LATVIJAS UNIVERSITĀTE
FIZIKAS UN MATEMĀTIKAS FAKULTĀTE
DATORIKAS NODAĻA

Elipšu atpazīšana attēlos

MAĢISTRA DARBS

Autors:
Gunārs Grundmanis
Studenta apliecība:
DatZ020009

Darba vadītājs:
LU docents Dr.sc.comp. Kārlis Freivalds

RĪGA 2008

Anotācija

Mūsdienās viena no aktuālākajām problēmām attēlu elektroniskā apstrādē un analīzē ir – pārveidot attēlu augstāka līmeņa objektos – taisnēs, riņķos, elipsēs utml. Šāda attēlu pārveidošana atvieglo tālāku attēlu analīzi un palīdz iegūt papildus informāciju par attēlu.

Maģistra darbā tiek aplūkota elipšu atpazīšana attēlos, tas ir, elipses atrašana un tās matemātiskā vienādojuma atrašana. Tiek apskatīti vairāki elipšu atpazīšanas algoritmi, to darbības principi. Liela daļa algoritmu balstās uz Hafa transformāciju, tāpēc dziļāk ir izpētīta elipšu atpazīšana ar Hafa transformācijas palīdzību. Darba ietvaros tika izstrādāta datorprogramma, kas spēj atpazīt elipses attēlos, izmantojot Hafa transformāciju.

Atslēgvārdi: attēlu apstrāde, kontūru atrašana, elipšu atpazīšana, Hafa transformācija, algoritmi.

Abstract

Nowadays, one of the most common problems in digital image processing and analysis is the transformation of digital image into higher level objects – lines, circles, ellipses and so on. Such transformation of digital image makes it easier to analyse the image further and helps to get additional information about the image.

In master thesis the ellipses' recognition in images is examined, i.e., finding contour of ellipse and its mathematical equation. Several ellipses' recognition algorithms are examined and their workings described. Many of them are based on Hough transform; that is why ellipses' recognition using Hough transform is more thoroughly investigated. A computer application, which can recognize ellipses using Hough transform, was developed.

Keywords: image processing, edge detection, ellipse detection, Hough transform, algorithms.

Autoreferāts

Autora paveiktais:

- izpētīja populārākos kontūru atrašanas algoritmus un to darbības principus,
- izpētīja pazīstamākos elipšu atpazīšanas algoritmus,
- veica populārāko elipšu atpazīšanas algoritmu salīdzināšanu,
- uzlaboja Hafa transformāciju, lai to būtu praktiski iespējams pielietot reālu attēlu apstrādē,
- izstrādāja datorprogrammu (700 koda rindiņas), kas spēj reālos attēlos atpazīt elipses.

Saturs

IEVADS

Bieži vien ir nepieciešams analizēt digitālos attēlus un meklēt tajos elipses un riņķus. Viens no šādiem piemēriem ir automatizēta šautuves sistēma, kas reālā laikā skaitītu iegūtos punktus. Šādas programmas izstrādei ir nepieciešams ātrs un precīzs elipšu atpazīšanas algoritms, jo nofotogrāfēto vai arī nofilmēto mērķi vajag sadalīt vērtību joslās. Pēc tam izanalizēt, kurā joslā šāviens ir trāpījis un cik daudz punktu par to pienākas.

Darba galvenais mērķis bija izstrādāt datorprogrammu, kas spētu atpazīt šautuves mērķu fotogrāfijās elipses. Lai šo mērķi sasniegtu, darba gaitā tika izpētīti vairāki kontūru atrašanas algoritmi, kā arī dažādas elipšu atpazīšanas metodes. Datorprogrammā ir izmantota Hafa transformācija, tāpēc tika dziļāk izpētīti dažādi uz Hafa transformāciju bāzēti algoritmi.

Datorprogrammas izstrādes laikā, padziļināti tika pētīti uz Hafa transformāciju bāzēti algoritmi, kā rezultātā tika izstrādātas divas uzlabotas Hafa transformācijas metodes. Šīs metodes radās, apvienojot dažādu algoritmu daļas. Tā varēja darīt, jo uzlabotajās metodēs darbība sastāv it kā no diviem soļiem – elipses centra atrašanas un elipses parametru atrašanas. Datorprogrammā tika izstrādāti divi algoritmi elipšu centru atrašanai un viens Hafa transformācijas variants, bez centru meklēšanas.

Rezultātā tika iegūta datorprogramma, kas reāliem attēliem spēj noteikt kontūru līnijas ar Sobela operatora palīdzību. Pēc tam no atrastajiem kontūru punktiem tiek atrasti iespējamie elipšu centri, kurus izmantojot ar Hafa transformācijas palīdzību tiek atrastas visas attēlā esošās elipses.

Maģistra darbs sastāv no trīs nodaļām – teorijas apskata, datorprogrammas apraksta un rezultātu analīzes – un pielikuma. Pielikumā ir izstrādātās datorprogrammas kods programmēšanas valodā c++.

Teorijas apskatā tika izpētīti populārākie kontūru atrašanas algoritmi un to darbības principi. Apskatītas populārākās elipšu atpazīšanas metodes, uzsvaru liekot uz metodēm, kas balstītas uz Hafa transformāciju. Tika izanalizēti algoritmu darbības principi un veikta dažādu elipšu atpazīšanas algoritmu salīdzināšana.

Datorprogrammas apraksts sevī ietver visu trīs izstrādāto algoritmu aprakstus un galvenās idejas, uz kurām balstās programmas kods. Katra izstrādātā metode ir soli pa solim izskaidrota. Pie kam sarežģītākā metode ar centru meklēšanu, izmantojot ģeometrisko simetriju, ir demonstrēta ar rezultātiem, kas ir iegūti pēc katra algoritma soļa.

Praktiskās daļas ietvaros tika apstrādāti divu veidu attēli. Pirmā veida attēli bija paša ģenerēti, izmantojot zīmēšanas programmu *Paint*. Tas tika darīts, lai izpētītu Hafa transformācija stiprās un vājās puses, kā rezultātā optimizējot un pilnveidojot programmu. Tika pētīta iegūto rezultātu atkarība no dažādām Hafa transformācijas parametru vērtībām, kā rezultātā tika iegūtas vairākas vērtīgas atziņas. Apstrādājot reālos attēlus, tika parādīts, ka Hafa transformācija ir pietiekoši spēcīgs līdzeklis, lai reālos attēlos atrastu elipses un riņķus.

1. TEORIJAS APSKATS

1.1. Kontūru atrašana

Jebkura attēla apstrāde sākas ar kontūru atrašanu. Pēc tam, balstoties uz atrastajām kontūrām, var veikt tālāku attēlu analīzi – meklēt taisnes, riņķus, elipses u. c. ģeometriskus objektus.

Liela daļa kontūru atrašanas algoritmu balstās uz vienkāršiem konvolūciju filtriem, jo tas ir samērā ātrs un vienkāršs process. Tomēr sarežģītākos gadījumos, lai iegūtās kontūras būtu pēc iespējas precīzākas, jāveic papildus attēla apstrāde pirms konvolūcijas filtru pielietošanas, un arī pēc tās atrastās kontūras vajag apstrādāt papildus.

Turpmāk darbā apskatīšu vienkāršākās metodes kontūru atrašanai, izmantojot Sobela un Laplasa operatorus. Kā arī Canny piedāvāto kontūru atrašanas metodi, kas izmanto Sobel operatoru, bet veic vēl papildus attēla sagatavošanu un pēcapstrādi, tā uzlabojot iegūtos rezultātus.

1.1.1. Sobela un Laplasa operatori.

Ir daudz veidu, kā atrast kontūras attēlos, tomēr lielāko daļu algoritmu var iedalīt divās grupās – uz gradientu balstītās metodes un uz laplasiāni (zināmu arī kā Laplasa operatoru) balstītās metodes. Visas plašāk lietotās metodes ir aprakstītas attēlu apstrādes rokasgrāmatā [1], bet es tālāk darbā apskatīšu divas populārākās metodes, no kurām viena balstās uz pirmo atvasinājumu, bet otra uz otro atvasinājumu.

Uz gradientu balstītās metodes meklē kontūras, skatoties uz attēla intensitātes pirmā atvasinājuma minimuma un maksimuma vērtībām. Bet metodes, kurās izmanto Laplasa operatoru, meklē punktus, kuros attēla intensitātes otrais atvasinājums pieņem vērtību „0”, jo punktos, kuros attēla intensitātes pirmais atvasinājums pieņem minimālo vai maksimālo vērtība, tā otrais atvasinājums

Sobela operators.

Sobela operators ir viens no visplašāk lietotajiem filtriem gradienta, jeb pirmā atvasinājuma, atrašanai. Sobela operators sastāv no diviem filtriem, viens atrod gradientu x ass virzienā, otrs y ass virzienā. Sobela operatora svaru maskas redzamas zemāk 1. attēlā.

-1	0	+1
-2	0	+2
-1	0	+1

Gx

+1	+2	+1
0	0	0
-1	-2	-1

Gy

1.1. att. **Sobela operatora svaru maskas**

Šie filtri ir veidoti tā, lai maksimāli labi reaģētu uz kontūrām, kas iet horizontālā un vertikālā virzienā attiecībā pret pikseļu režģi. Pielietojot katru no augstāk redzamajiem filtriem, var iegūt gradienta vērtības Gx un Gy katras ass virzienā. Lai aprēķinātu gradienta absolūto vērtību, varam izmantot sekojošu formulu:

$$|G| = \sqrt{Gx^2 + Gy^2} \qquad (1.1)$$

Parasti, lai samazinātu aprēķinu ilgumu, tiek izmantots šāds tuvinājums:

$$|G| = |Gx| + |Gy| \qquad (1.2)$$

Kontūra līnijasd virzienu var aprēķināt no Gx un Gy vērtībām pēc šādas formulas:

$$\theta = \arctan\left(\frac{Gy}{Gx}\right) \qquad (1.3)$$

Kad gradientu vērtības ir atrastas, tad atliek izvēlēties minimālo gradienta vērtību, lai konkrēto punktu pasludinātu par kontūras punktu attēlā. Tā apskatot visas iegūtās gradientu vērtības mēs iegūstam kontūru punktus attēlā.

Laplasa operators.

Laplasiānis ir attēla intensitātes otrais atvasinājums, kuru formāli pierakstām šādi:

$$L(x,y) = \frac{\partial^2 I}{\partial^2 x} + \frac{\partial^2 I}{\partial^2 y}. \qquad (1.4)$$

Laplasiānis izceļ attēla apgabalus, kuros ir strauja intensitātes maiņa, kas ļauj to izmantot kontūru atrašanai attēlos. Tā kā attēls tiek attēlots kā diskrētu pikseļu kopa, tad mums vajag atrast diskrētu konvolūciju filtru, kas apraksta attēla intensitātes otro atvasinājumu. Zemāk 2. attēlā redzamas trīs visbiežāk lietoto Laplasa operatoru svaru maskas.

0	1	0
1	−4	1
0	1	0

1	1	1
1	−8	1
1	1	1

−1	2	−1
2	−4	2
−1	2	−1

1.2. att. **Laplasa operatora svaru maskas**

Laplasa operators ir jūtīgs pret troksni, tāpēc, lai cīnītos pret trokšņa radītajām neprecizitātēm, pirms tam parasti tiek lietots kaut kāds gludinošs filtrs, piemēram, Gausa gludināšanas filtrs. Tā kā konvolūciju operācijas ir asociatīvas, tad mēs varam aprēķināt Laplasa un Gausa operatoru konvolūciju (apzīmēsim šo filtru ar LoG) un pēc tam izmantot jauniegūto operatoru. Tas mums palīdz ietaupīt aprēķinu laiku, jo divu filtru vietā pietiek ar vienu. 3. attēlā redzama LoG filtra diskrēta aproksimācija ar Gausa standartnovirzi Sigma = 1.4.

0	1	1	2	2	2	1	1	0
1	2	4	5	5	5	4	2	1
1	4	5	3	0	3	5	4	1
2	5	3	-12	-24	-12	3	5	2
2	5	0	-24	-40	-24	0	5	2
2	5	3	-12	-24	-12	3	5	2
1	4	5	3	0	3	5	4	1
1	2	4	5	5	5	4	2	1
0	1	1	2	2	2	1	1	0

1.3. att. **LoG operatora svaru maska**

1.1.2. Canny metode kontūru atrašanai.

Canny kontūru atrašanas algoritms daudziem ir zināms kā optimālais šķautņu noteicējs. Kā galveno savu uzdevumu John F. Canny izvirzīja jau esošo kontūru meklēšanas algoritmu

uzlabošanu. Tā rezultātā viņš nonāca pie tālāk aprakstītās metodes kontūru noteikšanai, šo metodi viņš aprakstīja savā publikācijā žurnālā „Pattern Analysis and Machine Intelligence" [2].

Uzlabojot jau esošos algoritmus, viņš pievērsa uzmanību trīs problēmām. Pirmkārt vajadzēja novērst kļūdas: kādas kontūras neatrašanu, vai gluži otrādi – atrast kontūru, kur attēlā tās nemaz nav. Otra lieta, kurai Canny pievērsa vērību, bija, lai visas kontūras tiktu atrastas precīzi, t.i., lai atrastā kontūras pikseļi būtu pēc iespējas tuvāk attēlā redzamajai kontūrai. Un treškārt viņš vēlējās panākt, lai katrai kontūrai attēlā tiktu atrasta ne vairāk kā viena kontūra, jo pirmie divi nosacījumi nenovērš iespēju, ka vienai attēla objektam var atrast divas kontūras.

Lai novērstu katru no iepriekš minētajām problēmām, Canny piedāvātajā metodē tika ieviesti vairāki papildus soļi. No sākuma attēls tiek nogludināts, lai novērstu kļūdainas kontūras. Tad tiek aprēķināts gradients visiem attēla punktiem, izmantojot Sobela operatoru. Pēc tam tiek atrasti pikseļi caur kuriem varētu iet kontūras līnijas, balstoties uz gradientu virzieniem. Atrastajiem punktiem tiek veikta kontūru meklēšana, izmantojot divus gradienta moduļa sliekšņus.

Tagad apskatīsim visus Canny algoritma soļus mazliet detalizētāk.

1. Solis.

Pirmais solis ir nofiltrēt visu nevajadzīgo troksni. Canny izvēlējās Gausa gludinošo filtru, jo to var pielietot ar vienas vienkāršas maskas palīdzību. Kad piemērota maska ir izrēķināta, tad to ir iespējams pielietot kā parastu konvolūciju filtru. Gausa filtra maskas izmēri var būt dažādi, jo maska būs lielāka, jo algoritmam būs mazāka jūtība pret trokšņiem. Attēlā 2.4 redzama 5*5 Gausa filtra maska ar standartnovirzi $\sigma = 0.4$.

$$\frac{1}{159}\begin{bmatrix} 2 & 4 & 5 & 4 & 2 \\ 4 & 9 & 12 & 9 & 4 \\ 5 & 12 & 15 & 12 & 5 \\ 4 & 9 & 12 & 9 & 4 \\ 2 & 4 & 5 & 4 & 2 \end{bmatrix}$$

2.4. att. **Gausa filtra maska**

2. solis.

Pēc attēla nogludināšanas vajag atrast gradientu visiem attēla punktiem. To izdarām ar Sobela operatora palīdzību, tāpat kā tas tika aprakstīts iepriekšējā apakšnodaļā.

3. solis.

Nākamais solis ir kontūras līnijas virziena noteikšana. Arī šeit rīkojamies, kā jau iepriekš aprakstīts – gradienta virzienu aprēķinām pēc formulas (1.3).

4. solis.

Kad kontūras līnijas virziens ir zināms, tad to vajag pielāgot virzienam, kuram var izsekot līdzi attēlā, t.i., leņķis θ ir jānoapaļo līdz tuvākajiem 45^0 (0^0, 45^0, 90^0 vai 135^0).

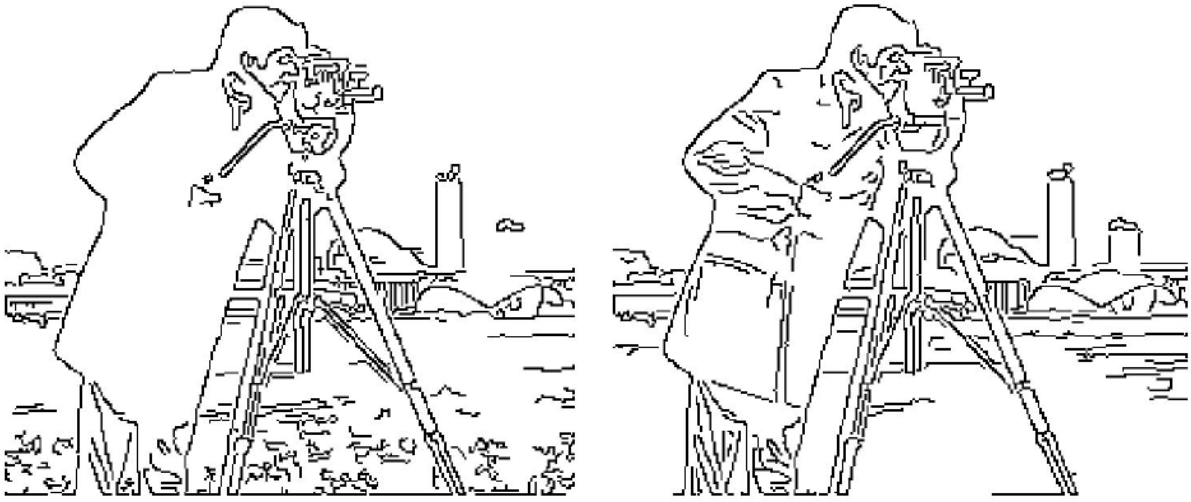
5. solis.

Kad jaunie kontūru virzieni ir zināmi, ejam pa visām kontūru līnijām iepriekš aprēķinātajā virzienā un visiem pikseļiem, kuri netiek uzskatīti par kontūras līniju, piešķiram vērtību 0. Rezultātā mēs būsim ieguvuši tievu līniju rezultāta attēlā.

6. solis.

Visbeidzot mums atliek noskaidrot, kuras no atrastajām līnijām ir īstas kontūras un kuras nē. Lai to izdarītu, mēs vēlamies atrast līnijas ar vidējo gradienta vērtību lielāku par sliekšņa vērtību. Dēļ trokšņa katra atsevišķā pikseļa gradienta vērtība var svārstīties gan virs sliekšņa vērtības, gan zem tās. Lai novērstu šo problēmu, lietosim divas sliekšņa vērtīas T1 > T2. Jebkurš pikselis ar gradienta vērtību lielāku par T1 uzreiz tiek pasludināts par kontūras līnijas punktu. Visi pikseļi, kas ir savienoti ar kādu kontūras līnijas pikseli un to gradientu vērtības ir lielākas par T2, arī tiek pasludināti par kontūras līnijas punktiem. Ja tiek lietota metode, kurā virzāmies pa jau 5. solī atrastajām kontūrām, tad lai sāktu līniju, mums ir nepieciešams gradients lielāks par T1, bet kontūras meklēšanu mēs turpinām, līdz sasniedzam pikseli ar gradientu mazāku par T2.

Ļoti līdzīgs Canny algoritmam, ir Peter Meer un Bogdan Georgescu piedāvātais algoritms [3]. Arī šis algoritms balstās uz tādiem pašiem trīs galvenajiem soļiem – gradienta noteikšana, nemaksimumu nomākšana un mērogošana. Autori savu piedāvāto algoritmu ir papildinājuši ar uzticēšanās varbūtības mērījumiem, kā rezultātā, viņi ieguva algoritmu, kurš spēj atpazīt ļoti vājas kontūras, kā arī neatpazīst daduz liekas līnijas. Canny un Peter Meer un Bogdan Georgescu piedāvāto algoritmu salīdzinājumu var redzēt 2.5. attēlā (attēls iegūts no Peter Meer un Bogdan Georgescu publikācijas [3]). Augšā redzams orģinālais attēls, pa kreisi ar Canny algoritmu iegūtās kontūras un pa labi ar Peter Meer un Bogdan Georgescu piedāvāto algoritmu atrastās kontūras. Varam redzēt, ka Canny algoritms ļoti labi tiek galā ar kontūrām, kuras var viegli izdalīt. Tur pretī Peter Meer un Bogdan Georgescu piedāvātais algoritms ne tikai atpazīst ļoti vājas kontūras, bet arī neatpazīst daudz liekas līnijas, kuras atpazina Canny algoritms.



2.5. att. **Ar Canny un Peter Meer un Bogdan Georgescu algoritmiem iegūtās kontūru līnijas**

1.2. Hafa transformācija

Viena no populārākajām metodēm dažādu līniju un līkņu atpazīšanā attēlos ir Hafa transformācija. Tās pamata ideja balstās uz P. V. C. Hough patentu [4], kurā ir aprakstīta metode, kā attēlos atpazīst sarežģītus tēlus. Vēlāk šo metodi nedaudz sīkāk un vieglāk saprotami aprakstīja Richard O. Duda un Peter E. Hart savā publikācijā „Hafa transformācijas izmantošana līniju un līkņu atrašanai attēlos" [5]. Viņi arī bija pirmie, kuri šo metodi nodēvēja par Hafa transformāciju.

Sākotnēji Hafa transformācija bija paredzēta taisnu līniju atpazīšanai. Tā balstījās uz visu līnijas punktu pārveidošanu uz parametru telpu. Šī parametru telpa ir definēta ar līnijas parametrisko reprezentāciju attēlā. Hafs savā patentā taisnu līniju aprakstīšanai izmantoja vienādojumu $y = ax + b$, tāpēc viņam bija divu dimensiju parametru telpa – viena dimensija parametram a un otra parametram b.

Ja mēs apskatām patvaļīgi izvēlētu taisni, tad to viennozīmīgi var aprakstīt ar šiem parametriem a un b, tātad taisne atbilst vienam punktam parametru telpā. Galvenā ideja arī balstās uz šāda punkta atrašanu, kurš apraksta konkrētu taisni.

Tātad, ja mums ir n punkti $\{(x_1, y_1), ..., (x_n, y_n)\}$ un mēs gribam noskaidrot taisnes, kuras atbilst šai punktu kopai, tad katru punktu (x, y) transformējam par taisni $b = y - ax$ parametru telpā. Patiesībā šī taisne apraksta visas tās taisnes, kuras iet caur punktu (x, y) attēlā. Kad visiem punktiem tiem atbilstošās taisnes ir atrastas, tad meklējam to krustpunktus(ja n taisnes parametru telpā krustojas punktā (a, b), tas nozīmē, ka attēlā ir n punkti, kuri atrodas uz taisnes $y = ax + b$). Tātad galvenā problēma ir atrast punktu, caur kuru iet visvairāk taisnes.

