Technická univerzita v Košiciach

Fakulta elektrotechniky a informatiky

Využitie Houghovej transformácie pri rozpoznávaní spŕšok tvorených časticami ultravysokých energií

Diplomová práca

2015 **Jozef Vasilko**

Technická univerzita v Košiciach

Fakulta elektrotechniky a informatiky

Využitie Houghovej transformácie pri rozpoznávaní spŕšok tvorených časticami ultravysokých energií

Diplomová práca

Študijný program:	Informatika
Študijný odbor:	9.2.1 Informatika
Školiace pracovisko:	Katedra počítačov a informatiky (KPI)
Školiteľ:	doc. Ing. Ján Genči, PhD.
Konzultant:	RNDr. Pavol Bobik, PhD.

Košice 2015 **Jozef Vasilko**

Erráta

Využitie Houghovej transformácie pri rozpoznávaní spŕšok tvorených časticami ultravysokých energií

Jozef Vasilko

Košice 2015

Abstrakt v SJ

JEM-EUSO je pripravovaný medzinárodný vesmírny experiment, ktorého cieľmi sú identifikácia hlavných zdrojov kozmického žiarenia ultravysokých energií a určenie jeho charakteristík. Táto práca sa, v rámci softvérovej prípravy experimentu JEM-EUSO, venuje návrhu, implementácii a analýze výsledkov algoritmov rozpoznávania vzorov, založených na vybraných metódach Houghovej transformácie, pre spŕšky tvorené časticami ultravysokých energií. Hlavným cieľom je, po integrácii navrhnutých algoritmov do softvérového rámca ESAF, ich aplikovanie na údaje spŕšok generovaných simulačnou časťou rámca ESAF. Na základe výstupu algoritmov sa určia závislosti presnosti rekonštrukcie spŕšok pri štandardnom a rastúcom UV pozadí.

Kľúčové slová

Algoritmy rozpoznávania vzorov, JEM-EUSO experiment, Houghova transformácia, kozmické žiarenie ultravysokých energií

Abstrakt v AJ

JEM-EUSO is the international space experiment in the phase of preparations, with intentions of finding the sources of ultra-high energy cosmic rays and studying their characteristics. This diploma focuses on design, implementation and analysis of results of pattern recognition algorithms, based on selected methods of Hough transform, for extensive air showers from ultra-high energy cosmic rays, in the scope of JEM-EUSO software preparations. After the integration of proposed algorithms into software framework ESAF, there is intention of application the algorithms on simulated data of extensive air showers. Based on the output of algorithms, the accuracy of reconstruction and its dependencies will be determined under the conditions of standard and rising UV background.

Kľúčové slová v AJ

Pattern recognition algorithms, JEM-EUSO experiment, Hough transform, ultra-high energy cosmic rays

TECHNICKÁ UNIVERZITA V KOŠICIACH
FAKULTA ELEKTROTECHNIKY A INFORMATIKY
Katedra počítačov a informatiky

ZADANIE DIPLOMOVEJ PRÁCE

Študijný odbor: **9.2.1 Informatika**
Študijný program: **Informatika**

Názov práce:

Využitie Houghovej transformácie pri rozpoznávaní spŕšok tvorených časticami ultravysokých energií
Application of Hough transformation for detecting showers of ultra high energy particles

Študent:	**Bc. Jozef Vasilko**
Školiteľ:	**doc. Ing. Ján Genči, PhD.**
Školiace pracovisko:	**Katedra počítačov a informatiky**
Konzultant práce:	**RNDr. Pavol Bobik, PhD.**
Pracovisko konzultanta:	**Ústav experimentálnej fyziky SAV, Košice**

Pokyny na vypracovanie diplomovej práce:

1. Oboznámiť sa so základmi fyzikálnej problematiky detekcie častíc ultravysokých energií.
2. Preštudovať si princípy fungovania JEM-EUSO experimentu so zameraním sa na rozlíšenie vzorov s charakteristikami spŕšok kozmického žiarenia v registrovanom UV signále.
3. Aplikácia Houghovej transformácie na signál tvorený spŕškami častíc ultravysokých energií (UHECR) v UV pozadí na nočnej strane Zeme.
4. Vývoj kódu pre rozpoznávanie UHECR spŕšok na základe Houghovej transformácie pre spŕšky generované simulačnou časťou frameworku ESAF.
5. Určenie faktorov ovplyvňujúcich rozpoznanie signálu spŕšky vzhľadom na presnosť rekonštrukcie spŕšky z bodov/pixelov nájdených rôznymi verziami vyvíjanej metódy. Stanovenie optimálneho nastavenia parametrov rozpoznávania obrazcov pre dsiahnutie minimálnych požiadaviek pre splnenie cieľov JEM-EUSO experimentu.
6. Určenie závislosti presnosti rekonštruovania spŕšok pre optimálnu verziu kódu od intenzity UV pozadia.

Jazyk, v ktorom sa práca vypracuje: slovenský
Termín pre odovzdanie práce: 30.04.2015
Dátum zadania diplomovej práce: 31.10.2014

doc. Ing. Jaroslav Porubän, PhD.
vedúci garantujúceho pracoviska

TECHNICKÁ UNIVERZITA V KOŠICIACH FAKULTA ELEKTROTECHNIKY A INFORMATIKY -13-

prof. Ing. Liberios Vokorokos, PhD.
dekan fakulty

Čestné vyhlásenie

Vyhlasujem, že som diplomovú prácu vypracoval(a) samostatne s použitím uvedenej odbornej literatúry.

Košice 30. 4. 2015 .....

Vlastnoručný podpis

Poďakovanie

Vďaka RNDr. Pavlovi Bobikovi, PhD. za pekný rok strávený pri experimente JEM-EUSO, doc. Ing. Jánovi Genčimu, PhD. za príjemné vedenie diplomovej práce, a v prvom rade Ing. Michalovi Vrábelovi, bez ktorého úsilia by táto práca nevznikla.

Obsah

Zoznam obrázkov

Zoznam tabuliek

Zoznam symbolov a skratiek

ESA European Space Agency

ESAF EUSO Simulation And Analysis Framework

EUSO Extreme Universe Space Observatory

GTU Gate Time Unit

GZK Greisen Zatsepin Kuzmin

ISS International Space Station

JEM-EUSO Japanese Experiment Module-Extreme Universe Space Observatory

LTT The Linear Tracking Trigger

MAPMT Multi Anode Photo Multiplier Tube

PDM Photo-Detector Module

PMT Photo Multiplier Tube

PWISE The Peak And Window Searching Technique

RMS Root Mean Square

SNR Signal To Noise Ratio

UHECP Ultra-High Energy Cosmic Particles

UHECR Ultra-High Energy Cosmic Rays

UV Ultraviolet

Úvod

Od objavu kozmického žiarenia uplynulo už viac ako 100 rokov. Napriek tomu zostáva otázka pôvodu žiarenia nezodpovedanou. Jednou z misií, ktoré sa na túto otázku budú snažiť dať odpoveď je experiment JEM-EUSO, realizovaný prostredníctvom medzinárodnej spolupráce výskumníkov momentálne zo 16 krajín. JEM-EUSO predstavuje inovatívnu misiu v spôsobe sledovania kozmického žiarenia ultravysokých energií. Za týmto účelom bol špeciálne zostrojený UV teleskop, ktorý bude umiestnený na palube japonského experimentálneho modulu Medzinárodnej vesmírnej stanice ISS. Tento teleskop bude zhora sledovať atmosféru Zeme a zaznamenávať stopy UV žiarenia, vyprodukovaného pri javoch spojených so vznikom a vývojom atmosférických časticových spŕšok ultravysokých energií. Určovanie pôvodu týchto spŕšok sa vykonáva v rámci údajovej analýzy zaznamenaných udalostí. Jej dôležitou súčasťou je rozpoznanie vzoru spŕšky v údajoch udalosti, čo predstavuje vybranie len relevantných údajov udalosti s pôvodom v spŕške. Od toho, aké údaje budú vybrané a posunuté do ďalších fáz analýzy, v podstatnej miere závisia odchýlky výsledných hodnôt analýzy. V rámci kolaborácie JEM-EUSO už bolo vyvinutých zopár algoritmov rozpoznávania vzorov. Táto práca má predstavovať odpoveď na ich požiadavku vytvorenia algoritmu pre rozpoznávanie vzorov, postaveného na Houghovej transformácii - metóde detekcie geometrických útvarov v priestore.

