentelė Daubechies žemo dažnio filtro koeficientai.. 20

1.2 lentelė Bangelių savybės.. 20

2.1 lentelė Suspaudimo rezultatų palyginimas naudojant skirtingus vaizdus...................................... 28

2.2 lentelė Suspaudimo rezultatų palyginimas, vaizdų atkūrimui naudojant skirtingus iteracijų kiekius 28

2.3 lentelė Suspaudimo rezultatų palyginimas, naudojant skirtingas bangelių transformacijas............ 30

Paveikslų sąrašas

1.1 pav. JPEG algoritmas – nuoseklus skaitmeninio vaizdo apdorojimas 12
1.2 pav. Pradinio vaizdo nuskaitymas ir suskaidymas 8x8 blokeliais... 13
1.3 pav. Zigzaginė 8x8 blokelio nuskaitymo tvarka 14
1.4 pav. Diskrečiojo bangelių spektro suskirstymas poaibiais 16
1.5 pav. Haaro mastelio funkcija (kairėje) ir motininė bangelė (dešinėje)... 19
1.6 pav. Daubechies *db*2 mastelio funkcija (kairėje) ir motininė bangelė (dešinėje) 20
1.7 pav. Skaitmeninis vaizdas $[X(m_1, m_2)]$, kai $N = 16$ 21
1.8 pav. a) HT spektras $[Y(k_1, k_2)]$; b) sąryšis tarp bangelių koeficientų skirtinguose lygiuose (medis), kai $N = 16$ 22
2.1 pav. Skaitmeninių vaizdų EZW kodavimo blokinė schema 24
2.2 pav. Masyvo nuskaitymo tvarka – Morton seka 25
2.3 pav. EZW algoritmo pagrindinės iteracijos schema.. 26
2.4 pav. Vaizdo suspaudimas taikant Haaro transformaciją ir EZW kodavimo algoritmą: a) originalus vaizdas; b) atkurtas vaizdas, gautas naudojant 90% iteracijų, MSE=0.30, PSNR=53.37; c) naudojant 70% iteracijų, MSE=19.83, PSNR=35.16; d) naudojant 50% iteracijų, MSE=210.37, PSNR=24.90 .. 27
2.5 pav. Pilkos šviesos intensyvumo skalė: a) originalus vaizdas; b) atkurtas vaizdas gautas naudojant 0 iteracijų; c) naudojant 1 iteraciją; d) naudojant 2 iteracijas; e) naudojant 3 iteracijas; f) naudojant 4 iteracijas; g) naudojant 5 iteracijas; h) naudojant 6 iteracijas; i) naudojant 7 iteracijas; j) naudojant 8 iteracijas – pilnai atkurtas vaizdas... 29
3.1 pav. Programos langas 31
3.2 pav. Pradinių parametrų parinkimas.. 32
3.3 pav. Pranešimas apie klaidą 33
3.4 pav. Statistiniai duomenys 34
3.5 pav. Transformacijos duomenys.. 35
3.6 pav. Originalus ir atkurtas vaizdas 36

ĮVADAS

Vaizdai yra viena pagrindinių priemonių grafinei informacijai perteikti. Kuo daugiau grafinės informacijos bendrame informacijos (duomenų) sraute, tuo lengviau tuos duomenis įsisavinti, suvokti. Šiuolaikiniame informaciniame pasaulyje svarbu ne tiktai perduodamos informacijos kokybė, bet ir perdavimo greičiai bei apimtys. Siekiant sumažinti perduodamos grafinės informacijos (vaizdų) apimtis, siūlomos įvairios vaizdų kodavimo (glaudinimo) technologijos. Bendrame informacinių technologijų, skirtų glaudinti skaitmeniniams vaizdams, pakete vis daugiau dėmesio skiriama progresyviojo skaitmeninių vaizdų kodavimo idėjos įgyvendinimui bei tobulinimui. Progresyviojo vaizdų kodavimo idėjos esmė – nedideli papildomi kiekiai informacijos leidžia atskleisti naujas detales perduodamuose vaizduose, t.y. galima nuosekliai gerinti atkurto vaizdo kokybę. Beje, ši idėja neatsiejama nuo diskrečiųjų bangelių transformacijų.

Šiame darbe pateikiama trumpa efektyvaus vaizdų kodavimo metodų ir požiūrių apžvalga. Susipažįstama su diskrečiosiomis (Haaro, Daubechies) bangelių transformacijomis bei įvertinamos jų specifinės savybės: diskretaus bangelių spektro koeficiento sąsaja su vaizdo fragmentu, nulinių medžių egzistavimas spektre.

Darbo tikslas – įvertinus diskrečiųjų bangelių transformacijos savybes, algoritmizuoti, realizuoti ir ištirti vieną perspektyviausių progresyviojo skaitmeninių vaizdų kodavimo algoritmų EZW (Embedded-Zerotree-Wavelet).

Darbas pristatytas konferencijoje „Matematika ir matematikos dėstymas – 2007“.

1. BENDROJI DALIS

1.1. Trumpa efektyvaus vaizdų kodavimo metodų ir požiūrių apžvalga

Efektyviam skaitmeninių vaizdų kodavimui šiuo metu keliami nemaži reikalavimai – tai didelis vaizdo suglaudinimo (suspaudimo) laipsnis, priimtini vaizdo kodavimo ir dekodavimo laikai bei gera po kodavimo atkurto vaizdo kokybė. Vaizdų suspaudimo būdai gali būti be informacijos praradimo ir su informacijos praradimu.

Vaizdų suglaudinimo be informacijos praradimo būdai naudojami tuomet, kai reikalinga itin aukšta vaizdo kokybė arba specifiniams vaizdams koduoti (grafikai, brėžiniai ir kt.). Realaus pasaulio vaizdams šie metodai leidžia pasiekti tik 2-3 kartų suspaudimo laipsnį, bet specifiniams vaizdams (atskiroms vaizdų klasėms) pasiekiamas kur kas geresnis rodiklis.

Vaizdų suglaudimo metodai, prarandantys dalį informacijos, remiasi pastebėjimais, jog žmogaus akis nėra jautri mažiems vaizdo pokyčiams. Kitaip tarus, kodavimo metu galima atmesti aukštas vaizdą sudarančias harmonikas. Šie metodai tinka realaus pasaulio vaizdų (pvz. nuotraukoms, paveikslams) efektyviam suspaudimui. Tačiau, kuo didesnis vaizdo suspaudimo laipsnis, tuo daugiau duomenų prarandama – prastėja atkurto vaizdo kokybė.

Skaitmeninių vaizdų suglaudinimas galimas tiek duomenų, tiek spektrinėje srityje. Antruoju atveju, apdorojamas vaizdas prieš kodavimą pervedamas į spektrinę sritį. Tam panaudojamos diskrečiosios transformacijos (kosinusinė (DKT), Volšo ir Adamaro (VAT), bangelių (BT) ir pan., [1]).

Žemiau trumpai apžvelgsime kai kuriuos plačiau praktikoje taikomus, efektyviam vaizdų kodavimui skirtus, metodus.

1.1.1 Vaizdų kodavimo duomenų srityje ypatumai

Realaus pasaulio vaizdui galima sukonstruoti norimo detalizacijos lygio matematinį modelį (skaitmeninį vaizdą). Realiai imant, vaizdo detalizacijos lygis negali būti begalinis. Tačiau, pasirinkus aukštą vaizdų detalizacijos lygį, duomenų srityje galima išgauti didelį suspaudimo efektą.

Vaizdų kodavimui (suglaudinimui) duomenų srityje naudojami įvairūs metodai bei procedūros. Jų tarpe:

- Baigtinių automatų (BA) teorija grįsti vaizdų kodavimo metodai;
- BTC (Block-Truncation-Coding) algoritmas ir jo modifikacijos;

- Fraktalinės technologijos;
- Kitos specializuotos procedūros.

Baigtinių automatų teorija grįstas vaizdų kodavimo metodas

Realaus pasaulio vaizdui $V \in \mathbf{R}$ ($\mathbf{R}$ – realaus pasaulio vaizdų aibė) galima sukonstruoti bet kurio jo detalizacijos lygio matematinį modelį $V_n = [V_n(\omega)]$. Vaizdo fragmentas $V_n(\omega)$ charakterizuojamas skaitine fiksuota šviesos intensyvumo reikšme iš aibės $\{0, 1, \ldots, 2^p - 1\}$ bei adresu $\omega = \omega_1\omega_2 \cdots \omega_n$, kur $\omega_i = \{0, 1, 2, 3\}$, $i = 0, 1, \ldots, n$ ($|\omega| = n$ – adreso ilgis). Tokių adresų aibės yra reguliarios ir jas galima nusakyti baigtiniais automatais (BA) arba reguliariosiomis išraiškomis.

Efektyvaus dvimačio vaizdo kodavimo, taikant BA teoriją arba reguliariąsias išraiškas, idėja yra tokia, [2]:

1. Vaizdui konstruojamas baigtinis automatas. Tai atliekama vaizdą skaidant į fragmentus ir fragmentus tapatinant su BA padėtimis (būsenomis). Perėjimas iš vienos padėties į kitą fiksuojamas tik tuo atveju, jeigu atitinkami vaizdo fragmentai yra panašūs. To paties detalizacijos lygio vaizdai $U_n = [U_n(\omega)]$ ir $V_n = [V_n(\omega)]$ yra panašūs jeigu vidutinė kvadratinė paklaida (VKP) yra mažesnė už maksimalią leistiną reikšmę (ε_0), t.y.

$$VKP = \left(\frac{1}{4^n} \sum_{\omega(|\omega|=n)} \left(U_n(\omega) - V_n(\omega) \right)^2 \right)^{1/2} \le \varepsilon_0 . \qquad \textbf{(1.1)}$$

2. Sukonstruotam BA užrašoma ekvivalenti reguliarioji išraiška.
3. Vaizdas atkuriamas, taikant procedūras, kurios „užpildo“ vaizdą pagal reguliariąją išraišką.

BTC algoritmas

Nespalvotiems skaitmeniniams vaizdams suspausti gali būti naudojamas BTC (Block-Truncation-Coding) algoritmas.

Skaitmeninis dvimatis vaizdas $[X(m_1, m_2)]$ aprašomas kaip pikselių aibė $x(m_1, m_2) \in \{0, 1, \ldots, 2^p - 1\}$, $m_1, m_2 = 0, 1, \ldots, N-1$. Tada vaizdas suskaidomas į $(l \times l)$ dydžio blokelius, kurie vėliau nuskaitomi nuoseklia tvarka. Paprastai, $l \in [4, 8]$.Tą pačią informaciją apie blokelį kaupia pirmieji pradiniai momentai $\nu_r = \frac{1}{m}\sum_{i=1}^{m} x_i^r$, kai $r = 1, 2, \ldots, m$ (patogumo sumetimais, blokelio taškai pažymėti $x_1, x_2, \ldots, x_m$, t.y. $m = l^2$).

BTC algoritmo idėja, [3]:

1. Vaizdas suskaidomas $(l \times l)$ blokeliais.

2. Blokelio taško reikšmės koduojamos dviem lygiais a ir b taip, kad būtų išsaugoti pirmieji du pradiniai momentai, t.y. $\nu_1 = \frac{1}{m}\sum_{i=1}^{m} x_i$ ir $\nu_2 = \frac{1}{m}\sum_{i=1}^{m} x_i^2$.

   Taškui $x_i \left(i \in \{1, 2, \ldots, m\}\right)$ priskiriama reikšmė a, jeigu $x_i \le x_{slenkstis}$ ir reikšmė b, jei $x_i > x_{slenkstis}$ (paprastai kvantavimo slenkstis $x_{slenkstis} = \nu_1$).

   Tarkime, kad q yra pikselių, kuriems buvo priskirta reikšmė b, skaičius. Tuomet $(m-q)$ pikseliams priskirta reikšmė a .Sudarome lygčių sistemą:

$$\begin{cases} \nu_1 = \frac{1}{m}\left((m-q)\cdot a + q\cdot b\right), \\ \nu_2 = \frac{1}{m}\left((m-q)\cdot a^2 + q\cdot b^2\right). \end{cases}$$

   Išsprendę, gauname

$$a = \nu_1 - \sigma\sqrt{\frac{m-q}{q}}, \; b = \nu_1 + \sigma\sqrt{\frac{m-q}{q}}, \quad \textbf{(1.2)}$$

   kai $\sigma = \sqrt{\nu_2 - \nu_1^2}$.

3. Konstruojama bitinė plokštuma (dydžio $(l \times l)$ masyvas) B, kurios elementams priskiriama reikšmė 0, kai atitinkami blokelio taškai koduojami reikšme a, ir 1, kai koduojami reikšme b.
4. Vaizdo fragmentui (blokeliui) saugoti imama informacija (ν_1, σ, B).

Tarkime, kad pradinio vaizdo taškui (pikseliui) koduoti buvo skirta p bitų. Tada vaizdo suspaudimo laipsnis β išreiškiamas formule:

$$\beta = \frac{mp}{m+2p}. \quad \textbf{(1.3)}$$

BTC algoritmo trūkumai:

- Mažas vaizdo suspaudimo efektas.
- Imant didesnius blokelius, pasireiškia blokinė atkurto vaizdo struktūra.

Galimos algoritmo modifikacijos:

- Kvantavimo slenksčio bei lygių parinkimas;
- Kvantavimo rezultatų kodavimas;
- Bitinės plokštumos problema (atsisakymas).

Fraktalinės vaizdų kodavimo technologijos

Fraktalinio vaizdų kodavimo idėja grindžiama panašių fragmentų apdorojamame vaizde nustatymu bei gerai žinomos (fraktalinėje geometrijoje) koliažo teoremos taikymu, [4]. Bazinė (praktiškai įgyvendinama) fraktalinio vaizdų kodavimo idėja pirmąkart buvo pristatyta 1992 m. A. Jacquin, [5]:

1. Apdorojamas vaizdas skaidomas į N vienodo dydžio nepersidengiančius blokelius (reikšmių sritis).
2. Kiekvienam blokeliui (8×8) tame pačiame vaizde ieškomas į šį panašus, du kartus didesnis fragmentas. Didesni fragmentai (blokeliai (16×16)) priklauso apibrėžimo sričiai. Jie nebūtinai dengia visą vaizdą, tačiau gali persidengti.
3. Panašių (reikšmių ir apibrėžimo sričių) blokelių atitikčiai fiksuoti naudojamos afiniosios transformacijos ω_i, kurios vėliau koduojamos skaičių trejetais, būtent:

$$\omega_i(x, y, z) = \begin{pmatrix} a_i & b_i & 0 \\ c_i & d_i & 0 \\ 0 & 0 & s_i \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} + \begin{pmatrix} e_i \\ f_i \\ o_i \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \end{pmatrix}, \qquad \textbf{(1.4)}$$

čia: s_i – keičia pikselio kontrastiškumą (jei $s_i = 0$, tai blokelis iš apibrėžimo srities atvaizduojamas į visiškai juodą blokelį reikšmių srityje; jei $s_i = 1$ – be pakeitimų; kai $0 < s_i < 1$ – vaizdas praranda kontrastiškumą; jei $s_i > 1$, tai fragmento kontrastiškumas didėja);

o_i – keičia pikselio šviesumą (teigiamas o_i didina šviesumą, neigiamas o_i – mažina).

4. Saugoma tokia informacija:

$$\omega_i \leftrightarrow \langle e_i, f_i, t_i, s_i, o_i \rangle \ (i = 1, 2, \ldots, N),$$

čia: e_i, f_i – koordinatės; t_i – transformacija.

Dekodavimas:

1. Pradinis atkuriamo vaizdo įvertis yra vidutinio intensyvumo pilkas vaizdas (fonas).
2. Vienos iteracijos metu apskaičiuojamos kiekvieno reikšmių srities blokelio pikselių reikšmės (panaudojus išsaugotus skaičių trejetus). Gautas tarpinis vaizdas naudojamas kaip pradinis vaizdas kitai iteracijai.
3. Po ketvirtos iteracijos blokelio (8×8) atkūrimo procesas užbaigiamas.

Pagrindinis fraktalinio vaizdų kodavimo trūkumas – ilgas kodavimo (panašių fragmentų vaizde paieška) etapas ir trumpas dekodavimo etapas.

Įvairių autorių mokslinėse publikacijose siūloma daug priemonių, kaip paspartinti kodavimo etapą. Plačiau apie tai nekalbėsime.

1.1.2 Vaizdų kodavimas spektrinėje srityje

Vaizdų kodavimo (suglaudinimo) spektrinėje srityje specifika ta, jog apdorojamas ne pats vaizdas, o jo diskretusis spektras, gautas panaudojus vieną ar kitą diskrečiąją transformaciją. Kitaip tarus, prieš kodavimą išryškinamas dažnuminis vaizdo turinys.