Viens no vienkāršākajiem veidiem kā atrast krustpunktus ir izmantot divu dimensiju masīvu ar skaitītājiem, kur viena dimensija atbilst parametram a, otra parametram b. Grūtākais uzdevums ir izvēlēties parametru minimālās un maksimālās vērtības, kā arī soli starp masīva blakus šūnām. Jo vairāk šūnas masīvā, jo precīzāk varēs atrast līnijas, toties ilgāks darbības laiks. Kad skaitītāju masīvs ir izveidots, tad atliek to aizpildīt. Katram punktam aizpildām tam atbilstošās šūnas – ejam cauri visām parametra a vērtībām(no minimālās līdz maksimālajai ar izvēlēto soli) un aprēķinām parametra b vērtību. Ja parametri a un b atbilst kādai divu dimensiju masīva šūnai(ir robežās no minimālās vērtības līdz maksimālajai), tad šīs šūnas vērtību palielinām par viens. Rezultātā mēs iegūsim tabulu, kur katrā šūnā ir punktu skaits, kuri atrodas uz taisnes, kura atbilst šīs šūnas parametriem.

Kad tabula ir iegūta, atliek tikai atrast šūnas, kurās ir pietiekošs skaits punktu, tad arī šai šūnai atbilstošie parametri ir atrastās taisnes parametri. Šī metode ir izmantojama arī sarežģītāku līniju un objektu meklēšanā, galvenais, ka šos objektu var aprakstīt ar matemātisku vienādojumu. Sarežģītāku līniju gadījumā vienkārši parametru telpai būs vairāk dimensiju un tabulas aizpildīšanas procedūra būs krietni sarežģītāka.

Vairāku dimensiju gadījumā, parasti, tiek izmantotas optimizētas vai uzlabotas Hafa transformācijas metodes. Pazīstamākās no tām ir gadījuma Hafa transformācija, varbūtiskā Hafa transformācija. Speciāliem gadījumiem, kā piemēram elipses gadījumā, var izveidot īpašu uzlabotu Hafa transformāciju, kas labi darbojas tikai konkrētā gadījumā. Viens no šādiem algoritmiem priekš elipses ir Hafa transformācija, kas balstīta uz ģeometrisko simetriju. Tālāk darbā apskatīsim šīs dažādās Hafa transformācijas.

1.2.1. Hafa transformācija elipses gadījumā.

Lai veiktu Hafa transformāciju elipsei, vispirms ir jāizvēlas elipses parametrizētais vienādojums, uz kura pamata izveidosim parametru telpu. Elipses gadījumā ir piecu dimensiju parametru telpa. Kādi precīzi ir šie parametri, var redzēt šajā elipses vienādojumā:

$$\frac{((x-x_c)\cos\Phi-(y-y_c)\sin\Phi)^2}{a^2}+\frac{((x-x_c)\sin\Phi+(y-y_c)\cos\Phi)^2}{b^2}=1, \qquad (1.5)$$

kur pieci elipses parametri apraksta sekojošus lielumus – (x_c, y_c) elipses centra koordinātas, (a, b) elipses lielā un mazā pusass, Φ elipses pagrieziena leņķis.

Tā kā elipses vienādojumā ir pieci parametri, tad Hafa transformācijai nepieciešama piecu dimensiju skaitītāju tabula. Piecu dimensiju tabula aizņem diezgan daudz vietu, kā arī aprēķini būs lēni, ja tabula būs pārāk liela, tāpēc uzmanīgi jāizvēlas katra parametra robežas un solis. Tā kā mūsdienās parasti tiek apstrādāti digitālie attēli, kuros mazākā vienība ir pikselis, tad priekš centra un pusass parametriem par soli loģiski ir ņemt vienu pikseli. Pagrieziena leņķim Φ nepieciešams apskatīt vērtības no 0^0 līdz 45^0, jo elipse ir simetriska pret tās centru un atrastajām elipsēm lielā pusass var būt gan vertikāli, gan horizontāli. Ja ātrdarbība ir pietiekoši laba, tad par soli var izvēlēties vienu grādu.

Kad parametru telpa un skaitītāju tabula ir precizēta, atliek apskatīt procedūru attēla punktu transformēšanai uz parametru telpu. Vienkāršākais veids ir veidot vairākus ciklus, kuros ejam

cauri visām pieļaujamajām parametru a, x_c, y_c, Φ vērtībām un aprēķinām parametru b. No vienādojuma (1) izsakām parametru b, iegūta izteiksme ir šāda:

$$b = \sqrt{\frac{a^2 \cdot ((x - x_c)\sin\Phi + (y - y_c)\cos\Phi)^2}{a^2 + ((x - x_c)\cos\Phi - (y - y_c)\sin\Phi)^2}}. \qquad (1.6)$$

Ja atrastais parametrs b atbilst mūsu prasībām, tad skaitītāju tabulā šiem parametriem atbilstošo skaitītāju palielinām par viens. Šādu procedūru veicam visiem attēla punktiem.

Šī parametru atrašanas procedūra ir samērā sarežģīta, un tā kā mums ir pieci parametri, tad kopējas Hafa transformācijas darbības ilgums ir samērā liels. Tāpēc rodas vēlme kaut kādā veidā samazināt dimensiju skaitu. Viens variants būtu šāds – no sākuma atrodam iespējamos elipšu centrs un tikai pēc tam, katram no šiem centriem veicam Hafa transformāciju.

Rodas aktuāls jautājums, kā atrast elipses centru. Vienkāršākais risinājums balstās uz faktu, ka elipse ir simetriska pret tās centru. Līdzīgi kā Hafa transformācijā, mēs izveidojam skaitītāju tabulu, lai atrastu populārākos iespējamos elipšu centrus. Ņemam visus iespējamos punktu pārus un uzskatām, ka tie ir pretējie elipses punkti. Saskaņā ar elipses simetriju, tās centrs atrodas šo divu punktu savienojošā nogriežņa viduspunktā, kuru var viegli aprēķināt. Pēc tam mēs šim punktam atbilstošo skaitītāju palielinām par viens. Tā kā mums ir zināmi pusasu izmēru ierobežojumi, mēs arī zinām minimālo un maksimālo pieļaujamo attālumu starp pretējiem elipses punktiem. To mēs izmantojam, lai nebūtu jāpārbauda punkti, starp kuriem attālums ir pārāk mazs vai pārāk liels. Kad visi pāri ir apskatīti, tad atrodam tās šūnas, kurās ir pietiekoši liels skaits un uzskatām tos par iespējamajiem elipšu centriem.

Ja elipses centrs ir zināms, tad elipses vienādojumu var pārveidot formā:

$$\frac{(x\cos\Phi - y\sin\Phi)^2}{a^2} + \frac{(x\sin\Phi + y\cos\Phi)^2}{b^2} = 1. \qquad (1.7)$$

Šādā gadījumā mums ir tikai trīs parametri - abas pusasis a un b un pagrieziena leņķis Φ. Tātad arī skaitītājiem nepieciešama trīs dimensiju tabula, kas ievērojami samazina patērēto atmiņu. Veicot Hafa transformāciju, darbojamies līdzīgi kā iepriekšējā gadījumā – apskatām visas iespējamās pusass a un pagrieziena leņķa Φ vērtības un aprēķinām otru pusasi b pēc šādas formulas:

$$b = \sqrt{\frac{a^2 \cdot (x\sin\Phi + y\cos\Phi)^2}{a^2 + (x\cos\Phi - y\sin\Phi)^2}}. \qquad (1.8)$$

Ja ir atrasti vairāki iespējamie elipses centri, tad katram punktam veicam Hafa transformāciju, tieši tāpat kā iepriekšējā gadījumā, tikai ar vienu izņēmumu – piecu dimensiju

vietā tagad mums ir trīs. Kad iegūta skaitītāju tabula, tad tāpat kā iepriekšējā gadījumā meklējam šūnas ar pietiekošu skaitu un attiecīgajiem parametriem atbilstošā elipse ir ar Hafa transformāciju atrastā elipse. Kad visi iespējamie centri ir apskatīti, darbu beidzam.

1.2.2. Uz ģeometrisko simetriju balstīta Hafa transformācija

Šīs metodes pamatā ir ideja daudzdimensionālu problēmu sadalīt divās divdimensionālās problēmās. Tās pamatā ir jauna metode, kā no attēla iegūt simetrijas asis. Šo metodi var apskatīt Yiwu Lei un Kok Cheong Wong rakstā [6].

Parastajā Hafa transformācijā tiek izmantota informācija tikai par vienu pikseli, tā atrašanās vietu un izliekumu. Bet, ja mēs apskatām visus pikseļu pārus, tad varam iegūt vairāk noderīgu informāciju. Šajā metodē elipses simetrijas ass atrašanai arī izmantosim inforāciju no visiem punktu pāriem. Tā kā mēs apskatīsim punktu pārus, tad mums nav nepieciešams meklēt pieskares vai izliekumu, jo šāds process ir jūtīgs pret troksni attēlos.

Lai atrastu elipses simetrijas asis, mēs katram punktu pārim atradīsim tos savienojošās līnijas viduspunktu un pēc tam taisni, kas ir perpendikulāra šai līnijai un iet caur tās viduspunktu. Taisnes aprakstīšanai, kas iet caur divus pikseļus savienojošās līnijas viduspunktu (x_m, y_m) izmantosim šādu taisnes vienādojumu:

$$s = x_m \cos\theta + y_m \sin\theta . \qquad (1.9)$$

Lai atrastu elipses simetrijas asis, konstruēsim līdzīgu skaitītāju tabulu, kā Hafa transformācijai, priekš abiem taisnes parametriem s un θ. Pēc tam atradīsim populārākās vērtības, kas tad arī būs mūsu meklētās elipses simetrijas asis. Detalizētāks simetrijas asu atrašanas algoritma apraksts aplūkojams zemāk. Šajā algoritma aprakstā ar D_{min} apzīmēts minimālais nepieciešamais attālums starp aplūkojamajiem punktiem un ar ACC[θ][s] skaitītāju masīvs parametru telpai (s, θ).

Simetrijas asu atrašanas algoritms:

1. Notīrām skaitītāju masīvu ACC;
2. Katram attēla punktam (x_1, y_1) izpildām soļus 3 līdz 6;
3. Katram no atlikušajiem punktiem (x_2, y_2), ja $|x_2 - x_1| > D_{min}$ vai $|y_2 - y_1| > D_{min}$, izpildām soļus 4 līdz 6;

4. No šiem diviem pikseļiem (x_1, y_1) un (x_2, y_2) atrodam taisni, kas iet caur to savienojošā nogriežņa viduspunktu. Taisnes parametrus var atrast sekojoši:

$$\theta = \arctan\left(\frac{y_2 - y_1}{x_2 - x_1}\right), \qquad (1.10)$$

$$s = \frac{\cos\theta \times (x_1 + x_2) + \sin\theta \times (y_1 + y_2)}{2}. \qquad (1.11)$$

5. Palielinām skaitītāju masīvā atrastajiem parametriem s un θ atbilstošo elementu par vērtību $\frac{1}{L}$, kur L var aprēķināt sekojoši:

$$L = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}. \qquad (1.12)$$

6. Turpinām līdz visi punktu pāri ir apskatīti.
7. Sameklējam visus nozīmīgos maksimumus skaitītāju masīvā ACC. Katram atrastajam maksimumam atbilstošā taisne ar parametriem s un θ arī ir iespējamā elipses simetrijas ass.
8. Darbu beidzam.

Simetrijas asu meklēšanā tuvi punkti netiek apskatītu divu iemeslu dēļ. Centrālās ass atrašana tuviem punktiem in jūtīga pret troksni un neprecīza. Kā arī aprēķinu laiks ir ievērojami mazāks, ja nav jāapskata visu punktu pāri.

Tagad apskatīsim, kā no atrastajām elipses simetrijas asīm atrast tās piecus parametrus - (x_c, y_c) elipses centra koordinātas, (a, b) elipses lielā un mazā pusass, α elipses pagrieziena leņķi. Tā kā atrastās simetrijas asis atbilst gan mazajai pusasij, gan lielajai, tad mums vajag atrast visas savstarpēji perpendikulārās taisnes, kas arī būs abas elipses simetrijas asis. Protams, var tikt atrastas arī nederīgas kombinācijas, taču tām netiks atrastas elipses, izmantojot Hafa transformāciju. Kad elipses asis ir atrastas, atliek vienu no asīm izvēlēties kā elipses orientāciju. Tad atrodam elipses centru (x_c, y_c) un pagrieziena leņķi α. Tagad atliek tikai atrast pārējos divus parametrus – abu pusasu garumus.

Tagad apskatīsim algoritmu abu pusasu garumu atrašanai. Ar D_{min} atzīmēsim minimālo nepieciešamo punktu attālumu līdz centram un ar ACC[a][b] skaitītāju masīvu Hafa transformācijai elipses pusasu garumiem a un b. Elipses pusasu garumus var atrast, izmantojot sekojošu algoritmu:

1. Notīrām skaitītāju masīvu ACC;
2. Katram attēla punktam (x, y) izpildām soļus 3 līdz 6;

3. Punktam veicam translācijas un rotācijas transformāciju, lai tas atbilstu nepagrieztai elipsei ar centru punktā (*0*, *0*):

$$x_1 = (x - x_c) \times \cos\alpha + (y - y_c) \times \sin\alpha\,, \qquad (1.13)$$

$$y_1 = -(x - x_c) \times \sin\alpha + (y - y_c) \times \cos\alpha\,. \qquad (1.14)$$

4. Aprēķinām visas iespējamās *a* un *b* vērtības, izmantojot vienādojumu

$$b = \frac{|y_1|}{\sqrt{1 - \frac{x_1^2}{a^2}}}\,. \qquad (1.15)$$

5. Atrastajiem parametriem *a* un *b* atbilstošo elementu palielinām par vērtību $\frac{1}{\sqrt{a+b}}$.
6. Turpinām, līdz visi punkti ir apskatīti;
7. Sameklējam visus nozīmīgos maksimumus skaitītāju masīvā ACC. Ja tādi tika atrasti, tad iegūtie parametri arī atbilst elipsei, bet, ja netika atrasti, tad izmantotais simetrijas asu pāris bija kļūdains;
8. Darbu beidzam.

1.2.3. Varbūtiskā Hafa transformācija

Varbūtiskā Hafa transformācija balstās uz ideju, ka Hafa transformācijai izmantotajā skaitītāju tabulā mūs interesē tikai tās vietas, kur ir maksimumi. Tas ir, tajās vietās, kur Hafa transformācijas vērtības ir mazas, mums nav nepieciešams iegūt labu aproksimāciju. Šo atziņu tad savā algoritmā arī izmanto J. R. Bergens un H. Shvaytser [7].

Hafa transformācijas aproksimāciju (varbūtisko Hafa transformāciju) m punktiem definē ar trīs parametriem ε, δ un μ. Parametrs μ ir izmēra parametrs, tas ir, daļa no m, kas ir pārāk maza, lai uzskatītu par vērā ņemamu vērtību. Parametri ε un δ ir konfidences un precizitātes parametri. Zīmīgo vērtību aproksimācijas kļūdu ierobežo ar pieļaujamo kļūdu ε ar uzticēšanās varbūtību vismaz $1 - \delta$. Formālā definīcija ir šāda:

Definīcija. Ja mums ir dota kopa C ar m punktiem un $H(\phi)$ ir punktu kopas m Hafa transformācija parametru vektoram ϕ ar dimensiju skaitu *w*. Tad parametriem $0 \le \varepsilon, \delta, \mu \le 1$ $\widetilde{H}(\phi)$ ir Hafa transformācijas $H(\phi)$ ε, δ, μ aproksimācija, ja:

$$\forall \phi \in \Theta^w, H(\phi) \Rightarrow \Pr ob(\widetilde{H}(\phi) \ge (1+\varepsilon)\mu m) \le \delta \qquad (1.16)$$

$$\forall \phi \in \Theta^w, H(\phi) \Rightarrow \Pr ob(\left|\widetilde{H}(\phi) - H(\phi)\right| \geq \varepsilon H(\phi)) \leq \delta \qquad (1.17)$$

Izteiksme (1.16) nodrošina (ar uzticēšanās varbūtību 1 – δ), ka netiks atrastas nepareizas maksimumu vietas. Izteiksme (1.17) nodrošina (ar uzticēšanās varbūtību 1 – δ), ka īstās maksimumu vietas tiks atrastas. Aizstājot parametru vektorus ar Hafa transformācijas skaitītāju tabulas šūnām, iegūsim tādu pašu diskrētās Hafa transformācijas aproksimācijas definīciju.

Tagad apskatīsim pašu varbūstiskās Hafa transformācijas darbības algoritmu, ja mums ir dots attēls ar m punktiem:

1. Uzstādām visus skaitītājus skaitītāju tabulā $\widetilde{H}(\phi)$ uz 0;
2. Soļus 3. un 4. atkārtojam n reizes;
3. Izvēlamies patvaļīgu attēla punktu p;
4. Atrodam visas šūnas skaitītāju tabulā, kuras atbilst izvēlētajam punktam un to vērtības palielinām par 1;
5. Normalizējam visas skaitītāju tabulas šūnas pēc šādas formulas:

$$\widetilde{H}(\phi) = \frac{m \cdot \widetilde{H}(\phi)}{n} \qquad (1.18)$$

Parametra n vērtību var aprēķināt sekojoši:

$$m = \min\left\{ 3\frac{\ln\left(\frac{1}{\delta}\right)}{\mu\varepsilon^2}, \frac{1-\mu}{\mu\varepsilon^2\delta} \right\} \qquad (1.19)$$

kur parametri ε, δ un μ ir ņemti no varbūtiskās Hafa transformācijas definīcijas.

Kad skaitītāju tabula ir aizpildīta, tālāk rīkojamies tāpat kā parastās Hafa transformācijas gadījumā.

Šīs metodes galvenais plus ir algoritma ātrdarbības uzlabošanās salīdzinot ar parasto Hafa transformāciju, jo nav jāaplūko visi attēla punkti. Tomēr nepareiza varbūtiskās Hafa transformācijas parametru izvēle var novest pie neprecīzu rezultātu iegūšanas.

1.2.4. Gadījuma Hafa transformācija

Gadījuma Hafa transformācija ir viena no efektīvākajām elipšu atpazīšanas metodēm, kura izmanto Hafa transformāciju. Tā balstās uz metodi, kuru savā publikācijā aprakstījuši L. Xu, E. Oja un P. Kultanen [8].

Diemžēl, šī metode nav praktiski pielietojama gadījumos, kad meklējamos objektus nevar aprakstīt ar lineāru vienādojumu. Elipse ir viens no šādiem nevēlamajiem gadījumiem. Lai tomēr būtu iespējams elipšu atpazīšanā izmantot gadījuma Hafa transformāciju, mēs izmantojam speciālu metodi elipses centra atrašanai un pēc tam izmantojam jauniegūto elipses vienādojumu, kurš nu jau ir lineārs. Pilnībā šī metode ir aprakstīta Roberta A. McLaughlin rakstā [9].

Šīs metodes vajadzībām vislabāk izmantot sekojošu elipses vienādojumu:

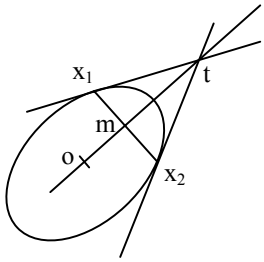
$$a(x-p)^2 + 2b(x-p)(y-q) + c(y-q)^2 = 1, \qquad (1.20)$$

kur x, y ir elipses punkta koordinātas, p un q ir elipses centra koordinātas un pārējiem elipses parametriem a, b un c spēkā nevienādība $ac - b^2 > 0$.

Ja mēs izmantotu šo elipses vienādojumu bez pārveidojumiem, tad mums vajadzētu izvēlēties piecus patvaļīgus punktus un atrisināt piecu nelineāru vienādojumu sistēmu, kur katram punktam atbilstu savs nelineārais vienādojums. Šādi aprēķini katriem pieciem punktiem rēķināšanas procesu ļoti salēnina un padara gadījuma Hafa transformācijas izmantošanu nepraktisku.

Lai padarītu procesu ātrāku, izmantosim tikai trīs attēla punktus – x_1, x_2 un x_3. Aprēķinus varam iedalīt divos soļos – vispirms atradīsim elipses centru, pēc tam atlikušos trīs parametrus.

Lai atrastu elipses centru katram no šiem trīs izvēlētajiem punktiem atradīsim pieskari. Apskatīsim katra punkta tuvu apkārtni un noteiksim līniju, kas vislabāk atbilst punktiem tuvā apkārtnē(to var viegli izdarīt, izmantojot mazāko kvadrātu metodi). Elipses centra atrašanai izmantosim elipses īpašību – ja mums ir zināmi divi elipses punkti, tad elipses centrs(punkts *o* 2.6. attēlā) atrodas uz taisnes, kur iet caur šo abu punktu savienojošā nogriežņa viduspunktu(punktu *m* 2.6. attēlā) un šiem punktiem vilkto pieskaru krustpunktu(punktu *t* 2.6. attēlā), kā tas ir redzams 2.6. attēlā.



2.6. att. **Elipse ar divām tai novilktām pieskarēm**

Šo līniju, uz kuras atrodas elipses centrs, mēs izrēķinām punktiem x_1 un x_2, pēc tam punktiem x_2 un x_3. Šo abu līniju krustpunkts tad arī ir meklētais elipses centrs.

Kad elipses centrs ir atrasts, sameklēsim arī pārējos elipses parametrus. No sākuma pārbīdīsim elipsi tā, lai tās centrs būtu koordinātu sākumpunktā. Pēc šī soļa veikšanas elipses jaunais vienādojums izskatās sekojoši:

$$ax^2 + 2bxy + cy^2 = 1. \qquad (1.21)$$

Ievietojot mūsu trīs punktu($x_1 = (x_1, y_1)$, $x_2 = (x_2, y_2)$ un $x_3 = (x_3, y_3)$) koordinātas jaunajā elipses vienādojumā, iegūstam trīs vienādojumu lineāru sistēmu:

$$\begin{bmatrix} x_1^2 & 2x_1y_1 & y_1^2 \\ x_2^2 & 2x_2y_2 & y_2^2 \\ x_3^2 & 2x_3y_3 & y_3^2 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}. \qquad (1.22)$$

Atrisinot šo vienādojumus sistēmu, mēs iegūstam atlikušos trīs elipses parametrus. Tālāk mēs pārbaudām nevienādību $ac - b^2 > 0$, kura ir patiesa, ja parametri apraksta īstu elipsi. Ja šī nevienādība nav patiesa, tad vai nu šie trīs punkti nav uz vienas elipses, vai arī pieskaru aprēķināšana bijusi neprecīza. Jebkurā no šiem gadījumiem, mēs atrastos parametrus atmetam un ņemam nākošos trīs punktus.