1 Formulácia úlohy

Hlavným cieľom diplomovej práce je navrhnúť algoritmus pre rozpoznávanie spŕšok tvorených časticami ultravysokých energií. Tento algoritmus má byť založený na vybraných metódach Houghovej transformácie. Návrh tohto algoritmu prebieha za participácie v rámci experimentu JEM-EUSO, ktorý je zdrojom potrebných teoretických poznatkov v oblasti UHECR spŕšok, v oblasti algoritmov pre ich rozpoznávanie a poskytuje nevyhnutné softvérové vybavenie potrebné pri vývoji a testovaní navrhovaného algoritmu. Pred návrhom algoritmu je potrebné nadobudnúť dostatočné teoretické poznatky o Houghovej transformácii a UHECR spŕškach, čo by malo viesť k vhodnej voľbe metód Houghovej transformácie a vhodnému návrhu algoritmu. Navrhnutý algoritmus by mal byť implementovaný a integrovaný do rámca ESAF, vyvinutého v rámci JEM-EUSO, tak, aby bolo možné algoritmus aplikovať na spŕšky generované simulačnou časťou rámca ESAF. Cieľom je nájsť optimálne nastavenie parametrov algoritmu a určenie faktorov ovplyvňujúcich rozpoznanie signálu spŕšky vzhľadom na presnosť rekonštrukcie spŕšky. Po optimalizovaní parametrov algoritmu sa určí závislosť presnosti rekonštruovania spŕšok od intenzity UV pozadia.

2 Experiment JEM-EUSO

Táto práca, zaoberajúca sa návrhom a vývojom algoritmov pre rozpoznávanie vzorov spŕšok tvorených časticami ultravysokých energií, bola tvorená v rámci participácie a softvérovej podpory vyvinutej v rámci medzinárodného experimentu JEM-EUSO. Hlavným cieľom tohto experimentu je výskum pôvodu a povahy kozmického žiarenia ultravysokých energií. Predpokladom tohto výskumu je dopravenie a umiestnenie navrhnutej techniky do vesmíru, odkiaľ by sledovala stopy UV žiarenia, vyprodukovaného pri javoch spojených so vznikom a vývojom atmosférických časticových spŕšok ultravysokých energií. Detailnejší popis tohto experimentu poskytujú nasledujúce podkapitoly.

2.1 Popis experimentu

Kozmické Observatórium Extrémneho Vesmíru na palube Japonského Experimentálneho Modulu (JEM-EUSO) Medzinárodnej vesmírnej stanice ISS je vesmírnou misiou zameranou na štúdium kozmického žiarenia ultravysokých energií (UHECR), a to kozmických častíc s energiou $E \geq 5 \cdot 10^{19}$ eV, nad takzvaným GZK limitom, pričom svoju vedeckú pozornosť obracia najmä k udalostiam o energii $E \sim 10^{20}$ eV. JEM-EUSO je navrhnuté k monitorovaniu Zemskej atmosféry z vesmíru a detegovaniu UV (290-430 nm) stôp atmosférických časticových spŕšok, šíriacich sa atmosférou (Santangelo, Picozza a Ebisuzaki, 2013). Pozorovateľné UV stopy spŕšok vznikajú v dôsledku dvoch javov:

Fluorescencia dusíka Pri prechode atmosférou spôsobuje spŕška sekundárnych častíc pozdĺž dráhy letu excitáciu atómov dusíka. Pomalý návrat týchto atómov z exitovaného stavu do základného stavu vedie k vzniku žiarenia v ultrafialovej a viditeľnej oblasti elektromagnetického spektra (Gajdoš, 2014).

Čerenkovov jav K Čerenkovovmu javu dochádza, pokiaľ sa nabitá častica šíri

médiom rýchlejšie než je rýchlosť svetla v tom prostredí c/n. Pohybujúci sa náboj spôsobuje oscilácie atómov, ktoré produkujú elektromagnetické vlnenie. Akonáhle rýchlosť nabitej častice presiahne rýchlosť šírenia signálu c/n, vlny začnú spolu interferovať a vytvoria rázovú vlnu, čím vzniká žiarenie označované ako Čerenkovovo žiarenie. V spŕške sú za produkciu tohto žiarenia zodpovedné mióny (Blaschke, 2009).

2.2 Nástroje JEM-EUSO

Konceptuálny pohľad na celý systém JEM-EUSO poskytuje Obrázok 2 – 1 (Gajdoš, 2014). Tento systém pozostáva z:

- Letiaceho segmentu,
- Podporných zariadení na Zemi,
- Pozemného segmentu,
- a Globálneho svetelného systému.

Letiaci segment pozostáva hlavne zo Segmentu vedeckých nástrojov, ktorý v podstate pozostáva z:

- Teleskopu,
- atmosférického monitorovacieho systému,
- a kalibračného systému.

JEM-EUSO teleskop, ktorého naposledy uvádzané parametre je možné nájsť v (Santangelo, Picozza a Ebisuzaki, 2013), principiálne pozostáva zo štyroch častí:

Optika Pozostáva z troch obojstranne zakrivených kruhových Fresnelových šošoviek s externým priemerom 2.65 m. Fresnelova šošovka je poloplochá šošovka, ktorá má kruhové drážky, ktoré eliminujú veľkú hmotnosť štandardných kon-

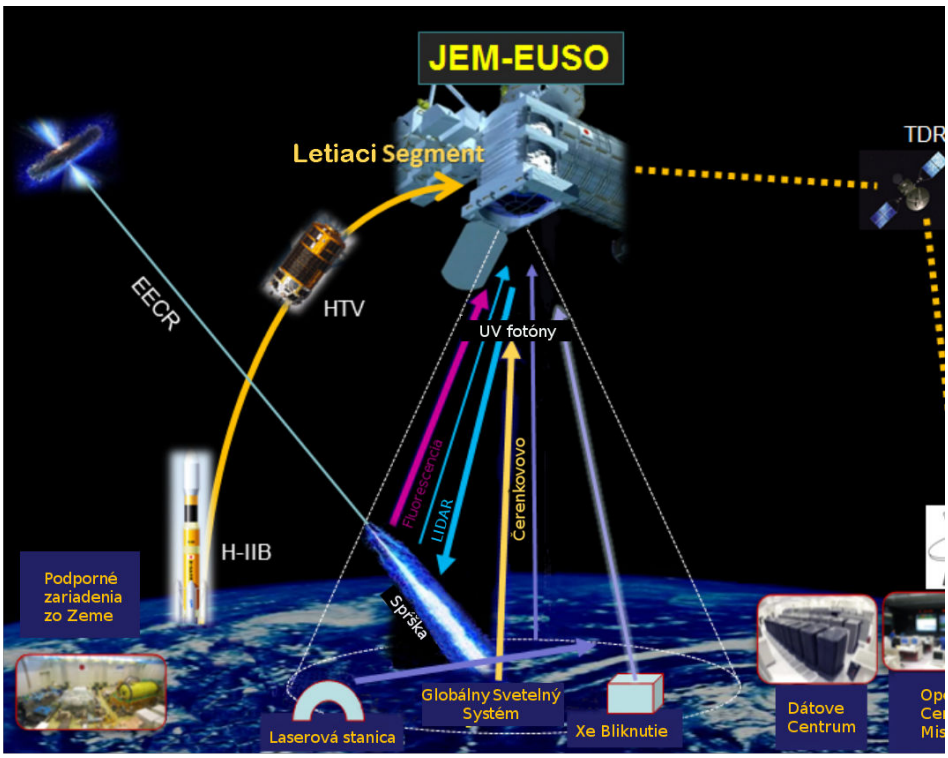
vektívnych a konkávnych šošoviek. Tenkosť a ľahkosť Fresnelovej šošovky je nevyhnutnou podmienkou jej využitia vo vesmíre, pričom ponúka rovnaké optické funkcie ako hrubé a ťažké šošovky (Staroň, 2013). Táto kombinácia troch Fresnelových šošoviek umožňuje plné využitie veľkého zorného poľa s uhlom 60 stupňov s rozlíšením 0.075 stupňa, čo pre jeden pixel približne odpovedá 550 m na Zemi (Kajino a iní, 2013).

Detektor ohniskovej plochy Detektor ohniskovej plochy (Obrázok 2 – 2) je navrhnutý tak, aby zachytil stopu produkovanú spŕškou sekundárneho kozmického žiarenia v rozsahu vlnových dĺžok 290-430 nm. Je schopný určiť pozíciu prichádzajúcich fotónov a sledovať vývoj spŕšky v čase a priestore. Ohnisková plocha je zakrivená s priemerom 2.5 m s plochou približne 4.5 m^2 a pokrytá multianódovými fotonásobičmi (MAPMT)(Obrázok 2 – 3). Na celej ohniskovej ploche sa ich nachádza okolo 5000 (Kajino a iní, 2013).