Aptarsime keletą spektrinėje srityje veikiančių vaizdų suglaudinimo metodų, būtent:

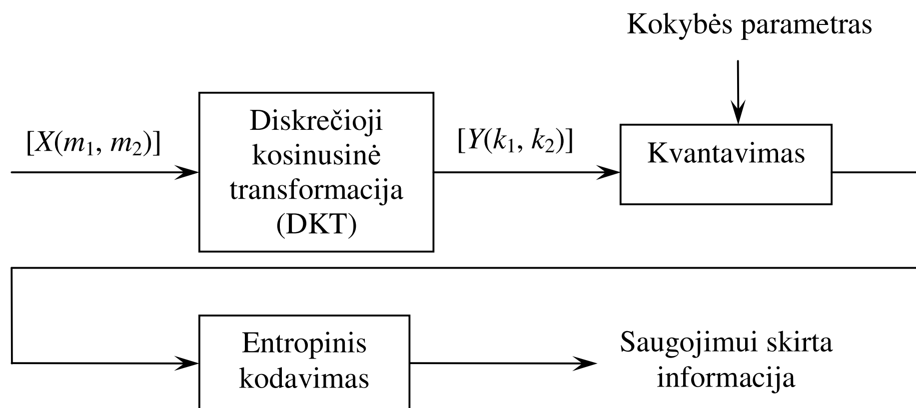
- JPEG, JPEG-2000;
- hiperbolinius filtrus (HF);
- progresyvųjį kodavimo algoritmą EZW.

JPEG, JPEG-2000

JPEG (*Joint Photographic Experts Group*) – fotografinių vaizdų išsaugojimo formatas (ir jų suspaudimo algoritmas). Nors dalis informacijos, esant didesniam suspaudimo laipsniui, prarandama, formato ypatybės tokios, jog šie praradimai yra sunkiai pastebimi.

JPEG standartas naudoja nuoseklų skenavimą (vienkartinį vaizdo nuskaitymą).

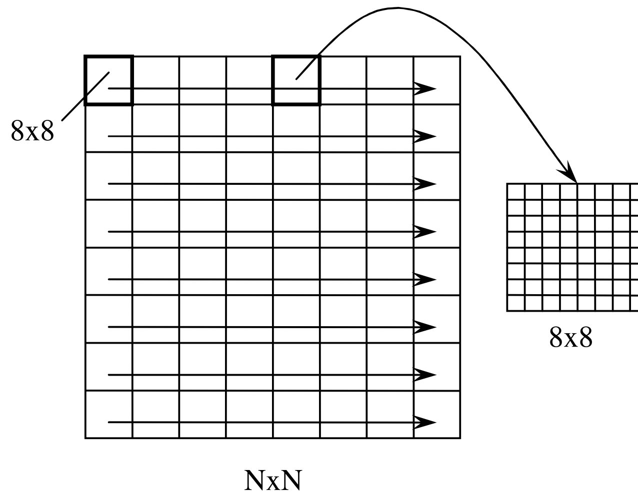
Tarkime, kad $[X(m_1, m_2)]$, $(m_1 = 0, 1, \ldots, N_1 - 1;\quad m_2 = 0, 1, \ldots, N_2 - 1)$ yra dvimatis skaitmeninis vaizdas, $[Y(k_1, k_2)]$, $(k_1 = 0, 1, \ldots, N_1 - 1;\quad k_2 = 0, 1, \ldots, N_2 - 1)$ yra spektras gautas pritaikius diskrečiąją kosinusinę transformaciją (DKT). Bendra vaizdo apdorojimo schema pateikta 1.1 paveikslėlyje.



1.1 pav. JPEG algoritmas – nuoseklus skaitmeninio vaizdo apdorojimas

JPEG šiuo metu yra vienas populiariausių dvimačių nespalvotų vaizdų suglaudinimo standartų, todėl šiek tiek praskleisime jo atskirų etapų specifiką.

1. Pradinis vaizdas $\left[X\left(m_1, m_2\right)\right]$ dalijamas į vienodo dydžio 8×8 blokelius.



1.2 pav. Pradinio vaizdo nuskaitymas ir suskaidymas 8x8 blokeliais

2. Kiekvienam blokeliui taikoma DKT:

$$Y\left(k_1, k_2\right) = \frac{1}{4} C_{k_1} C_{k_k} \sum_{m_2=0}^{7} \sum_{m_1=0}^{7} X\left(m_1, m_2\right) \cos \frac{\left(2m_1 + 1\right) k_1 \pi}{16} \cos \frac{\left(2m_2 + 1\right) k_2 \pi}{16}, \qquad \textbf{(1.5)}$$

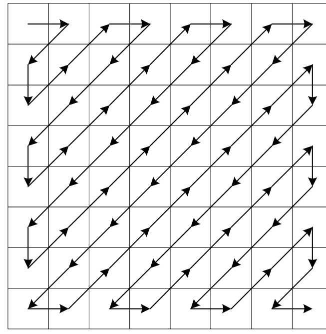
čia: $k_1, k_2 = 0, 1, \ldots, 7$; $C_0 = \frac{1}{\sqrt{2}}$, $C_i = 1$, $i = 1, 2, \ldots, 7$.

3. Gauti 64 DKT koeficientai kiekvienam blokeliui yra kvantuojami:

$$\tilde{Y}\left(k_1, k_2\right) = \text{round}\left(\frac{Y\left(k_1, k_2\right)}{q_{k_1, k_2}}\right), \qquad \textbf{(1.6)}$$

čia: $Q = \left\lfloor q_{k_1, k_2} \right\rfloor$ – kvantavimo matrica (nuo kvantavimo matricos parinkimo priklauso kiek bus prarandama informacijos).

4. Kvantuoti DKT spektro koeficientai nuskaitomi zigzagine tvarka (1.3 pav.). Kiekvienas nenulinis spektrinis koeficientas koduojamas prieš jį einančių nulinių spektrinių koeficientų skaičiumi ir paties nenulinio koeficiento reikšmė gaunama skaičių seka.



1.3 pav. Zigzaginė 8x8 blokelio nuskaitymo tvarka

5. Gautoji skaičių seka koduojama, taikant Hafmano (arba aritmetinį) kodavimą.
6. Išsaugoma informacija.

Atkuriant (dekoduojant) skaitmeninį vaizdą taikomi tie patys žingsniai atvirkščia tvarka su keliais pakeitimais:

1. Vietoj kvantavimo atliekamas dekvantavimas:

$$Y(k_1, k_2) = \widetilde{Y}(k_1, k_2) q_{k_1,k_2} . \qquad \textbf{(1.7)}$$

2. Vietoj diskrečiosios kosinusinės transformacijos taikoma atvirkštinė diskrečioji kosinusinė transformacija:

$$\widetilde{X}(m_1, m_2) = \frac{1}{4} \sum_{k_2=0}^{7} \sum_{k_1=0}^{7} C_{k_1} C_{k_k} \widetilde{Y}(k_1, k_2) \cos \frac{(2m_1 + 1)k_1 \pi}{16} \cos \frac{(2m_2 + 1)k_2 \pi}{16} , \qquad \textbf{(1.8)}$$

čia: $m_1, m_2 = 0, 1, \ldots, 7$.

JPEG užtikrina gana gerą vaizdo suspaudimo efektą (10-20 kartų) bei aukštą atkurto vaizdo kokybę. Metodo realizacija nesudėtinga ir greita.

Skaitmeninių vaizdų suspaudimo, taikant JPEG algoritmą, trūkumai:

- Netinka dvejetainiams (juodai-baltiems) vaizdams spausti.
- Didėjant suspaudimo efektui (>20), atkurtame vaizde išryškėja blokinė struktūra.
- Vaizdo negalima atkurti bet kuriuo norimu detalizacijos lygiu.

Tuo tarpu JPEG-2000 skaitmeninių vaizdų suspaudimui naudoja progresyvųjį kodavimą ir yra tobulesnis už JPEG. Šis vaizdų kodavimo algoritmas naudoja bangeles. JPEG-2000 užtikrina didelį vaizdo suspaudimo efektą su aukšta atkurto vaizdo kokybe, [6].

Hiperboliniai filtrai

Atliekant skaitmeninio vaizdo $[X(m_1, m_2)]$ hiperbolinį filtravimą, jo diskrečiojo spektro (DKT, VAT, HT) koeficientai $Y(k_1, k_2)$, kurių indeksai k_1 ir k_2 tenkina sąlygą $\bar{k}_1\bar{k}_2 > M$ (čia M – hiperbolinio filtro lygis) yra atmetami; $\bar{k}_i = \max\{1, k_i\}$, $i = 1, 2$. Spektro koeficientų keitimo nuliais procedūrą galima aprašyti taip:

$$\tilde{Y}(k_1, k_2) = Y(k_1, k_2)\hat{H}(M; k_1, k_2), \text{ su visais } k_1, k_2; \qquad \textbf{(1.9)}$$

čia: $\hat{H}(M; k_1, k_2) = \begin{cases} 1, \, kai \; \bar{k}_1\bar{k}_2 \le M, \\ 0, \, kai \; \bar{k}_1\bar{k}_2 > M, \end{cases}$ yra diskretusis $\hat{H}(M; k_1, k_2)$ – M – lygio hiperbolinis filtro spektras; $\bar{k}_1\bar{k}_2 = M$ – spektrinių koeficientų poaibius skirianti linija (hiperbolė).

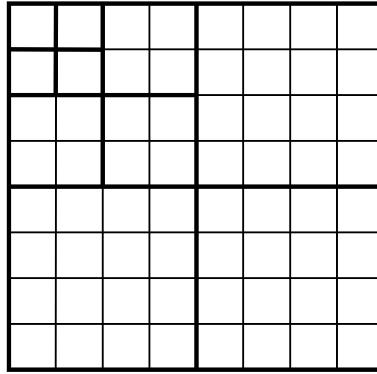
Atkuriant vaizdą, atmestieji koeficientai keičiami nuliais. Hiperbolinio filtro lygis M parenkamas, atsižvelgiant į norimą išgauti vaizdo suspaudimo efektą arba į norimą atkurto vaizdo kokybę.

Atmestieji koeficientai yra nykstamai maži, todėl jų nešama informacija apie vaizdą yra neesminė; be to, žmogaus akis nėra jautri pokyčiams aukštuose dažniuose.

Hiperbolinį filtravimą patogu naudoti, nes vaizdo kodavimui galima taikyti bet kurią diskrečiąją transformaciją (pavyzdžiui, DKT, VAT, HT). Svarbu tik, kad jos baziniai vektoriai būtų išdėstyti pagal dažnį. Hiperbolinio filtravimo efektyvumas tiesiogiai priklauso nuo apdorojamo vaizdo glodumo, t.y. kuo vaizdas glodesnis, tuo hiperbolinis filtras dirba efektyviau. Hiperbolinis vaizdų filtras leidžia suspausti vaizdą 5-210 kartų, kartu labai nepakenkiant vaizdo kokybei, [7].

Progresyviojo vaizdų kodavimo metodas EZW

EZW (Embedded-Zerotree-Wavelet) kodavimas – tai ganėtinai naujas, perspektyvus progresyvaus vaizdų kodavimo metodas. Šis kodavimas yra skirtas diskrečiosiomis bangelėmis transformuotiems skaitmeniniams vaizdams. Jis remiasi diskrečiųjų bangelių specifinėmis savybėmis: bangelių spektro koeficiento sąsaja su vaizdo fragmentu ir nulinių medžių egzistavimu spektre. Diskrečiosiomis bangelėmis transformuoti vaizdai pasižymi tuo, jog reikšmingiausi koeficientai sukoncentruoti žemuose dažniuose. Nuosekliam koeficientų nuskaitymui (skenavimui) diskretusis spektras yra suskirstomas į poaibius (1.4 pav.).



1.4 pav. Diskrečiojo bangelių spektro suskirstymas poaibiais

Toks koeficientų išsidėstymas yra labai patogus progresyviam kodavimui, nes mažiau reikšmingų koeficientų poaibiai tik padidina vaizdo tikslumą.

Kadangi progresyvusis vaizdų kodavimo metodas, jo algoritmizavimas ir realizacija yra šio darbo tikslas, tai nuosekliai pristatysime ne tiktai patį metodą, bet ir jo įgyvendinimui naudojamą matematinį aparatą (diskrečiąsias bangelių transformacijas, vaizdo skenavimo įrankius ir pan.)

1.2. Diskrečiosios bangelių transformacijos – naujas efektyvus vaizdų kodavimo įrankis

Bangelės – tai funkcijų klasė, lokalizuojanti duotą signalą tam tikroje erdvėje ar laike. Bangelių transformacija tinka nestacionariems, nedidelės trukmės signalams.

Dažniausiai, bangelės – tai funkcijos sugeneruotos vienos atskiros funkcijos $\psi(x)$, kuri yra vadinama motinine bangele, t.y.

$$\psi_{a,b}(x) = |a|^{-1/2}\psi\left(\frac{x-b}{a}\right), \qquad \textbf{(1.10)}$$

kur $\psi(x)$ tenkina $\int_{-\infty}^{\infty}\psi(x)dx = 0$.

Pagrindinė bangelių transformacijos idėja – signalo funkciją f užrašyti kaip integralą intervale [a, b] iš $\psi_{a,b}$, kai diskrečios reikšmės yra $a = a_0{}^m$, $b = nb_0a_0{}^m$, čia: $m, n \in \mathbf{Z}$, ir $a_0 > 1$, $b_0 > 0$ yra fiksuoti realieji skaičiai. Tada signalo funkcija bangelėmis suskaidoma taip:

$$f = \sum c_{m,n}\psi_{m,n}\,. \qquad \textbf{(1.11)}$$

Pasirinkus atitinkamą ψ ir $a_0 > 1$, $b_0 > 0$, funkcija $\psi_{m,n}$ sudarys ortonormuotą bazę erdvėje $L^2(\mathbf{R})$. Atskiru atveju, jei $a_0 = 2$, $b_0 = 1$, tai egzistuos ψ su tokiu laiko-dažnio lokalizavimo parametru, kad $\psi_{m,n}$ sudarys ortonormuotą bazę erdvėje $L^2(\mathbf{R})$. Tada,

$$c_{m,n} = \langle \psi_{m,n}, f \rangle = \int_{-\infty}^{\infty} \psi_{m,n}(x) f(x) dx . \quad \textbf{(1.12)}$$

Taigi, gavome, kad duomenys yra diskretūs laiko atžvilgiu. To mums ir reikėjo – gavome diskrečiąją bangelių transformaciją.

Bangelių generavimui naudojame mastelio funkciją $\varphi(x)$ ir daugiapakopio skaidymo analizę [8]. Tada **(1.12)** galime užrašyti tokiu algoritmu:

$$c_{m,n} = \sum_k g_{2n-k} a_{m-1,k} , \quad \textbf{(1.13)}$$

$$a_{m,n} = \sum_k h_{2n-k} a_{m-1,k} , \quad \textbf{(1.14)}$$

kur $g_l = (-1)^l h_{-l+1}$ ir $h_n = 2^{1/2} \int \varphi(2x-n)\varphi(x)dx$. Lygtis **(1.14)** duoda kodavimo algoritmą šioms paprastoms reikšmėms su žemo dažnio filtru h ir aukšto dažnio filtru g. Naudojant šiuos filtrus galima atlikti tikslų atstatymą pagal tokią išraišką:

$$a_{m-1,n}(f) = \sum_n \left[h_{2n-l} a_{m,n} + g_{2n-l} c_{m,n} \right]. \quad \textbf{(1.15)}$$

Suspaudimas, remiantis ortogonaliomis bangelėmis

Šioje dalyje apibrėšime ortogonalių bangelių filtrų koeficientus, naudojamus skaitmeniniams vaizdams spausti (glaudinti). Žinome mastelio funkciją

$$\phi(t) = \sum_k h_k \sqrt{2}\, \phi(2t-k). \quad \textbf{(1.16)}$$

Integruojant abi puses, gauname

$$\int_{-\infty}^{\infty} \phi(t) dt = \int_{-\infty}^{\infty} \sum_k h_k \sqrt{2}\, \phi(2t-k) dt ,$$

Atliekame keitimą $x = 2k - t$, $dx = 2dt$. Dešinėje pusėje gauname

$$\int_{-\infty}^{\infty} \phi(t) dt = \sum_k h_k \sqrt{2} \int_{-\infty}^{\infty} \phi(x) \frac{1}{2} dx = \sum_k h_k \frac{1}{\sqrt{2}} \int_{-\infty}^{\infty} \phi(x) dx .$$

Dabar, abi lygties puses padalinus iš $\frac{1}{\sqrt{2}} \int_{-\infty}^{\infty} \phi(x) dx$, gausime

$$\sum_k h_k = \sqrt{2} . \quad \textbf{(1.17)}$$

Mastelio funkcijoje panaudojus ortogonalumą, gauname naują filtrų koeficientus $\{h_k\}$ apibrėžiančią sąlygą:

$$\int_{-\infty}^{\infty} |\phi(t)|^2 \, dt = \int_{-\infty}^{\infty} \sum_k h_k \sqrt{2}\, \phi(2t-k) \sum_m h_m \sqrt{2}\, \phi(2t-m) dt =$$

$$= \sum_k \sum_m h_k h_m 2 \int_{-\infty}^{\infty} \phi(2t-k)\phi(2t-m) \, dt = \sum_k \sum_m h_k h_m \int_{-\infty}^{\infty} \phi(x-k)\phi(x-m) \, dx \ ,$$

kur paskutinėje išraiškoje $2t$ pakeista į x.