Robert A. McLaughlin praktiski noskaidrojis, ka Hafa transformāciju labāk veikt elipses parametriem (p, q, r_1, r_2, θ), kur r_1 un r_2 ir elipses lielā un mazā pusass un θ pagrieziena leņķis, tāpēc iegūtos parametrus (p, q, a, b, c), pirms Hafa transformācijas veikšanas pārveido formā (p, q, r_1, r_2, θ). Atmiņas taupības nolūkos piecu dimensiju histogramma, kas parasti tiek lietota ar Hafa transformāciju saistītos algoritmos, tiek aizstāta ar sakārtotu saistīto sarakstu. Priekšrocība ir tā, ka šajā sarakstā ir tikai atrastie elipses parametri, bet piecu dimensiju histogrammā tiek glabāta informācija par visiem iespējamajiem elipšu parametru komplektiem, pie kam, parasti, lielākā daļa šīs tabulas elementi ir nulles. Vēl viena problēma ir tā, ka vajag noteikt, kādu parametru apgabalu šī histogramma aptvers. Ja apgabals būs izvēlēts pārāk liels, tad būs daudz nelietderīgi izmantotas atmiņas. Tur pretī, ja tas būs pārāk mazs, tad atsevišķas elipses var neatrast.

Praktiskajā daļā Robert A. McLaughlin savu izstrādāto algoritmu salīdzināja ar citiem, jau iepriekš zināmiem algoritmiem, kuri arī izmanto Hafa transformāciju:

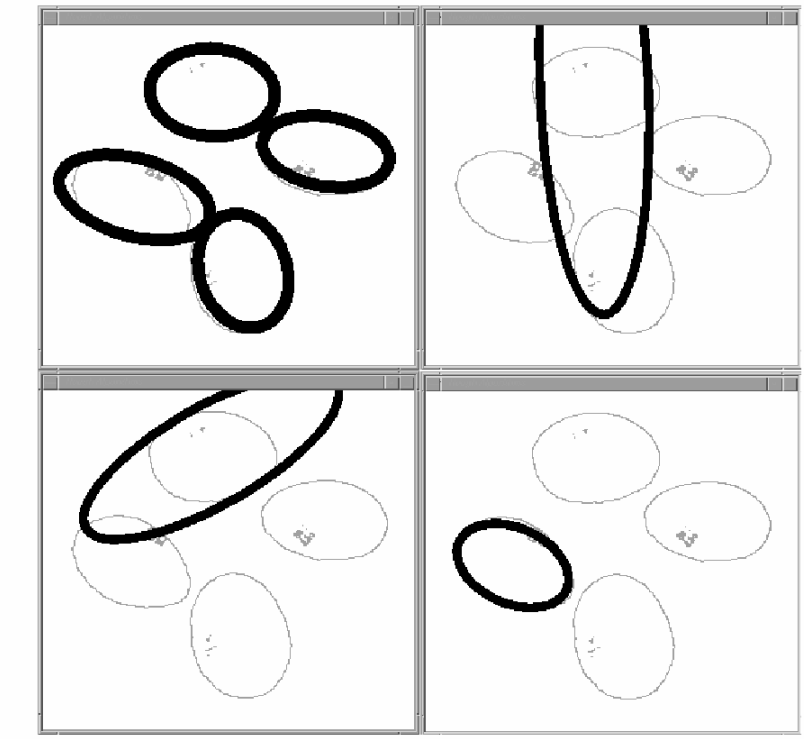
RHT: gadījuma Hafa transformācija,

SHT: parastā Hafa transformācija, kuru aprakstījis H. K. Yuen un pārējie [10],

PHT: varbūtiskā Hafa transformācija,

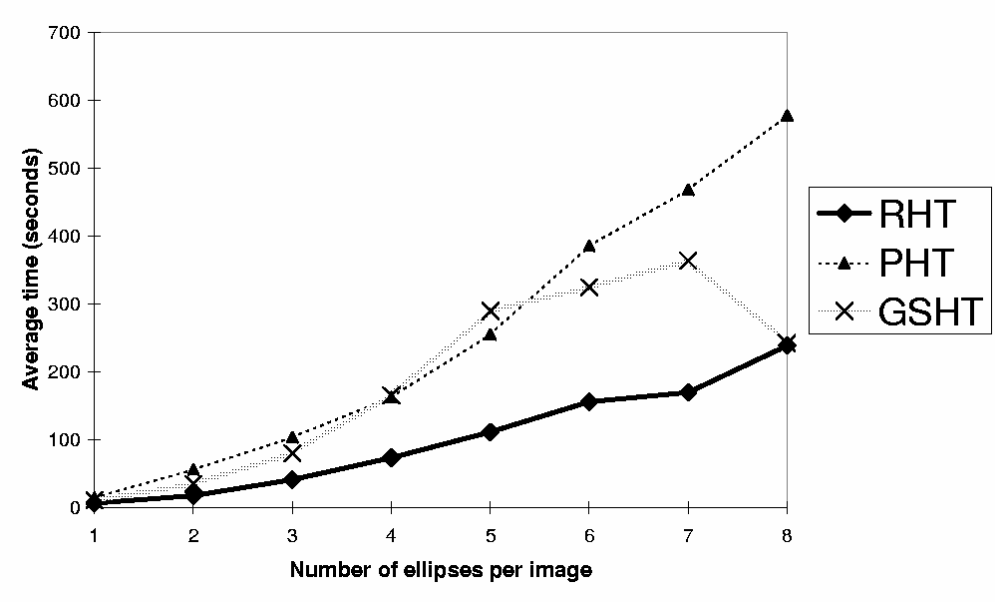
GSHT: Hafa transformācija, kura balstīta uz ģeometrisko simetriju.

Praktiskā pētījuma rezultātā Robert A. McLaughlin secināja, ka gadījuma Hafa transformācija strādā ātrāk kā citas populārākās uz Hafa transformāciju balstītās metodes. Kā arī atrastās elipses ir atsevišķos gadījumos ir daudz precīzākas. Tas redzams arī viņa publicētajā attēlā, kur ar pelēko atzīmēti punkti attēlā un ar melnu, treknu līniju atrastās elipses. Kreisajā augšējā stūrī RHT, labajā augšējā stūrī SHT, kreisajā apakšējā stūrī PHT un labajā apakšējā stūrī GSHT(skatīt 2.7. attēlu). 2.7. attēlā ar pelēko krāsu atzīmēti attēla punkti jeb meklējamās elipses un ar treknu melnu līniju atrastās elipses (2.7. attēls iegūts no Robert A. McLaughlin publikācijas [4]).



2.7. att. **Ar dažādām Hafa transformācijām atpazītās elipses**

Autors savā pētījumā arī apskatīja dažādu Hafa transformāciju ātrdarbību un veica skaitlisku salīdzināšanu. Attālā 2.8. (attēls iegūts no Robert A. McLaughlin publikācijas [4]) redzams vidējais algoritma darbības laiks sekundēs (uz vertikālās ass) atkarībā no elipšu skaita apstrādātajā attēlā (uz horizontālās ass).



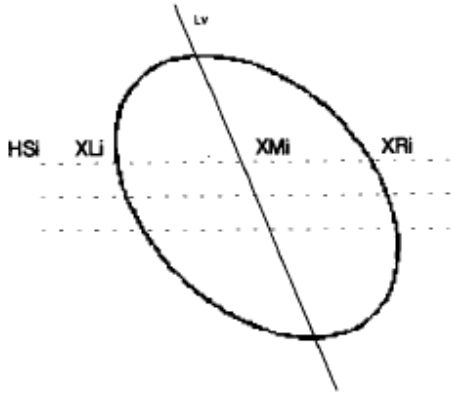
2.8. att. **Dažādu Hafa transformāciju darbības ilguma salīdzinājums**

1.3. Elipšu atrašana, balstoties uz simetriju

Šī metode ir interesanta ar to, ka arī šeit tiek izmantota Hafa transformācija, taču tai nav nekāda sakara ar elipšu atrašanu. Hafa transformācija tiek izmantota taišņu atrašanai un pārējie elipses parametri tiek aprēķināti balstoties uz dažādām elipses īpašībām. Šo metodi savā rakstā prezentēja Chun-Ta Ho un Ling-Hwei Chen [11].

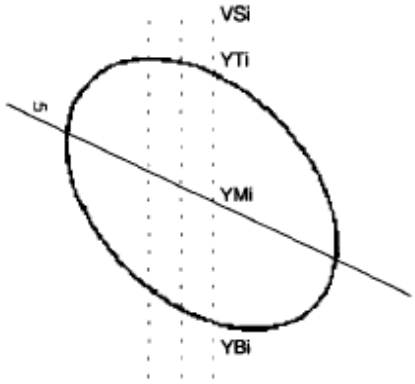
Šo metodi it kā var iedalīt divos soļos – pirmais ir elipses centra atrašana un otrais ir elipses parametru noteikšana. Gan pirmais, gan otrais solis balstās uz vairākām teorēmām, kas ir samērā vienkāršas un viegli pierādāmas, tāpēc šos pierādījumus darbā neiekļāvu. Ja ir interese, tad tos var aplūkot Chun-Ta Ho un Ling-Hwei Chen rakstā.

1. teorēma. Pieņemsim, ka mums ir dota elipse E, kas tiek skenēta no kreisās puses uz labo un no augšas uz leju. Katra horizontālā skenēšanas līnija HS_i krustojas ar elipsi punktos XL_i un XR_i, kā tas ir redzams 2.9. attēlā (attēls iegūts no Chun-Ta Ho un Ling-Hwei Chen publikācijas [11]). Ja XM_i ir viduspunkts starp XL_i un XR_i, tad visi XM_i atrodas uz vienas taisnes L_v, kas turpmāk tiks saukta par elipses vertikālo simetrijas asi.



2.9. att. **Elipse ar horizontālām skenēšanas līnijām un vertikālo simetrijas asi**

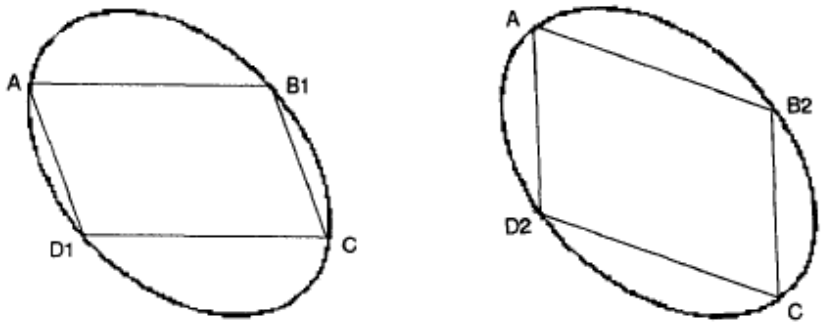
2. teorēma. Pieņemsim, ka mums ir dota elipse E, kas tiek skenēta no augšas uz leju un no kreisās puses uz labo. Katra verikālās skenēšanas līnija VS_i krustojas ar elipsi punktos YT_i un YB_i, kā tas ir redzams 2.10. attēlā (attēls iegūts no Chun-Ta Ho un Ling-Hwei Chen publikācijas [11]). Ja YM_i ir viduspunkts starp YT_i un YB_i, tad visi YM_i atrodas uz vienas taisnes L_h, kas turpmāk tiks saukta par elipses horizontālo simetrijas asi.



2.10. att. **Elipse ar vertikālām skenēšanas līnijām un horizontālo simetrijas asi**

3. teorēma. Pieņemsim, ka mums ir dota elipse E un ar iepriekš aprakstīto metodi iegūtas elipses simetrijas asis L_v un L_h, tad taišņu L_v un L_h krustpunkts ir elipses centrs.

4. teorēma. Pieņemsim, ka mums ir dota elipse E ar centru punktā (0, 0), A ir punkts uz elipses un C ir punkta A simetriski pretējais punkts pret elipses centru. Punkti B_1 un D_1 ir simetriski pretējie punktu A un C punkti pret vertikālo simetrijas asi L_v. Punkti B_2 un D_2 ir simetriski pretējie punktu A un C punkti pret horizontālo simetrijas asi L_h. Tad četrstūri AB_1CD_1 un AB_2CD_2 ir paralelogrami (redzams 2.11. attēlā, kurš iegūts no Chun-Ta Ho un Ling-Hwei Chen publikācijas [11]).



2.11. att. **Elipse ar ievilktu paralelogramu**

Tagad apskatīsim pašu algoritma darbības principu. Sāksim ar elipses centra atrašanas algoritmu.

Elipses centra atrašana balstās uz pirmajām trīs teorēmām. Pirms sākam meklēt elipšu centrus, no dotā attēla iegūstam kontūru punktus. Pēc tam šos punktus, izmantojot horizontālo skenēšanu (aprakstīta 1. teorēmā), sadalām vairākos apakšattēlos. To darām sekojoši:

1. Izveidojam tukšu attēlu G;
2. Skenējam attēlu no labās puses uz kreiso un no augšas uz leju. Katram attēla punktam (i, j), ja eksistē attēla punkts (k, j), attēlā G iezīmējam punktu ((i+k)/2, j);
3. Beidzam skenēšanu;
4. Attēlam G pielietojam Hafa transformāciju, lai atrastu visas taisnes (iespējamās vertikālās simetrijas asis);
5. Katrai atrastajai simetrijas asij L_v atrodam punktus, kuri ir simetriski pret šo līniju un izveidojam jaunu apakšattēlu F_h.

Tagad katram atrastajam apakšattēlam F_h veicam vertikālo skenēšanu (aprakstīta 2. teorēmā). Vertikālās skenēšanas procedūra:

1. Izveidojam tukšu attēlu G;
2. Skenējam attēlu no augšas uz leju un no labās puses uz kreiso. Katram attēla punktam (i, j), ja eksistē attēla punkts (i, k), attēlā G iezīmējam punktu (i, (j+k)/2);
3. Beidzam darbu;
4. Attēlam G pielietojam Hafa transformāciju, lai atrastu visas taisnes (iespējamās horizintālās simetrijas asis);
5. Katrai atrastajai simetrijas asij L_h atrodam punktus, kuri ir simetriski pret šo līniju un izveidojam jaunu apakšattēlu F_{hv}. Līniju L_h un L_v krustpunkts (x_c, y_c) ir iespējamās elipses centrs.

Kad iespējamie elipses centri ir atrasti, tad apskatām katram elipses centram atbilstošo apakšattēlu F_{hv}. Elipses parametriem (a, b, θ), līdzīgi kā Hafa transformācijas gadījumā, izveidojam skaitītāju masīvu. Katram apakšattēla punktam A skatāmies, vai apakšattēlā ir arī tam simetriskais punkts C attiecībā pret elipses centru (kā tas ir aprakstīts 4. teorēmā). Ja tāds punkts eksistē, tad atbilstoši 4. teorēmā aprakstītajam algoritmam, atrodam punktus D_1, D_2, B_1 un B_2. Kad tas ir izdarīts, pārbaudām, vai četrstūri AB_1CD_1 un AB_2CD_2 ir paralelogrami. To var izdarīt, pārbaudot, vai vai nogriežņi $AB_1 = CD_1$ un $AD_2 = B_2C$. Ja šie četrstūri ir paralelogrami, tad punktus A, B_1 un B_2 pārbīdām par $(-x_c, -y_c)$, lai iegūtu punktus uz elipses ar centru (0, 0).

Elipsi ar centru punktā (0, 0) var aprakstīt ar vienādojumu

$$dx^2 + exy + fy^2 = 1 \qquad (1.23)$$

Ja mums ir zināmi trīs punkti, tad atrisinot trīs lineāru vienādojumu sistēmu iegūstam elipses parametrus (d, e, f). Savukārt no šiem parametriem pēc zemāk redzamajām formulām var atrast abas elipses pusasis a un b, kā arī pagrieziena leņķi θ.

$$\theta = \frac{\arctan\left(\frac{e}{d-f}\right)}{2} \tag{1.24}$$

$$a = \sqrt{\frac{1}{d\cos^2\theta + e\sin\theta\cos\theta + f\sin^2\theta}} \tag{1.25}$$

$$b = \sqrt{\frac{1}{f\cos^2\theta - e\sin\theta\cos\theta + d\sin^2\theta}} \tag{1.26}$$

Kad elipses parametri ir atrasti, tad skaitītāju masīvā šiem parametriem atbilstošo elementu palielinām par viens. Kad visi apakšattēla punkti ir apstrādāti, atliek tikai skaitītāju masīvā atrast maksimumus, kas arī ir mūsu meklēto elipšu parametri.

1.4. Elipšu atrašana, balstoties uz elipses lielo pusasi

Šo metodi pirmie apskatīja Yonghong Xie un Qiang Ji savā rakstā „Jauna efektīva elipses atrašanas metode" [12]. Šī metode balstās uz elipses īpašību, ka, zinot elipses lielās ass abus galapunktus, var viegli atrast elipses mazo pusasi, kas ļauj viennozīmīgi aprakstīt konkrēto elipsi. Šai metodei nevajag aprēķināt elipses pagrieziena leņķi vai tās aprakstošās līnijas izliekumu, kas, vispārīgi ņemot, ir ļoti jūtīga pret trokšņiem. Vēl šai metodei nav nepieciešami nekādi sarežģīti aprēķini, kā arī aprēķiniem nepieciešamā atmiņa ir salīdzinoši maza, salīdzinot ar metodēm, kuru pamatā ir Hafa transformācija.

Algoritmos, kuri izmanto Hafa transformāciju, nepieciešama piecu dimensiju parametru telpa, kas prasa daudz laika un atmiņas. Tā kā šajā metodē nepieciešams tikai viens saraksts, kurā glabāt mazās pusass garumu, tai aprēķinu veikšanai nepieciešamā atmiņa ir daudz mazāka, salīdzinot ar Hafa transformāciju.

Patvaļīgai elipsei ir pieci nezināmi parametri: (x_0, y_0) – centra koordinātas, α – pagrieziena leņķis un (a, b) – lielā un mazā pusass. Parasti, lai aprēķinātu šos piecus parametrus, mums nepieciešami pieci elipses punkti. Ja izmantojam kādu papildus informāciju par izvēlētajiem punktiem, vai arī izvēlamies kādus speciālus punktus, tad elipses parametrus iespējams izrēķināt, izmantojot mazāk punktus. Šajā metodē mēs izmantosim otro gadījumu – izvēlēsimies speciālus punktus(lielās ass abus galapunktus un patvaļīgi izvēlētu punktu uz elipses).

Katram punktu pārim (x_1, y_1) un (x_2, y_2) pieņemam, ka tie ir lielās ass galapunkti. No šiem diviem punktiem mēs varam aprēķināt četrus elipses parametrus pēc sekojošām formulām:

$$x_0 = \frac{x_1 + x_2}{2} \qquad (1.27)$$

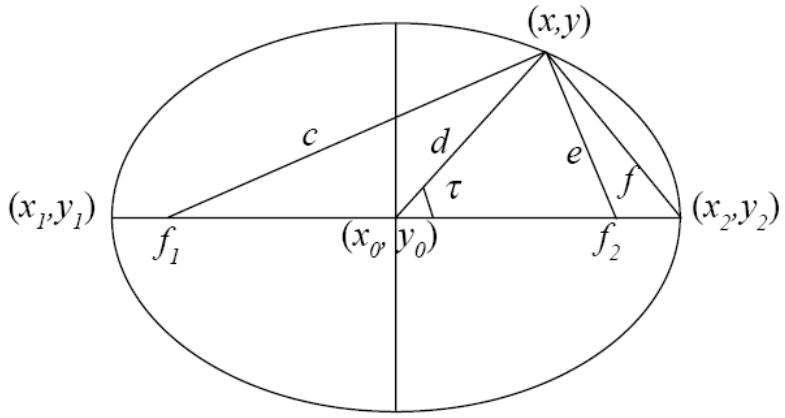
$$y_0 = \frac{y_1 + y_2}{2} \qquad (1.28)$$

$$a = \frac{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}}{2} \qquad (1.29)$$

$$\alpha = \arctan\left(\frac{y_2 - y_1}{x_2 - x_1}\right) \qquad (1.30)$$

Kur (x_0, y_0) ir elipses centrs, a – elipses lielās pusass garums un α – elipse pagrieziena leņķis.

Lai atrastu elipses mazo pusasi, ņemsim vēl vienu punktu (x, y) uz elipses, kā tas redzams 2.12. attēlā.



2.12. att. **Elipse ar svarīgākajiem tās elementiem**

Attēlā 2.12. f_1 un f_2 ir elipses fokusi, d – attālums starp elipses centru (x_0, y_0) un punktu (x, y) uz elipses. Ievērosim, ka attālumam d ir jābūt mazākam par lielās pusass garumu a. To ievērojot, elipses mazās pusass garumu varam izteikt no vienādojuma:

$$b^2 = \frac{a^2 d^2 \sin^2 \tau}{a^2 - d^2 \cos^2 \tau}, \qquad (1.31)$$

kur $\cos(\tau)$ var aprēķināt šādi:

$$\cos(\tau) = \frac{a^2 + d^2 - f^2}{2ad}. \qquad (1.32)$$

Izmantojot vienādojumus (1.27) – (1.32), ir iespējams viennozīmīgi aprēķināt visus piecus elipses parametrus.

Elipšu meklēšanas algoritms īsumā izskatās šādi:

1. Ejam cauri visiem punktu pāriem(pieņemot tos par elipses lielās pusass galapunktiem). Pārbaudām, vai tie atbilst elipses lielās ass galapunktiem – vai attālums starp punktiem ir pa vidu starp minimālo un maksimālo lielās ass garumu. Ja ir, tad izpildām soļus 2. – 4.
2. Ejam cauri viesiem atlikušajiem punktiem(ievērojam, ka punktus, kuri atrodas tālāk par lielās pusass garumu no elipses centra, nevajag apskatīt) un aprēķinām mazās pusass garumu. Iegūto vērtību glabājam sarakstā, kur ir punktu skaits, kuriem sanāca konkrētais pusass garums.
3. Kad visi punkti ir apskatīti, atrodam populārākā pusass garumu un pārbaudām, vai punktu skaits priekš šīs pusass ir pietiekoši liels(lielāks par minimālo nepieciešamo, lai uzskatītu, ka elipse ir atpazīta).

4. Ja elipse ir atrasta, tad izdrukājam elipses parametrus un izmetam visus tos punktus, kuri ir uz šīs elipses. Pēc tam turpinām ar nākošo punktu pāri
5. Darbu beidzam, kad visi punktu pāri ir apskatīti.