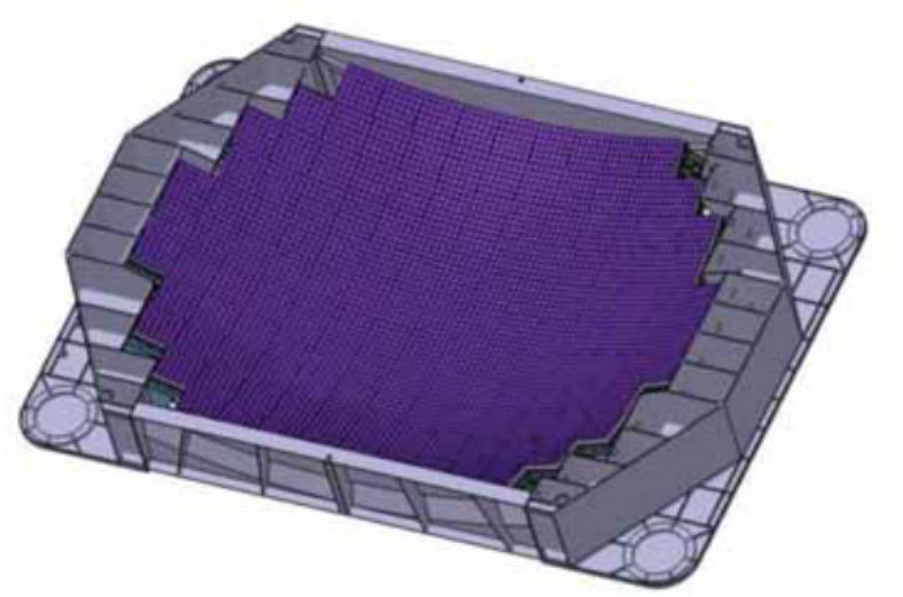
Elektronika ohniskovej plochy Elektronika nahráva počty signálov pre časové intervaly 2.5 μs (Kajino a iní, 2013).

Mechanická štruktúra ohniskovej plochy Ohnisková plocha je časťou sféry s rádiom 2505 mm (Obrázok 2 – 4). Hlavnou štruktúrou ohniskovej plochy je vrchný rám. K nemu sú pripevnené PDM štruktúry. Táto spojená štruktúra je zobrazená na Obrázku 2 – 5 (Ricci, Franceschi a Napolitano, 2011).

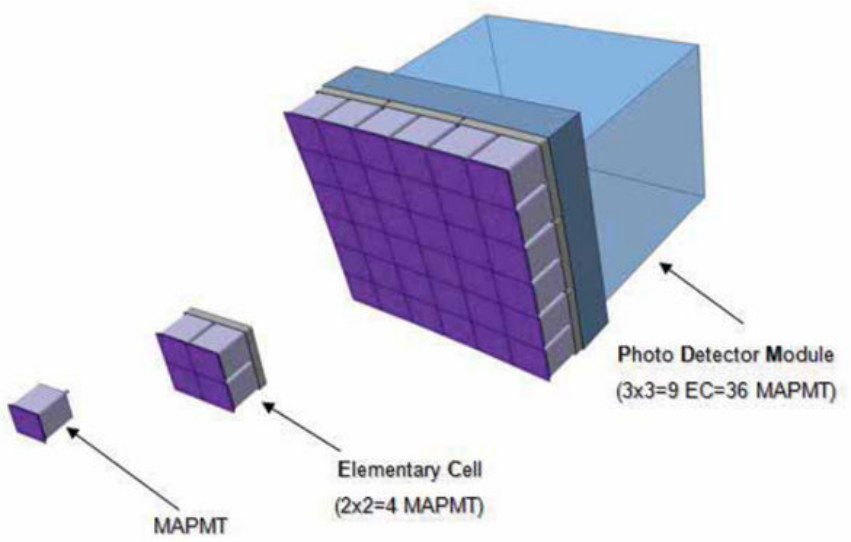
Medzi podporné zariadenia patria mechanické, optické, elektrické a kalibračné zariadenia, podporujúce Letiaci segment vo fáze výroby. Pozemný segment pozostáva z odpaľovacej základne a operačného strediska misie. Globálny svetelný systém je určený ku kalibrácii vybavenia pomocou xenónových svetiel a UV laserov (Kajino a iní, 2013).



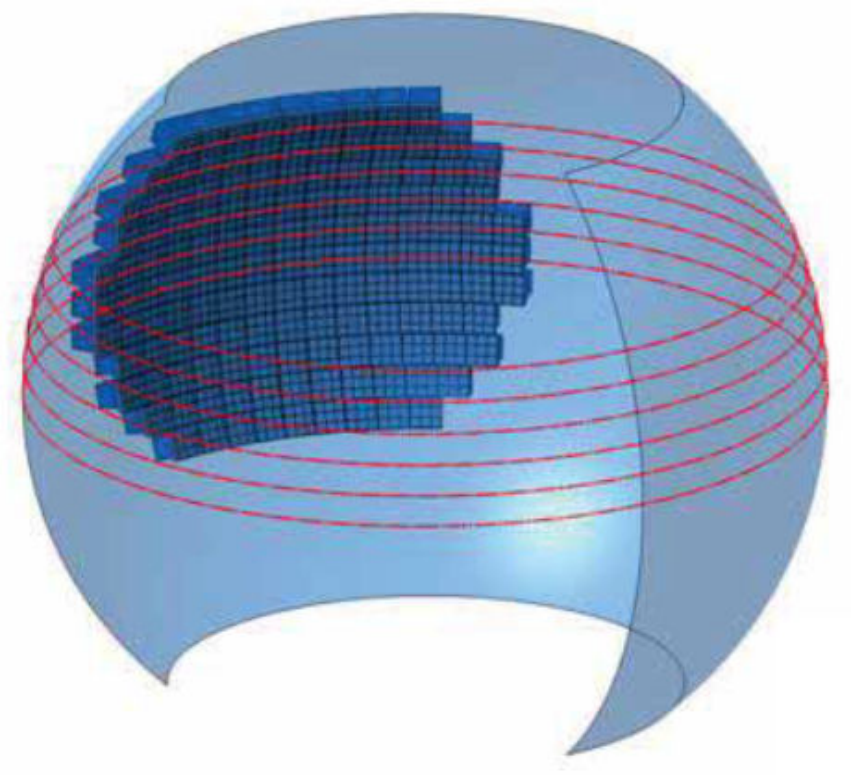
Obr. 2 – 1 Konceptuálny pohľad na systém JEM-EUSO (Gajdoš, 2014).



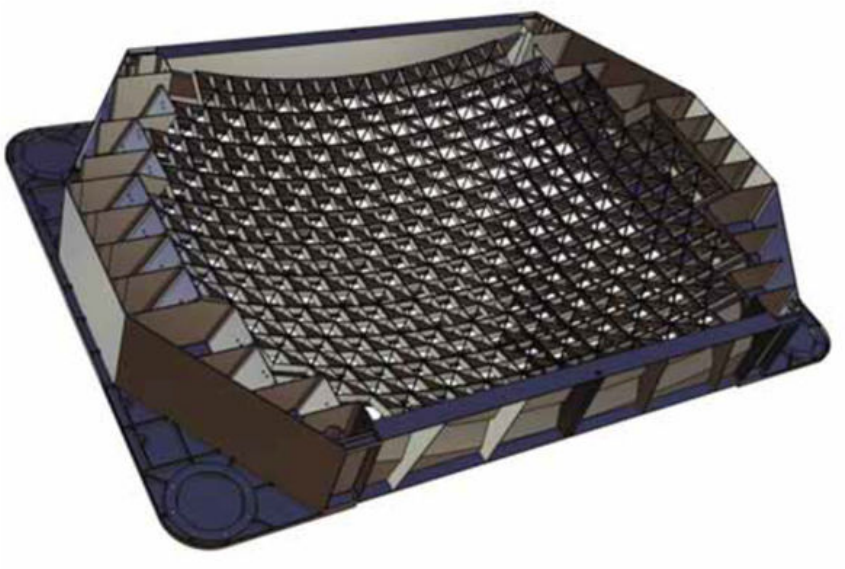
Obr. 2 – 2 Detektor ohniskovej plochy (Kawasaki a iní, 2011), (Macarone a iní, 2010).



Obr. 2 – 3 Fotonásobiče (Macarone a iní, 2010).



Obr. 2 – 4 PDM umiestnenie na ohniskovej ploche (Ricci, Franceschi a Napolitano, 2011).