Dešinėje pusėje esantis integralas yra lygus nuliui, išskyrus atvejį, kai $k = m$. Kai $k = m$, gauname

$$\sum_k h_k^2 = 1 . \qquad \textbf{(1.18)}$$

Išvedime naudojome

$$\int \phi(t)\phi(t-m) \, dt = \delta_m \ ; \qquad \textbf{(1.19)}$$

$\varphi(x)$ pakeitus mastelio funkcija gauname

$$\int \left[\sum_k h_k \sqrt{2}\, \phi(2t-k) \right] \left[\sum_k h_k \sqrt{2}\, \phi(2t-2m-l) \right] dt = \sum_k \sum_l h_k h_l 2 \int \phi(2t-k)\phi(2t-2m-l) dt \ .$$

Tada pakeitus $x = 2t$, gauname

$$\int \phi(t)\phi(t-m) \, dt = \sum_k \sum_l h_k h_l \int \phi(x-k)\phi(x-2m-l) dx = \sum_k \sum_l h_k h_l \, \delta_{k-(2m+l)} = \sum_k h_k h_{k-2m} \ .$$

Taigi, turime

$$\sum_k h_k h_{k-2m} = \delta_m \ . \qquad \textbf{(1.20)}$$

Naudojant **(1.17)**, **(1.18)** ir **(1.20)** lygybes, galima sugeneruoti mastelio funkcijų filtrų koeficientus.

Kai $k = 2$, iš **(1.17)** ir **(1.18)** lygčių gauname

$$\begin{cases} h_0 + h_1 = \sqrt{2} \ , \\ h_0^2 + h_1^2 = 1 \ . \end{cases} \qquad \textbf{(1.21)}$$

Vienintelis sprendinys yra $h_0 = h_1 = \dfrac{1}{\sqrt{2}}$, o tai Haaro mastelio funkcijos filtrų koeficientai.

Kai $k = 2$, iš **(1.17)**, **(1.18)** ir **(1.20)** lygčių gauname

$$\begin{cases} h_0 + h_1 + h_2 = \sqrt{2} \ , \\ h_0^2 + h_1^2 + h_2^2 = 1 \ , \\ h_0 h_2 + h_1 h_3 = 0 \ . \end{cases} \qquad \textbf{(1.22)}$$

Šios sistemos sprendinys priklauso Daubechies mastelio funkcijai: $h_0 = \dfrac{1+\sqrt{3}}{4\sqrt{2}}$, $h_1 = \dfrac{3+\sqrt{3}}{4\sqrt{2}}$, $h_2 = \dfrac{3-\sqrt{3}}{4\sqrt{2}}$, $h_3 = \dfrac{1-\sqrt{3}}{4\sqrt{2}}$.

Žinome, kad bangelės funkcija yra

$$\psi(t) = \sum_k w_k \sqrt{2}\, \phi(2t - k). \qquad \textbf{(1.23)}$$

Jei to paties mastelio bangelės yra ortogonalios mastelio funkcijai, tai turime

$$\int \phi(t-k)\psi(t-m)\, dt = 0. \qquad \textbf{(1.24)}$$

Tada galime gauti bangelės filtro koeficientus iš mastelio filtro koeficientų [9]

$$w_k = \pm(-1)^k h_{N-k}. \qquad \textbf{(1.25)}$$

Haaro bangelė

Haaro mastelio funkcija (1.5 pav.) apibrėžiama kaip:

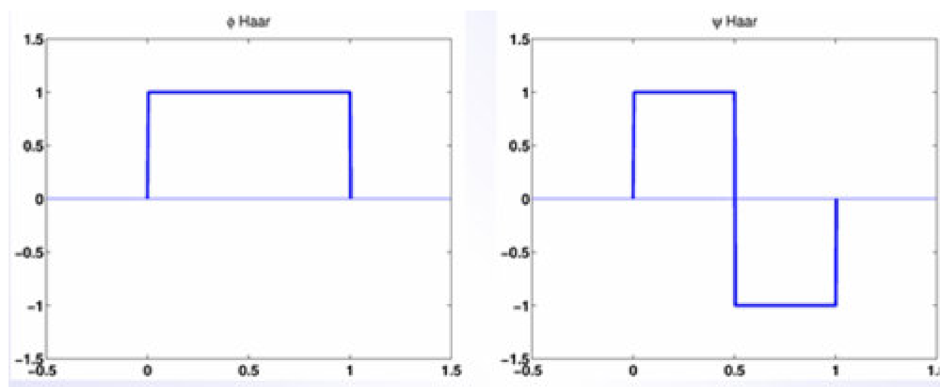
$$\phi(x) = \begin{cases} 1, & kai \;\; 0 \le x < 1, \\ 0, & kitu \; atveju. \end{cases} \qquad \textbf{(1.26)}$$

Mastelio lygtyje $\varphi(x) = \sum_{k=0}^{N-1} C_k \varphi(2x - k)$ tik $c_0 = c_1 = 1$, visi kiti koeficientai yra nuliai. Haaro bangelė apibrėžiama kaip:

$$\psi_{a,b}(x) = 2^{a/2}\psi(2^a x - b), \quad kai\; b = 0, 1, \ldots, 2^a - 1, \qquad \textbf{(1.27)}$$

kur Haaro motininė bangelė (1.5 pav.) yra

$$\psi(x) = \begin{cases} 1, & kai \;\; 0 \le x < 1/2, \\ -1, & kai \;\; 1/2 \le x < 1, \\ 0, & kitu \; atveju. \end{cases} \qquad \textbf{(1.28)}$$



1.5 pav. Haaro mastelio funkcija (kairėje) ir motininė bangelė (dešinėje)

Ortogonali Daubechies bangelė

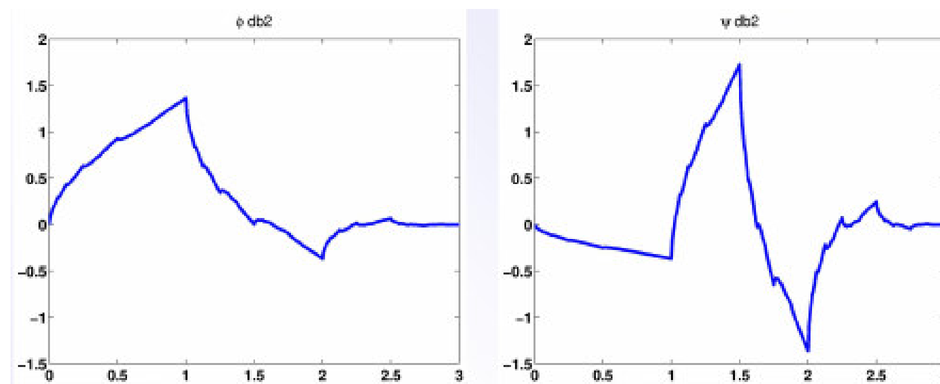
Daubechies bangelių transformacijos žymimos $db\left(\frac{N}{2}\right)$, kur N – filtro ilgis. Nėra tikslios ortogonalios Daubechies bangelės ir su ja susijusios mastelio funkcijos išraiškos, tačiau Daubechies

bangelių ir mastelio filtrų koeficientai yra unikalūs tuo, kad jie turi aukštą glodumo laipsnį. 1.1 lentelėje Daubechies bangelės filtro koeficientai, kai $N = 4$. 1.6 paveiksle parodyta $db2$.

1.1 lentelė

Daubechies žemo dažnio filtro koeficientai

Koeficientas	Koeficiento reikšmė
h_0	0,4829629131445341
h_1	0,8365163037378077
h_2	0,2241438680420134
h_3	-0,1294095225512603



1.6 pav. Daubechies *db*2 mastelio funkcija (kairėje) ir motininė bangelė (dešinėje)

1.2 lentelėje įvardytos Haaro ir Daubechies bangelių pagrindinės savybės.

1.2 lentelė

Bangelių savybės

Savybės	Haaro	Ortogonalioji Daubechies
Tiksli funkcija	Taip	Ne
Ortogonalumas	Taip	Taip
Simetriškumas	Taip	Ne
Tolydumas	Ne	Taip
Suspaudžiančioji atrama	Taip	Taip

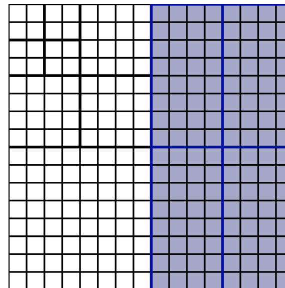
Haaro ir Daubechies bangelės ortogonalios, be to, mastelio ir bangelės funkcijos yra tokios pačios tiek tiesioginei ir atvirkštinei transformacijai.

Tačiau Haaro bangelių transformacija yra greičiausia ir paprasčiausiai realizuojama. Pagrindinis Haaro bangelių trūkumas – jos netolydumas, kuris apsunkina tolydaus signalo imitavimą.

Dar keletas diskrečiųjų bangelių transformacijų savybių, svarbių skaitmeninių vaizdų kodavimui. Tarkime, kad $\left[X\left(m_1, m_2\right)\right]$ $(m_1 = 0, 1, \ldots, N_1 - 1; \quad m_2 = 0, 1, \ldots, N_2 - 1)$ yra dvimatis skaitmeninis

vaizdas, $[Y(k_1, k_2)]$ ($k_1 = 0, 1, \ldots, N_1 - 1;\ \ k_2 = 0, 1, \ldots, N_2 - 1$) – diskretusis Haaro transformacijos (HT) spektras šiam vaizdui. Kiekvienam spektro koeficientui $Y(k_1, k_2)$ būdinga:

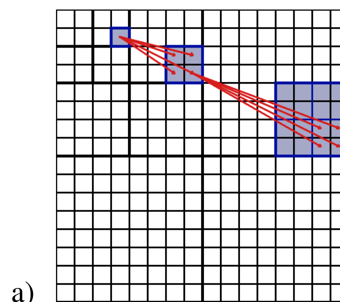
- Spektro koeficientas $Y(k_1, k_2)$ susijęs su vaizdo $[X(m_1, m_2)]$ fragmentu $X_{k_1,k_2} = \{X(m_1, m_2) | (m_1, m_2) \in J_{k_1} \times J_{k_2}\}$, čia $J_{k_i} = \{l_i \cdot 2^{s_i}, l_i \cdot 2^{s_i} + 1, \ldots, (l_i + 1) \cdot 2^{s_i} - 1\}$. Kitaip tarus, skaitinė koeficiento $Y(k_1, k_2)$ reikšmė priklauso tik nuo X_{k_1,k_2}. Pavyzdžiui, 1.7 paveikslėlyje tamsiau pažymėtas vaizdo fragmentas atitinka koeficientą $Y(1, 3)$.

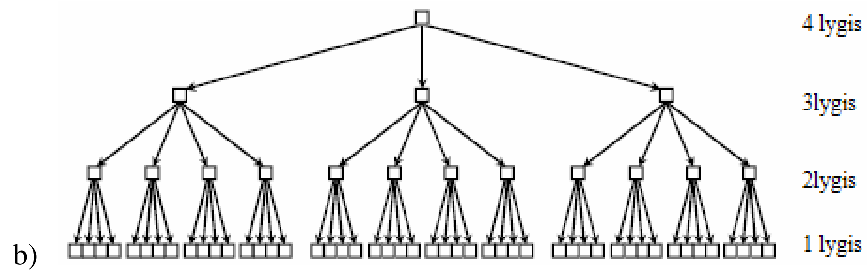


1.7 pav. Skaitmeninis vaizdas $[X(m_1, m_2)]$, kai $N = 16$

- Spektro koeficientui $Y(k_1, k_2)$ visada galima priskirti medį, kurio viršūnės (spektro koeficientai $Y(k_1^*, k_2^*)$ $\left(k_i^* \in \bigcup_{t=1}^{s_i - 1} \{2^t k_i, 2^t k_i + 1, \ldots, 2^t (k_i + 1) - 1\} \right)$, $i = 1, 2$) yra susiję su to paties vaizdo fragmento poaibiais. Be to, spektro koeficientus žemesniajame poaibyje galime įsivaizduoti kaip turinčius keturis palikuonis aukštesniajame poaibyje (1.8 pav., b)). Kiekvienas iš keturių palikuonių taip pat turi keturis palikuonis dar aukštesniame poaibyje, ir gauname medį, iš kurio bet kurios viršūnės gali išeiti tik po keturias šakas (1.8 pav.).

a)



b) 

1.8 pav. a) HT spektras $[Y(k_1, k_2)]$; b) sąryšis tarp bangelių koeficientų skirtinguose lygiuose (medis), kai $N = 16$

2. TIRIAMOJI DALIS

2.1. Progresyviojo vaizdų kodavimo metodo (EZW) algoritmizavimas, realizacija ir tyrimas

Progresyvusis kodavimas reiškia, kad imant (pasiunčiant) papildomus bitus, po kodavimo atkurtas vaizdas įgauna daugiau detalių. Tikslumas gali būti didinamas iki norimo detalizacijos lygio ir gali būti bet kuriuo metu nutraukiamas.

EZW (Embedded-Zerotree-Wavelet) [10] kodavimo algoritmas yra pagrįstas progresyviuoju skaitmeninių vaizdų kodavimu.

EZW kodavimo algoritmas grindžiamas nulinių medžių paieška diskrečiajame bangelių spektre ir problemiškai orientuotu spektro koeficientų kodavimu.

Nulinis medis – medis, kurio viršūnes atitinkantys spektriniai koeficientai yra nedidesni (absoliutine reikšme) už medžio šaknį, o pati šaknis (spektrinis koeficientas) yra mažesnė už tam tikrą fiksuotą (slenksčio) reikšmę. Progresyviojo vaizdų kodavimo idėja įgyvendinama, pervedant apdorojamą vaizdą į spektrinę diskrečiųjų bangelių sritį, skenuojant diskretųjį vaizdo spektrą bei įvedant specialų reikšmingų spektrinių koeficientų žymėjimą.

2.1.1 Struktūrinė progresyviojo vaizdų kodavimo schema

Skaitmeninio vaizdo suspaudimo struktūrinė schema pateikta 2.1 paveiksle.

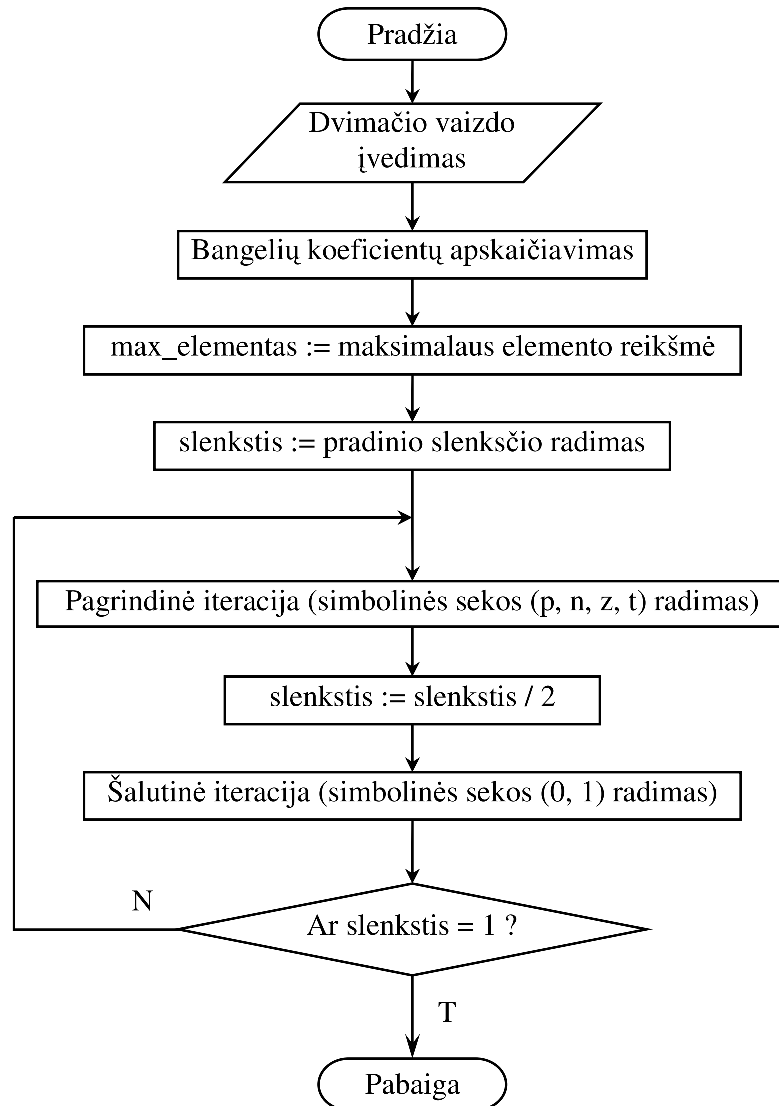
Atkuriant (dekoduojant) skaitmeninį vaizdą, taikomi tie patys pagrindiniai žingsniai atvirkščia tvarka.