Šī metode ļoti labi darbojas vairāku elipšu un samērā liela trokšņa gadījumā, kamēr tieši šīs situācijas ir galvenā problēma uz Hafa transformāciju balstītiem algoritmiem. Par to var pārliecināties arī zemāk redzamajā 2.13 attēlā (attēls iegūts no Yonghong Xie un Qiang Ji raksta [12]), kur ar baltu līniju atzīmētas atrastās elipses. Vienīgais šīs metodes trūkums ir tas, ka jāpārbauda visi punktu pāri, kas ir laikietilpīgs process.

2.13 att. **Atpazītās elipses, izmantojot uz lielo pusasi balstīto elipšu meklēšanas algoritmu**

1.5. Ģenētiskais algoritms.

Hafa transformācija ir ātrs un efektīvs veids, lai atrastu vienkāršas ģeometriskas formas, bet, kad jāatpazīst kaut kas sarežģītāks, tad tā kļūst lēna un ļoti prasīga pret atmiņu. Ģenētiskais algoritms darbojas uz līdzīgiem principiem – mēs meklējam optimumu parametru telpā, tomēr šī algoritma darbības principi ir savādāki. Sīkāk ar tiem var iepazīties E. Lutton un Patrice Martinez rakstā [13].

Ģenētiskā algoritma pluss neregulāru funkciju optimizēšanā ir tas, ka tas darbojas varbūtiski lielā meklēšanas apgabalā, liekot vairākiem risinājumiem (sauktiem par populāciju) attīstīties vienlaicīgi. Ģenētiskie algoritmi darbojas ļoti līdzīgi dabā redzamajai evolūcijai – ir kaut kāda vecāku populācija, no kuras izveidojas bērnu populācija. Ja šī bērnu populācija nav gana laba(mūsu interesējošajā gadījumā – elipses vienādojums nav atrasts), tad tā kļūst par jauno vecāku populāciju un viss sākas no sākuma.

No sākuma vajag inicializēt katru populācijas indivīdu. Parasti tas tiek izdarīts, izvēloties patvaļīgus attēla punktus un no tiem izveido populācijas indivīdu(sauktu par hromosomu) ar šiem punktiem atbilstošajiem gēniem. Gēni ir tādi, lai viennozīmīgi aprakstītu meklēto objektu. Dažreiz šos punktus mēdz izraudzīties determinēti, tā, lai tie būtu vienmērīgi sadalīti pa visu attēlu.

Bērna veidošana sastāv no trīs soļiem. Pirmais ir vecāku izvēlēšanās, atsaucoties uz to *derīgumu*. Otrais ir vecāku ģenētiskā materiāla sakrustošanu, lai iegūtu divus dažādus bērnus. Trešais ir iegūto bērnu gēnu mutācija. Krustošanās un mutācija notiek varbūtiski – krustošanās notiek ar lielu varbūtību (parasti 0.7 līdz 0.9), bet mutācija ar mazu, tā lai mainītos tikai daži gēni. Krustošanās nodrošina populācijas konverģenci uz risinājumu, kurš ir tuvu populācijas indivīdiem ar lielāku *derīgumu*, tur pretī mutācija it kā attālina populāciju no šī maksimuma. Tas tiek darīts ar domu, lai populācija netiektos uz lokālu, bet gan uz globālu maksimumu.

Apstāšanās problēma ir diezgan sarežģīta, jo grūti noteikt, kad atrastā populācija ir pietiekoši laba, tāpēc parasti veic noteiktu skaitu soļu un pēc tam analizē iegūto rezultāta populāciju.

Ar konkrētā indivīda *derīgumu* var saprast to, cik labi viņa aprakstītā elipse sakrīt ar attēlu. Viens no veidiem, kā to pārbaudīt, ir saskaitīt visus attēla punktus, kuri atrodas uz šīs elipses. Ir vēl arī citi sarežģītāki risinājumi.

Vēl ir iespējams jauno vecāku populāciju veidot ne tikai no bērnu populācijas, bet arī no vecāku populācijas, izlasot indivīdus ar lielāko *derīgumu*.

Kad nepieciešamais soļu skaits ir veikts, tad no iegūtās kopas mēģinām noskaidrot uz kādu punktu dotā populācija konverģē un šis tad arī ir meklētais elipses vienādojums.

1.6. Daudzpopulāciju ģenētiskais algoritms.

Jie Yao, Nawwaf Kharma un Peter Gordon savā rakstā [14] aprakstīja efektīvu tehniku elipšu atpazīšanai attēlos, izmantojot ģenētisku algoritmu ar vairākām populācijām. Daudzpopulāciju ģenētiskais algoritms izmanto gan evolūciju, gan klasterizāciju. Algoritmā *derīgumam* tiek izmantoti divi lielumi(*līdzība* un *distance*), kā arī speciāli krustošanās un mutācijas algoritmi. Tā rezultātā ir iegūts algoritms, kas ļoti labi darbojas uz nepilnām elipsēm un attēliem ar lielu troksni.

Populācijas hromosomas gēni ir pieci attēla punkti, no kuriem viennozīmīgi var aprēķināt elipses parametrus. Sākumā mēs izveidojam populācija ar kaut kādu iepriekš fiksētu skaitu hromosomu, kuru gēnus patvaļīgi izvēlamies no attēla punktiem. Tad populācija tiek novērtēta pēc *līdzības* un *distances* un tiek meklēti labi kandidāti. Ja tādu nav, tad, izmantojot klasterizācijas tehniku, hromosomas tiek sadalītas klasteros jeb apakšpopulācijās. Ja kāda apakš populācija konverģē uz kādu elipsi, tad visa apakšpopulācija tiek izņemta un atrastā elipse tiek piefiksēta.

Derīgums.

Lai noskaidrotu, ko iesākt ar konkrēto hromosomu, tiek izmantoti divi *derīgums* raksturlielumi – *līdzība* un *distance*. *Līdzība* raksturo, cik labi kandidāt elipses perimetrs sakrīt ar ideālo elipsi, un *distance* rāda, cik tālu perimetrs ir no ideālās elipses perimetra.

Līdzību(S) aprēķinām pēc šādas formulas:

$$S = \frac{\sum_{(x,y)} \frac{E(x+i, y+j)}{d_{i,j}}}{\#total} \qquad (1.33)$$

kur *#total* ir kandidāt elipses punktu skaits, *E(x+i, y+j)* atgriež 1, ja attēlā ir punkts, kas sakrīt vai ir tuvu punktam (x, y) un $d_{i,j}$ ir mēs aprēķinām pēc sekojošas formulas:

$$d_{i,j} = e^{\frac{|i|+|j|}{4}} \qquad (1.34)$$

Skaitļi *i* un *j* raksturo horizontālo un vertikālo nobīdi starp punktu uz ideālās elipses un atbilstošo punktu attēlā.

Distanci(D) aprēķinām šādi:

$$D(x, y) = \frac{\sum_{(x,y)} d_{i,j}}{\#eff} \qquad (1.35)$$

kur $d_{i,j}$ aprēķinām pēc formulas (1.34) un *#eff* ir punktu skaits, kas uz attēla elipses sakrīt ar ideālo elipsi.

Mūsu mērķis ir atrast tādus kandidātus, kuriem ir laba *līdzība* un maza *distance*, vai arī pieņemama *līdzība* un teicama *distance*.

Klasterizācija.

Apakšpopulāciju sauksim par klasteri un tā centrs ir hromosoma ar vislielāko līdzību. Ja ir vairākas hromosomas ar vienādu līdzību, tad par centru tiek ņemta hromosoma ar mazāko distanci.

Daudzpopulāciju ģenētiskā algoritma sākumā ir viena populācija, kurā hromosomas ir sakārtotas pēc līdzības un distances. Vēlāk šī populācija un arī pārējās apakšpopulācijas tiek mainītas klasterizācijas procesā. Klasterizācija sastāv no *migrācijas*, *dalīšanas* un *apvienošanas*.

Katrā apakšpopulācijā visas labās hromosomas(S > 0.7 un D < 10) tiek paturētas tajā pašā populācijā vai tiek pārvietotas uz citu esošo vai jaunizveidotu klasteri. Visas pārējās hromosomas vienkārši tiek atstātas tajos klasteros, kuros viņas jau ir.

Eiklīda distance *ED* ir tas raksturlielums, kas nosaka, vai hromosoma paliks tajā pašā klasterī vai tiks pārvietota uz citu. *ED* starp divām hromosomām , kuru atbilstošo elipšu vienādojumu parametri ir (a_1, b_1, x_1, y_1, ω_1) un (a_2, b_2, x_2, y_2, ω_2) aprēķinām pēc sekojošas formulas:

$$ED = \frac{5}{(a_1 - a_2)^2 + (b_1 - b_2)^2 + (x_1 - x_2)^2 + (y_1 - y_2)^2 + (\omega_1 - \omega_2)^2} \qquad (1.36)$$

Elipses parametri līdzīgi kā iepriekš ir lielā un mazā pusass(a un b), elipses centra koordinātas (x, y) un pagrieziena leņķis ω. Kad kāda hromosoma tiek pārvietota no viena klastera uz citu, tad tā aizstāj jaunā klastera vājāko hromosomu(hromosomu ar vismazāko līdzību). Vēlāk tukšā vieta tiek aizpildīta evolūcijas procesā.

Migrācija notiek tad, kad kādai labai hromosomai *ED* ar tās klastera centru ir mazāka par iepriekš noteiktu minimālo vērtību. Lai noteiktu uz kuru klasteri jāpārvieto hromosoma, jāsalīdzina *ED* ar visu klasteru centriem. Hromosomu pārvieto uz to klasteri, kuram *ED* ir maksimāla. Ja nav tāda klastera, ar kura centru aplūkotās hromosomas *ED* ir lielāka par minimālo nepieciešamo, tad izveidojam jaunu klasteri, kura centrs ir aplūkotā hromosoma. Šo darbību mēs sauksim par *dalīšanu*.

Apvienošana ir tad, kad *ED* starp diviem centriem ir mazāka par noteiktu minimālo vērtību. Tādā gadījumā izveidojam jaunu klasteri, kurā ieliekam 50% labākās hromosomas(balstoties uz

līdzību). Visi klasteri periodiski tiek pārbaudīti, vai gadījumā kādi nav jāapvieno. To dara ik pēc 30 paaudzēm.

Evolūcija.

Evolūcijas procesā mēs izvēlamies hromosomas ar lielāku derīgumu, jo tā ir lielākas cerības atrast elipses. Tomēr ar *krustošanās* un *mutācijas* palīdzību mēs nodrošinām hromosomu mainīgumu, kas palīdz doties jaunā, potenciāli daudzsološā virzienā.

Evolūcijas procesā, veidojot nākošo paaudzi, 15% labākās (balstoties uz līdzību) hromosomas tiek pārkopētas no vecās paaudzes uz jauno. Lai aizpildītu atlikušās vietas, tiek ņemtas divas hromosomas (hromosomas varbūtība tikt paņemtai ir tieši proporcionāla tās relatīvajai līdzībai) un ar varbūtību 0.6 krustotas, un ar varbūtību 0.1 mutētas.

Kad jaunā populācija ir pabeigta, pārbaudām, vai tās labās hromosomas nevajag pārvietot, vai dalīt.

Krustošanās procesā bērnam pirmie n gēni tiek no viena vecāka un pēdējie 5 – n gēni no otra vecāka. Otram bērnam tiek pretējie gēni no tiem pašiem vecākiem.

Mutācijai ir izveidots speciāls operators, kas izvēlas patvaļīgu gēnu, no kura sāk ceļu pa patvaļīgi izvēlētu elipsi, līdz atrod vēl četrus punktus, kuri tad arī ir jaunās, mutācijas procesā iegūtās hromosomas gēni.

Algoritma darbības beigas.

Kad visi algoritma soļi ir izskaidroti, atliek noskaidrot, cik ilgi algoritms ir jādarbina.

Katras apakšpopulācijas evolūcija ir neatkarīga no visām pārējām, tāpat neatkarīgas ir tās attīstības beigas. Apakšpopulācija tiek izbeigta, ja ir sasniegts viens no zemāk minētajiem nosacījumiem:

1. Optimāla konverģence – ja apakšpopulācijā ir vismaz viena „optimāla” hromosoma. Par „optimālu” hromosomu sauksim hromosomu, kurai $S > 0.95$ un $D < 10$;
2. Pus-optimāla konverģence – ja apakšpopulācijā ir vismaz viena „laba” hromosoma un 30 paaudžu laikā klasterī nav parādījusies neviena „optimāla” hromosoma. Par „labu” hromosomu sauksim hromosomu, kurai $S > 0.7$ un $D < 10$;
3. Stagnācija – 500 paaudzes ir pagājušas un nav izpildījies neviens no iepriekš minētajiem nosacījumiem.

Autori uzsver šī algoritma ātrdarbību un precīzo elipšu atrašanu vairāku elipšu un liela trokšņa gadījumā. Tomēr pats algoritms ir krietni sarežģītāks, ja salīdzina ar algoritmiem, kuri izmanto Hafa transformāciju.

2. DATORPROGRAMMAS APRAKSTS

Praktiskās daļas mērķis bija izstrādāt datorprogrammu, kas spētu reālos attēlos atpazīt elipses. No sākuma tika strādāts ar ģenerētiem attēliem, t.i., melnbaltiem attēliem, kur ar melno krāsu ir atzīmēti visi attēla kontūru punkti. Tas tika darīts, lai varētu pilnveidot elipšu atpazīšanas algoritmu un saprast galvenās problēmas darbā ar Hafa transformācijām, jo tieši Hafa transformācija tika izvēlēta kā tā metode, ar kuru būtu visvieglāk atpazīt elipses. Vēlāk tika strādāts pie reāliem attēliem, konkrēti pie šautuves mērķa fotogrāfiju analīzes. Tas tika darīts ar mērķi izstrādāt automātisku punktu skaitīšanas sistēmu, kurai nepieciešams ir atrast visus riņķus, kas atdala zonas ar dažādu punktu skaitu.

Izstrādājot programmu, nonācu pie secinājuma, ka parasto Hafa transformāciju nav iespējams pielietot reāliem attēliem, jo tās darbība aizņem pārāk ilgu laiku. Tāpēc tika izstrādātas divas uzlabotas Hafa transformācijas versijas, kas tomēr nedaudz atšķiras no teorētiskajā daļā apskatītajām. Uzlabotās Hafa transformācijas metodes sadalīja elipšu atpazīšanu divās fāzēs – no sākuma tika atrasti elipšu centri un pēc tam, izmantojot Hafa transformāciju, arī pašas elipses. Tā rezultātā programma sevī ietver šāda trīs Hafa transformācijas:

1. Parastā Hafa transformācija;
2. Hafa transformācija ar centru meklēšanu, izmantojot simetrisku punktu pārus (turpmāk tekstā sauksim par pirmo metodi);
3. Hafa transformācija ar centru meklēšanu, izmantojot ģeometrisko simetriju(turpmāk tekstā sauksim par otro metodi).

Datorprogramma ir izstrādāta c++ valodā, izmantojot .NET bibliotēku bitmap failu apstrādei. Programmas rakstīšanai izmantota Microsoft Visual Studio 2005 vide. Kā ievadu programma saņem bitmap failu un tāda paša formāta failu izdod kā gala rezultātu. Datorprogrammas darbību varētu iedalīt 4 soļos:

1. Attēla ielādēšana;
2. Kontūru līniju punktu iegūšana;
3. Elipšu atrašana;
4. Iegūtā attēla saglabāšana.

Izmantotajās Hafa transformācijas variācijās, atšķirīgs ir tikai 3. solis, kurš katrai no metodēm tiks aprakstīts vēlāk. Vēl jāatzīmē, ka ģenerētajiem un reālajiem attēliem kontūru līniju punktu iegūšanai, tika izmantotas dažādas metodes. Pašas Hafa transformācijas algoritms visām

metodēm ir vienāds, un tas balstās uz punktā 1.2.1. aprakstītā algoritma. Tomēr nelielas atšķirības var būt skaitītāju masīva maksimumu meklēšanā un analīzē dažādām Hafa transformācijas metodēm.

Lai uzlabotu programmas ātrdarbību, sarežģītākās operācijas – tādas kās *sin*, *cos* – tika iepriekš izrēķinātas, un tad, kad bija vajadzība pēc tām, tās tika realizētas kā apskatīšanās tabulā.

2.1. Kontūru punktu atrašana

Lai varētu pielietot Hafa transformāciju, vispirms vajag atrast visus attēla punktus, kuri apraksta kaut kādas attēla kontūras. Programmā tiek izdalīti divi principiāli atšķirīgi gadījumi – ģenerētie attēli un reālie attēli.

Ģenerētie attēli vairāk tika izmantoti programmas un paša Hafa transformācijas algoritma testēšanai, pētīšanai. Tāpēc maksimāli tika atvieglota attēlu struktūra – uz balta fona ar melnu krāsu tika uzzīmētas visas kontūras. Tātad pēc attēla ielādēšanas atlika sameklēt visus attēla melnos punktus, un tad tos arī apstrādāt.

Reālos attēlos tik vienkārši nav, jo kontūras var nebūt novilktas ar kaut kādu līniju, tās var būt tikai robeža starp dažādu krāsu laukumiem. Tāpēc, lai atrastu kontūras, tika izstrādāta metode, kas izmanto Sobela operatoru (aprakstīts punktā 1.1.1.). Vispirms visiem attēla punktiem tiek aprēķināta gradienta moduļa skaitliskā vērtība un, ja tā ir pietiekoši liela, tad šis punkts tiek uzskatīts par kontūras punktu. Attēlā 2.1. varam redzēt, ka ar Sobela operatoru atrastās kontūras (redzamas ar sarkano krāsu) diezgan labi atbilst oriģinālajam attēlam.

2.1. att. **Ar Sobela operatoru atrastās kontūras**

Ja mēs tā kārtīgi paskatāmies uz iegūtajiem rezultātiem, tad varam redzēt, ka vietās, kur pārejas no vienas krāsas apgabala uz otras krāsas apgabalu bija pakāpeniskākas, kontūras netika atrastas, tomēr kopumā atrastās kontūras ir apmierinošas.

2.2. Parastā Hafa transformācija

Kā vienkāršākais algoritms tika izvēlēta parastā Hafa transformācija, bez nekādiem uzlabojumiem. Tā kā parastā Hafa transformācija ir ļoti lēna, tad, lai būtu iespējams iegūt kaut kādus rezultātus arī uz samērā maziem ģenerētajiem attēliem, tika meklētas tikai taisni novietotas elipses. Lai to nodrošinātu, visur, kur punktā 1.2.1. tika lietots elipses orientācijas leņķis, šis leņķis tika ņemts kā 0^0.

Tagad apskatīsim pašu elipšu atrašanas algoritmu. Tas sastāv no trīs soļiem:

1. Skaitītāju tabulas aizpildīšana;
2. Elipses parametru atrašana;
3. Elipšu iezīmēšana oriģinālajā attēlā.

Skaitītāju tabulas aizpildīšana ir samērā viegls, taču laikietilpīgs process. Rīkojamies, kā aprakstīts punktā 1.2.1. – ejam cauri visām iespējamajām parametru a, x_c, y_c vērtībām un pēc vienādojuma (1.6.) izrēķinām parametra b vērtību (atceramies, ka leņķa Φ vietā jāliek 0^0). Pēc tam šiem parametriem atbilstošo skaitītāju palielinām par viens. Tā turpinām, līdz visas parametru a, x_c, y_c kombinācijas ir apskatītas.

Elipšu parametru atrašana ir samērā bīstams pasākums, jo tajā ir viegli kļūdīties, patiesībā, grūtāk ir iegūt pareizus rezultātu, ne kā nepareizus. Meklējot elipses parametrus, ir jāizvēlas minimālais skaitītāja slieksnis, kuru sasniedzot, elipse tiek uzskatīta par atrastu. Ja šo slieksni izvēlas pārāk mazu, tad var tikt atrastas liekas elipses, piemēram, mazākas elipses, kas ļoti labi piekļaujas kādam lielākas elipses lokam. Bet, ja šo slieksni izvēlas pārāk lielu, tad dažas elipses var netikt atrastas. Šī problēma ir aktuāla, ja ir dažāda izmēra elipses, jo tādā gadījumā, vai nu mazākās elipses netiks atrastas, vai arī tiks atrastas nevajadzīgas elipses.

Zinot elipses parametrus, elipses iezīmēšana attēlā ir vienkārša, jo izmantotajā bitmap bibliotēkā, ir automatizēta elipšu zīmēšana.

2.3. Hafa transformācija ar centru meklēšanu

No sākuma apskatīsim, ar ko tad īsti labāka ir Hafa transformācijas modifikācija ar centru meklēšanu. Ja mums ir attēls ar izmēriem 200*200 pikseļi, tad meklētās elipses centrs var atrasties jebkurā no attēla 40’000 punktiem (ja mēs gribētu meklēt arī nepilnas elipses). Ja mums iespējamais elipses centrs ir jau atrasts, tad mums nav jāpārbauda 40’000 punkti, bet tikai viens. Tātad pati Hafa transformācija ir 40’000 reižu ātrāka, kas tiešām ir fantastisks ātrdarbības uzlabojums. Lielākos attēlos ieguvums ir vēl ievērojamāks.

Ko vēl mēs iegūstam no tā, ka ir atrasts elipses centrs? Parasti attēlos vienam centram atbilst viena elipse, tātad, atrodot tās centru, ar Hafa transformācijas palīdzību mēs varam precīzi atrast šo elipsi, ja protams, tāda ir. Nezinot elipses centru, mēs varam tikai atrast vairākas īstajai elipsei ļoti tuvas elipses, no kurām izvēlēties īsto ir samērā sarežģīti.

Nedaudz sarežģītāks ir gadījums, kad jāmeklē koncentriskas elipses – elipses ar vienu centru. Šādā gadījumā mēs nevaram meklēt tikai populārāko elipsi, jo tad no visām koncentriskajām elipsēm tiks atrasta tikai viena. No otras puses, mēs vēlamies, lai katrai no koncentriskajām elipsēm tiktu atrasta tieši viena elipse. Lai to nodrošinātu, tiek atrasta tuvā apkārtnē populārākā elipse un pasludināta par atpazītu. Par tuvu apkārtni tiek uzskatīta pusasu garumu atšķirība mazāka par 20 pikseļiem un elipses pagrieziena leņķa atšķirība mazāka par 20^0.

2.3.1. Hafa transformācija ar centru meklēšanu, izmantojot simetrisku punktu pārus

Šīs metode balstās uz punktā 1.2.1. aprakstīto metode, kad vispirms tiek atrasti elipšu centri un tikai pēc tam pielietota Hafa transformācija elipšu atrašanai. No sākuma apskatīsim algoritma galvenos soļus, un pēc tam katru no zemāk redzamajiem soļiem izanalizēsim nedaudz smalkāk. Kontūru punktu iegūšana jau ir iepriekš apskatīta, tāpēc uzreiz sāksim ar centru meklēšanu:

1. Visu punktu pāru viduspunktu atrašana;
2. Atrodam populārākos viduspunktus un tos pasludinām par iespējamajiem elipses centriem;
3. Katram elipses centram aizpildām soļus 4. līdz 6.;
4. Hafa transformācijas skaitītāju tabulas aizpildīšana;
5. Elipšu parametru atrašana;
6. Elipšu iezīmēšana attēlā.