Obr. 2 – 5 Vrchný rám a PDM rámy (Ricci, Franceschi a Napolitano, 2011).

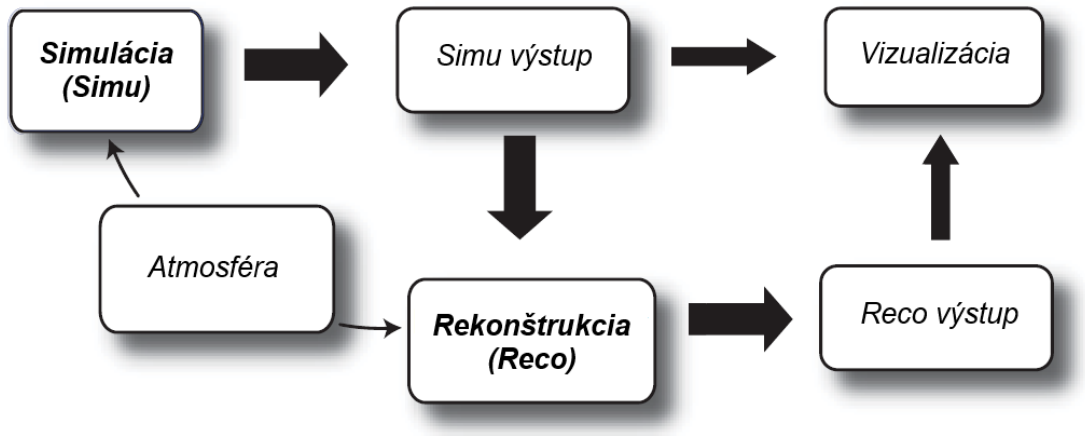
3 EUSO Simulation and Analysis Framework (ESAF)

ESAF predstavuje softvérový rámec, navrhnutý pre simulovanie fungovania, výkonu a fyziky javov registrovaných vesmírnymi pozorovacími stanicami pre pozorovanie kozmického žiarenia ultravysokých energií, vyvinutý v rámci projektu EUSO počas štúdie ESA s pomenovaním Fáza A (Mernik, 2013). Tento rámec kompletne softvérovo pokrýva simuláciu a rekonštrukciu od interakcie primárnej častice v atmosfére Zeme až po koncovú rekonštrukciu udalosti. Kód rámca zahŕňa (Berat a iní, 2010):

- Simuláciu rozsiahlych atmosférických spŕšok,
- kompletný popis atmosféry, vrátane aerosolov a mrakov,
- produkciu fluorescenčného a Čerenkovovho žiarenia,
- kompletnú simuláciu šírenia fotónu od vzniku až po teleskop, vrátane interakcií s atmosférou a povrchom Zeme,
- simuláciu optiky teleskopu,
- geometriu teleskopu,
- simuláciu detektora fotónov,
- simuláciu spúšťačov a elektroniky,
- simuláciu pozadia,
- rozpoznávanie vzorov a identifikáciu signálu spŕšky nad hranicou pozadia,
- rekonštrukciu smeru a energie.

Celý softvér bol vyvinutý držiac sa paradigmy objektovo-orientovaného programovania, čo umožňuje flexibilitu, modulárnosť a udržiavateľnosť kódu. Väčšina kódu

je napísaná v jazyku C++ s niekoľkými externými knižnicami, ktoré sú v jazyku FORTRAN. Kód je rozdelený do dvoch nezávislých častí - simulácie (Simu) a rekonštrukcie (Reco) (Berat a iní, 2010). Na Obrázku 3 – 1 je možné vidieť vrchnú úroveň štruktúry ESAF-u.

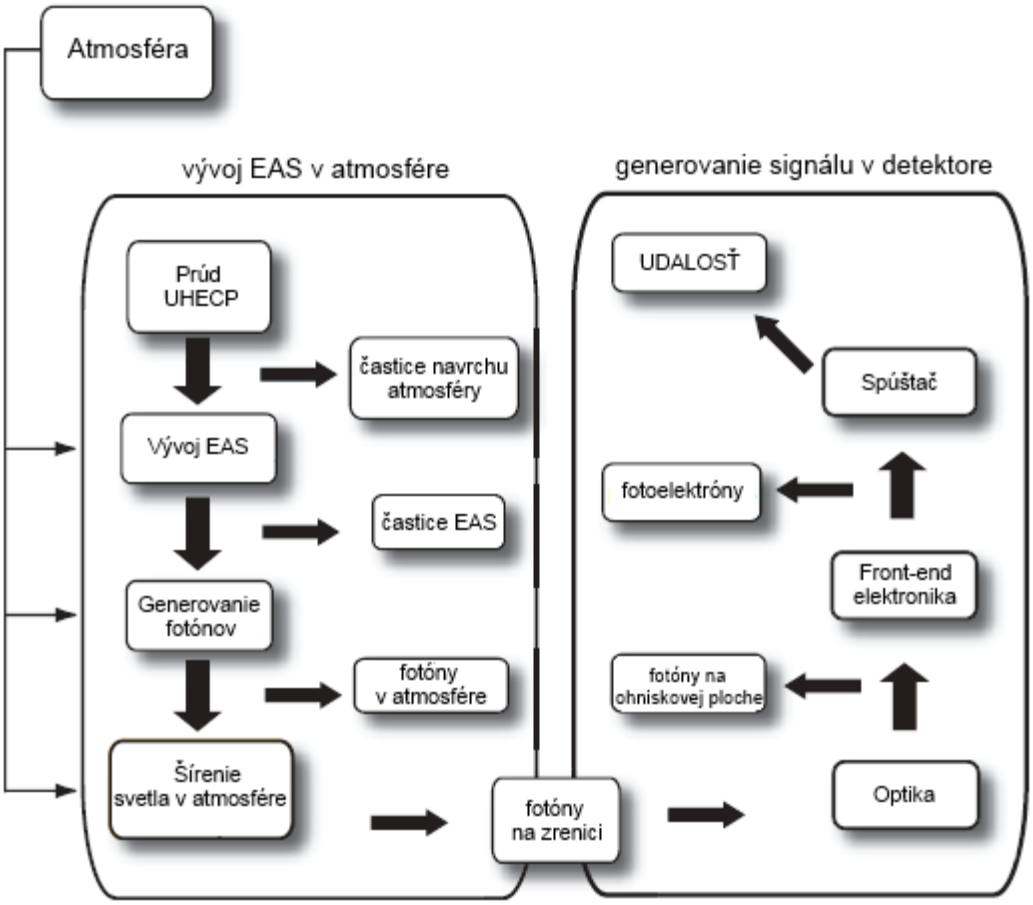


Obr. 3 – 1 Vrchná úroveň štruktúry ESAF-u (Berat a iní, 2010).

Simulácia pozostáva zo 6 hlavných častí (Obrázok 3 – 2) (Berat a iní, 2010):

- Simulácie prúdu UHECP,
- simulácie vzniku fluorescencie a Čerenkovových fotónov v atmosfére,
- simulácie šírenia svetla, od bodu vzniku až po teleskop, a interakcií v atmosfére,
- simulácie optiky teleskopu,
- simulácie front-end elektroniky,
- simulácie spúšťača.

Výstupom simulačného modulu je ROOT súbor obsahujúci údaje simulovanej udalosti. Tieto údaje sú tak dostupné pre následnú údajovú analýzu, ktorá je k dispozícii prostredníctvom rekonštrukčného rámca. Rekonštrukčný rámec používa pre



Obr. 3 – 2 Štruktúra simulácie (Berat a iní, 2010).

načítanie súborov s údajmi udalosti vstupný modul a ďalej poskytuje reťaz modulov, zodpovedných za rekonštrukčnú analýzu nad údajmi udalosti. Každý z týchto modulov má svoju špecifickú funkciu analýzy. Medzi najzákladnejšiu analýzu patria:

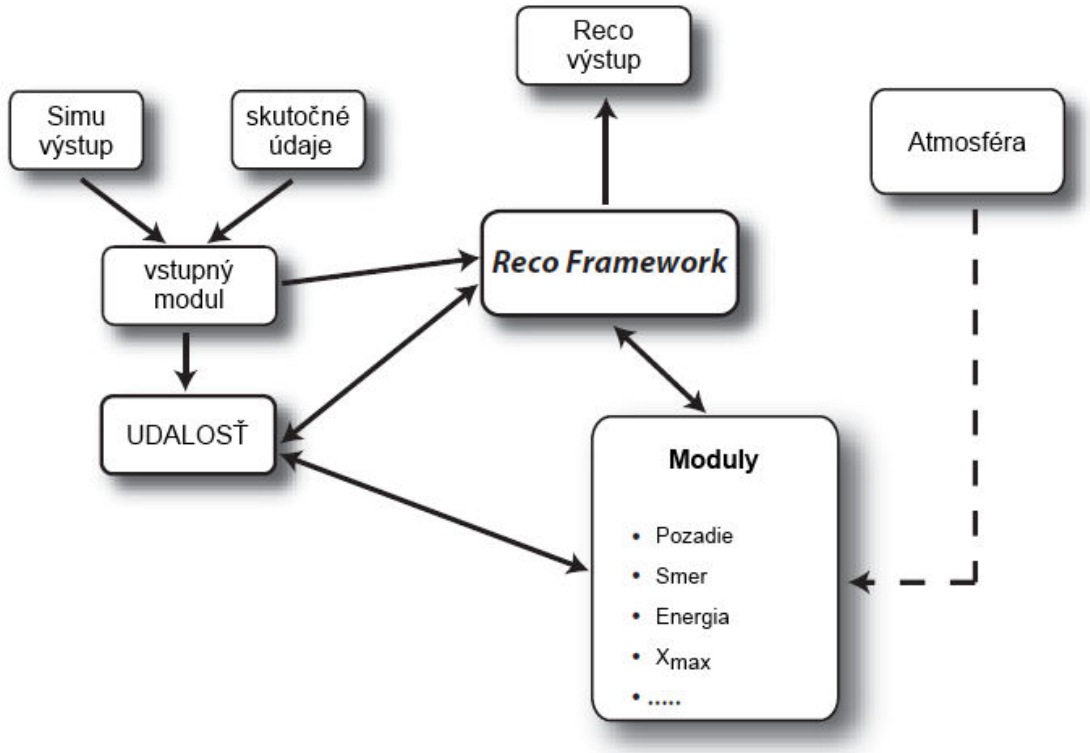
Rozpoznávanie vzorov Cieľom je z údajov udalosti oddeliť časovo previazané údaje spŕšky od údajov s pôvodom v pozadí. Tento nájdený vzor spŕšky sa stáva zdrojom údajov pre ďalšiu analýzu a od jeho „kvality“ závisí podstatná miera presnosti rekonštrukcie.

3D smerová rekonštrukcia Cieľom je nájsť smer primárnej častice v priestore. Pri jeho zisťovaní sa využívajú smer stopy spŕšky na ohniskovej ploche a časové

informácie príchodu fotónov (Berat a iní, 2010).

Rekonštrukcia energie a X_{max} Cieľom je nájsť miesto, kde má spŕška najviac sekundárnych častíc (X_{max}), a rekonštrukcia energie primárnej častice. Od týchto charakteristík, ktoré sú navzájom prepojené, sa odvíja identifikácia primárnej častice (Berat a iní, 2010).

Modulárnosť štruktúry umožňuje jednoduché pridanie alebo vynechanie jednotlivých modulov z rekonštrukcie, čo dáva možnosť vyskúšať rôzne kombinácie modulov alebo vytvárať porovnania výsledkov rôznych algoritmov aplikovaných na rovnaké zdrojové údaje.



Obr. 3 – 3 Štruktúra rekonštrukcie (Berat a iní, 2010).

Vzhľadom na to, že cieľom tejto práce bol vývoj alternatívneho algoritmu rozpoznávania vzorov, založeného na metóde Houghovej transformácie, ktorý by sa eventuálne stal súčasťou implementácie modulu rozpoznávania vzorov rámca ESAF, je vhodné v

nasledujúcich podkapitolách v krátkosti predstaviť štandardne používané algoritmy tohto modulu.

3.1 The peak and window searching technique (PWISE)

Názov tohto algoritmu je možné preložiť ako „Technika vyhľadávania vrcholov a okien“. „Vrcholy“ v názve sa viažu k prvému kroku algoritmu, kedy sa pre ďalšiu analýzu vyberú iba pixely, ktorých maximálne hodnoty („vrcholy“) počtu signálov sa nachádzajú nad definovanou prahovou hodnotou (Obrázok 3 – 4). „Okná“ v názve predstavujú časové rámce okolo „vrcholov“ vybraných pixelov, využívané pri časovej analýze. Celkovo prebieha tento algoritmus v 3. krokoch (Biktemerova a iní, 2013):

1. Vyberú sa iba pixely, ktorých maximálne hodnoty počtu signálov sa nachádzajú nad definovanou prahovou hodnotou.

2. Okolo údajov, s maximálnymi hodnotami počtu signálov na jednotlivých vybraných pixeloch, sa hľadá časové okno s najlepším pomerom signálu k šumu, definovaným ako funkcia SNR

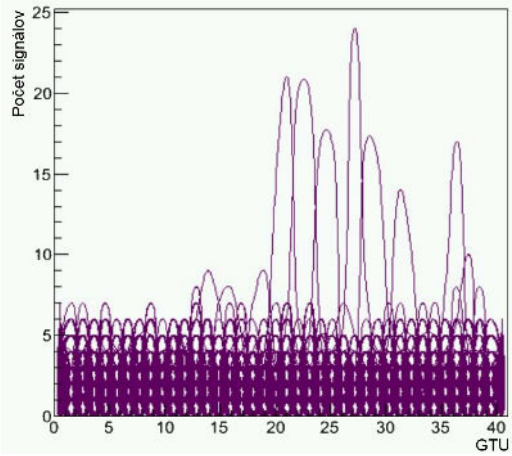
$$SNR = \frac{\sum_{\Delta\tau} pc(t)}{\Delta\tau RMS} \tag{3.1}$$

   , kde $pc(t)$ je počet signálov ako funkcia času, $\Delta\tau$ je vopred zadefinovaná množina časových okien okolo údaja s maximálnou hodnotou počtu signálov, a RMS je stredná kvadratická odchýlka počtu signálov na pixeli.

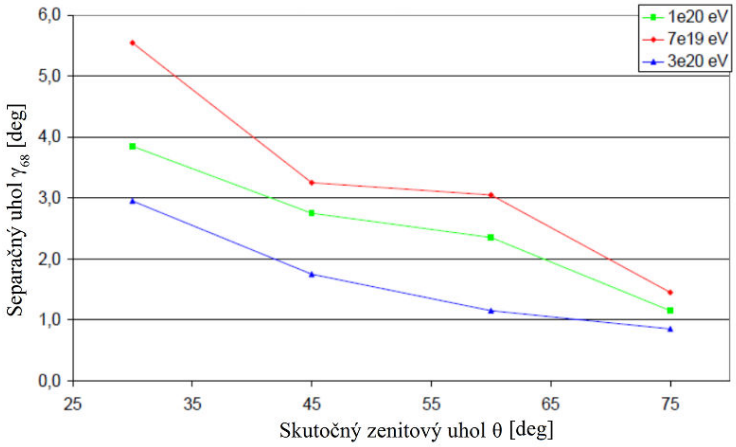
3. Ak je maximálna hodnota funkcie SNR nad definovaným prahom, vyberú sa údaje v rámci časového okna, ktoré maximalizovalo hodnotu funkcie SNR, a posunú sa do nasledujúceho rekonštrukčného modulu.

Ako jedna z výhod tohto algoritmu je uvádzaná schopnosť odfiltrovať mnohonásobne rozptýlené fotóny, ktoré stoja za „výbežkovitým“ vzhľadom obrazovej stopy spŕšky (Biktemerova a iní, 2013). Toto filtrovanie prebieha v prvom kroku algoritmu výberom pixelov a je spojené s dosť podstatným nedostatkom. Definovaný prah pre

výber pixelov má predstavovať hodnotu, ktorá dostatočne oddelí šum od signálu spŕšky. Týmto spôsobom sa ale vyberie iba „najjasnejšia“ centrálna línia pixelov spŕšky, ktorá nemusí byť dostatočným zdrojom údajov pre uhlovú rekonštrukciu. To do určitej miery naznačujú aj štatistické údaje uhlovej rekonštrukcie spŕšok pre rôzne zenitové uhly (Obrázok 3 – 5). Odchýlka je tým väčšia, čím kratšie sú zachytené spŕšky, čím kratšie sú zachytené spŕšky, tým menší počet pixelov a údajov spŕšky, a tým aj nedokonalejšia línia spŕšky je PWISE-om posúvaná do rekonštrukčného modulu.



Obr. 3 – 4 Graf GTU - Počet signálov (Mernik a iní, 2013).



Obr. 3 – 5 Výsledky uhlovej analýzy pre algoritmus PWISE (Guzman a iní, 2013).