2.1.2 Atskirų vaizdo kodavimo, taikant diskrečiąją bangelių transformaciją, etapų organizavimas ir algoritmavimas

Prieš pradedant aprašyti EZW algoritmą, reikia turėti minimalią informaciją, kuri bus reikalinga koduojant. Tai – naudojamas bangelių transformacijos lygių skaičius ir pradinis slenkstis, kuris apskaičiuojamas pagal formulę:

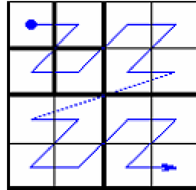
$$T_0 = 2^{\lfloor \log_2 \left(\max \left| Y(k_1, k_2) \right| \right) \rfloor}; \qquad \textbf{(2.1)}$$

čia : $\max(.)$ – maksimali spektrinio koeficiento reikšmė (spektre).



2.1 pav. Skaitmeninių vaizdų EZW kodavimo blokinė schema

Su šiuo slenksčiu vykdoma pagrindinė kodavimo dalis. Spektro koeficientai nuskaitomi, naudojant Morton seką (2.2 pav.), tokiu būdu nuosekliai nuskaitomi nuliniai medžiai.



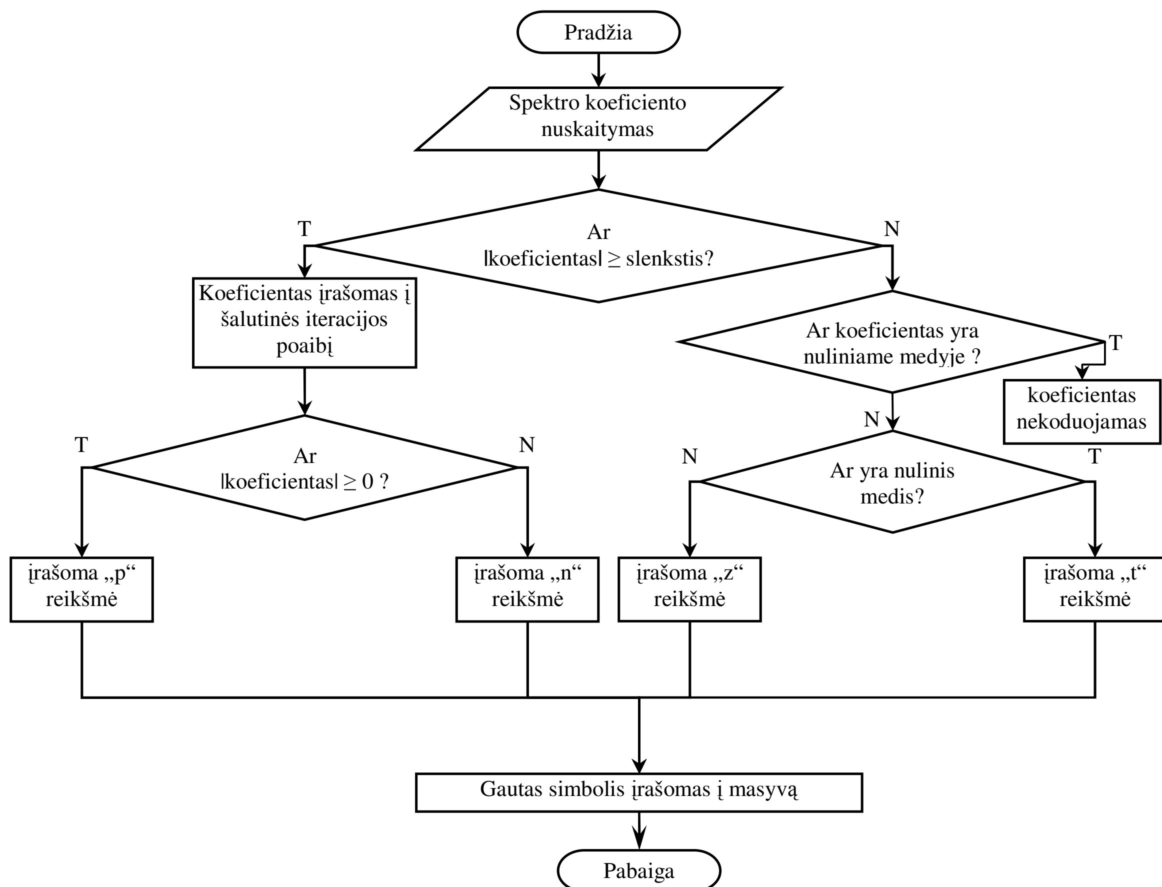
2.2 pav. Masyvo nuskaitymo tvarka – Morton seka

Tada įvertinami ir koduojami bangelių transformacijos koeficientai:

- p – jei $Y(k_1, k_2) > T_i$ (koeficientas reikšmingas), tai jis yra koduojamas ir šalinamas iš vaizdo;
- n – jei $Y(k_1, k_2) < -T_i$ (koeficientas reikšmingas), tai jis yra koduojamas ir šalinamas iš vaizdo;
- t – jei $|Y(k_1, k_2)| \leq T_i$ ir $[Y(k_1^*, k_2^*)] \leq T_i$, tai nulinis medis koduojamas vienu simboliu ir jam priklausantys koeficientai paliekami sekančiai iteracijai;
- z – jei koeficientas nereikšmingas ir nėra nulinio medžio viršūnė (šaknis), jis koduojamas kaip izoliuota viršūnė ir paliekamas sekančiai iteracijai.

Kai visi bangelių koeficientai yra įvertinti, slenkstis žeminamas ir procedūra kartojama (mažiausia leistina slenksčio reikšmė yra 1).

Pagrindinės EZW kodavimo dalies schema parodyta 2.3 paveiksle.



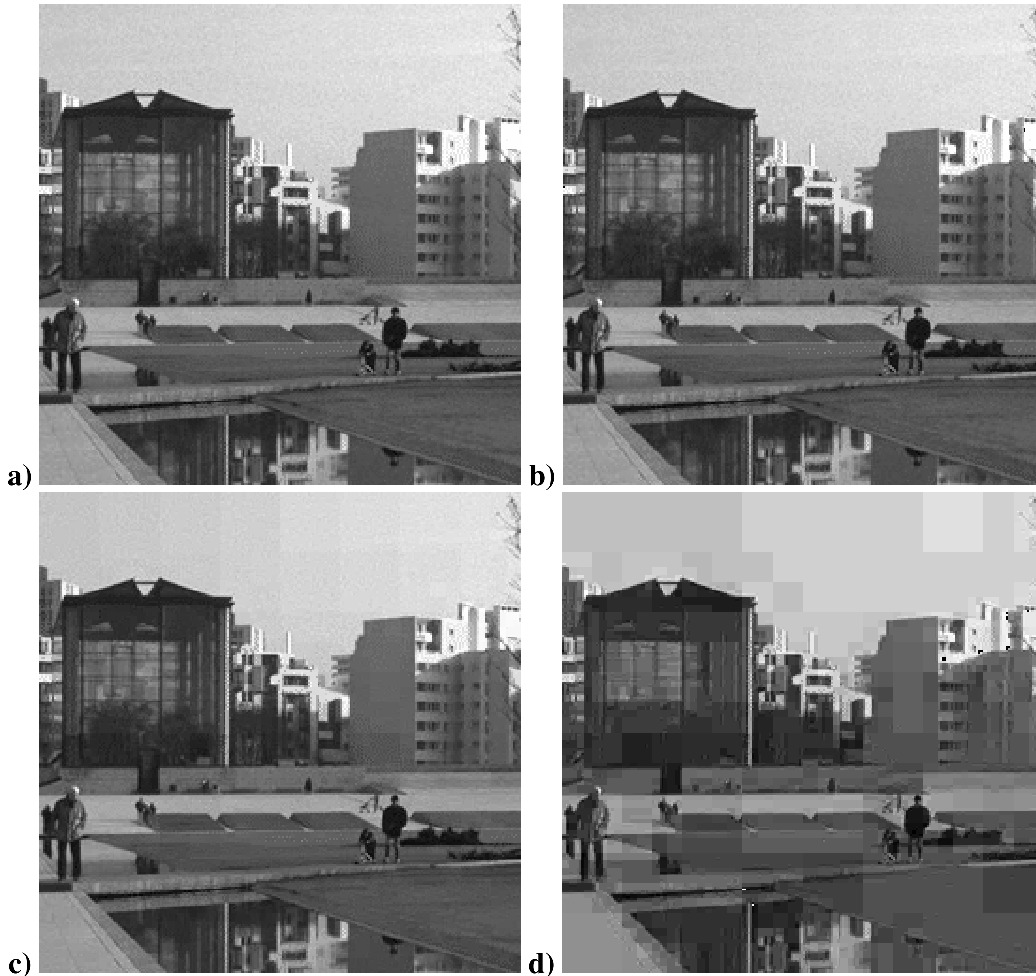
2.3 pav. EZW algoritmo pagrindinės iteracijos schema

Norint atkurti vaizdą, visi EZW algoritmo žingsniai vykdomi atvirkštine tvarka ir pritaikoma atvirkštinė bangelių transformacija.

2.2. Palyginamoji eksperimento rezultatų analizė

2.2.1 Vaizdų suspaudimas taikant skirtingas bangelių transformacijas

Nagrinėsime dvimačius skaitmeninius realaus pasaulio vaizdus $[X(m_1, m_2)]$ ($m_1 = 0, 1, \ldots, N_1 - 1$; $m_2 = 0, 1, \ldots, N_2 - 1$). Vaizdų suspaudimui naudosime Haaro ir Daubechies bangelių transformacijas bei EZW (progresyvųjį) kodavimo algoritmą (2.4 pav.).



2.4 pav. Vaizdo suspaudimas taikant Haaro transformaciją ir EZW kodavimo algoritmą: a) originalus vaizdas; b) atkurtas vaizdas, gautas naudojant 90% iteracijų, MSE=0.30, PSNR=53.37; c) naudojant 70% iteracijų, MSE=19.83, PSNR=35.16; d) naudojant 50% iteracijų, MSE=210.37, PSNR=24.90

2.2.2 Rezultatų įvertinimas

Šiame skyriuje apžvelgsime realaus ir atkurto vaizdo statistiką: EZW užkodavimo ir dekodavimo trukmę skirtingoms bangelių transformacijoms, skirtingiems suspaudimams ir skirtingiems vaizdams. Taip pat įvertinsime vidutinę kvadratinę paklaidą (MSE):

$$MSE = \frac{1}{N^2} \sum_{m_1, m_2 = 0}^{N-1} \left(\tilde{X}(m_1, m_2) - X(m_1, m_2) \right)^2 ; \quad \textbf{(2.2)}$$

čia $\left[\tilde{X}(m_1, m_2)\right]$ – atkurtas vaizdas; $\left[X(m_1, m_2)\right]$ – atkurtas vaizdas.

PSNR (peak-signal-to-noise-ratio) – tai koeficientas naudojamas vaizdo suspaudimo kokybės įvertinimui. Jis apskaičiuojamas taip:

$$PSNR = 10\log_{10}\frac{255^2}{MSE}; \qquad \textbf{(2.3)}$$

2.1 lentelėje pateikti skirtingi vaizdai, kurių atkūrimui naudojama 90% iteracijų.

2.1 lentelė

Suspaudimo rezultatų palyginimas naudojant skirtingus vaizdus

Vaizdai	Failo dydis (baitais)	Užkoduotų duomenų kiekis (baitais)	vaizdo užkodavimo trukmė (ms)	vaizdo atkūrimo trukmė (ms)	MSE	PSNR (dB)
Japan	65536	86883	703	516	0,12	57,17
Difference	65536	47533	521	219	0,30	53,40
Acura	65536	38123	438	156	0,79	49,16
Nature	65536	94220	765	579	0,13	56,98
Fractal	65536	69117	516	359	0,13	56,83
Earth	262144	288758	5656	4360	0,13	56,85
Gray_scale	262144	57455	3108	328	0,51	51,04

Nuo parinkto vaizdo priklauso: koduotų duomenų kiekis, vaizdo užkodavimo ir atkūrimo trukmė. Tuo tarpu MSE ir PSNR priklauso nuo atkurto vaizdo kokybės, t.y. nuo iteracijų kiekio pasirinkto vaizdo atkūrimui.

Paimkime pilkos šviesos intensyvumo skalės vaizdą (Gray_scale), kurio dydis (512x512), o failo, kuriame jis saugojamas, dydis 262144 baitai 2.2 lentelėje pateikta atkurto pasirinkto vaizdo priklausomybės nuo atkūrimui pasirinktų iteracijų kiekio statistika.

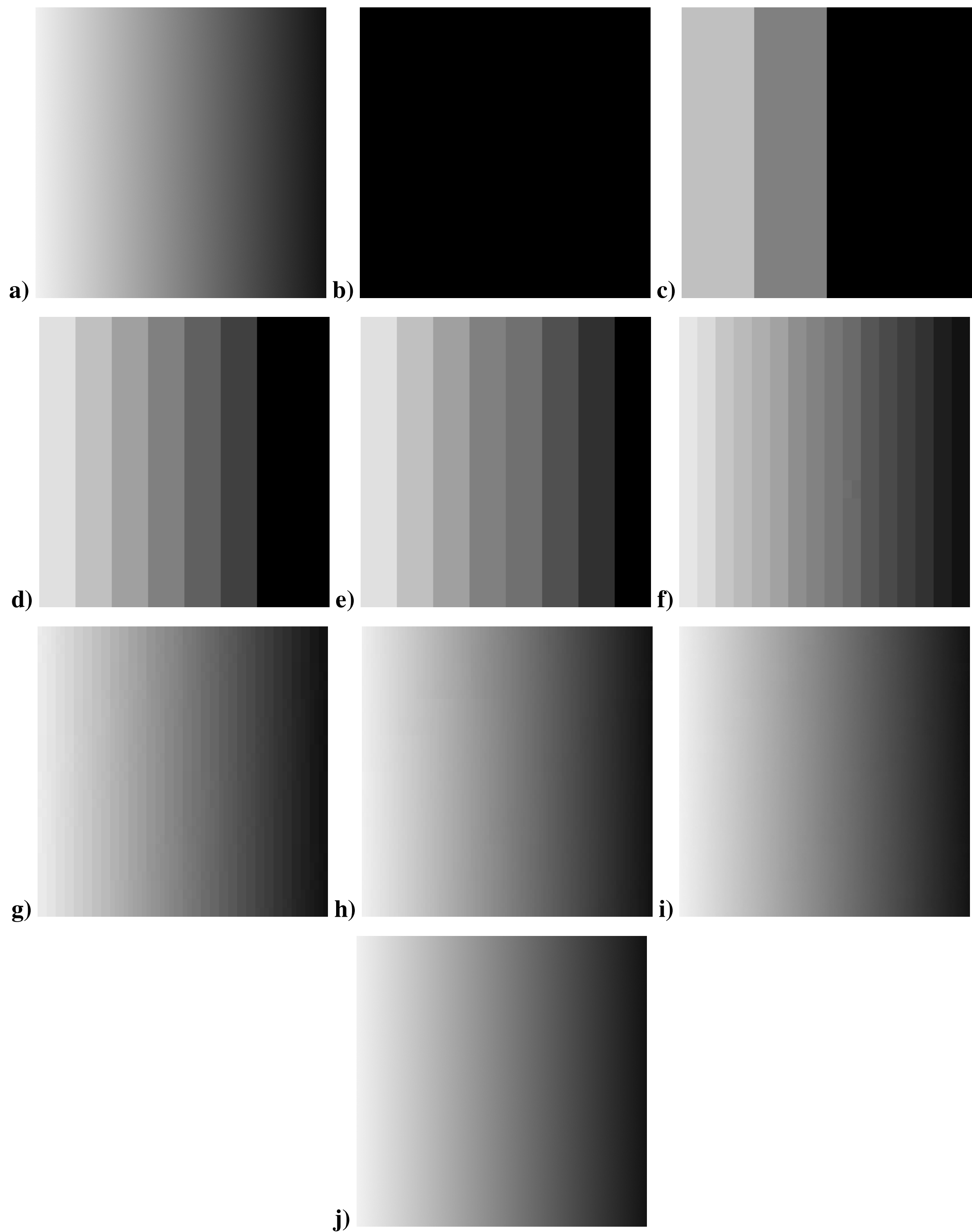
2.2 lentelė

Suspaudimo rezultatų palyginimas, vaizdų atkūrimui naudojant skirtingus iteracijų kiekius

Pasirinktų iteracijų kiekis	Užkoduotų duomenų kiekis (baitais)	Vaizdo užkodavimo trukmė (ms)	Vaizdo atkūrimo trukmė (ms)	MSE	PSNR (dB)
0	12	3079	0	21037,34	4,90
1	176	3078	0	3752,55	12,39
2	285	3093	0	838,37	18,90
3	347	3063	0	286,02	23,57
4	767	3016	0	58,29	30,48
5	3091	3047	0	20,22	35,07
6	12404	3125	32	3,44	42,77
7	57455	3078	313	0,51	51,40
8	111736	3140	2453	0	-

Nuo iteracijų kiekio, parinkto vaizdo atkūrimui, priklauso koduotų duomenų kiekis, vaizdo atkūrimo trukmė, MSE ir PSNR.