Lai noskaidrotu iespējamos elipšu centrus, apskatām visus iespējamos punktu pārus. Ja minimālais attālums starp šiem punktiem ir pietiekams, tad atrodam to savienojošā nogriežņa viduspunktu. Pēc tam palielinām šim viduspunktam atbilstošo skaitītāju par 1. Lai atrastu iespējamos centrus, iepriekš ir jāizvēlas sliekšņa vērtība, kuru pārsniedzot, punkts tiek uzskatīts par iespējamo centru. Kad meklējam elipses centrus, tad ne tikai skatāmies, lai skaitītāja vērtība būtu lielāka par sliekšņa vērtību, bet, lai arī šis punkt būtu lokālais maksimums.

Kad iespējamie elipses centri ir atrasti, tad aizpildām Hafa transformācijas skaitītāju tabulu tieši tāpat kā parastās Hafa transformācijas gadījumā. Vienīgā atšķirība ir tā, ka elipses centra parametri x_c un y_c ir fiksēti, bet elipses pagrieziena leņķis mainās robežās no 0^0 līdz 45^0.

Beidzot esam nonākuši līdz elipšu parametru noskaidrošanas, tad var būt divi gadījumi. Ja mēs nemeklējam koncentriskas elipses, tad atrodam lielāko Hafa transformācijas tabulas skaitītāju, un, ja tas pārsniedz sliekšņa robežu, tad elipse ir atrasta. Ja mēs tomēr gribam atrast koncentriskas elipses, tad sakārtojam elipses parametrus atbilstoši tiem piekārtoto skaitītāju vērtībām (parametrus, kuriem atbilstošais skaitītājs ir mazāks par sliekšņa vērtību, nav vērts apskatīt). Pēc tam ejam, sākot ar lielāko vērtību, visam sarakstam cauri. Ja vēl neviena elipse pārāk tuvu (pusasu garumu atšķirība mazāka par 20 pikseļiem un orientācijas leņķa atšķirība mazāka par 20^0) nav atrasta, tad atzīmējam elipsi ar tekošajiem parametriem par atrastu.

2.3.2. Hafa transformācija ar centru meklēšanu, izmantojot ģeometrisko simetriju

Šī metode no iepriekšējā punktā aprakstītās atšķiras tikai ar elipšu centru atrašanu, tāpēc atkārtoti neaprakstīšu elipšu parametru atrašanu. Elipšu centru atrašana balstās uz 2.3. apakšnodaļā aprakstīto metodi. No sākuma atrodam visas vertikālās simetrijas asis. Pēc tam katrai vertikālajai simetrijas asij atrodam horizontālo simetrijas asi, un, ja tāda eksistē, tad vertikālās un horizontālās simetrijas ass krustpunkts ir elipses centrs. Pēc tam pielietojam Hafa transformāciju, lai atrastu pārējos elipses parametrus.

Algoritma soļi:

1. Veicam horizontālo skenēšanu un atrodam visu vienas līnijas punktu pāru viduspunktus;
2. Atrodam visas vertikālās simetrijas asis;
3. Katrai vertikālajai simetrijas asij izpildām soļus 4. līdz 6.;
4. Atrodam visus pret vertikālo simetrijas asi simetriskos punktus;
5. Izmantojot atrastos punktus, atrodam visas iespējamās horizontālās simetrijas asis;

6. Katrai horizontālajai simetrijas asij izpildām soļus 7. līdz 10.;
7. Atrodam krustpunktu ar vertikālo simetrijas asi (krustpunkts ir elipses centrs);
8. Aizpildām Hafa transformācijas skaitītāju tabulu, izmantojot 5. solī atrastos punktus;
9. Elipšu parametru atrašana;
10. Elipšu iezīmēšana attēlā.

Horizontālās skenēšanas vajadzībām, punktus glabājam tabulā ar sarakstiem, kur n-tajā sarakstā ir visu attēla n-tās rindas punktu y koordinātas. Vertikālajai skenēšanai ir tāda pati tabula, tikai n-tajā sarakstā ir n-tās attēla kolonnas punktu x koordinātas.

Taišņu atrašanai izmantoju Hafa transformāciju, kur taisni aprakstu ar vienādojumu

$$r = x\cos\varphi + y\sin\varphi \qquad (2.1)$$

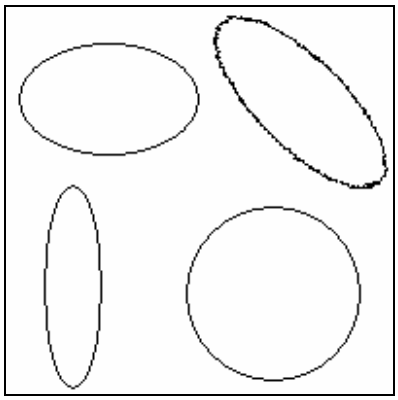
Attiecīgi Hafa transformācijai taisnes atrašanai ir divu dimensiju parametru telpa. Lai aizpildītu Hafa transformācijas skaitītāju tabulu, katram punktam eju cauri visām leņķa φ vērtībām un aprēķinu parametra r vērtību. Pēc tam atrodu leilākās skaitītāju vērtības, kurām atbilstošie parametri arī ir meklētās simetrijas asis.

Ja mums ir dotas divas taisnes ar parametriem (r_1, φ_1) un (r_2, φ_2), tad šo taišņu krustpunkta koordinātas var atrast pēc formulām (2.2) un (2.3), kuras iegūtas no izteiksmes (2.1).

$$y = \frac{r_1\cos\varphi_1 - r_1\cos\varphi_2}{\cos\varphi_1\sin\varphi_2 - \sin\varphi_1\cos\varphi_2} \qquad (2.2)$$

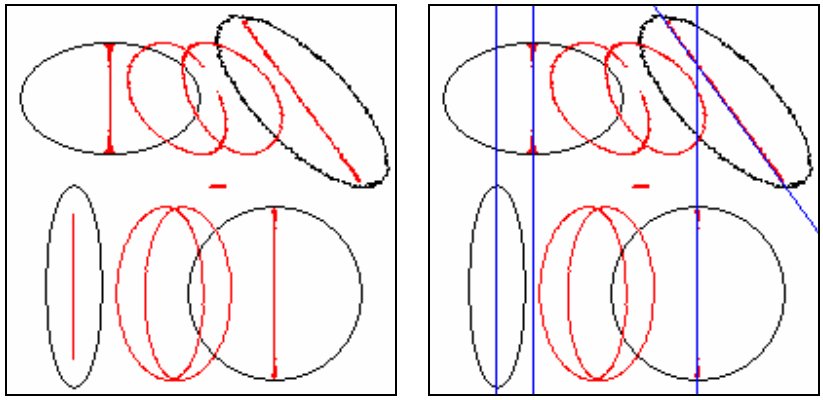
$$x = \frac{r_1 - y\sin\varphi_1}{\cos\varphi_1} \qquad (2.3)$$

Lai saprastu, kā īsti algoritms darbojas, apskatīsim, ko mēs īsti iegūstam pēc katra no soļiem. Kā piemēru ņemsim 2.2. attēlā redzamo bildi.



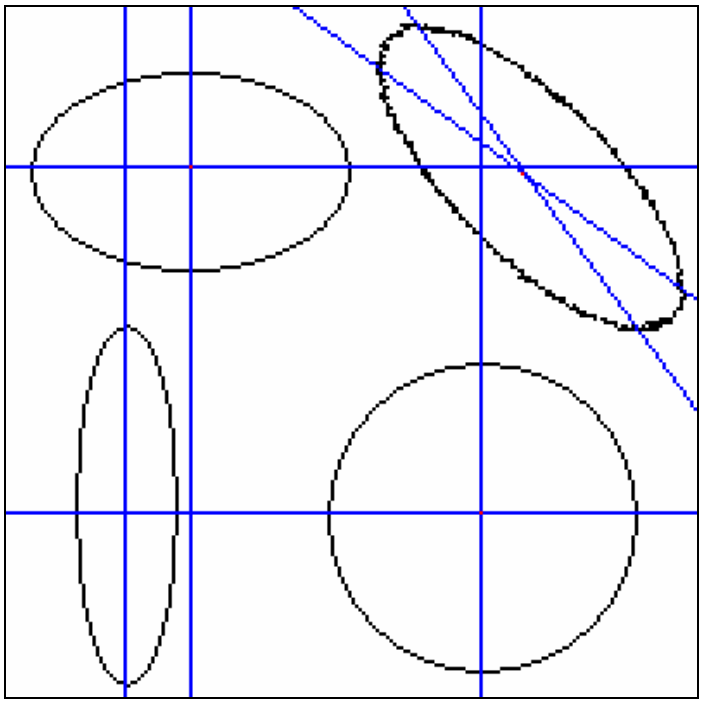
2.2. att. **Oriģinālais attēls**

Pēc pirmo divu soļu izpildes (skat. Attēlu 2.3.), mums ir atrasti visu vienas līnijas punktu pāru viduspunkti (attēlā pa labi ar sarkanu krāsu) un arī elipšu vertikālās simetrijas asis (attēlā pa kreisi ar zilu krāsu).



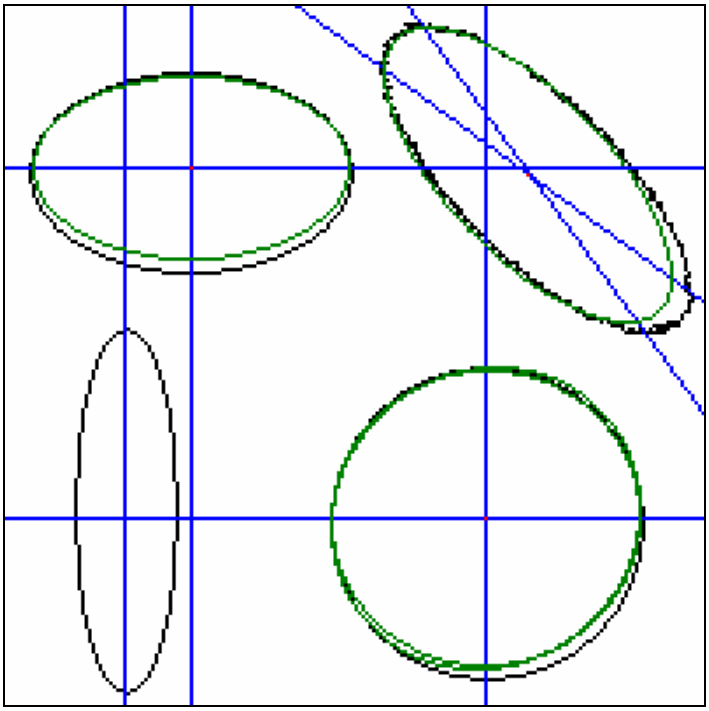
2.3. att. **Oriģinālais attēls ar atrastām vertikālajām simetrijas asīm**

Pēc tam, kad ir atrastas vertikālās simetrijas asis, atliek atrast horizontālās simetrijas asis. 2.4. attēlā redzama situācija, kad visas vertikālās simetrijas asis ir apstrādātas. Ar zilo krāsu ir redzamas visas simetrijas asis, un ar sarkano krāsu atrastie centri.



2.4. att. **Oriģinālais attēls ar atrastām simetrijas asīm un elipšu centriem**

Tagad tikai atliek atrast pašas elipses. Beigās iegūtais rezultāts redzams attēlā 2.5., kur ar zaļo krāsu atzīmētas atrastās elipses.



2.5. att. **Oriģinālais attēls ar atrastām simetrijas asīm un elipsēm**

3. REZULTĀTU ANALĪZE

Ar izstrādāto datorprogrammu tika apstrādāti divu veidu attēli – ģenerētie un reālie. Ģenerētie attēli tika apskatīti, lai saprastu Hafa transformācijas stiprās puses un trūkumus. Reālie attēli tika apstrādāti, lai parādītu, ka Hafa transformācija var būt noderīga arī reālā dzīves situācijā.

Hafa transformācijā un centru meklēšanā tiek izmantoti divi parametri. Minimālo punktu skaitu, kas nepieciešams, lai elipsi atzītu par atpazītu, sauksim par elipses slieksni. Bet minimālo punktu skaitu, kas nepieciešams, lai punktu atzītu par iespējamo centru, sauksim par centra slieksni.

Tā kā parastā Hafa transformācija ir ļoti lēna, tad tā tika izmantota tikai ģenerētajiem attēliem, kuri ir mazāki un arī punktu skaits tajos ir daudz mazāks. Uz reālajiem attēliem tika pielietotas abas Hafa transformācijas ar centru meklēšanu.

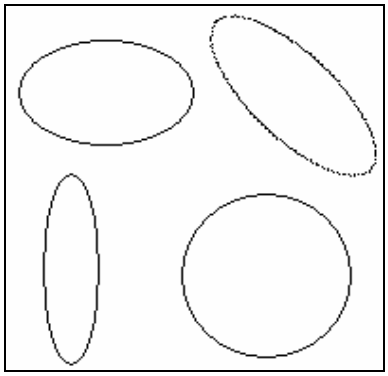
3.1. Ģenerētie attēli

Ģenerēto attēlu analīzē galvenā uzmanība tika pievērsta programmas darbības pētīšanai atkarībā no dažādām Hafa transformācijas parametru vērtībām. Visi attēli tika apstrādāti ar visām trīs programmā realizētajām metodēm, tā parādot to plusus un mīnusus.

Oriģinālie attēli ir melni punkti uz balta fona. Lai vieglāk būtu apjaust atpazīto elipšu skaitu, katra atrastā elipse tika zīmēta ar patvaļīgi izvēlētu krāsu. Ja tika lietots algoritms ar centru meklēšanu, tad atrastie centri ir atzīmēti kā sarkani punkti.

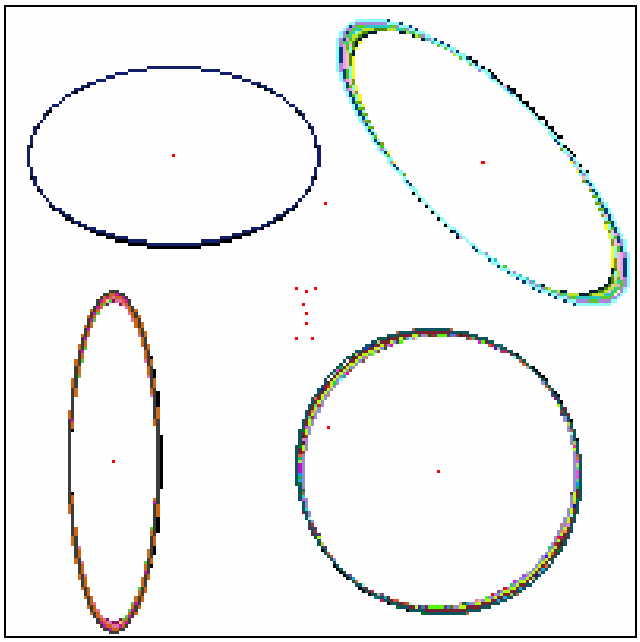
3.1.1. Pirmais piemērs

Pirmajā piemērā apskatīsim trīs elipses un vienu riņķa līniju. Viena elipse ir horizontāla, otra vertikāla un trešā slīpa, kā tas redzams 3.1. attēlā. Attēla izmērs ir 200*200 pikseļi.

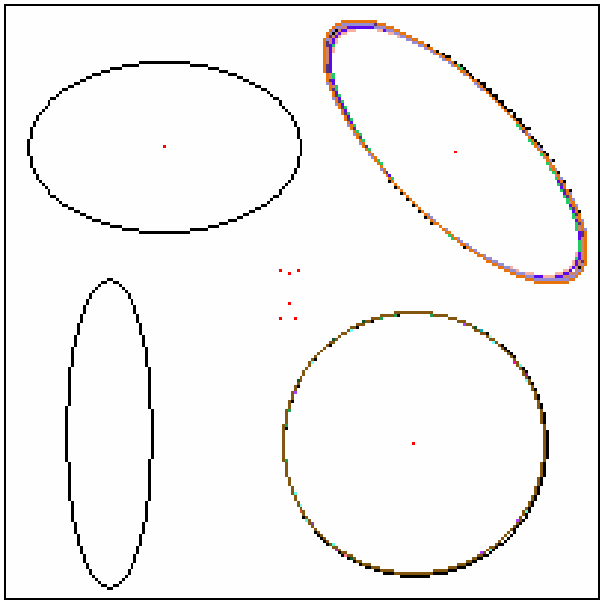


3.1. att. **Ģenerētais attēls ar 3 elipsēm un riņķi**

Aplūkosim iegūtos rezultātus, izmantojot Hafa transformāciju ar centru atrašanu. Atzīmēšu, ka katram centram tiek meklētas visas koncentriskās elipses, kurām atbilstošais skaitītājs ir lielāks par elipses slieksni. Tālāk redzamie rezultāti ir iegūti ar dažādiem centru un elipses sliekšņiem.

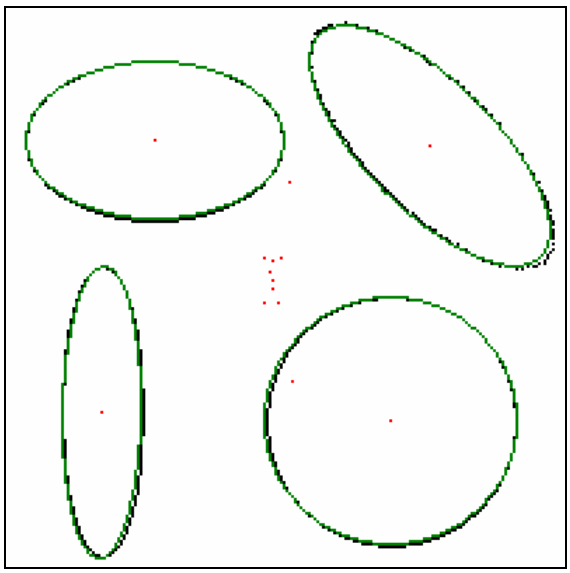


3.2. att. **Atrastās elipses ar centra slieksni 500 un elipses slieksni 80**



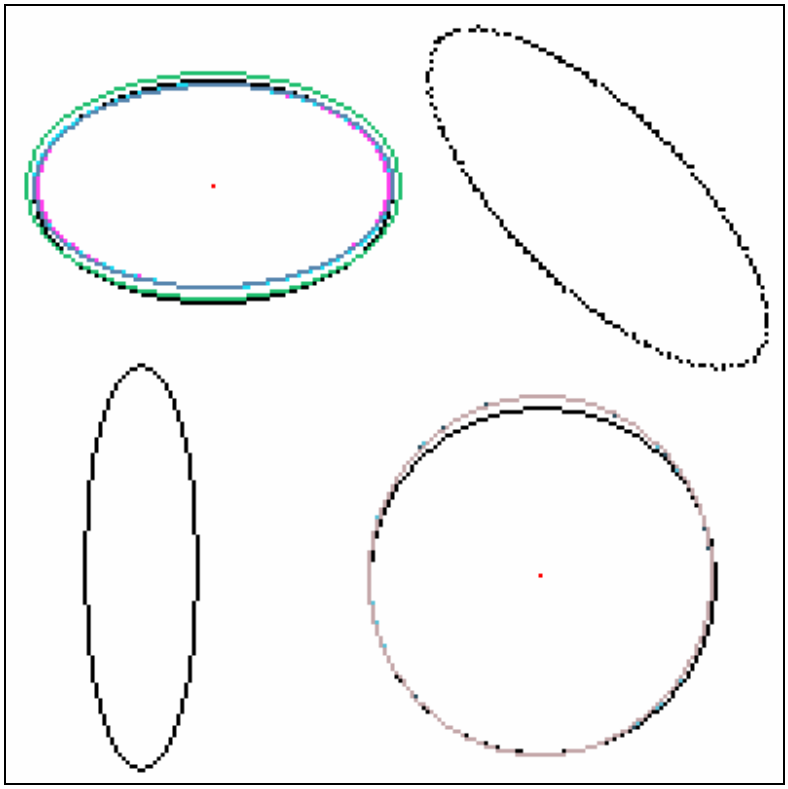
3.3. att. **Atrastās elipses ar centra slieksni 600 un elipses slieksni 90**

Kā redzams 3.2. attēlā, tad četru atsevišķu elipšu gadījumā, šī metode itin labi strādā, vienīgi atrastās elipses ir diezgan daudz, it īpaši slīpās elipses un riņķa gadījumā. To ievērojot, mēs varam palielināt gan minimālo centru parametru, gan minimālo elipses parametru. Rezultāts ir redzams 3.3. attēlā – riņķis un slīpā elipse ir labāk atpazītas, nav vairs tik daudz atrastās elipses, bet horizontālā un vertikālā elipses netika atrastas. Lai tiktu vaļā no liekajām elipsēm, t.i., katru elipsi atpazītu kā tieši vienu objektu, iespējams meklēt tikai pašas populārākās elipses (skat. Attēlu 3.4.).



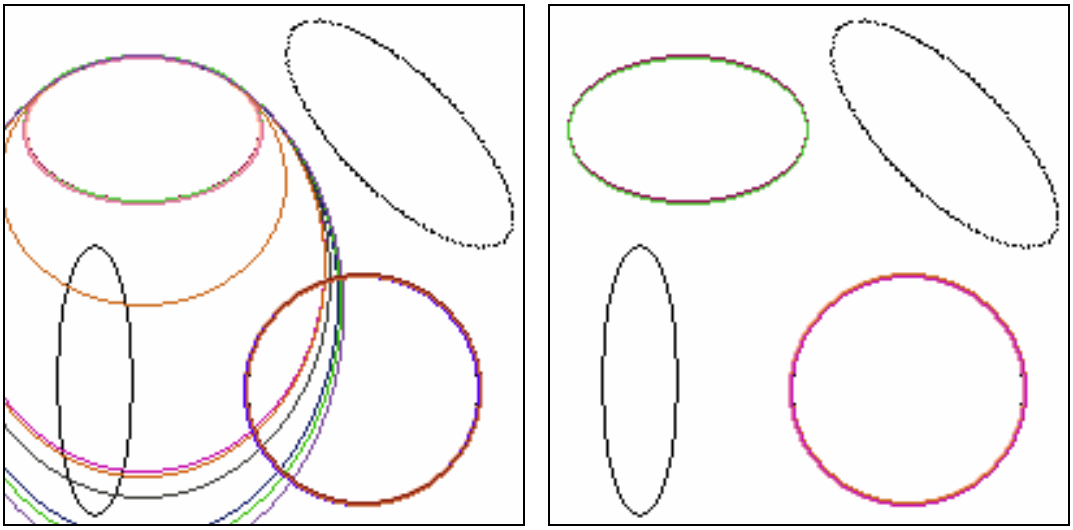
3.4. att. **Atrastās elipses, lietojot algoritmu, kurš katram centram meklē tieši vienu elipsi**

Arī ar otru uzlabotu Hafa transformāciju, rezultāti ir ļoti līdzīgi. Vienīgi centru atrašana strādā labi izteiktām elipsēm (skat. attēlu 3.5). Izmantotais elipses slieksnis ir 60.