3.2 The Linear Tracking Trigger (LTT) Pre-Clustering

LTT Pre-Clustering predstavuje techniku hľadania priamych ciest hustej akumulácie signálu. Ide o vylepšenú logiku spúšťača druhého stupňa (Mernik, 2013). Najprv vyberá pixely na ohniskovej ploche, ktoré obsahujú najvyššie hodnoty počtu signálov. Vzhľadom na tieto pixely sa potom vo vopred definovaných smeroch hľadá cesta s najhustejšou akumuláciou signálu. Ako uvádza Fenu (Fenu, 2013), definuje sa objekt o rozmere 3x3 pixelov a umiestni sa v čase podľa maximálnej sumy počtu signálov objektu v piatich po sebe idúcich GTU. Následne sa tento objekt posúva v predefinovaných smeroch do dĺžky odpovedajúcej 200 GTU. Cesta, ktorá maximalizuje sumu počtu signálov, je vybraná, a následne sú okolo nej do definovanej vzdialenosti vybrané údaje. Uvádza sa, že použitie tejto metódy pred použitím modulov pre rozpoznávanie, môže viesť k zvýšeniu ich rýchlosti (Biktemerova a iní, 2013) a k zvýšeniu presnosti uhlovej rekonštrukcie (Mernik, 2013).

3.3 Track finding method

Tento algoritmus pracuje v každom GTU iba s pixelmi, ktorých údaj o počte signálov v tomto čase, je nad určitou prahovou hodnotou počtu signálov. Tieto pixely predstavujú tzv. snímok ohniskovej plochy v danom čase. Tieto snímky potom slúžia pre nájdenie optimálnej cesty v čase pohybujúceho sa bodu pozdĺž priamky po ohniskovej ploche. Ako sa detailnejšie uvádza (Biktemerova, Gonchar a Sharakin, 2013), algoritmus spracováva snímky sekvenčne, pričom sa snaží spojiť všetky možné páry pixelov aktuálnej a nasledujúcej snímky. Tieto páry pixelov pritom musia spĺňať dané podmienky:

Vzdialenosť Maximálna povolená vzdialenosť medzi dvoma spojenými pixelmi.

Trvanie Maximálny povolený časový rozdiel medzi údajmi dvoch spojených pixelov.

Odchýlka od línie cesty Maximálna povolená vzdialenosť pixelov od línie cesty. Podmienka začína platiť po tom, čo cesta obsahuje aspoň dva pixely.

Ak sú tieto podmienky splnené, pixel je pridaný k segmentu cesty. Vzhľadom na podmienku trvania, ktorej hodnota je uvádzaná 5 GTU (Biktemerova, Gonchar a Sharakin, 2013), volí algoritmus kandidátov pre pridanie k segmentom cesty nielen na základe dvoch po sebe idúcich snímkov, ale aj na základe segmentov vyskytujúcich sa v snímkoch do 5 GTU spätne. Na konci algoritmu je zo všetkých vytvorených ciest vybraná najdlhšia s maximálnou sumou počtu signálov, neobsahujúca časové skoky.

4 Houghova transformácia

Houghova transformácia predstavuje metódu detekcie geometrických útvarov v priestore. Pôvodnú myšlienku transformácie popísal v roku 1962 vo svojom patente P.V.C. Hough (Hough, 1962), keď pri detekcii priamok použil pre transformačný priestor parametre smernice a posunutia. Avšak tento spôsob parametrizácie viedol k neohraničenému priestoru transformácie, keďže hodnoty smernice mohli ísť k nekonečnu. K modernej podobe Houghovej transformácie viedla práca Dudu a Harta (Duda a Hart, 1972), v ktorej pri detekcii priamok poukázali na vhodnosť použitia parametrov uhla otočenia a kolmej vzdialenosti priamky od počiatku súradnicovej sústavy, a taktiež ukázali možnosti metódy pri fitovaní všeobecnejších kriviek. Všeobecná Houghova transformácia pre detekciu ľubovoľných útvarov bola navrhnutá Dana H. Ballardom (Ballard, 1981). Zovšeobecnenie Houghovej transformácie pre detekciu analytických útvarov v n-dimenzionálnych priestoroch popísali Fernandes a Oliveira (Fernandes a Oliveira, 2012). Ako vhodné metódy pri návrhu algoritmov rozpoznávania spŕšok boli zvolené nasledujúce metódy Houghovej transformácie pre detekciu priamok v 2D a 3D priestore a pre detekciu rovín.

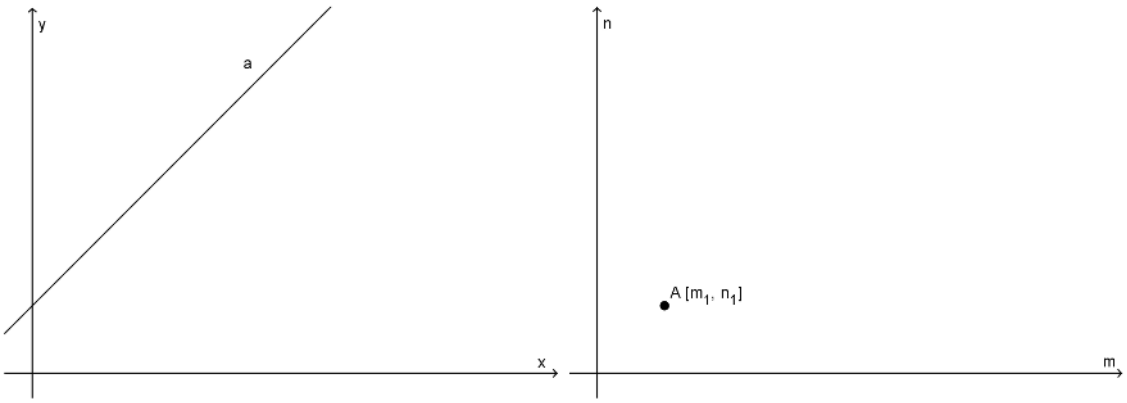
4.1 Houghova transformácia pre detekciu priamok v 2D priestore

Houghova technika (Hough, 1962)

V obrazovom priestore môžeme popísať každú priamku parametrickou rovnicou

$$y_i = mx_i + n \tag{4.1}$$

, kde x_i a y_i sú súradnice bodov ležiacich na priamke v priestore XY, m je smernica a n je posunutie priamky po y-ovej osi. To umožňuje reprezentáciu priamky prostredníctvom jedinečného bodu $(m,\ n)$ v parametrickom priestore smernice a posunutia (Obrázok 4 – 1).



Obr. 4 – 1 Reprezentácia priamky v parametrickom priestore MN.

Ak by sme predchádzajúcu rovnicu upravili na tvar

$$n_i = -xm_i + y \tag{4.2}$$

, kde m_i a n_i sú súradnice bodov ležiacich na priamke v priestore MN, x a y sú súradnice bodu v priestore XY, v priestore MN predstavujú smernicu a posunutie priamky po n-ovej osi, potom by priamka, popísaná touto rovnicou, bola reprezentáciou všetkých priamok prechádzajúcich bodom $(x,\ y)$ v priestore XY (Obrázok 4 – 2). Z toho vyplýva, že ak dva ľubovoľné body ležia na tej istej priamke, jej parametre je možné určiť ako priesečník priamok v parametrickom priestore MN, kde tieto priamky reprezentujú všetky priamky prechádzajúce danými bodmi v priestore XY (Obrázok 4 – 3). Tento vzťah medzi obrazovým a parametrickým priestorom bodu sa ponúka pre nasledujúci algoritmus (Ballard, 1981):

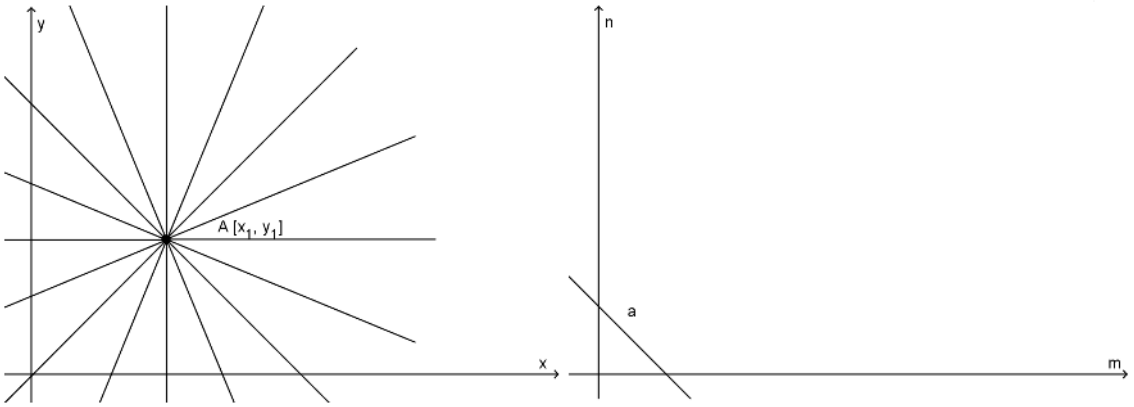
1. Diskretizuj parametrický priestor medzi vhodnými maximálnymi a minimálnymi hodnotami parametrov m a n.