Naudojant pilkos šviesos intensyvumo skalės vaizdą (Gray_scale) galima pamatyti atkurto vaizdo priklausomybę nuo atkūrimui pasirinktų iteracijų kiekio (2.5 pav.).



2.5 pav. Pilkos šviesos intensyvumo skalė: a) originalus vaizdas; b) atkurtas vaizdas gautas naudojant 0 iteracijų; c) naudojant 1 iteraciją; d) naudojant 2 iteracijas; e) naudojant 3 iteracijas; f) naudojant 4 iteracijas; g) naudojant 5 iteracijas; h) naudojant 6 iteracijas; i) naudojant 7 iteracijas; j) naudojant 8 iteracijas – pilnai atkurtas vaizdas

Nagrinėjant realaus pasaulio vaizdus, pastebime, jog nuo parinkto vaizdo priklauso suspausto vaizdo failo dydis, vaizdo užkodavimo ir atkūrimo trukmė. Tuo tarpu MSE ir PSNR priklauso nuo atkurto vaizdo kokybės, t.y. nuo iteracijų kiekio pasirinkto vaizdo atkūrimui. Nuo pasirinktos atkurto vaizdo kokybės taip pat priklauso ir vaizdo atkūrimo trukmė.

Paėmus tą patį skaitmeninį vaizdą (256x256) pritaikome skirtingas bangelių transformacijas (Haaro ir Daubechies). 2.3 lentelėje pateikti rezultatai, gauti naudojant skirtingas bangelių transformacijas ir EZW kodavimą bei keičiant iteracijų skaičių.

2.3 lentelė

Suspaudimo rezultatų palyginimas, naudojant skirtingas bangelių transformacijas

Bangelių transformacija	Iteracijų skaičius	Užkoduotų duomenų kiekis (baitais)	vaizdo užkodavimo trukmė (ms)	vaizdo atkūrimo trukmė (ms)	MSE	PSNR (dB)
Haaro (HT)	7/8	38123	438	172	0,79	49,16
Daubechies (db2)	7/8	24372	391	94	41,54	31,95
Haaro (HT)	8/8	50294	453	329	0	0
Daubechies (db2)	8/8	38259	422	297	0	0

Nuo parinktos bangelių transformacijos priklauso: koduotų duomenų kiekis, vaizdo užkodavimo ir atkūrimo trukmė bei MSE ir PSNR. Galima pastebėti, kad EZW kodavimas naudojant Daubechies bangelių transformaciją vyksta greičiau, o koduotų duomenų kiekis yra mažesnis. Tačiau dalinai atkuriant vaizdą, gaunama prastesnė atkurto vaizdo kokybė. Tad norint gauti geresnę dalinai atkurto vaizdo kokybę, reikėtų naudoti Haaro transformaciją.

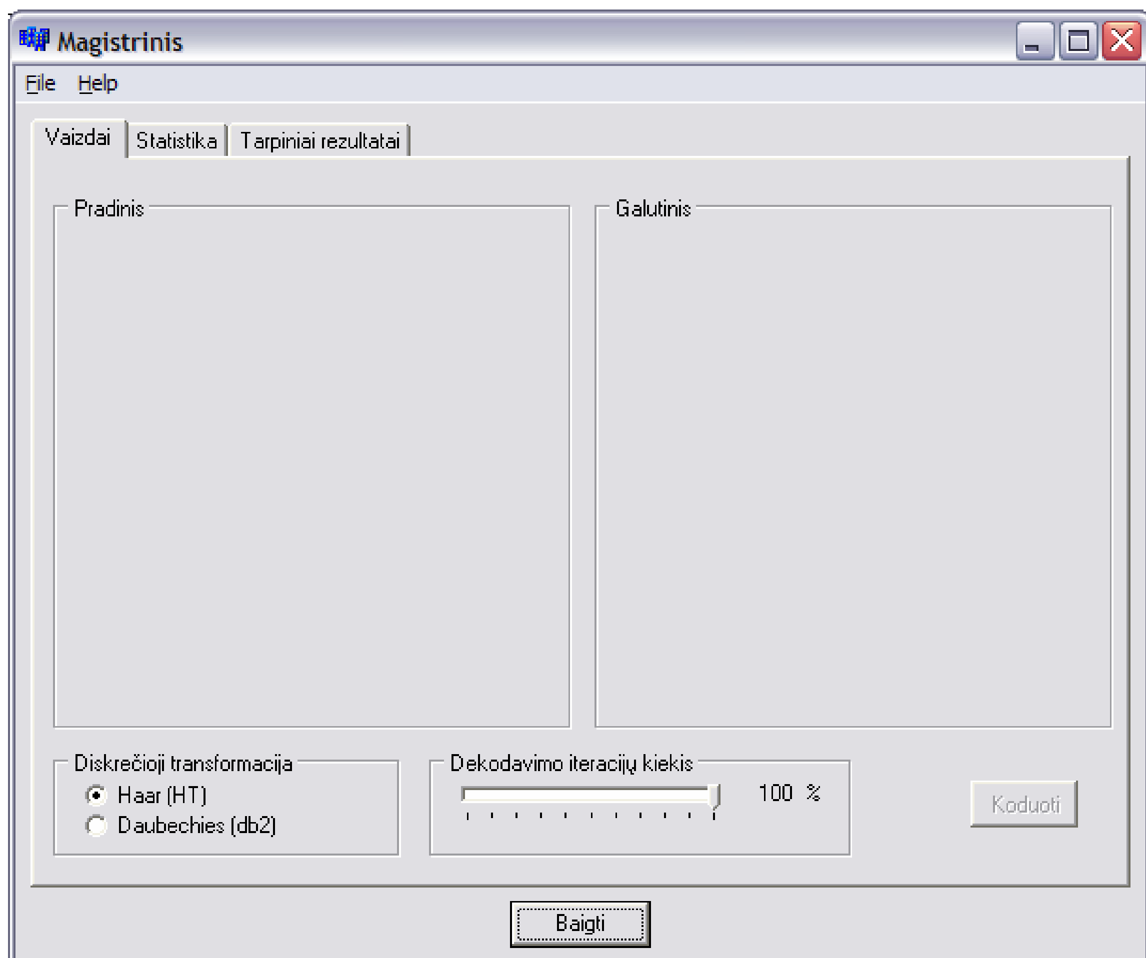
3. PROGRAMINĖ REALIZACIJA IR INSTRUKCIJA VARTOTOJUI

Naudojant diskrečiąsias bangelių transformacijas realizuotas skaitmeninių vaizdų EZW kodavimo algoritmas. Buvo naudotas Borland C++ Builder (version 6.0) programinis paketas. Sukurta grafinė vartotojo sąsaja, kur galima:

- nurodyti pradinius programos vykdymo parametrus;
- matyti pradinius, tarpinius ir galutinius skaitmeninius vaizdus bei statistinius duomenis.

Norint užkoduoti dvimatį skaitmeninį vaizdą, reikia:

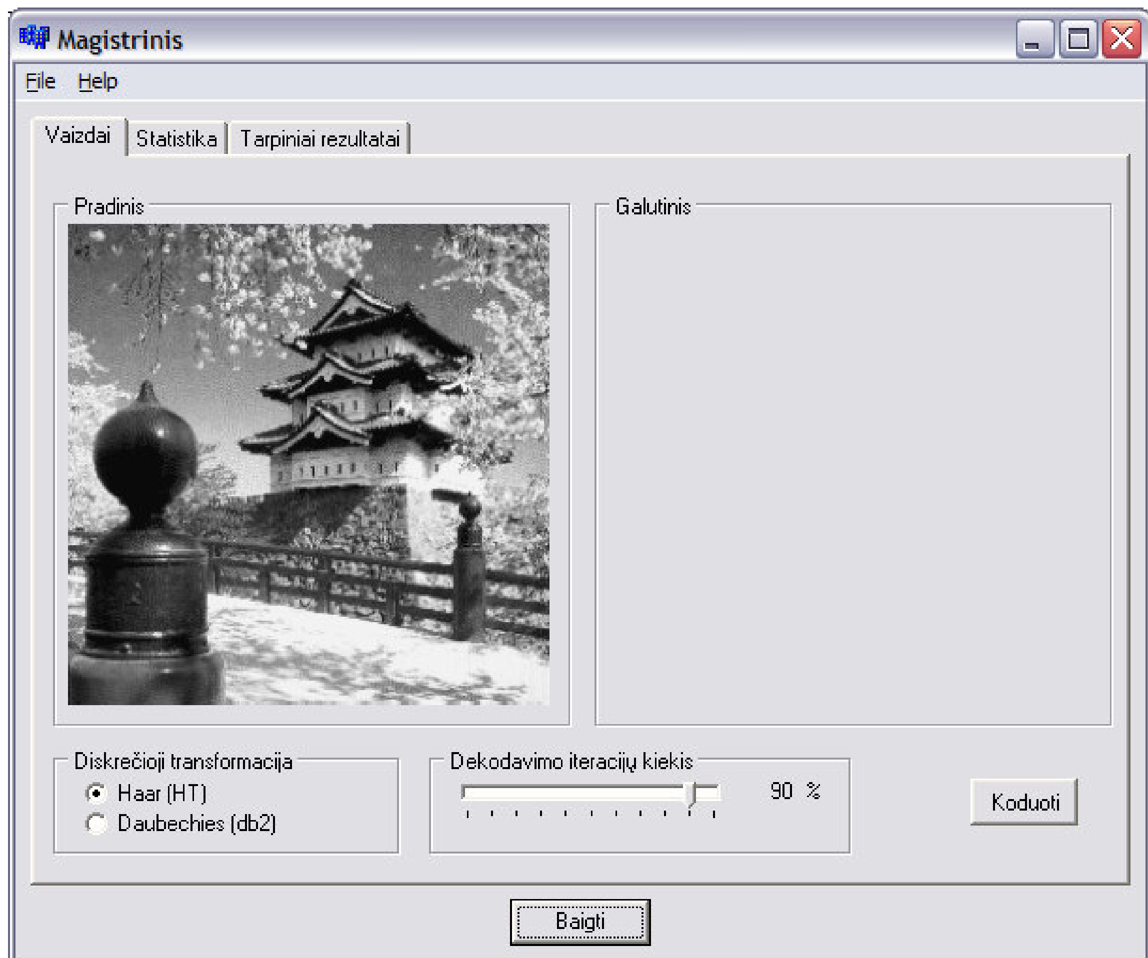
1. Atsiversti programos *proj.exe* „Paveikslai" langą (3.1 pav.);



3.1 pav. Programos langas

2. Užėjus ant menių juostos, pasirinkti **File** → **Open** (arba paspaudus **Ctrl+O**) ir iš esančių duomenų failų išsirinkti vaizdą, kurį norite užkoduoti;

3. Reikia pasirinkti pradinius parametrus: Haaro ar Daubechies diskrečiąją transformaciją bei vaizdo atkūrimui naudojamą iteracijų kiekį (procentais, nes iteracijų skaičius nėra žinomas iš anksto – jis nustatomas tik atlikus bangelių transformaciją) (3.2 pav.);



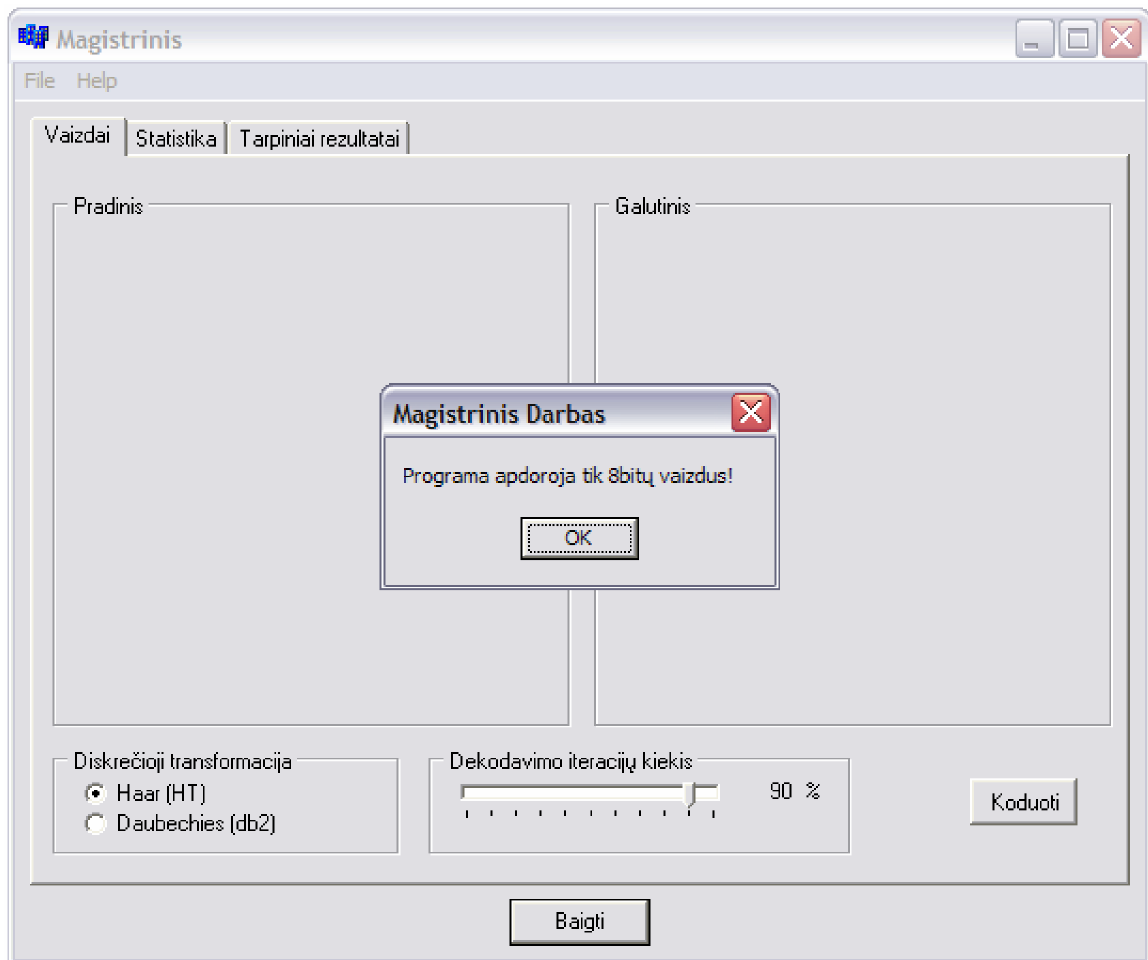
3.2 pav. Pradinių parametrų parinkimas

4. Paspaudus mygtuką **Koduoti**, pasirinktas vaizdas užkoduojamas bei atkoduojamas ir gautas vaizdas parodomas lange **Galutinis**.
5. Paspaudus mygtuką **Baigti**, uždaromas programos vykdymo langas.