3.5. att. Atrastās elipses, izmantojot otro metodi

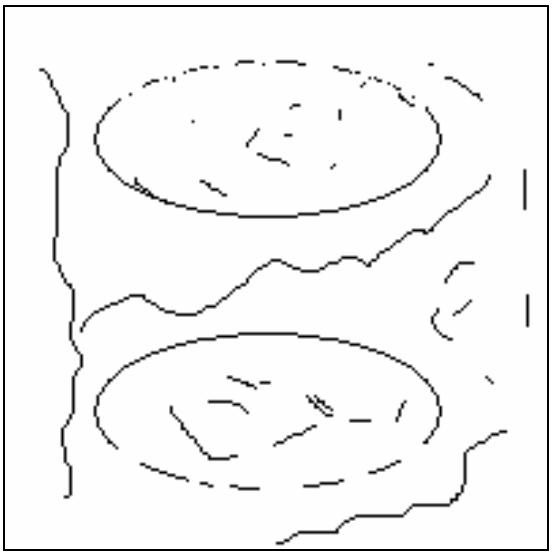
Tagad apskatīsimies, kā ar šīm elipsēm tiek galā implementētā parastā Hafa transformācija horizontālu elipšu atpazīšanai. Redzam, ka izvēloties precīzus parametrus, labi izdodas atrast elipses, tomēr pat neliela parametru izmaiņa var veicināt daudz nepareizu elipšu atrašanu, kā tas ir redzams 3.6. attēlā. Attēlam pa kreisi elipses slieksnis ir 70, bet attēlam pa labi 73.



3.6. att. Atrastās elipses, izmantojot parasto Hafa transformāciju

3.1.2. Otrais piemērs

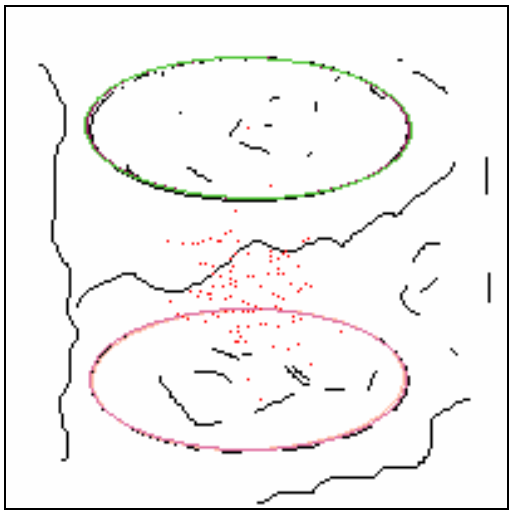
Tagad apskatīsim, kā izstrādātā programma spēj atpazīt izkropļotas elipses ar citām traucējošām līnijām, kā tas redzams 3.7. attēlā. Paredzams, ka šajā gadījumā tiks atrastas diezgan daudz neīstas elipses, kā arī izkropļotās elipses var netikt atrastas.



3.7. att. Divas izkropļotas elipses ar papildus troksni

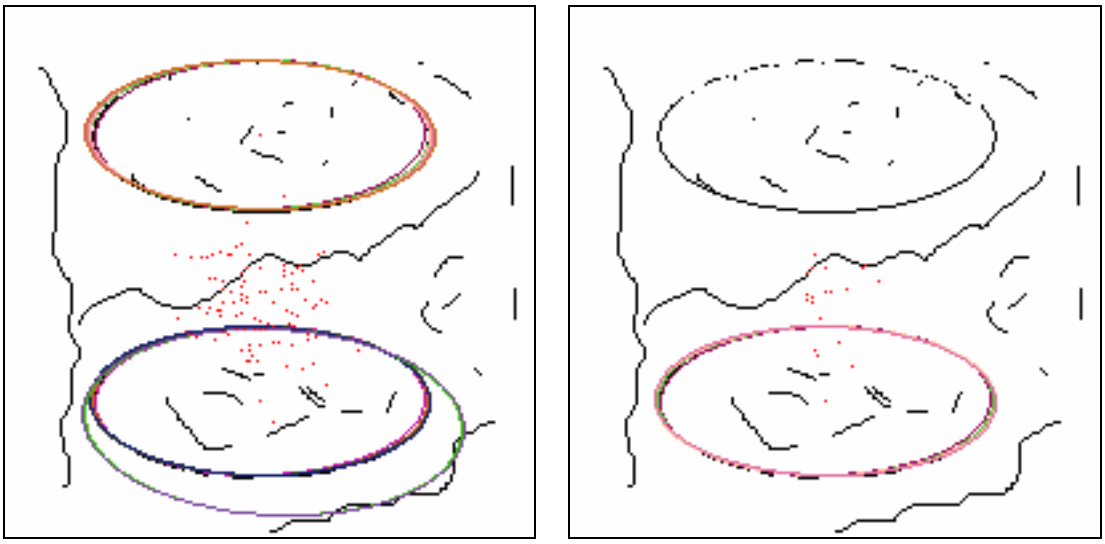
Šajā piemērā pirmās problēmas radās ar centru atrašanu, ja izmantojam centru meklēšanu ar simetrisko punktu pāru palīdzību. Augšējā elipse ir diezgan izkropļota, tāpēc centra slieksnis ir

jāliek samērā mazs, lai varētu atrast tās centru. No otras puses, ja šis parametrs ir mazs, tad dēļ samērā lielā trokšņa tiks atrasti daudzi neīsti elipses centri, kas jūtami palielina programmas darbības laiku. Kā redzams attēlā 3.8, tad piemeklējot īstos parametrus (centra slieksnis 600, elipses slieksnis 85), iespējams ļoti labi atrast tikai mūs interesējošās elipses.



3.8. att. Atrastās izkropļotās elipses piemeklējot ideālus parametrus

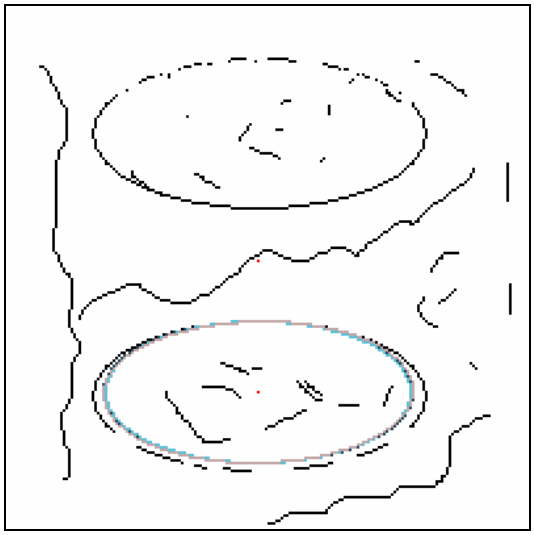
Bet, ja parametri nav gluži piemēroti, tad augšējā elipse var netikt atrasta, toties var parādīties citas neīstas elipses(redzams 3.9. attēlā). Attēlā pa kreisi centra slieksnis 600 un elipses slieksnis 80, bet pa labi centra slieksnis 750 un elipses slieksnis 80.



3.9. att. Atrastās izkropļotās elipses, izmantojot nepiemērotus parametrus

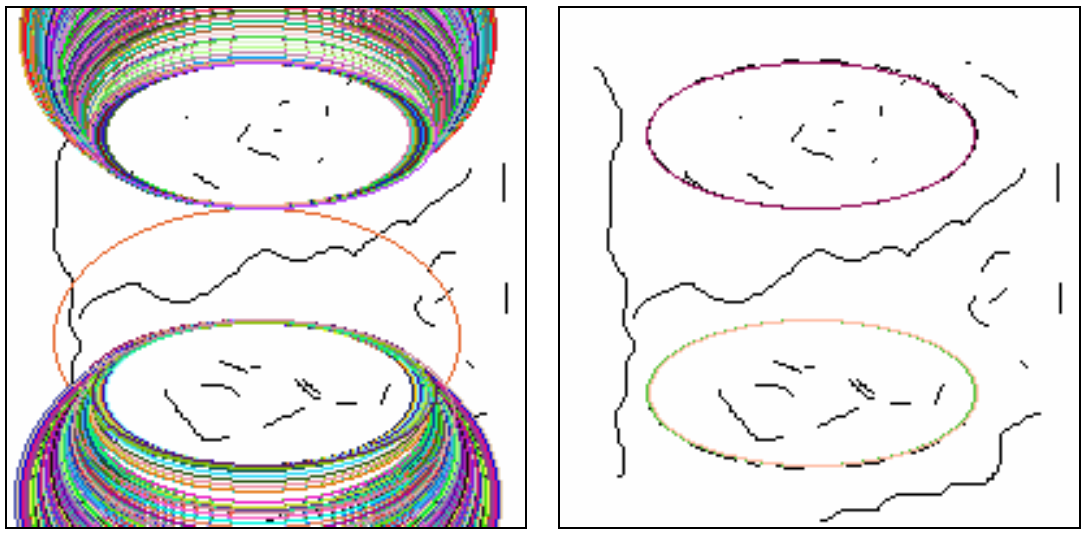
.

Ja centru meklēšanai izmanto ģeometrisko simetriju, tad atrasto centru skaits ir daudz mazāks, tomēr augšējās elipses centrs netika atrast, jo tā ir pārāk daudz izkropļota (skat. attēlu 3.10.). Izmantotais elipses slieksnis ir 60. Apakšējo elipsi gan izdevās diezgan labi atrast, tomēr jāatzīst, ka ar uzlaboto Hafa transformāciju atrastās elipses bija precīzākas.



3.10. att. Atrastās izkropļotās elipses, izmantojot otro metodi

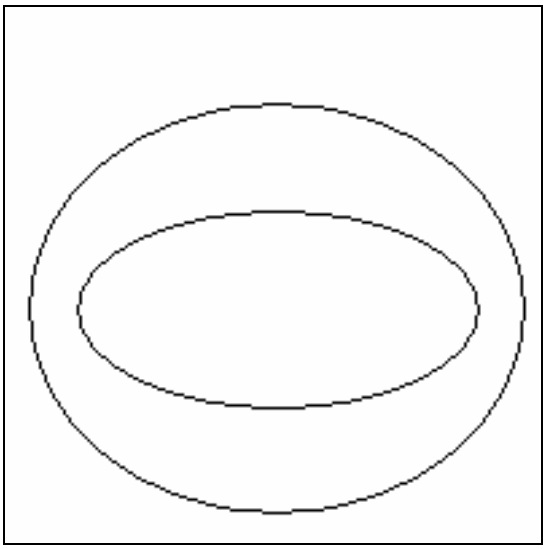
Parastās Hafa transformācijas gadījumā ir labāk, jo nav jāmeklē elipšu centri. Grūtības var sagādāt parametru izvēle, jo var tikt atrastas daudz neīstas elipses, vai arī neatrastas meklētās. Divi rezultāti ir redzami 3.11. attēlā. Attēlam pa kreisi elipses slieksnis ir 70, pa labi 85.



3.11. att. Atrastās izkropļotās elipses, izmantojot parasto Hafa transformāciju

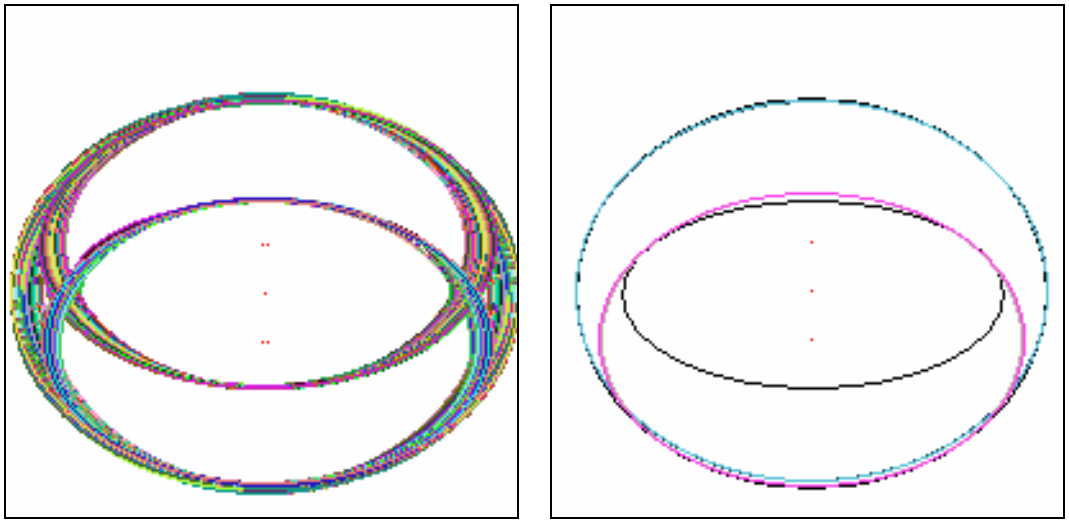
3.1.3 Trešais piemērs

Apskatīsim divas koncentriskas elipses (skat. attēlu 3.12.). Sagaidāms, ka tiks atrastas daudz neīstas elipses, kas sevī ietvers lokus no abām elipsēm. Šo problēmu ne kādi nevar novērst, jo šis algoritms nemēģina kaut kā skatīties uz apkārtējiem punktiem, kas varētu palīdzēt saprast, kurai elipsei kurš punkts pieder.



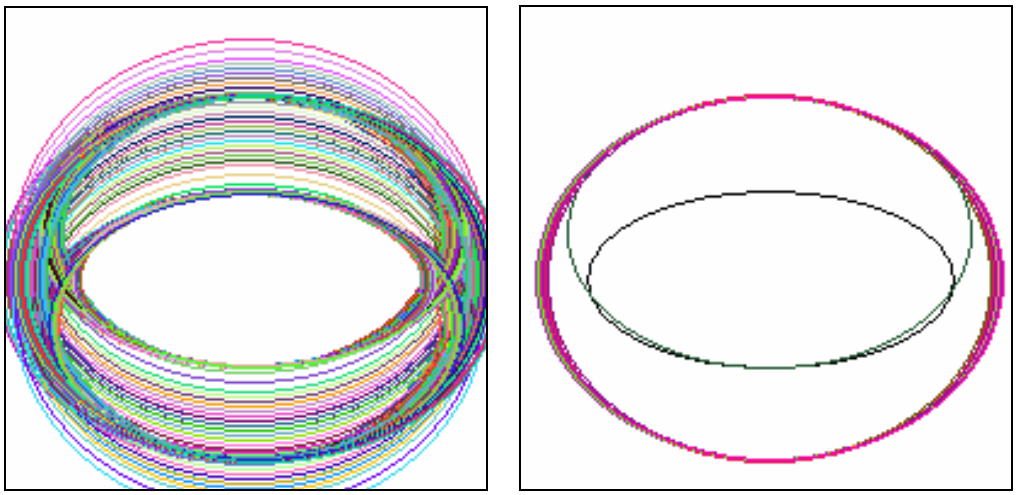
3.12. att. Divas koncentriskas elipses

Abas uzlabotās Hafa transformācijas uzvedās ļoti līdzīgi, jo tās būtiski atšķiras tikai ar centru meklēšanu. Attēlā 3.13. varam redzēt, ka ar abām metodēm atrastie centri ir gandrīz identiski – pa kreisi ir centru meklēšana ar simetrisko pāru palīdzību, bet pa labi ar ģeometriskās simetrijas palīdzību. Vēl no šī attēla var spriest, ka, brīvi izvēloties parametrus, atrasto elipšu skaits ir liels; attiecīgi, attēlā pa kreisi izvēlētais elipses slieksnis ir 100, bet pa labi 150. Attēlā pa kreisi var redzēt, ka ir atrasta arī mazā elipse. Toties attēls pa labi parāda to, ka, samazinot atrasto elipšu skaitu (to var izdarīt palielinot elipses slieksni), tiek atrasta lielākā elipse, bet mazākā elipse nav atrasta. Tas norāda uz problēmu, ka grūti ir atrast dažāda izmēra elipses, jo viņām ir ļoti atšķirīgs punktu skaits.



3.13. att. Atpazītās koncentriskās elipses ar uzlabotajām Hafa transformācijām

Arī ar parasto Hafa transformāciju iegūtie rezultāti ir ļoti līdzīgi (skat. attēlu 3.14). Attēlā pa kreisi izvēlētais elipses slieksnis ir 100, bet pa labi 150

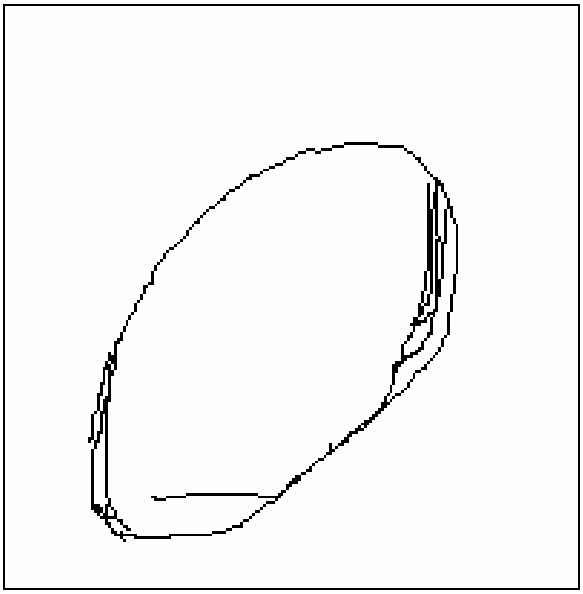


3.14. att. Atpazītās koncentriskās elipses ar parasto Hafa transformāciju

3.1.4 Ceturtais piemērs

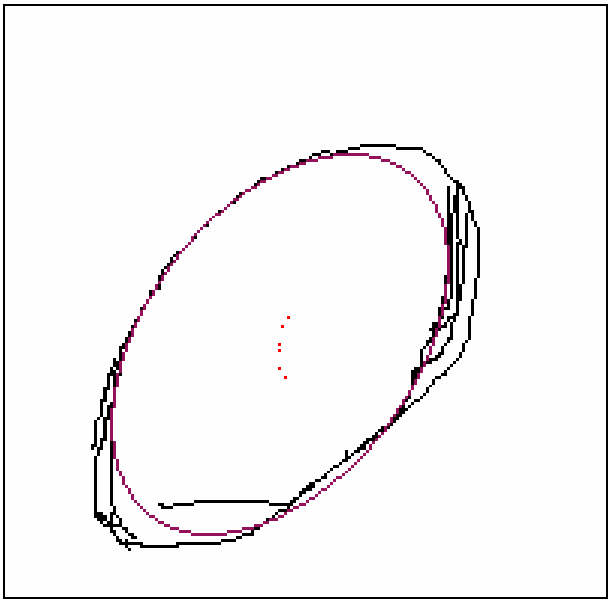
Apskatīsim manis paša zīmētu elipsi zīmēšanas programmā *Paint* (redzama 3.15. attēlā). Elipsi vajadzētu labi atrast, jo nav daudz trokšņa un ir tikai viena elipse. Vienīgi var tikt atrastas vairākas elipses, jo manis zīmētais objekts tikai attāli līdzinās elipsei.

Tā kā elipse ir slīpa, tad nav jēga izmantot parasto Hafa transformāciju, jo implementētā parastā Hafa transformācija spēj atpazīt tikai taisnas elipses. Problēmas radās arī centru atrašanai ar ģeometriskās simetrijas palīdzību, jo šis objekts tikai attāli līdzinās ideālai elipsei.



3.15. att. Ar brīvu roku zīmētā elipse

Attēlā 3.16 arī labi redzams, ka, izvēloties labus parametrus, arī šādai, tikai nedaudz elipsei līdzīgai figūrai var atrast atbilstošu elipsi. Ja parametrus padara nedaudz vaļīgākus, tad var atrast vēl vairākas elipses, kuras labi piekļaujas kādai attēla līnijai, kā tas redzams 19. attēlā.



3.16. att. Atrastā elipse ar centra slieksni 1000 un elipses slieksni 105

3.2. Reālie attēli

Apstrādājot reālus attēlus, gribēju pārliecināties par to, ka, izmantojot Hafa transformāciju, iespējams ar foto kameru fotografētos attēlos atrast elipses. Kā piemērs tika ņemtas šautuves mērķu fotogrāfijas, ar veseliem, un jau sašautiem mērķiem. Veiksmīga elipšu atrašana šajos attēlos varētu palīdzēt automatizētas, reālā laikā strādājošas punktu skaitīšanas sistēmas izstrādē. Gribēju salīdzināt uzlaboto Hafa transformāciju algoritmu darbību uz vienādiem attēliem..

Apskatīju trīs dažādas šautuves mērķu fotogrāfijas – bez izdarītiem šāvieniem, pēc 10 izdarītiem šāvieniem un pēc 30 izdarītiem šāvieniem. No sākuma apskatīsimies, kādi ir iegūtie rezultāti, izmantojot Hafa transformāciju ar centru meklēšanu, izmantojot ģeometrisko simetriju, pielietojot veselajam mērķim (skat. att. 3.17.). Lai parādītu, cik labi šis algoritms atrod elipšu centrus, attēlā ar zilu līniju iezīmētas horizontālās un vertikālās simetrijas asis, ar sarkano punktu elipšu centrs un atrastās elipses ar zilu līniju.