2. Vytvor akumulačnú maticu $A(m,\ n)$ a inicializuj prvky matice hodnotou 0.

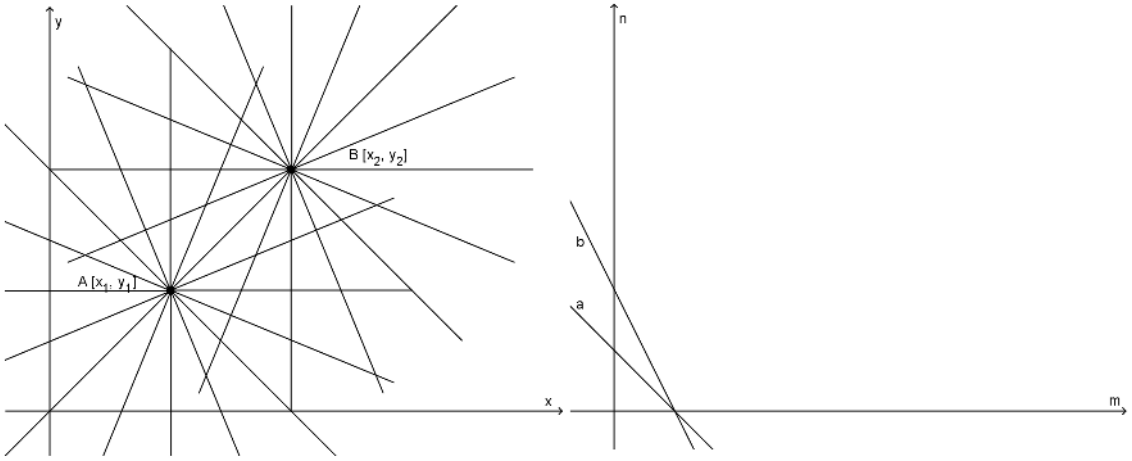
3. Pre každý bod $(x,\ y)$ vstupného obrazu, s hodnotou nad definovaným prahom, inkrementuj všetky prvky akumulačnej matice vzhľadom na vypočítané parametre:

$$A(m_i,\ n_i) = A(m_i,\ n_i) + 1,\ \{(m_i,\ n_i) \mid n_i = -xm_i + y\} \tag{4.3}$$

4. Lokálne maximá v akumulačnej matici korešpondujú s kolineárnymi bodmi vstupného obrazu. Hodnoty akumulačnej matice poskytujú mieru počtu bodov ležiacich na priamke.



Obr. 4 – 2 Reprezentácia všetkých priamok, prechádzajúcich daným bodom, v parametrickom priestore MN.



Obr. 4 – 3 Grafické riešenie hľadania parametrov spoločnej priamky dvoch bodov v parametrickom priestore MN.

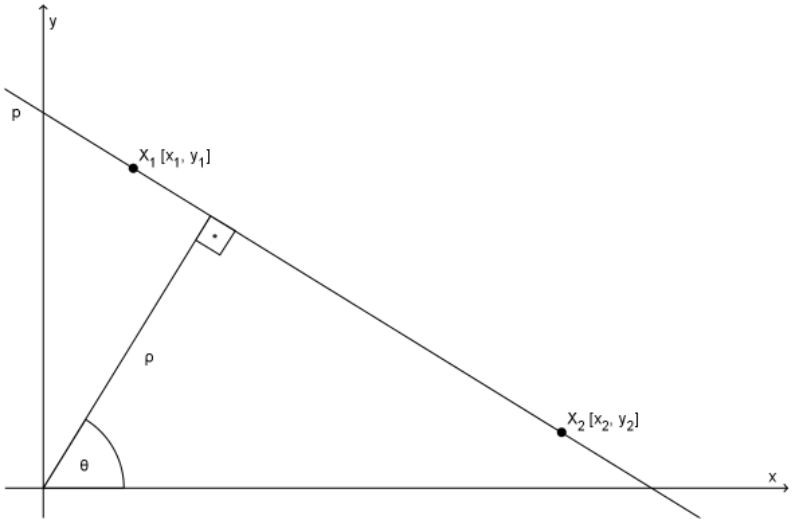
Avšak takáto reprezentácia priamok má svoje nevýhody – priveľké hodnoty m pre približne vertikálne priamky a žiadna reprezentácia vertikálnych priamok. Napriek tomu, že tieto nevýhody je možné prekonať (Houghov návrh prejsť obrázok dvakrát v pravom uhle (Hough, 1962), vyjadrený prehodením súradníc bodov $(x_i,\ y_i) \rightarrow (y_i,\ x_i)$ (Rosenfeld, 1962)), existuje aj vhodnejšie riešenie.

Moderná podoba Houghovej transformácie (Duda a Hart, 1972)

Riešením predchádzajúcich problémov Hougovej techniky je popis priamky prostredníctvom polárnych súradníc. Rovnica priamky má tak vyjadrenie

$$y_i = -\frac{\cos\theta}{\sin\theta}x_i + \frac{\rho}{\sin\theta} \tag{4.4}$$

, kde x_i a y_i sú súradnice bodov ležiacich na priamke, ρ je kolmá vzdialenosť priamky od počiatku súradnicovej sústavy a θ je polárny uhol (Obrázok 4 – 4).



Obr. 4 – 4 Grafické zobrazenie popisu priamky prostredníctvom polárnych súradníc.

Pre popis priamky tak môžeme využiť parametrický priestor určený kolmou vzdialenosťou priamky od počiatku súradnicovej sústavy a polárnym uhlom. Týmto spôsobom je možné popísať každú priamku jedinečným bodom $(\rho,\ \theta)$ parametrického priestoru (Obrázok 4 – 5), napr. práve vtedy, keď $\theta \in [0,\ \pi) \wedge \rho \in R$, alebo $\theta \in [0,\ 2\pi) \wedge \rho \in [0,\ R^+)$. Ak by sme predchádzajúcu rovnicu upravili na tvar

$$\rho_i = x\cos\theta_i + y\sin\theta_i \tag{4.5}$$

, kde ρ_i a θ_i sú súradnice bodov ležiacich na sinusoidálnej krivke v priestore ΘP, x a y sú súradnice bodu v priestore XY, potom by sinusoidálna krivka, popísaná touto rovnicou, predstavovala reprezentáciu všetkých priamok prechádzajúcich bodom $(x,\ y)$ v priestore XY (Obrázok 4 – 6). Z toho vyplýva, že ak dva ľubovoľné

body ležia na tej istej priamke, jej parametre je možné určiť ako priesečník sinusoidálnych kriviek v parametrickom priestore, kde tieto krivky reprezentujú všetky priamky prechádzajúce danými bodmi v priestore XY (Obrázok 4 – 7). Algoritmus, popísaný pri Houghovej technike, platí podobne aj po zmene parametrickej reprezentácie priamok. Na základe tohto algoritmu bol vytvorený nasledujúci pseudokód Houghovej transformácie pre detekciu priamok v 2D priestore.