Skaitmeninio vaizdo failo parametrai:

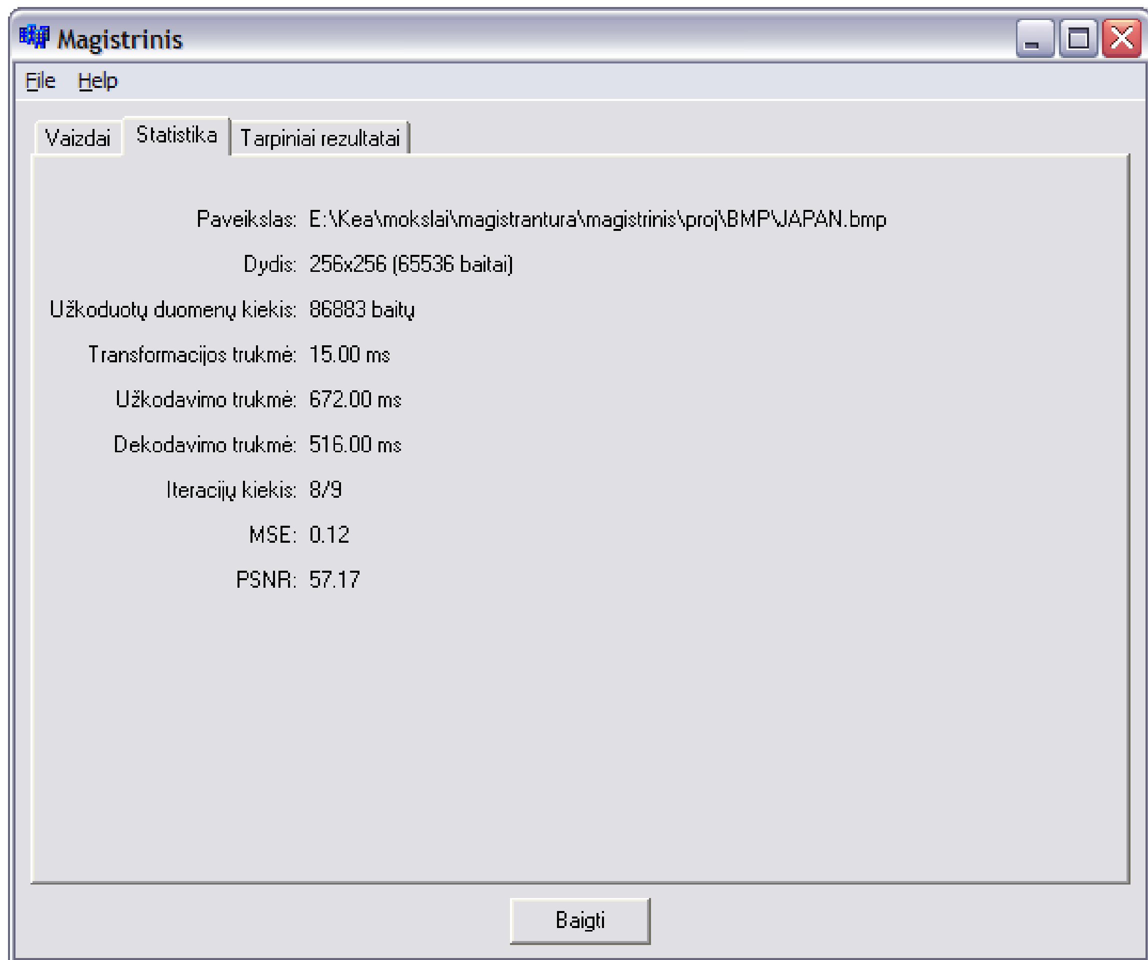
- skaitmeninis vaizdas turi būti su pilka šviesos intensyvumo skale;
- svarbu, kad vaizdai būtų kvadratiniai, t.y. $N \times N$ dydžio.

Užkrovus vaizdą su neteisingais parametrais, išvedamas pranešimas apie klaidą (3.3 pav.).



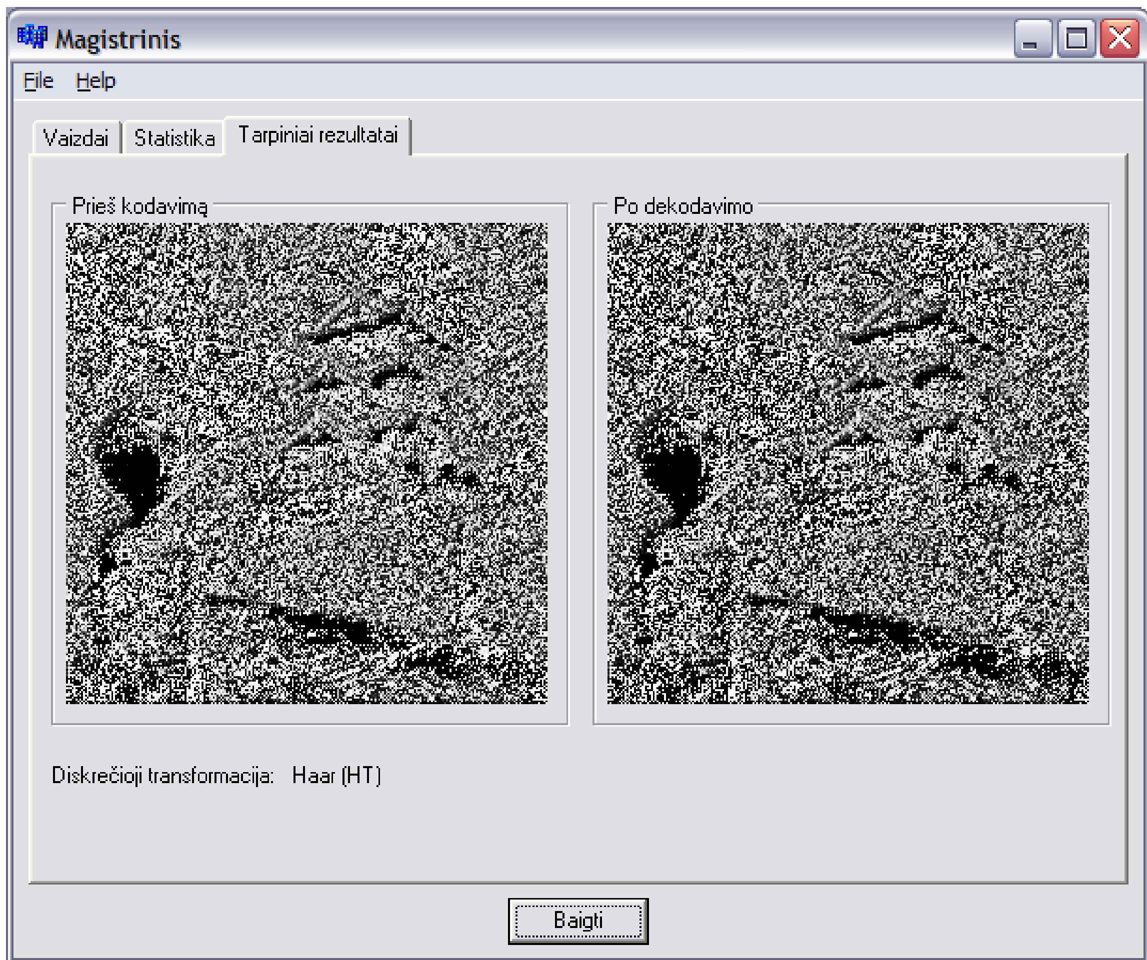
3.3 pav. Pranešimas apie klaidą

Atsidarius kitą langą, galima pažiūrėti programos vykdymo statistinius duomenis: vaizdo pavadinimą, dydį, užkrautų duomenų kiekį, transformacijos trukmę, užkodavimo ir dekodavimo trukmę, vaizdo atkūrimui pasirinktų iteracijų kiekį iš visų galimų bei MSE ir PSNR (3.4 pav.).



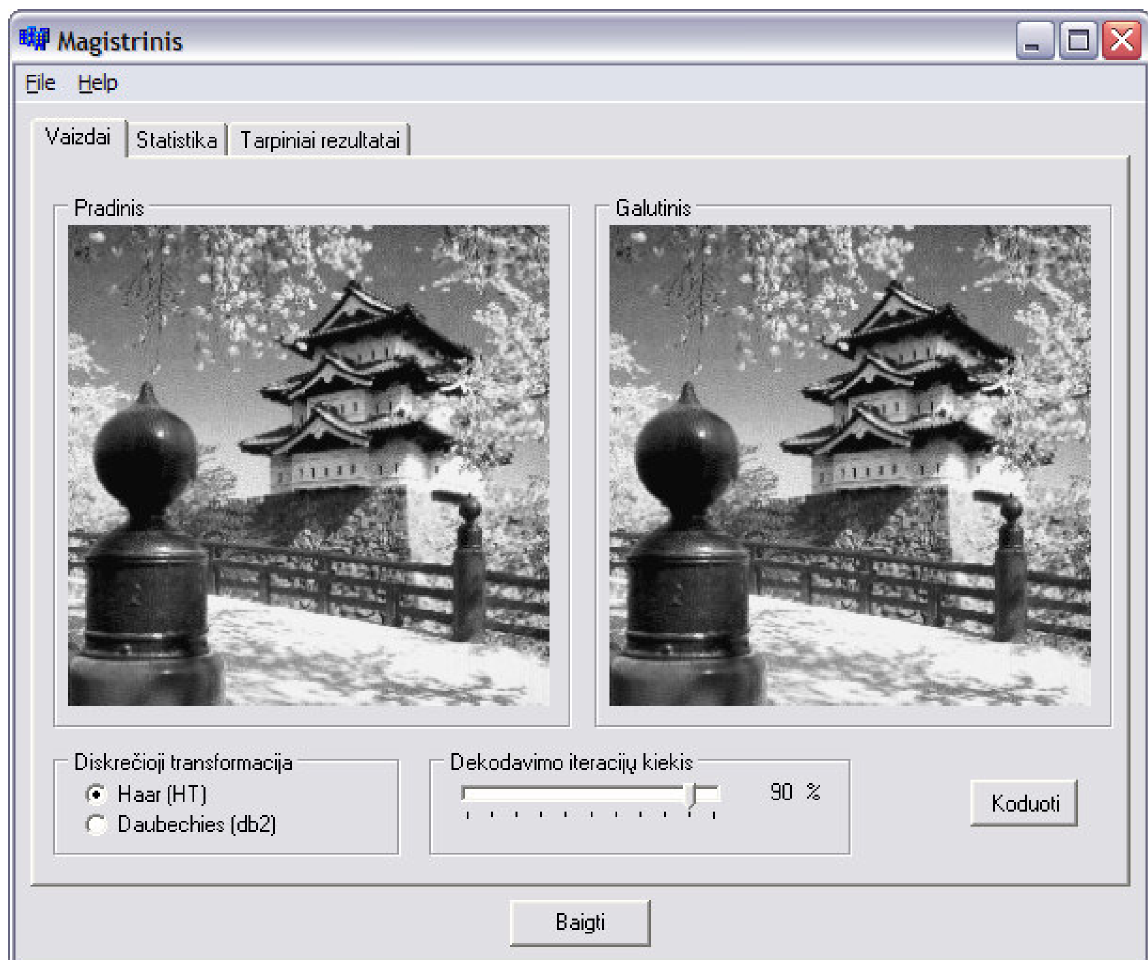
3.4 pav. Statistiniai duomenys

Taip pat, galima pažiūrėti ir tarpinius rezultatus: transformuotą vaizdą naudojant pasirinktą bangelę prieš EZW kodavimą ir po jo (3.5 pav.).



3.5 pav. Transformacijos duomenys

Norint palyginti originalų ir atkurtą vaizdą įvykdžius kodavimą atverčiamas langas **Vaizdai** (3.6 pav.).



3.6 pav. Originalus ir atkurtas vaizdas

DISKUSIJA

Šiame darbe trumpai supažindinama su įvairiais vaizdų kodavimo metodais duomenų ir spektrinėje srityje. Plačiau pristatomas progresyviojo vaizdų kodavimo metodo EZW algoritmas, grįstas nulinių medžių paieška diskrečiajame bangelių spektre ir spektro koeficientų kodavimu.

Naudojant progresyvius kodavimo algoritmus, galima pasirinkti atkuriamo vaizdo detalizacijos lygį (vaizdo kokybę). EZW kodavime vaizdo kokybę apsprendžia vaizdo atkūrimui pasirinktų iteracijų skaičius.

Atlikus eilę eksperimentų su skirtingais skaitmeniniais vaizdais, nustatyta, kad vidutinė kvadratinė paklaida (MSE) bei signalo ir triukšmo santykinė reikšmė (PSNR) priklauso nuo vaizdo atkūrimui naudojamų iteracijų kiekio ir nežymiai nuo pačio vaizdo, tačiau nepriklauso nuo vaizdo dydžio $N \times N$. Pasirinkus daugiau iteracijų vaizdo atkūrimui gaunama mažesnė vidutinė kvadratinė paklaida.

Taip pat pastebėta, kad, naudojant EZW kodavimo algoritmą, skaitmeniniai vaizdai atkuriami sparčiau, negu buvo užkoduojami – tai gerai specialistams žinoma asimetrijos problema. Užkodavimo ir dekodavimo laikas labiausiai priklauso nuo kompiuterio, kuriame vykdoma programa, parametrų. Vaizdo užkodavimo trukmė taipogi priklauso nuo vaizdo dydžio ir nežymiai nuo pačio vaizdo turinio. Dekodavimo trukmė tiesiogiai priklauso nuo vaizdo atkūrimui pasirinkto iteracijų skaičiaus.

Naudojant skirtingas bangelių transformacijas (Haaro ir Daubechies), pastebėta, kad taikant Daubechies transformaciją, EZW kodavimas užtrunka trumpiau ir gaunamas mažesnis koduotų duomenų kiekis, tačiau dalinai atkuriant vaizdą, gaunama prastesnė atkurto vaizdo kokybė.

EZW algoritmo efektyvumas priklauso jo realizacijos optimizavimo bei pritaikymo kompiuterių architektūros.

IŠVADOS

1. Diskrečiųjų bangelių transformacijos turi labai įdomių (praktinio panaudojimo prasme) savybių: spektro koeficientų sąsają su vaizdo fragmentais ir nulinių medžių egzistavimą spektre.
2. EZW algoritmas išnaudoja specifines bangelių transformacijos savybes – tai leidžia sparčiai surikiuoti reikšmingus bangelių koeficientus taip, kad vaizdą būtų galima efektyviai suglaudinti.
3. EZW algoritmui būdinga gerai specialistams žinoma asimetrijos problema (lėtokas užkodavimo ir greitas dekodavimo etapas). Tai susiję su nulinių medžių diskrečiajame bangelių spektre paieška ir identifikavimu.
4. Skaitmeninių vaizdų su pilka šviesos intensyvumo skale suglaudinimo kokybė priklauso nuo vaizdo atkūrimui naudojamų iteracijų skaičiaus, bet nepriklauso nuo apdorojamo vaizdo dydžio.
5. EZW kodavimo algoritmas skirtas dvimačiams skaitmeniniams signalams (vaizdams) koduoti, tačiau gali būti nesunkiai pritaikomas kitokio matavimo signalams.

LITERATŪRA

1. Ahmed N., Rao K. R. Orthogonal Transforms for Digital Signal Processing. – Springer-Verlag, Berlin, Heidelberg-New York, 1975. – 263 p.
2. Culik K., Kari I. Image compression using weighted finite automata. – Computer and Graphics, 17. 1993. – 305-313 p.
3. Frantini P., Nevalainen O., Kaukoranta t. Compression of digital images by black truncation coding: A survey. – The Computer Journal, 37 (4). 1994.
4. Fisher Y. Fractal Image Compression – Theory and Applications. – Springer-Verlag, New York, 1994.
5. Jacquin A. Image Coding Based on a Fractal Theory of Iterated Contractive Image Transformations. – IEEE Transactions on Image Processing, Vol. 1. 1992. – 18-30 p.
6. Marcelin M. W., Gormish M. J., Birgin A., Bolick M. P. An Overview of JPEG-2000. – IEEE Data Compression Conference. 2000. – 523-541 p.
7. Zinterhof P. Uber die schnelle Losung von hochdimensioalen Fredholm – Gleichungen vom Faltungsryp mitzahlentheoretschen Methoden. – Sonderdruck aus Sitzungsberichte, Arb. 2, Mathematische, Physikalische, Band 196, Wien. 1987 – 4-7 h.
8. Walker J. S. Wavelet-based Image Compression. http://www.uwec.edu/walkerjs/media /imagecompchap.pdf
9. Daubechies I. Ten lectures on Wavelets. – SIAM, Philadelphia, Pennsylvania, 1992. – 460 p.
10. Burrus C. S., Gopinath R. A., Guo H. Introduction to wavelets and wavelet transform. – Prentice Hall, Englewood Cliffs, New Jersey, 1998.
11. Shapiro J. M. Embedded image coding using zerotrees of wavelet coefficients. – IEEE transactions on signal processing. 41(12). December, 1993. – 3445-3462 p. http://www.csee.wvu.edu/%7Exinl/courses/ee591b/EZW.pdf
12. Gonzalez R. C., Woods R. E. Digital image processing. – Prentice Hall, Upper Saddle River, New Jersey, 2nd edition, 2002. – 1072 p.
13. Блатер К. Вейвлет-анализ. Основы теории. – Москва: Техносфера, 2004. – 280 с.
14. Saha S. Image compression – from DCT to Wavelet. http://www.acm.org/crossroads /xrds6-3/sahaimgcoding.html, 2000.