3.17. att. Atrastās elipses ar mainīgu elipses orientācijas leņķi

Kā redzams attēlā 3.17., tika atrastas liekas elipses ar kaut kādu slīpu orientāciju. Patiesībā, pirms šo attēļu apstrādes mēš jau zinām, ka visas elipses ir taisnas, tāpēc to pagrieziena leņķi varam uzskatīt kā 0^0. Tas ne tikai uzlabos programmas ātrdarbību, bet arī netiks atrastas nepareizas elipses. Kā vēlāk izrādījās, taisno elipšu meklēšana šim konkrētajam attēlam ir daudz piemērotāka un iegūtie rezultāti ir precīzāki (skat. Attēlu 3.18)

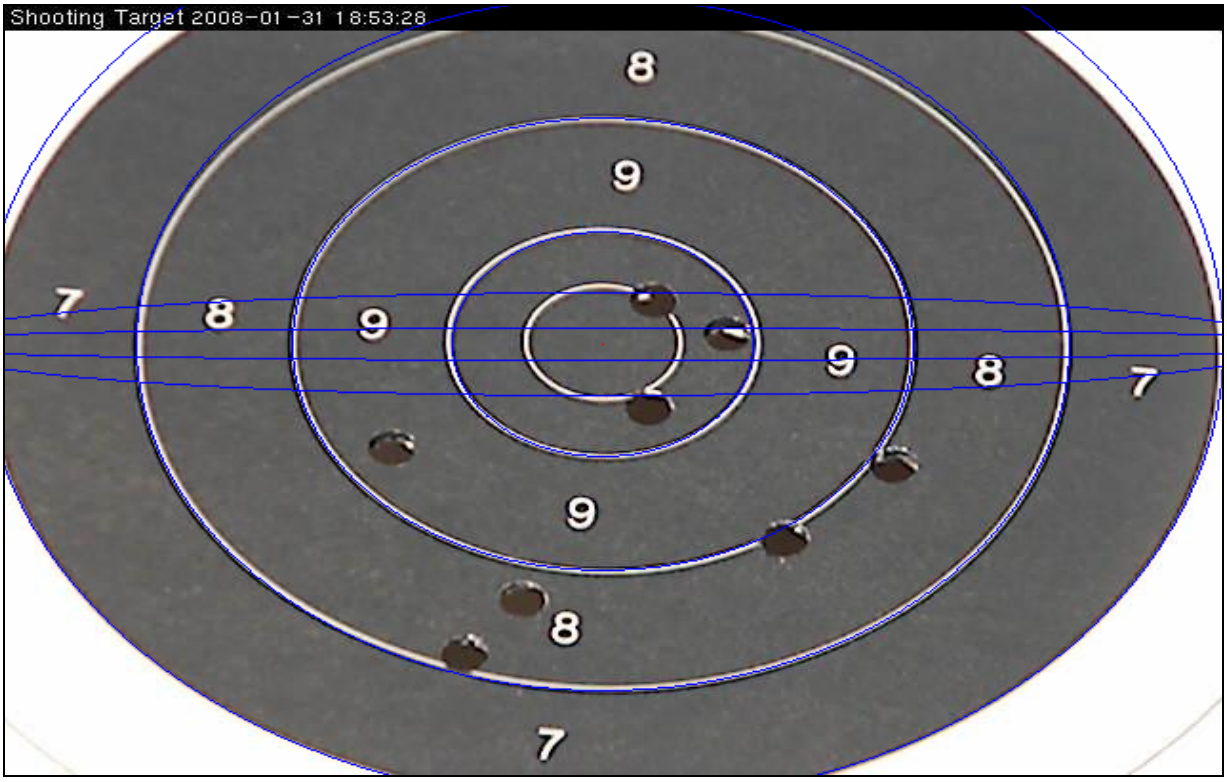
3.18. att. Atrastās taisnās elipses, izmantojot otro metodi

Tā kā tika noskaidrots, ka jāmeklē ir tikai taisnās elipses, tad turpmāk apskatīsim tikai taisno elipšu meklēšanu. Tagad paskatīsimies, kādus rezultātus iegūsim ar otru centru meklēšanas metodi, kurā izmantojam simetriskos punktu pārus. Arī ar šo metodi elipses orģinālajā attēlā var ļoti labi atrast, taču tika atrasta viena lieka elipse, kas pieskaras vairākām mērķu joslu vērtību ciparu kontūrām (skat. Attēlu 3.19). Ar otru uzlaboto Hafa transformāciju, šāda elipse netika atrasta.

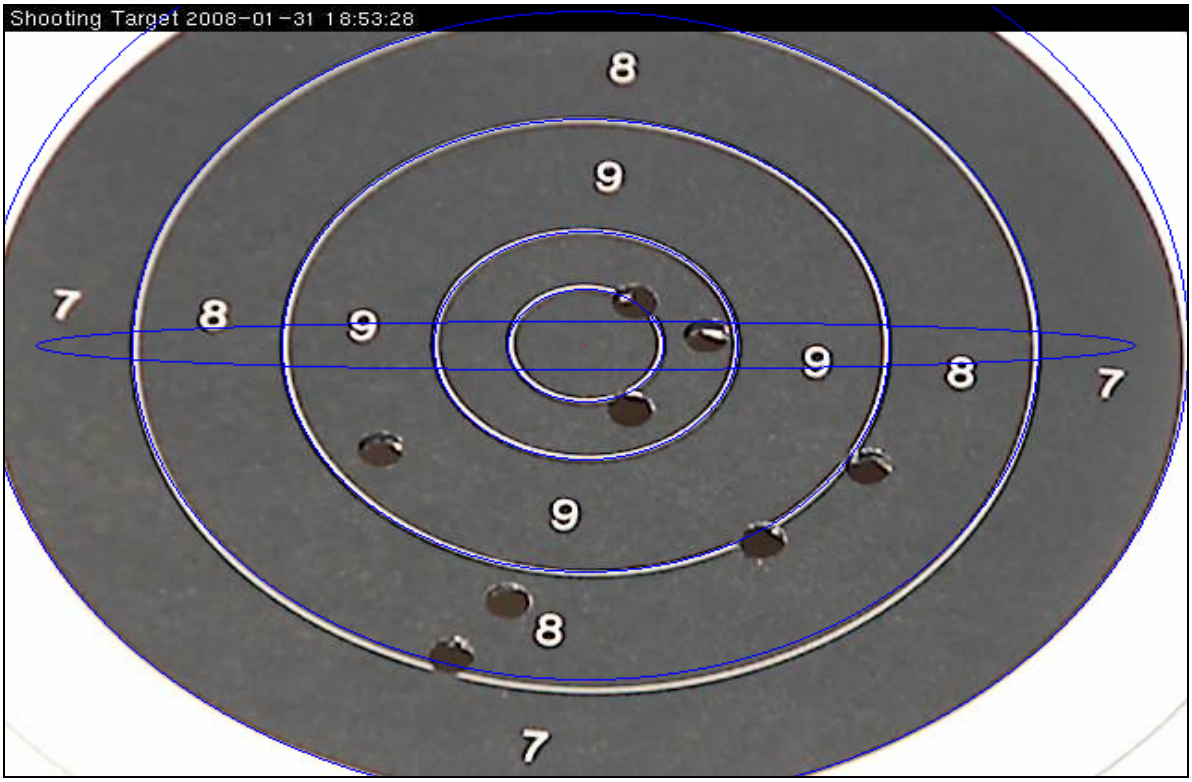
3.19. att. Atrastās taisnās elipses, izmantojot pirmo metodi

Arī sašautajos mērķos elipses tika tīri labi atrastas, problēmas tikai radās ar to, ka dažas elipšu līniijas nu ir pazudušas, kas traucē elipšu atpazīšanai. Ar pirmo metodi tika atrastas vairāk elipses, taču tika atrastas arī nevajadzīgas elipses. Tas ir redzams sašauto mērķu attālos 3.20., 3.21., 3.22. un 3.23.

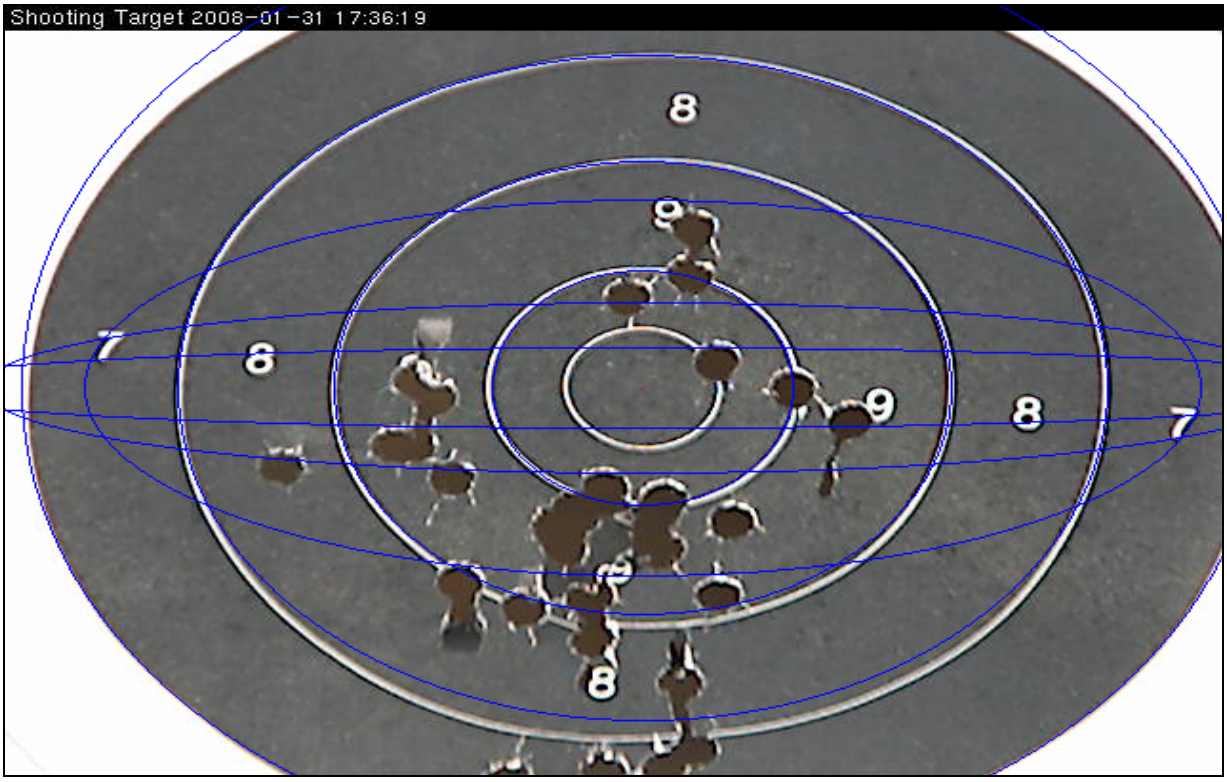
Vēl jāpiezīmē, ka otrās metodes darbības laiks bija īsāks kāpirmās metodes darbības laiks. To varētu izskaidrot ar to, ka otrajā metodē Hafa transformācijai tiek izmantoti tikai tie punkti, ka sir simetriski pret atrasto centru, bet pirmajā metodē tiek apskatīti visi attēla punkti.



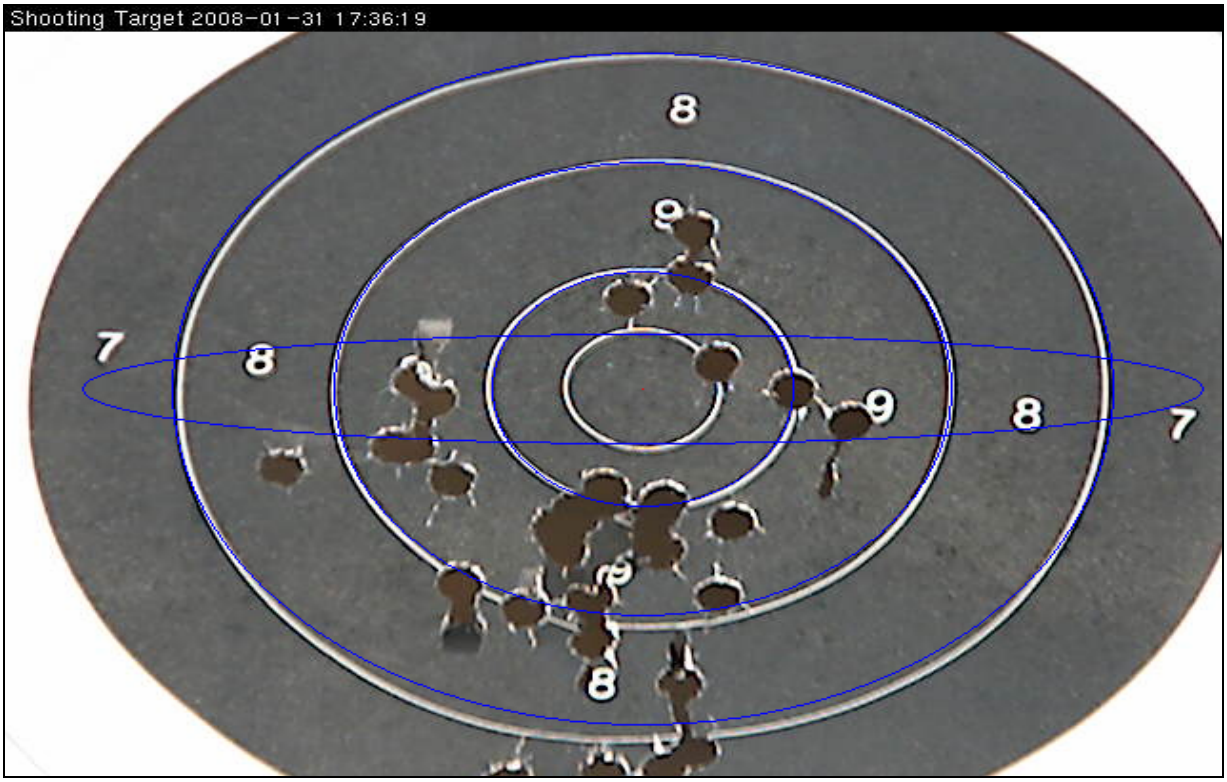
3.20. att. Mērķis ar desmit šāvieniem, apstrādāts ar pirmo metodi



3.21. att. Mērķis ar desmit šāvieniem, apstrādāts ar otro metodi



3.22. att. Mērķis ar trīsdesmit šāvieniem, apstrādāts ar pirmo metodi



3.23. att. Mērķis ar trīsdesmit šāvieniem, apstrādāts ar otro metodi

SECINĀJUMI

Darbā tika aplūkotas vairākas metodes elipšu atpazīšanai attēlos. Vairākas no tām balstījās uz Hafa transformāciju, kas ir viegli realizējama, bet samērā nepraktiska elipšu gadījumā. Analizējot dažādus uz Hafa transformāciju bāzētos algoritmus, atklājās, ka daudz labāka par parasto Hafa transformāciju ir gadījuma Hafa transformācija, kas ir gan ātrākā, gan arī precīzākā uz Hafa transformāciju bāzētā elipšu atpazīšanas metode. Vēl tika aplūkota ļoti vienkārša metode, kura balstīta uz elipses parametru atrašanu izmantojot elipses lielo asi – šī metode atsevišķos gadījumos ir labāka, bet atsevišķos gadījumos sliktāka par Hafa transformāciju.

Ātrākais un efektīvākais darbā apskatītais elipšu atpazīšanas algoritms ir daudzpopulāciju ģenētiskais algoritms, kurš ir ne tikai krietni ātrāks par visiem uz Hafa transformāciju bāzētajiem algoritmiem, bet arī darbojas daudz precīzāk vairāku elipšu un liela trokšņa gadījumos. Tomēr jāatzīst, ka šis algoritms ir sarežģīts un grūti implementējams.

Praktiskajā daļā izstrādātā datorprogramma ļoti veiksmīgi spēj atrast elipses reālos attēlos, tā pierādot, ka Hafa transformācija ir pietiekoši labs līdzeklis elipšu atpazīšanai. Pētot ģenerētos attēlus, atklājās, ka galvenā problēma Hafa transformācijai ir koncentrisku elipšu atrašana. Ja šādu elipšu nav, tad ir samērā viegli atrast katram iespējamajam centram atbilstošo elipsi. Problēmas rodas tad, ja ir vairākas elipses, kuru lokus sevī ietver viena elipse.

Apstrādājot samērā mazus attēlus, atklājās, ka parastā Hafa transformācija, pat ar diezgan lieliem ierobežojumiem, elipšu atpazīšanai aizņem ļoti daudz laika. Tāpēc uzsvars tika likts uz uzlabotu Hafa transformāciju izstrādi. Rezultātā tika izstrādāts jauns Hafa transformācijas paveids, kas sevī ietver elipšu centru meklēšanu no cita ar Hafa transformāciju nesaistīta algoritma. Kā izrādījās, tieši šis algoritms piedāvā samērā labus rezultātus un ātrdarbības ziņā ir tūkstošiem reižu ātrāks kā parastā Hafa transformācija.

Apskatot reālo attēlu apstrādi, var secināt, ka Hafa transformācija strādā ļoti labi. Reizēm gan var būt problēma ar piemērotu parametru izvēli, bet, ja apstrādājamie attēli ir līdzīgi (izmērs, elipšu skaits, orientācija u.c.), tad ar vieniem parametriem iespējams labi atpazīt elipses dažādos attēlos.

IZMANTOTĀ LITERATŪRA

[1] J.C. Russ, *The Image Processing Handbook Fourth Edition*. CRC Press, 2002, pp. 208-248.

[2] J. Canny *A Computational Approach To Edge Detection*, IEEE Trans. Pattern Analysis and Machine Intelligence, 1986, pp. 679-714.

[3] P. Meer, B. Georgescu, *Edge Detection with Embedded Confidence*, IEEE Trans. Pattern Analysis and Machine Intelligence, 23(12), 2001, pp. 1351-1365.

[4] P. V. C. Hough, *Method and means for recognizing complex patterns*. U. S. Patent 3069654, 1962.

[5] Richard O. Duda, Peter E. Hart *Use of the Hough transform to detect lines and curves in pictures*. Comm. ACM, Vol 15, No. 1, 1972.

[6] Y. Lei, K.C. Wong, *Ellipse detection based on symmetry*, Pattern Recognition Letters, 20(1), 1999, pp. 41-47.

[7] J. R. Bergen and H. Shvayster, *A probabilistic algorithm for computing Hough transforms*. Journal of Algorithms, 12, 1991, pp. 639-656.

[8] L. Xu, E. Oja and P. Kultanen, *A new curve detection method: Randomized hough transform (RHT)*. Pattern recognition Letters, 11(5), 1990, pp. 331-338.

[9] Robert A. McLaughlin, *Randomized Hough Transform: Improved ellipse detection with comparison*. Pattern Recognition Letters, Volume 19, 1998, pp. 299-305.

[10] H. K. Yuen, J. Illingworth and J. Kittler, *Detecting partially occluded ellipses using the Hough transform*, Image and Vision Computing 7(1), 1989, pp. 31-37.

[11] C.T. Ho, L.H. Chen, *A Fast Ellipse/Circle Detector Using Geometric Symmetry*, Pattern Recognition, 28, No.1, 1995, pp. 117-124.

[12] Younghong Xie and Qiang Ji, *A new efficient ellipse detection method*. Proceedings of the 16th International Conference on Pattern Recognition, 2002.

[13] Eveluyne Lutton and Patrice Martinez, *A genetic Algorithm for the detection of 2D Geometric Primitives in images,* Proceedings of the 12th International Conference on Pattern Recognition, 1994.

[14] Jie Yao, Nawwaf Kharma and Peter Grogono, *A multi-population genetic algorithm for robust and fast ellipse detection*. Pattern Analysis & Applications, Volume 8, 2005.