1 PRIEDAS. PROGRAMOS TEKSTAS

failas unit1.cpp

```
//---------------------------------------------------------------------------

#include <vcl.h>
#pragma hdrstop

#include <stdio.h>
#include <math.h>

#include "Unit1.h"
#include "Apie.h"
#include "image.h"
#include "coding.h"

#include "ezw.h"
#include "ezw2.h"
#include "haar.h"


//---------------------------------------------------------------------------
#pragma package(smart_init)
#pragma resource "*.dfm"
TLangas *Langas;
//---------------------------------------------------------------------------
__fastcall TLangas::TLangas(TComponent* Owner)
        : TForm(Owner)
{
}
//---------------------------------------------------------------------------

//---------------------------------------------------------------------------
void __fastcall TLangas::ExitButtonClick(TObject *Sender)
{
         Close();
}
//---------------------------------------------------------------------------
void __fastcall TLangas::FormClose(TObject *Sender, TCloseAction &Action)
{
        Action = caNone;
        Application->NormalizeTopMosts();
        int res = Application->MessageBox(" Ar tikrai norite baigti darba?  ",
"Pabaiga", MB_YESNO);
        Application->RestoreTopMosts();
        if (res == ID_YES)
        {
                Action = caFree;
        }
}
//---------------------------------------------------------------------------
void __fastcall TLangas::AboutClick(TObject *Sender)
{
     AboutBox->ShowModal();
}
//---------------------------------------------------------------------------
void __fastcall TLangas::ExitClick(TObject *Sender)
{
        Close();
}
//---------------------------------------------------------------------------
```

```
void __fastcall TLangas::FileOpenClick(TObject *Sender)
{
        OpenDialog->Filter = "Paveikslai (*.bmp)|*.bmp";
        if (OpenDialog->Execute() && FileExists(OpenDialog->FileName)) {
                char buf[256];
                RezImage->Picture->Assign(NULL);
                PradImage->Picture->LoadFromFile(OpenDialog->FileName);
                if (PradImage->Picture->Bitmap->PixelFormat != pf8bit) {
                        PradImage->Picture->Assign(NULL);
                        ShowMessage("Programa apdoroja tik 8bitø vaizdus!");
                        return;
                }
                if (PradImage->Picture->Width != PradImage->Picture->Height) {
                        PradImage->Picture->Assign(NULL);
                        ShowMessage("Programa apdoroja tik NxN dydþio vaizdus!");
                        return;
                }
                if (PradImage->Picture->Width % 16 ||
                    PradImage->Picture->Height % 16) {
                        PradImage->Picture->Assign(NULL);
                        ShowMessage("Programa apdoroja tik NxN dydþio vaizdus, kai
N dalijasi ið 16!");
                        return;
                }

                FileSave->Enabled = false;
                Transform->Enabled = true;
                PradPavName->Text = OpenDialog->FileName;
                sprintf(buf, "%dx%d (%d baitai)", PradImage->Picture->Width,
                                PradImage->Picture->Height,
                                PradImage->Picture->Width * PradImage->Picture-
>Height);
                PradPavDim->Text = buf;
        }
}
//---------------------------------------------------------------------------

void __fastcall TLangas::FileSaveClick(TObject *Sender)
{
        SaveDialog->Filter = "Paveikslai (*.bmp)|*.bmp";
        if (SaveDialog->Execute()) {
                RezImage->Picture->SaveToFile(SaveDialog->FileName);
        }
}
//---------------------------------------------------------------------------



static struct image_data* bitmap2image(TPicture* p) {
        int i, j;
        image_data_t* im;
        im = image_create(p->Bitmap->Width, p->Bitmap->Height);
        p->Bitmap->Canvas->Lock();
        for (i = 0; i < p->Height; i++)
                for (j = 0; j < p->Width; j++) {
                        TColor c = p->Bitmap->Canvas->Pixels[j][i];
                        im->data[i][j] = c & 0xff;
                }

        p->Bitmap->Canvas->Unlock();
        return im;
}

static  Graphics::TBitmap*
```

```
image2bitmap(const image_data_t* im, TPicture* src) {
        if (im == NULL)
                return NULL;
        Graphics::TBitmap* b = new Graphics::TBitmap();
        b->Width = im->w;
        b->Height = im->h;
        b->PixelFormat = src->Bitmap->PixelFormat;
        b->Palette = CopyPalette(src->Bitmap->Palette);
        b->Assign(src->Bitmap);

        b->Canvas->Lock();
        for (size_t i = 0; i < im->h; ++i) {
                for (size_t j = 0; j < im->w; j++) {
                        unsigned char bb = (unsigned char)((int)(im->data[i][j]));
                        TColor c = bb << 16 | bb << 8 | bb;
                        b->Canvas->Pixels[j][i] = c;
                }
        }
        b->Canvas->Unlock();
        return b;
}

void __fastcall TLangas::TransformClick(TObject *Sender)
{
        int i, j;
        RezImage->Picture->Assign(NULL);
        char buf[32];
        // isiminkim kursoriu ir padarykim ji laikroduku tam, kad
        // parodyti, jog vyksta skaiciavimas
        TCursor Save_Cursor = Screen->Cursor;
        Screen->Cursor = crHourGlass;
        Transform->Enabled = false;
        RezImage->Picture->Assign(NULL);
        Application->ProcessMessages();

        // konvertuokim paveiksleli i duomenu matrica
        image_data_t* im = bitmap2image(PradImage->Picture);
        transcode_data* out;
        int transtype;

        if (htRadio->Checked) {
                transtype = TRANS_HT;
        } else {
                transtype = TRANS_DB2;
        }
        out = transcode(transtype, im, ProcBar->Position);

        // duomenu matrica konvertuojam atgal i paveiksleli atvaizdavimui
        Graphics::TBitmap* b = image2bitmap(out->result, PradImage->Picture);
        if (b != NULL) {
                RezImage->Picture->Bitmap->Assign(b);
        }

        b = image2bitmap(out->trans1, PradImage->Picture);
        if (b != NULL) {
                TransformImage1->Picture->Bitmap->Assign(b);
        }
        b = image2bitmap(out->trans2, PradImage->Picture);
        if (b != NULL) {
                TransformImage2->Picture->Bitmap->Assign(b);
        }

        TransformName->Caption = out->transName;
        sprintf(buf, "%d baitø",
```

```
                ezw_calculate_required_space(out->ezw_info, ProcBar->Position));
        CodedSize->Text = buf;

        sprintf(buf, "%4.2f ms", out->transformDuration);
        TransformDuration->Text = buf;
        sprintf(buf, "%4.2f ms", out->codeDuration);
        CodingDuration->Text = buf;
        sprintf(buf, "%4.2f ms", out->decodeDuration);
        DecodingDuration->Text = buf;
        sprintf(buf, "%d/%d", out->ezw_info->it, out->ezw_info->iterations);
        Iterations->Text = buf;

        image_diff_t diff;
        image_diff(im, out->result, &diff);
        sprintf(buf, "%4.2f", diff.mse);
        DiffMSE->Text = buf;
        sprintf(buf, "%4.2f", diff.psnr);
        DiffPSNR->Text = buf;

        // atstatykim kursoriu i buvusi
        Screen->Cursor = Save_Cursor;
        Transform->Enabled = true;
        FileSave->Enabled = true;

        image_destroy(im);
        transdata_destroy(out);
}
//---------------------------------------------------------------------------


void __fastcall TLangas::ProcBarChange(TObject *Sender)
{
        char buf[16];
        sprintf(buf, "%d", ProcBar->Position);
        ProcLabel->Caption = buf;
}
//---------------------------------------------------------------------------
```

failas fifo.h

```
void put_in_fifo(ezw_element d);
ezw_element get_from_fifo(void);
void destroy_fifo(void);
void initialize_fifo(void);

#endif /* __FIFO_H__ */
```

failas fifo.c

```
/*
FIFO.C

Simple implementation of a fifo in ANSI-C.

This file is part of my Embedded Zerotree Wavelet Encoder Tutorial.
```

```
(C) C. Valens, <c.valens@mindless.com>

Created    : 04/09/1999
Last update: 29/09/1999
*/


#include "fifo.h"
#include <stdlib.h>
#include <mem.h>
#include <string.h>

typedef struct __fifo_element {
  ezw_element data;
  struct __fifo_element *previous;
} fifo_element;


fifo_element *fifo_first = NULL;
fifo_element *fifo_last = NULL;
char fifo_empty = 1;


void put_in_fifo(ezw_element d)
{
  fifo_element *p;

  p = malloc(sizeof(fifo_element));
  if (p!=NULL) {
    p->data = d;
    p->previous = NULL;
    if (fifo_last!=NULL) fifo_last->previous = p;
    fifo_last = p;
    if (fifo_first==NULL) fifo_first = p;
    fifo_empty = 0;
  }
}

ezw_element get_from_fifo(void)
{
  ezw_element d;
  fifo_element *p;

  p = fifo_first;
  if (p!=NULL) {
    d = p->data;
    fifo_first = p->previous;
    free(p);
  }
  else {
    fifo_last = NULL;
    fifo_empty = 1;
  }
  return d;
}

void destroy_fifo(void)
{
  fifo_element *p;
  while (fifo_first!=NULL) {
    p = fifo_first;
    fifo_first = p->previous;
    free(p);
  }
```

```
  fifo_last = NULL;
  fifo_empty = 1;
}

void initialize_fifo(void)
{
  fifo_first = NULL;
  fifo_last = NULL;
  fifo_empty = 1;
}
```

failas ezw2.h

```
#ifndef EZW2_H_INCLUDED
#define EZW2_H_INCLUDED

#include "image.h"
#include "ezw.h"

#ifdef __cplusplus
extern "C"{
#endif

ezw_data_t* ezw2_encode(const image_data_t* src);
image_data_t* ezw2_decode(ezw_data_t* data, unsigned int qual);

#ifdef __cplusplus
}
#endif


#endif
```

failas ezw2.c

```
#include <alloc.h>
#include <stdlib.h>
#include <math.h>

#include "ezw2.h"
#include "fifo.h"

#define min_element_type -32768
#define max_element_type  32767

/* ALTERNATIVE EZW coding/decoding */


/* --- ENCODE -------- */
void output_code(ezw_data_t* e, byte code)
{
        it_row_t* dit = e->doms[e->it];
        it_row_t* sit = e->subs[e->it];

        if (code == EZW_ZERO || code == EZW_ONE) {
                it_row_append(sit, code);
```

```
        } else {
                it_row_append(dit, code);
        }
}
/*
 * Returns 1 if descendance tree is a zerotree.
 */
char Zerotree(image_data_t* src, int x, int y, int threshold)
{
  int i, j, min_x, max_x, min_y, max_y;
  float temp, max;
  char stop;

  stop = 0;
  if ((x==0) && (y==0)) {
    temp = src->data[0][0];
    src->data[0][0] = min_element_type;
    max = image_findmaxabs(src);
    src->data[0][0] = temp;
    if (max>=threshold) stop = 1;
  }
  else {
    min_x = x << 1;
    min_y = y << 1;
    max_x = (x+1) << 1;
    max_y = (y+1) << 1;
    if ((min_x==src->w) || (min_y==src->h)) {
      return (1);
    }

    max = 0;
    while ((max_y<=src->h) && (max_x<=src->w)) {
      for (i=min_y; i<max_y; i++) {
        for (j=min_x; j<max_x; j++) {
          temp = abs((int)src->data[i][j]);
          if (temp>=threshold) {
            stop = 1;
            break;
          }
        }
        if (stop==1) break;
      }
      if (stop==1) break;
      min_x <<= 1;
      max_x <<= 1;
      min_y <<= 1;
      max_y <<= 1;
    }
  }
  if (stop==1) return (0);
  return (1);
}

/*
 * Returns a dominant-pass code from the alphabet [POS,NEG,ZTR,IZ].
 */
int code(image_data_t* src, int x, int y, unsigned int threshold)
{
  float temp;
  temp = src->data[y][x];
  if (abs(temp)>=threshold) {
    if (temp>=0) return (EZW_POS);
    else return (EZW_NEG);
  }
```

```
  else {
    if (Zerotree(src,x,y,threshold)==1) return (EZW_ZTR);
    else return (EZW_IZO);
  }
}
/*
 * Appends a value to the subordinate list.
 */
void to_sub_list(ezw_data_t* e, float* value)
{
  xy_t d;
  /*
   * Put only coefficient magnitude in list, sign is allready coded.
   */
  d.x = abs((int)*value);
  d.y = 0;
  xy_row_append(e->subxy, &d);
}

/*
 * Builds a dominant pass EZW-element from a matrix element and a threshold.
 */
void process_element(ezw_data_t* e, image_data_t* src,
                unsigned int threshold, unsigned int row, unsigned int col,
                ezw_element *s)
{
        float* val;
        s->x = col; s->y = row;
        s->code = code(src,s->x,s->y,threshold);
        val = &src->data[row][col];
        if ((s->code==EZW_POS) || (s->code==EZW_NEG)) {
                to_sub_list(e, val);
                *val = 0;
        }
}
/*
 * Performs one complete dominant pass. Dominant-pass-codes are sent to the
 * output stream and the subordinate list is updated.
 */
void dominant_pass(ezw_data_t* e, image_data_t* src, unsigned int thresh)
{
  ezw_element s;
  int min_x, max_x, min_y, max_y;
/*  int level;*/

  process_element(e,src,thresh,0,0,&s);
  output_code(e, s.code);

  process_element(e,src,thresh, 0, 1, &s);
  put_in_fifo(s);
  process_element(e,src,thresh, 1, 0, &s);
  put_in_fifo(s);
  process_element(e,src,thresh, 1, 1, &s);
  put_in_fifo(s);

  s = get_from_fifo();
  if (fifo_empty==0)
        output_code(e, s.code);

  while (fifo_empty==0) {
    if (s.code!=EZW_ZTR) {
      min_x = s.x << 1;
      max_x = min_x+1;
      min_y = s.y << 1;
```

```
      max_y = min_y+1;
      if ((max_x<=src->w) && (max_y<=src->h)) {
        unsigned int c,r;
        for (r=min_y; r<max_y+1; r++) {
          for (c=min_x; c<max_x+1; c++) {
            process_element(e,src,thresh,r,c,&s);
            put_in_fifo(s);
          }
        }
      }
    }
    s = get_from_fifo();
    if (fifo_empty==0) output_code(e,s.code);
  }
}

/*
 * Performs one subordinate pass.
 */
void subordinate_pass(ezw_data_t* e, unsigned int threshold)
{
  xy_t* d;
  int i;
  char found;

  if (threshold>0) {
    for (i=0; i<e->subxy->count; i++) {
        d = &e->subxy->data[i];
        if ((d->x&threshold)!=0) output_code(e, EZW_ONE);
        else output_code(e,EZW_ZERO);
    }
  }
}
extern float log2(float x);
extern unsigned int two_x(unsigned int x);
ezw_data_t* ezw2_encode(const image_data_t* src) {
        ezw_data_t* e;
        image_data_t* copy;
        unsigned int it;
        unsigned int max_depth = log2(src->w);
        unsigned int thresh;
        float max_el = image_findmaxabs(src);
        float log2_max_el;

        log2_max_el = log2(abs(max_el));

        // slenkstis - threshold
        thresh = two_x(floor(log2_max_el));
        // iteraciju kiekis
        it = ceil(log2_max_el);

        destroy_fifo();
        initialize_fifo();

        e = ezw_data_create(it, src);
        e->max_depth = max_depth;
        e->max_element = max_el;
        e->threshold = thresh;
        e->test = log2_max_el;


        copy = image_copy(src);
        // suformuojam iteracijas
        for (e->it = 0; e->it < e->iterations; ++(e->it)) {
```

```
                dominant_pass(e, copy, thresh);
                thresh >>= 1;
                subordinate_pass(e, thresh);
        }
        it_row_reset(e->coefs);


        image_destroy(copy);
        destroy_fifo();

        return e;

}

/* --- DECODE -------- */
static long int pixels;

int input_code(ezw_data_t* e, int count)
{
        it_row_t* dit = e->doms[e->it];

        if (count == 1) {
                it_row_t* sit = e->subs[e->it];
                return sit->data[sit->rid++];
        }

        return dit->data[dit->rid++];
}

/*
 * Builds a matrix element from dominant pass EZW-element and a threshold.
 */
void input_element(ezw_data_t* e, image_data_t* out,
        unsigned int thresh, unsigned int row, unsigned int col,
        ezw_element *s)
{
        xy_t d;
        d.x = col; d.y = row;
        s->x = col; s->y = row;
        s->code = input_code(e, 2);

        if ((s->code==EZW_POS)) {
                out->data[s->y][s->x] = (float)thresh;
                xy_row_append(e->xy, &d);
                //append_to_list(d);
        } else if ((s->code==EZW_NEG)) {
                out->data[s->y][s->x] = -1 * (float)thresh;
                xy_row_append(e->xy, &d);
                //append_to_list(d);
        }
}

void
b2_dom_pass(ezw_data_t* e, image_data_t* out, unsigned int thresh) {
        ezw_element s;
        int min_x, max_x, min_y, max_y;

        input_element(e,out,thresh, 0, 0, &s);
        if ((s.code==EZW_POS) || (s.code==EZW_NEG)) pixels++;

        input_element(e,out,thresh, 0, 1, &s);
        put_in_fifo(s);
        //fifo_push(fifo, &s);
```

```
        input_element(e,out,thresh, 1, 0, &s);
        put_in_fifo(s);
        //fifo_push(fifo, &s);

        input_element(e,out,thresh, 1, 1, &s);
        put_in_fifo(s);
        //fifo_push(fifo, &s);

        s = get_from_fifo();
        if (fifo_empty==0) {
                if ((s.code==EZW_POS) || (s.code==EZW_NEG)) pixels++;
        }

  while (fifo_empty==0) {
        if (s.code!=EZW_ZTR) {
                min_x = s.x << 1;
                max_x = min_x+1;
                min_y = s.y << 1;
                max_y = min_y+1;
                if ((max_x<=out->w) && (max_y<=out->h)) {
                        int r,c;
                        for (r=min_y; r<=max_y; r++) {
                                for (c=min_x; c<=max_x; c++) {
                                        input_element(e,out,thresh, r, c, &s);
                                        put_in_fifo(s);
                                }
                        }
                }
        }

        s = get_from_fifo();
        if (fifo_empty==0) {
                if ((s.code==EZW_POS) || (s.code==EZW_NEG)) pixels++;
        }
  }
}

void
s2_dom_pass(ezw_data_t* e, image_data_t* out, unsigned int thresh) {
        long int i;
        float temp;
        xy_t* d;
        char found;

        if (thresh == 0)
                return;

        for (i=0; i<e->xy->count; i++) {
                d = &e->xy->data[i];
                temp = out->data[d->y][d->x];
                if (input_code(e, 1)==EZW_ONE) {
                                if (temp<0) {
                                    out->data[d->y][d->x] = temp - thresh;
                                } else {
                                    out->data[d->y][d->x] = temp + thresh;
                                }
                }
        }
}

image_data_t*
ezw2_decode(ezw_data_t* e, unsigned int qual) {
        image_data_t* img = image_create(e->w, e->h);
        unsigned int iterations;
```

```
        unsigned int thresh;

        destroy_fifo();
        initialize_fifo();

        pixels = 0;
        xy_row_reset(e->xy);

        thresh = e->threshold;
        iterations = e->iterations * qual / 100;

        for (e->it = 0; e->it < iterations; ++(e->it)) {
                it_row_t* sit = e->subs[e->it];
                it_row_t* dit = e->doms[e->it];
                sit->rid = dit->rid = 0;

                b2_dom_pass(e, img, thresh);
                thresh >>= 1;
                s2_dom_pass(e, img, thresh);
        }
        destroy_fifo();
        initialize_fifo();
        return img;
}
```

failas coding.h

```
#ifndef CODING_H_INCLUDED
#define CODING_H_INCLUDED

#include "image.h"
#include "ezw.h"

#ifdef __cplusplus
extern "C"{
#endif

#define TRANS_NONE      0
#define TRANS_HT        1
#define TRANS_DB2       2

typedef struct transcode_data {
        image_data_t* trans1; // transform before encoding
        image_data_t* trans2; // transform after decoding
        char transName[32];

        float codeDuration;
        float decodeDuration;
        float transformDuration;

        image_data_t* result; // result image after transform + encode + decode +
inverse transform

        unsigned int quality;
        ezw_data_t* ezw_info;
} transcode_data_t;

transcode_data_t* transdata_create(void);
void transdata_destroy(transcode_data_t* d);
```

```
transcode_data_t* transcode(int type, const image_data_t* src, unsigned int qual);

#ifdef __cplusplus
}
#endif


#endif
```

failas coding.c

```
#include <alloc.h>
#include <stdlib.h>
#include <math.h>
#include <stdio.h>
#include <time.h>

#include "coding.h"
#include "ezw2.h"
#include "haar.h"
#include "d4.h"

#define D4_LEVELS       4
#define HT_LEVELS       4


unsigned long
laikas(void)
{
        unsigned long ms;
        ms = clock();
        return ms;
}

transcode_data_t*
transdata_create(void) {
        transcode_data_t* d = calloc(1, sizeof(transcode_data_t));
        return d;
}

void
transdata_destroy(transcode_data_t* d) {
        if (d == NULL)
                return;
        image_destroy(d->trans1);
        image_destroy(d->trans2);
        image_destroy(d->result);
        ezw_data_destroy(d->ezw_info);
        free(d);
}


// transform + EZW and back
transcode_data_t*
transcode(int type, const image_data_t* src, unsigned int qual) {
        transcode_data_t* out = transdata_create();
        int i, N;
        unsigned long start, end;

        // apply transform
```

```
        out->trans1 = image_copy(src);
        start = laikas();
        switch (type) {
        case TRANS_HT:
                sprintf(out->transName, "Haar (HT)");
                haar_transform(out->trans1, HT_LEVELS, 0);
                break;
        case TRANS_DB2:
                sprintf(out->transName, "Daubechies (db2)");
                d4_transform(out->trans1, D4_LEVELS, 0);
                break;
        default:
                sprintf(out->transName, "none");
        }
        end = laikas();
        out->transformDuration = end - start;
        // apply encode

        start = laikas();
        for (i=0; i< 1; ++i) {
                if (out->ezw_info) {
                        ezw_data_destroy(out->ezw_info);
                }
                out->ezw_info = ezw2_encode(out->trans1);
        }
        end = laikas();
        out->codeDuration = (end - start) ;
        // apply decode
#if 0
        out->trans2 = image_copy(out->trans1);
        out->result = image_copy(src);
#else
        start = laikas();
        N = 1;
        for (i=0; i < N; ++i) {
                out->trans2 = ezw2_decode(out->ezw_info, qual);
        }
        end = laikas();
        out->decodeDuration = (end - start) / N;

        out->result = image_copy(out->trans2);
        // detransform
        switch (type) {
        case TRANS_HT:
                haar_transform(out->result, HT_LEVELS, 1);
                break;
        case TRANS_DB2:
                d4_transform(out->result, D4_LEVELS, 1);
                break;
        }

#endif

        out->quality = qual;
        return out;
}
```

failas image.h

```
#ifndef IMAGE_H_INCLUDED
```

```
#define IMAGE_H_INCLUDED

#include <sys/types.h>
#ifdef __cplusplus
extern "C"{
#endif

typedef struct image_data {
        unsigned int w, h;
        float **data;
} image_data_t;

image_data_t* image_create(unsigned int width, unsigned int height);
image_data_t* image_copy(const image_data_t* src);
void  image_destroy(image_data_t* im);

float image_findmaxabs(const image_data_t* im);
void image_fill(image_data_t* im, int val);
void image_vflip(image_data_t* im);

typedef struct {
        float mse;
        float psnr;
} image_diff_t;

void image_diff(const image_data_t* src,
                const image_data_t* result, image_diff_t* diff);

#ifdef __cplusplus
}
#endif


#endif
```

failas image.c

```
#include <alloc.h>
#include <stdlib.h>
#include <mem.h>
#include <string.h>
#include <math.h>
#include "image.h"

image_data_t*
image_create(unsigned int width, unsigned int height) {
        unsigned int i;
        image_data_t* im = (image_data_t*)calloc(1, sizeof(image_data_t));
        if (im == NULL) {
                return NULL;
        }
        im->w = width;
        im->h = height;
        im->data = (float**)calloc(height, sizeof(float*));
        if (im->data == NULL) {
                free(im);
                return NULL;
        }
        for (i = 0; i < height; ++i) {
                im->data[i] = (float*)calloc(width, sizeof(float));
```

```
        }

        return im;
}

image_data_t*
image_copy(const image_data_t* src) {
        unsigned int i;
        image_data_t* im;

        if (src == NULL)
                return NULL;
        im = image_create(src->w, src->h);
        if (im == NULL)
                return NULL;

        for (i = 0; i < im->h; ++i) {
                memcpy(im->data[i], src->data[i], im->w * sizeof(float));
        }
        return im;
}

void
image_destroy(image_data_t* im) {
        unsigned int i;
        if (im == NULL)
                return;
        if (im->data != NULL) {
                for (i = 0; i < im->h; ++i) {
                        if (im->data[i] != NULL)
                                free(im->data[i]);
                }
        }
        free(im->data);
        free(im);
        return;
}

void
image_fill(image_data_t* im, int val) {
        unsigned int i, j;
        if (im == NULL)
                return;
        if (im->data != NULL) {
                for (i = 0; i < im->h; ++i) {
                        for (j = 0; j < im->w; ++j) {
                                im->data[i][j] = val;
                        }
                }
        }
        return;
}

float
image_findmaxabs(const image_data_t* im) {
        float maxval = 0;
        unsigned int i, j;
        for (i = 0; i < im->h; ++i) {
                for (j = 0; j < im->w; ++j) {
                        float val = abs(im->data[i][j]);
                        if (val > maxval) {
                                maxval = val;
                        }
                }
```

```
        }

        return maxval;
}

void
image_vflip(image_data_t* im) {
        unsigned int i;
        if (im == NULL)
                return;
        if (im->data == NULL)
                return;
        for (i = 0; i < im->h / 2; ++i) {
                float* x = im->data[i];
                im->data[i] = im->data[im->h - i -1];
                im->data[im->h - i - 1] = x;
        }
}

void
image_diff(const image_data_t* src, const image_data_t* dst, image_diff_t* d) {
        float sum = 0;
        unsigned int i, j;

        if (d == NULL)
                return;

        for (i = 0; i < src->h; ++i)
                for (j = 0; j < src->w; ++j) {
                        float a = dst->data[i][j] - src->data[i][j];
                        sum += a * a;
                }

        d->mse = sum / (src->w * src->h);
        d->psnr = 0;
        if (d->mse != 0) {
                d->psnr = 10 * log10(65025 / d->mse);
        }
}
```

failas haar.h

```
#ifndef HAAR_H
#define HAAR_H

#include "image.h"
#ifdef __cplusplus
extern "C"{
#endif

void haar_transform(image_data_t* img, int levels, int inverse);

#ifdef __cplusplus
}
#endif

#endif
```

failas haar.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "haar.h"

void haar_transform(image_data_t* img, int levels, int inverse)
{
float *line,*nextline;
int step,step2,x,y,ll_width;
int a,b,c,d,A,B,C,D;

          ll_width = img->w >> levels;

          if ( ! inverse ) {

                    for(step=1;step<=ll_width;step<<=1) {
                              step2 = step + step;
                              for(y=0;(y+step)<img->h;y += step2) {
                                  line = img->data[y];
                                  nextline = img->data[y + step];
                                        for(x=0;(x+step)<img->w;x += step2) {

                                                  a = line[x]; b = line[x+step];
                                                  c = nextline[x]; d =
nextline[x+step];

                                                  A = (a+b+c+d)>>2;
                                                  B = a-b-c+d;
                                                  C = a+b-c-d;
                                                  D = a-b+c-d;

                                                  line[x] = A; // will be coded in
next pass
                                                  line[x+step] = B;
                                                  nextline[x] = C;
                                                  nextline[x+step] = D;
                                        }
                              }
                    }

          } else {

                    /** go backwards in scale **/
                    for(step=ll_width;step>=1;step>>=1) {
                              step2 = step + step;

                              for(y=0;(y+step)<img->h;y += step2) {
                                  line = img->data[y];
                                  nextline = img->data[y + step];

                                        for(x=0;(x+step)<img->w;x += step2) {

                                                  A = line[x]; // already decoded.
                                                  B = line[x+step];
                                                  C = nextline[x];
                                                  D = nextline[x+step];

                                                  a = A + ((3 + B + C + D)>>2);
                                                  b = A + ((3 - B + C - D)>>2);
                                                  c = A + ((3 - B - C + D)>>2);
                                                  d = A + ((3 + B - C - D)>>2);
```

```
                                                  line[x]           = a;
                                                  line[x+step]      = b;
                                                  nextline[x]       = c;
                                                  nextline[x+step]= d;
                                        }
                              }
                    }
          }
}
```

failas db2.h

```
#ifndef D4_H
#define D4_H

#include "image.h"
#ifdef __cplusplus
extern "C"{
#endif

void d4_transform(image_data_t* img, int levels, int inverse);

#ifdef __cplusplus
}
#endif


#endif
```

failas db2.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include "d4.h"

#define SHIFT(x)   ((x)>>(12))

#define ROOT3      7094
#define ROOT2O2    2896
#define ROOT3O4    1774
#define ROOT3M2O4  (-275)

static void do_tdec_line(float * to, float *from,int len)
{
unsigned int x, half;
float *ptr,*low,*high;

//        if ( len <= 2 ) errexit("special-cased to len > 2 , scumbag");
//        if ( len&1 ) errexit("len shouldn't be odd");

          half = len>>1;

          ptr = from;        high = to + half;
```

```
        for(x=0;x<half;x++) {
                *high++ = ptr[1] - SHIFT( ROOT3 * (int)(ptr[0]) );
                ptr += 2;
        }

        ptr = from; low = to; high = to + half;
        for(x=0;x<half;x++) {
                if ( x == half-1 ) {
                        *low = ptr[0] + SHIFT( ROOT3O4 * (int)(high[0]) +
ROOT3M2O4 * (int)(high[0]) );
                } else {
                        *low = ptr[0] + SHIFT( ROOT3O4 * (int)(high[0]) +
ROOT3M2O4 * (int)(high[1]) );
                }
                if ( x == 0 )      *high += low[0] ;
                else                       *high += low[-1] ;
                ptr += 2; high++; low++;
        }
}

static void un_tdec_line(float *to,float *from,int len)
{
unsigned int x, half;
float *ptr,*low,*high;

        half = len>>1;

        ptr = to; low  = from;       high = from + half;

        low  = from;       high = from + half;
        for(x=0;x<half;x++) {
                if ( x == 0 )      *high -= low[0] ;
                else                       *high -= low[-1] ;
                high++; low++;
        }

        ptr = to; low  = from;       high = from + half;
        for(x=0;x<half;x++) {
                if ( x == half-1 ) {
                        ptr[0] = *low - SHIFT( ROOT3O4 * (int)(high[0]) +
ROOT3M2O4 * (int)(high[0]) );
                } else {
                        ptr[0] = *low - SHIFT( ROOT3O4 * (int)(high[0]) +
ROOT3M2O4 * (int)(high[1]) );
                }
                ptr += 2; high++; low++;
        }

        ptr = to; high = from + half;
        for(x=0;x<half;x++) {
                ptr[1] = (*high++) + SHIFT(ROOT3 * (int)(ptr[0]) );
                ptr += 2;
        }
}

void d4_transform(image_data_t* img, int levels, int inverse)
{
int x, y, w, h, l;
float *buffer,*tempbuf,*temprow;

       buffer = (float*)calloc(img->h * 3, sizeof(float*));
        temprow = buffer+img->h;
        tempbuf = buffer+img->h*2;
```

```
        if ( !inverse ) {
                for (l = 0; l < levels; l++) {
                     w = img->w >> l;
                     h = img->h >> l;
                         /* Rows */

                         do_tdec_line(temprow,img->data[h-1],w);
                         for (y = h-2; y >=0; y--) {
                                 do_tdec_line(img->data[y+1],img->data[y],w);
                         }

                         /* Columns */

                         for (x = 0; x < w; x++) {
                             for  (y  =  1;  y  <  h;  y++)  buffer[y-1]  =  img-
>data[y][x];

                             buffer[h-1] = temprow[x];
                                  do_tdec_line(tempbuf,buffer,h);
                                  for  (y  =  0;  y  <  h;  y++)  img->data[y][x]  =
tempbuf[y];
                         }
                }

     } else {
                for (l = levels-1; l >= 0; l--) { /** backwards in scale **/
                     w = img->w >> l;
                     h = img->h >> l;

                         /* Columns */

                         for (x = 0; x < w; x++) {
                             for  (y  =  0;  y  <  h;  y++)  buffer[y]  =  img-
>data[y][x];

                                  un_tdec_line(tempbuf,buffer,h);
                                  for  (y  =  0;  y  <  h-1;  y++)  img->data[y+1][x]
= tempbuf[y];

                                  temprow[x] = tempbuf[h-1];
                         }

                         /* Rows */
                         for (y = 0; y < h-1; y++) {
                                  un_tdec_line(img->data[y],img->data[y+1],w);
                         }
                         un_tdec_line(img->data[h-1],temprow,w);
                }
        }

        free(buffer);
}
```