PIELIKUMI

1. pielikums. Izstrādātās datorprogrammas kods

```
#include "stdafx.h"
#include <vector>
#define _USE_MATH_DEFINES
#include <math.h>
#include <fstream>

using namespace System;
using namespace System::Drawing;


// globaalas konstantes

static const unsigned int MIN_DISTANCE_BETWEEN_POINTS = 20;

// minimaalais nepiecieshamais gradients, lai punktu uzskatiitu par robezjpunktu
static const int MIN_SOBEL = 1300;
// bildes max izmeeri
static const unsigned int MAX_X = 640;
static const unsigned int MAX_Y = 480;
// pusasu a un b ierobezjojumi
static const unsigned int MIN_A = 10;
static const unsigned int MAX_A = 400;
static const unsigned int MIN_B = 10;
static const unsigned int MAX_B = 400;
// nepiecieshamais punktu skaits, lai punkts tiktu uzskatiits par Ellipses centru
static const unsigned int MIN_POINTS_FOR_CENTER = 70000;
// minimaalais punktu skaits Hafa 3dimensiju telpaa, lai sho punktu uzskatiitu par Ellipses
vienaadojumu
static const unsigned int MIN_POINTS_PER_CELL_WITH_CENTER_FINDING = 100;
// minimaalais punktu skaits Hafa 4dimensiju telpaa, lai sho punktu uzskatiitu par Ellipses
vienaadojumu
static const unsigned int MIN_POINTS_PER_CELL = 100;
// elipshu centru pieljaujamaas veertiibas priksh 4dimensiju Hafa transformaacijas
static const unsigned int MIN_CX = 10;
static const unsigned int MAX_CX = 390;
static const unsigned int MIN_CY = 10;
static const unsigned int MAX_CY = 390;

typedef std::pair<unsigned int, unsigned int> IntPair;
typedef std::vector<unsigned int> IntVector;
typedef std::vector<IntPair> IntPairVector;
typedef std::vector<IntVector> IntVectorVector;
typedef std::vector<IntPairVector> IntPairVectorVector;
typedef std::vector<IntPairVectorVector> IntPairVectorVectorVector;


static unsigned int centerMatrix[MAX_X][MAX_Y] = {{0}}; // centru matrica
static unsigned int EllipseMatrixForCenter[MAX_A-MIN_A][MAX_B-MIN_B][180] = {{{0}}}; // 3
dimensiju telpa
static unsigned int EllipseMatrix[MAX_CX-MIN_CX][MAX_CY-MIN_CY][MAX_A-MIN_A][MAX_B-MIN_B] =
{{{{0}}}}; // 4 dimensiju telpa
// tabulas trigonometriskajaam funkcijaam
static float sinTable[360];
static float cosTable[360];

static int sqrtTable[MAX_B*MAX_B];
static bool pointTable[MAX_X][MAX_Y] = {false};
static IntVectorVector horizontalPoints(MAX_Y);


//tiek izmantota, lai Ellipses parametrus dabuutu aaraa no 3dimensiju telpas
struct EllipseParameters{
       EllipseParameters(unsigned int a, unsigned int b, unsigned int fi): a(a), b(b), fi(fi) {}
       unsigned int a;
       unsigned int b;
```

```
	unsigned int fi;
};

typedef std::vector<EllipseParameters> EllipseParameterVector;

// aizpilda sin un cos tabulas
void fillConstantTables()
{
	for (int i = 0; i < 360; i++)
	{
		sinTable[i] = sin((float)M_PI*i/180);
		cosTable[i] = cos((float)M_PI*i/180);
	}
	for (int i = 0; i < MAX_B*MAX_B; i++)
	{
		sqrtTable[i] = (int)sqrt((float)i);
	}
}

// atrod attaalumu starp diviem punktiem
int getDistance(const IntPair& point1, const IntPair& point2)
{
	int x = point1.first - point2.first;
	int y = point1.second - point2.second;
	return abs(x)+abs(y);
}

// atrod viduspunktus starp katriem diviem punktiem un ieliek centerMatrix tabulaa
void getMiddlePoints(const IntPairVector& points)
{
	for (size_t i = 0; i < points.size(); i++)
	{
		for (size_t j = i; j < points.size(); j++)
		{
			unsigned int x = (points[i].first + points[j].first)/2;
			unsigned int y = (points[i].second + points[j].second)/2;
			if (getDistance(points[i], points[j]) > MIN_DISTANCE_BETWEEN_POINTS) //
neapskataam ljoti tuvus punktus
			{
			if (centerMatrix[x][y] < 65535)
				{
					centerMatrix[x][y]++;
				}
			}
		}
	}
}

// paarbauda vai punkts (x,y) ir vai nav centrs
bool checkForCenter(size_t x, size_t y)
{
	int value = 0;
	for (int i = -1; i < 2; i++)
	{
		for (int j = -1; j < 2; j++)
		{
			if (centerMatrix[x][y] < centerMatrix[x+i][y+j])
			{
				return false;
			}
			value += centerMatrix[x+i][y+j];
		}
	}

	if (value >= MIN_POINTS_FOR_CENTER)
	{
		return true;
	}

	return false;
}

// iet cauri viesiem centerMatrix punktiem un un atgriezj tos punktus, kuri var buut Ellipses
centri
```

```
IntPairVector getEllipseCenters()
{
	IntPairVector result;

	for (size_t x = 3; x < MAX_X - 3; x++)
	{
		for (size_t y = 3; y < MAX_Y - 3; y++)
		{
			if (checkForCenter(x, y))
			{
				result.push_back(IntPair(x, y));
			}
		}
	}

	return result;
}

// atrod vienu iespeejamo elipses centru visaa atteelaa
IntPair getEllipseCenter()
{
	int max_x = 0;
	int max_y = 0;
	unsigned int max_value = 0;

	for (size_t x = 3; x < MAX_X - 3; x++)
	{
		for (size_t y = 3; y < MAX_Y - 3; y++)
		{
			if (centerMatrix[x][y] > max_value)
			{
				max_x = x;
				max_y = y;
				max_value = centerMatrix[x][y];
			}
		}
	}

	return IntPair(max_x, max_y);
}

// atgriezj visus atteela melnos punktus sarakstaa
IntPairVector getPointsFromBitmap(Bitmap^ picture)
{
	IntPairVector result;
	for (int x = 1; x < picture->Width; x++)
	{
		for (int y = 1; y < picture->Height; y++)
		{
			Color^ color = picture->GetPixel(x, y);
			if (color->R == 0 && color->B == 0 && color->G == 0)
			{
				result.push_back(IntPair(x,y));
				pointTable[x][y] = true;
				horizontalPoints[y].push_back(x);
			}
		}
	}
	return result;
}

// ieziimee atteelaa visus atrastos centrus
void drawCenters(const IntPairVector& points, Bitmap^ picture, Color color = Color::Red)
{
	for (size_t i = 0; i < points.size(); i++)
	{
		picture->SetPixel(points[i].first, points[i].second, color);
	}
}

// celj kvadraataa
template <class T>
T sqr(T x)
{
```

```
	return x*x;
}

// atrod otru pusasi b, zinot punktu, Ellipses centru, pusasi a un rotaacijas lenjkji fi
unsigned int getB(const size_t a, const size_t fi, const IntPair& point, const IntPair& center)
{
	int x = point.first - center.first;
	int y = point.second - center.second;
	float temp1 = sqr(x*cosTable[fi] - y*sinTable[fi]);
	float temp2 = sqr(x*sinTable[fi] + y*cosTable[fi]);
	float bSquare = sqr(a)*temp2/(sqr(a)-temp1);
	if (bSquare < MAX_B*MAX_B && bSquare > MIN_B*MIN_B)
	{
		return sqrtTable[(int)bSquare];
	}
	else
	{
		return MAX_B;
	}
}

// ieziimee atteelaa elipsi, ja zinaams taas centrs, pusasis a un b, kaa arii rotaacijas lenjkjis
fi
void drawEllipse(const IntPair& center, int a, int b, int fi, Bitmap^ picture)
{
	Random^ random = gcnew Random(rand());
	Graphics^ g = Graphics::FromImage(picture);
	g->TranslateTransform((float)center.first, (float)center.second);
	g->RotateTransform((float)-fi);
	//g->DrawEllipse(gcnew Pen(Color::FromArgb(random->Next(0xff000000, 0xffffffff))), -a, -b,
a*2, b*2);
	g->DrawEllipse(gcnew Pen(Color::Blue), -a, -b, a*2, b*2);
}

// ieziimee atteelaa taisni
void drawLine(const IntPair& parameters, Bitmap^ picture)
{
	Random^ random = gcnew Random(rand());
	Graphics^ g = Graphics::FromImage(picture);
	g->RotateTransform((float)parameters.first);
	g->TranslateTransform((float)parameters.second, (float)0);

	g->DrawLine(gcnew Pen(Color::Blue), 0, -1000, 0, 1000);
}


// konkreetam centram veic Hafa transformaaciju
void fillEllipseMatrixForCenter(const IntPair& center, const IntPairVector& points)
{
	for (size_t a = MIN_A; a < MAX_A; a++)
	{
		for (size_t fi = 0; fi < 45; fi++)
		{
			for (size_t p = 0; p < points.size(); p++)
			{
				unsigned int b = getB(a, fi, points[p], center);
				if ((b < MAX_B) && (b >= MIN_B))
				{
					EllipseMatrixForCenter[a-MIN_A][b-MIN_B][fi]++;
				}
			}
		}
	}
}

// veic Hafa transformaaciju uz 4 dimensiju telpu
void fillEllipseMatrix(const IntPairVector& points)
{
	for (size_t cx = MIN_CX; cx < MAX_CX; cx++)
	{
		for (size_t cy = MIN_CY; cy < MAX_CY; cy++)
		{
			for (size_t a = MIN_A; a < MAX_A; a++)
			{
```

```
                                for (size_t p = 0; p < points.size(); p++)
                                {
                                        unsigned int b = getB(a, 0, points[p], IntPair(cx, cy));
                                        if ((b < MAX_B) && (b >= MIN_B))
                                        {
                                                EllipseMatrix[cx-MIN_CX][cy-MIN_CY][a-MIN_A][b-
MIN_B]++;
                                        }
                                }
                        }
                }
                Console::WriteLine(cx);
        }
}

// paarbauda vai max Hafa transformaacijas 3 dimensiju telpas populaaraakais punkts ir
pietiekoshi populaars
bool isGoodParameters(const EllipseParameters& par, unsigned int minCount)
{
        return minCount <= EllipseMatrixForCenter[par.a][par.b][par.fi];
}

// atrod populaaraako Hafa transformaacijas 3 dimensiju telpas punktu
EllipseParameterVector getEllipsesParametersForFixedCenter()
{
        EllipseParameterVector result;
        EllipseParameters tmp(0,0,0);
        unsigned int maxValue = 0;
        for (unsigned int a = 0; a < MAX_A - MIN_A; a++)
        {
                for (unsigned int b = 0; b < MAX_B - MIN_B; b++)
                {
                        for (unsigned int fi = 0; fi < 180; fi++)
                        {
                                //if (EllipseMatrixForCenter[a][b][fi] >
MIN_POINTS_PER_CELL_WITH_CENTER_FINDING)
                                //{
                                //      result.push_back(EllipseParameters(a, b, fi));
                                //}
                                if (EllipseMatrixForCenter[a][b][fi] > maxValue)
                                {
                                        maxValue = EllipseMatrixForCenter[a][b][fi];
                                        tmp.a = a;
                                        tmp.b = b;
                                        tmp.fi = fi;
                                }
                        }
                }
        }

        return result;
}

// aizpilda Hafa transformaacijas skaitiitaaju tabulu
void fillEllipsesParameterList(EllipseParameterVector* ellipsesParameterList, unsigned int
minCount)
{
        for (unsigned int a = 0; a < MAX_A - MIN_A; a++)
        {
                for (unsigned int b = 0; b < MAX_B - MIN_B; b++)
                {
                        for (unsigned int fi = 0; fi < 1; fi++)
                        {
                                if (EllipseMatrixForCenter[a][b][fi] > minCount)
                                {

        ellipsesParameterList[EllipseMatrixForCenter[a][b][fi]].push_back(EllipseParameters(a, b,
fi));
                                }
                        }
                }
        }
}
```

```
// nodroshina, lai netiktu atrastas divas paaraak tuvas elipses
bool checkParameters(const EllipseParameters& parameters, const EllipseParameterVector
foundEllipses)
{
	for (size_t i = 0; i < foundEllipses.size(); i++)
	{
		if (abs((int)(foundEllipses[i].a - parameters.a)) < 20)
		{
			if (abs((int)(foundEllipses[i].fi - parameters.fi)) < 20)
			{
				return false;
			}
		}
		if (abs((int)(foundEllipses[i].b - parameters.b)) < 20)
		{
			if (abs((int)(foundEllipses[i].fi - parameters.fi)) < 20)
			{
				return false;
			}
		}
	}
	return true;
}

// atrod koncentriskas elipses
EllipseParameterVector getEllipsesForConcentricEllipses(unsigned int minCount)
{
	EllipseParameterVector result;
	EllipseParameterVector ellipsesParameterList[1000] = {};
	fillEllipsesParameterList(ellipsesParameterList, minCount);
	EllipseParameterVector foundEllipses;
	for (int i = 1000; i > 0; i--)
	{
		for (unsigned int j = 0; j < ellipsesParameterList[i].size(); j++)
		{
			if (checkParameters(ellipsesParameterList[i][j], foundEllipses))
			{
				foundEllipses.push_back(ellipsesParameterList[i][j]);
				result.push_back(ellipsesParameterList[i][j]);
			}
		}
	}
	return result;
}

// atrod un ieziimee atteelaa visas Ellipses, ieejaa sanjem atrastos centrus un atteela melnos
punktus
void getEllipsesForFoundCenters(const IntPairVector& centers, const IntPairVector& points,
Bitmap^ picture, unsigned int minCount = MIN_POINTS_PER_CELL_WITH_CENTER_FINDING)
{
	Console::WriteLine("Atrasti {0} iespeejamie Ellipses centri", centers.size());
	for (size_t c = 0; c < centers.size(); c++)
	{
		memset(EllipseMatrixForCenter, 0, sizeof(EllipseMatrixForCenter));
		fillEllipseMatrixForCenter(centers[c], points);
		//EllipseParameterVector ellipses = getEllipsesParametersForFixedCenter();
		EllipseParameterVector ellipses = getEllipsesForConcentricEllipses(minCount);
		for (size_t i = 0; i < ellipses.size(); i++)
		{
			if (isGoodParameters(ellipses[i], minCount))
			{
				drawEllipse(centers[c], ellipses[i].a+MIN_A, ellipses[i].b+MIN_B,
ellipses[i].fi, picture);
			}
		}
		Console::WriteLine(c);
	}
}

// iet cauri visai Hafa transformaacijas 4 dimensiju telpai un ieziimee taas Ellipses, kuras ir
pietiekoshi populaaras
void getEllipses(const IntPairVector& points, Bitmap^ picture)
{
	fillEllipseMatrix(points);
```

```
	for (size_t cx = MIN_CX; cx < MAX_CX; cx++)
	{
		for (size_t cy = MIN_CY; cy < MAX_CY; cy++)
		{
			for (size_t a = MIN_A; a < MAX_A; a++)
			{
				for (size_t b = MIN_B; b < MAX_B; b++)
				{
					if (EllipseMatrix[cx-MIN_CX][cy-MIN_CY][a-MIN_A][b-MIN_B] >
MIN_POINTS_PER_CELL)
					{
						drawEllipse(IntPair(cx, cy), a, b, 0, picture);
					}
				}
			}
		}
	}
}

// atrod kontuuru punktus, izmantojot Sobela operatoru
IntPairVector getEdgesWithSobelOperator(Bitmap^ picture)
{
	IntPairVector result;
	IntVectorVector pixelArray(picture->Width, IntVector(picture->Height, 0));
	for (int x = 0; x < picture->Width; x++)
	{
		for (int y = 0; y < picture->Height; y++)
		{
			Color pixel = picture->GetPixel(x, y);
			pixelArray[x][y] = pixel.R + pixel.G + pixel.B;
		}
	}
	int Gx[3][3] = {{-1, 0, 1}, {-2, 0, 2}, {-1, 0, 1}};
	int Gy[3][3] = {{1, 2, 1}, {0, 0, 0}, {-1, -2, -1}};
	for (int x = 1; x < picture->Width - 1; x++)
	{
		for (int y = 1; y < picture->Height - 1; y++)
		{
			int g_x = 0;
			int g_y = 0;
			for (unsigned int i = 0; i < 3; i++)
			{
				for (unsigned int j = 0; j < 3; j++)
				{
					g_x += Gx[i][j]*pixelArray[x-1+i][y-1+j];
					g_y += Gy[i][j]*pixelArray[x-1+i][y-1+j];
				}
			}
			if (abs(g_x)+abs(g_y) > MIN_SOBEL)
			{
				result.push_back(IntPair(x, y));
				pointTable[x][y] = true;
				horizontalPoints[y].push_back(x);
			}
		}
	}
	return result;
}

// nodroshina, lai netiktu atrastas divas ljoti tuvas taisnes
bool checkLineParameters(const IntPair& parameters, const IntPairVector foundLines)
{
	for (size_t i = 0; i < foundLines.size(); i++)
	{
		if ( (abs((int)(foundLines[i].first - parameters.first)) < 5) ||
((abs((int)(foundLines[i].first - parameters.first)) > 355)) )
		{
			if (abs((int)(foundLines[i].second - parameters.second)) < 15)
			{
				return false;
			}
		}
	}
```

```
	return true;
}

// atrod taisnes parametrus, izmantojot Hafa transformaaciju
IntPairVectorVector makeHoughTransformForLines(IntPairVectorVectorVector& points, unsigned int
minTreshold, Bitmap^ picture)
{
	IntPairVectorVector result;
	IntPairVectorVectorVector acc(360, IntPairVectorVector(MAX_X, IntPairVector()));

	for (int x = 0; x < MAX_X; x++)
	{
		for (int y = 0; y < MAX_Y; y++)
		{
			if (points[x][y].size() > 0)
			{
				for (unsigned int fi = 0; fi < 360; fi++)
				{
					int r = (int)(x*cosTable[fi] + y*sinTable[fi]);
					if ((r > 0) && (r < MAX_X))
					{
						acc[fi][r].push_back(IntPair(x, y));
					}
				}
			}
		}
	}

	IntPairVectorVector lineParameterList(1000, IntPairVector());

	for (unsigned int fi = 0; fi < 360; fi++)
	{
		for (unsigned int r = 0; r < MAX_X; r++)
		{
			if (acc[fi][r].size() > minTreshold)
			{
				lineParameterList[acc[fi][r].size()].push_back(IntPair(fi, r));
			}
		}
	}

	IntPairVector foundLines;

	for (int i = 999; i > 0; i--)
	{
		for (unsigned int j = 0; j < lineParameterList[i].size(); j++)
		{
			if (checkLineParameters(lineParameterList[i][j], foundLines))
			{
				foundLines.push_back(lineParameterList[i][j]);
			}
		}
	}

	for (size_t i = 0; i < foundLines.size(); i++)
	{
		int fi = foundLines[i].first;
		int r = foundLines[i].second;
		Console::WriteLine("Atradam taisni {0} {1} {2}",acc[fi][r].size(), fi, r);
		IntPairVector tmp(1, foundLines[i]);
		tmp.insert(tmp.begin(), acc[fi][r].begin(), acc[fi][r].end());
		result.push_back(tmp);
		//drawLine(foundLines[i], picture);
	}


	return result;
}

IntPair getLinesCrossingPoint(int fi1, int r1, int fi2, int r2)
{
	int y = (int)((r2*cosTable[fi1]-r1*cosTable[fi2])/(cosTable[fi1]*sinTable[fi2]-
sinTable[fi1]*cosTable[fi2]));
	int x = (int)((r1-y*sinTable[fi1])/cosTable[fi1]);
```

```
	return IntPair(x, y);
}

void getElipsesForVerticalSymetricAxes(const IntPair& parameters, const IntVectorVector&
verticalPoints, Bitmap^ picture)
{
	IntPairVector middlePoints;
	IntPairVectorVectorVector middlePoinTable(MAX_X, IntPairVectorVector(MAX_Y,
IntPairVector()));
	IntPairVector points;
	for (size_t i = 0; i < verticalPoints.size(); i++)
	{
		for (size_t j = 0; j < verticalPoints[i].size(); j++)
		{
			points.push_back(IntPair(i, verticalPoints[i][j]));
			for (size_t k = j+1; k < verticalPoints[i].size(); k++)
			{
				if (verticalPoints[i][k] - verticalPoints[i][j] >
MIN_DISTANCE_BETWEEN_POINTS)
				{
					unsigned int y = (verticalPoints[i][k] +
verticalPoints[i][j]) / 2;
					middlePoinTable[i][y].push_back(IntPair(i,
verticalPoints[i][k]));
					middlePoinTable[i][y].push_back(IntPair(i,
verticalPoints[i][j]));
					middlePoints.push_back(IntPair(i ,y));
				}
			}
		}
	}

	IntPairVectorVector lineParameters = makeHoughTransformForLines(middlePoinTable, 300,
picture);

	int fi1 = parameters.first;
	int r1 = parameters.second;
	IntPairVector centers;
	for (size_t i = 0; i < lineParameters.size(); i++)
	{
		int fi2 = lineParameters[i][lineParameters[i].size()-1].first;
		if (fi2 == 0)
		{
			continue;
		}
		int r2 = lineParameters[i][lineParameters[i].size()-1].second;
		IntPair parameters = lineParameters[i][lineParameters[i].size()-1];
		centers.push_back(getLinesCrossingPoint(fi1, r1, fi2, r2));
	}

	drawCenters(centers, picture, Color::Red);

	picture->Save("picture_step_5.bmp");

	getEllipsesForFoundCenters(centers, points, picture, 100);

}

IntPairVector getCentersWithSimmetry(Bitmap^ picture)
{
	IntPairVector result;
	IntPairVector middlePoints;

	IntPairVectorVectorVector middlePoinTable(MAX_X, IntPairVectorVector(MAX_Y,
IntPairVector()));

	for (size_t i = 0; i < horizontalPoints.size(); i++)
	{
		for (size_t j = 0; j < horizontalPoints[i].size(); j++)
		{
			for (size_t k = j+1; k < horizontalPoints[i].size(); k++)
			{
				if (horizontalPoints[i][k] - horizontalPoints[i][j] >
MIN_DISTANCE_BETWEEN_POINTS)
```

```
                                {
                                        unsigned int x = (horizontalPoints[i][k] +
horizontalPoints[i][j]) / 2;

        middlePoinTable[x][i].push_back(IntPair(horizontalPoints[i][k] ,i));

        middlePoinTable[x][i].push_back(IntPair(horizontalPoints[i][j] ,i));
                                        middlePoints.push_back(IntPair(x, i));
                                }
                        }
                }
        }

        picture->Save("picture_step_1.bmp");

        IntPairVectorVector lineParameters = makeHoughTransformForLines(middlePoinTable, 300,
picture);

        picture->Save("picture_step_2.bmp");

        for (size_t i = 0; i < lineParameters.size(); i++)
        {
                IntPair parameters = lineParameters[i][lineParameters[i].size()-1];
                IntVectorVector verticalPoints(MAX_X);
                for (size_t j = 0; j < lineParameters[i].size() - 1; j++)
                {
                        int x = lineParameters[i][j].first;
                        int y = lineParameters[i][j].second;
                        for (size_t k = 0; k < middlePoinTable[x][y].size(); k++)
                        {

        verticalPoints[middlePoinTable[x][y][k].first].push_back(middlePoinTable[x][y][k].second);
                        }
                }
                getElipsesForVerticalSymetricAxes(parameters, verticalPoints, picture);
        }

        picture->Save("picture_with_midle_points.bmp");

        return result;
}

void findConcentricElipses()
{
        Bitmap^ picture = gcnew Bitmap("picture.bmp");

        picture->Save("pictureEdges.bmp");

        IntPairVector points = getEdgesWithSobelOperator(picture);

        getCentersWithSimmetry(picture);
        return;
        getMiddlePoints(points);
        IntPairVector centers = getEllipseCenters();
        drawCenters(centers, picture);
        getEllipsesForFoundCenters(centers, points, picture);
        picture->Save("pictureProcessedWithCenter.bmp");
}

int main(array<System::String ^> ^args)
{
        fillConstantTables();

        findConcentricElipses();

        Bitmap^ picture = gcnew Bitmap("picture.bmp");
        IntPairVector points;
        points = getEdgesWithSobelOperator(picture);
        drawCenters(points, picture);
        picture->Save("picture_with_edges.bmp");

        getMiddlePoints(points);
        IntPairVector centers = getEllipseCenters();
```

```
        getEllipsesForFoundCenters(centers, points, picture);
        drawCenters(centers, picture);
        picture->Save("pictureProcessedWithCenters.bmp");

        picture = gcnew Bitmap("picture.bmp");
        getEllipses(points, picture);
        picture->Save("pictureProcessed.bmp");

    return 0;
}
```

Maģistra darbs „Elipšu atpazīšana attēlos”

Ar savu parakstu apliecinu, ka pētījums veikts patstāvīgi, izmantoti tikai tajā norādītie informācijas avoti un iesniegtā darba elektroniskā kopija atbilst izdrukai. **Piekrītu sava darba publicēšanai internetā.**

Autors: _____
(Autora paraksts)

Ar savu parakstu apliecinu, ka esmu lasījis augstāk minēto maģistra darbu un atzīstu to par p i e m ē r o t u / n e p i e m ē r o t u (nevajadzīgo svītrot) aizstāvēšanai Latvijas Universitātes datorzinātņu maģistrantūrā.

Darba vadītājs: _____
(Vadītāja paraksts)

Darbs iesniegts Datorikas nodaļā _____.
(Iesniegšanas datums)

Ar šo es apliecinu, ka darba elektroniskā versija ir augšupielādēta LU informatīvajā sistēmā.
Metodiķe: _____.
(Metodiķes paraksts)

Recenzents: _____
(Recenzenta paraksts)

Darbs aizstāvēts maģistra gala pārbaudījuma komisijas sēdē

_____ prot. Nr. _____, vērtējums_____
(Darba aizstāvēšanas datums)

Komisijas sekretārs: _____
(Sekretāra paraksts)