Pseudokód metódy Houghovej transformácie pre detekciu priamok v 2D priestore

$Points$ vstupné údaje obrazového priestoru

$Lines$ množina dvojíc parametrov detegovaných priamok

$Accumulator$ akumulačné pole

$threshold_{point}$ prahová hodnota bodov

$threshold_{line}$ prahová hodnota počtu bodov priamky

$[\theta_{min},\ \theta_{max}]$ rozsah polárneho uhla

θ_{array} jednorozmerné pole hodnôt polárneho uhla

θ_{count} veľkosť jednorozmerného poľa hodnôt polárneho uhla

θ_{step} krok diskretizácie priestoru polárneho uhla

ρ_{max} absolútna hodnota maximálnej možnej kolmej vzdialenosti priamky

ρ_{array} jednorozmerné pole hodnôt kolmej vzdialenosti priamky

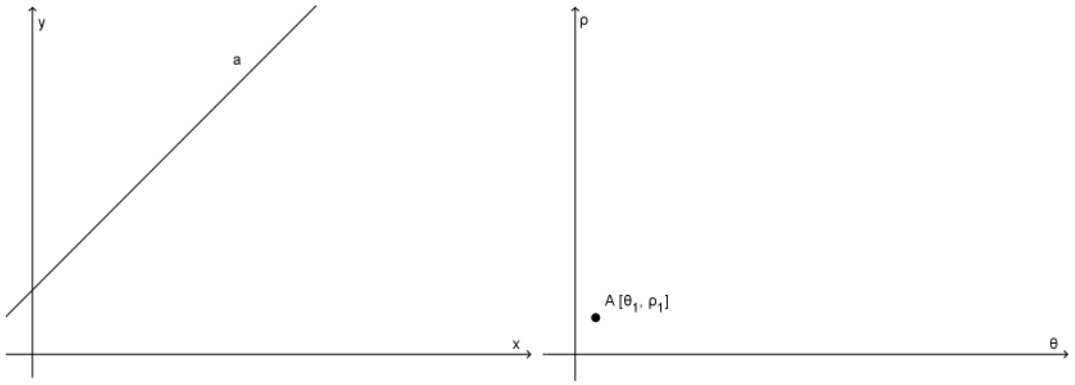
ρ_{count} veľkosť jednorozmerného poľa hodnôt kolmej vzdialenosti priamky

ρ_{step} krok diskretizácie priestoru kolmej vzdialenosti priamky

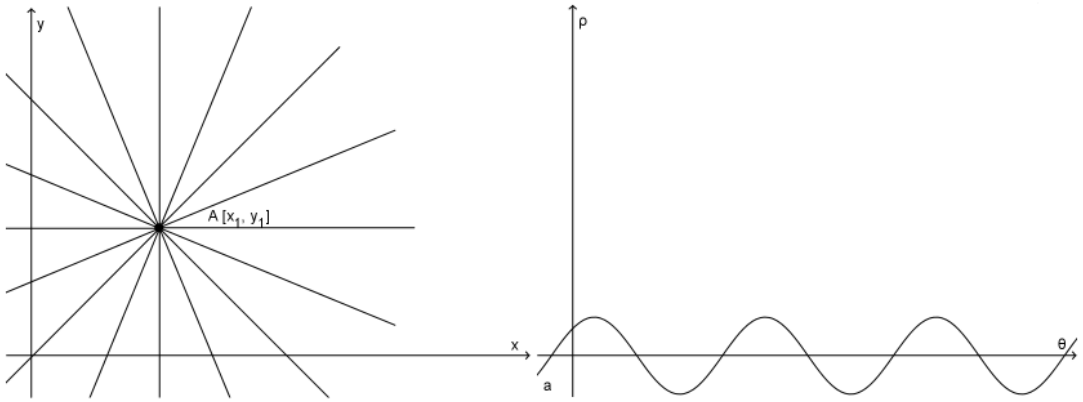
ρ_{value} vypočítaná hodnota kolmej vzdialenosti priamky

1. $\theta_{min} = -\frac{\pi}{2}$

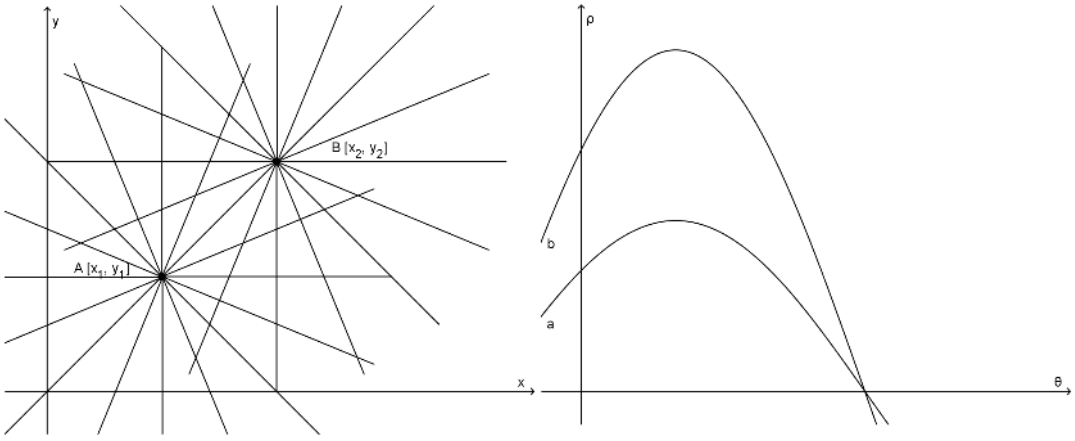
2. $\theta_{max} = \frac{\pi}{2}$
3. $X_{min} = min(Points_x)$
4. $X_{max} = max(Points_x)$
5. $Y_{min} = min(Points_y)$
6. $Y_{max} = max(Points_y)$
7. $\rho_{max} = \sqrt{X_{max}^2 + Y_{max}^2} > \sqrt{X_{min}^2 + Y_{min}^2} ? \sqrt{X_{max}^2 + Y_{max}^2} : \sqrt{X_{min}^2 + Y_{min}^2}$
8. $\rho_{count} = ceil(\frac{2\rho_{max}}{\rho_{step}}) + 1$
9. $\rho_{array} = [-\rho_{max}, -\rho_{max} + \rho_{step}, ..., -\rho_{max} + \rho_{step}(\rho_{count} - 1)]_{\rho_{count}}$
10. $\theta_{count} = floor(\frac{\theta_{max} - \theta_{min}}{\theta_{step}}) + 1$
11. $\theta_{array} = [\theta_{min}, \theta_{min} + \theta_{step}, ..., \theta_{min} + \theta_{step}(\theta_{count} - 1)]_{\theta_{count}}$
12. $Accumulator[]_{\rho_{count}}[]_{\theta_{count}} = 0$
13. $for\ point : Points$
14. $if\ point_{value} > threshold_{point}$
15. $for\ \theta_{index} = 0 : (\theta_{count} - 1)$
16. $\rho_{value} = point_x \cos(\theta_{array}[\theta_{index}]) + point_y \sin(\theta_{array}[\theta_{index}])$
17. $\rho_{index} = round(\frac{\rho_{value} - \rho_{array}[0]}{\rho_{step}})$
18. $Accumulator[\rho_{index}][\theta_{index}] += 1$
19. $end\ for$
20. $end\ if$
21. $end\ for$
22. $Lines =$
    $\{(\rho_{array}[\rho_{index}],\ \theta_{array}[\theta_{index}]) \mid Accumulator[\rho_{index}][\theta_{index}] > threshold_{line}\}$



Obr. 4 – 5 Reprezentácia priamky v parametrickom priestore ΘP.



Obr. 4 – 6 Reprezentácia všetkých priamok, prechádzajúcich daným bodom, v parametrickom priestore ΘP.



Obr. 4 – 7 Grafické riešenie hľadania parametrov spoločnej priamky dvoch bodov v parametrickom priestore ΘP.

4.2 Houghova transformácia pre detekciu priamok v 3D priestore

Algoritmus Houghovej transformácie pre detekciu priamok v 3D priestore je možné skonštruovať pomocou predchádzajúceho algoritmu pre detekciu priamok v 2D priestore. Princíp algoritmu spočíva v rozklade úlohy detekcie priamky v 3D priestore na dve podúlohy detekcie priamok v 2D priestore vzhľadom na to, že priamku v 3D priestore je možné popísať dvoma parametrickými rovnicami, napr.

$$z_i = -\frac{\cos\theta_{xz}}{\sin\theta_{xz}}x_i + \frac{\rho_{xz}}{\sin\theta_{xz}} \tag{4.6}$$

$$z_i = -\frac{\cos\theta_{yz}}{\sin\theta_{yz}}y_i + \frac{\rho_{yz}}{\sin\theta_{yz}} \tag{4.7}$$

, kde x_i, y_i a z_i sú súradnice bodov ležiacich na priamke v priestore XYZ, ρ_{xz} je kolmá vzdialenosť priamky od počiatku súradnicovej sústavy a θ_{xz} je polárny uhol v priestore XZ, ρ_{yz} je kolmá vzdialenosť priamky od počiatku súradnicovej sústavy a θ_{yz} je polárny uhol v priestore YZ. Geometrický význam týchto rovníc spočíva v definovaní priamky ako priesečníka dvoch rovín v 3D priestore, pričom jedna rovina je kolmá na rovinu XZ, a druhá rovina je kolmá na rovinu YZ (Obrázok 4 – 8). Úloha detekcie priamok v 3D priestore preto spočíva v nájdení parametrov ρ_{xz}, θ_{xz}, ρ_{yz} a θ_{yz}, ktoré by odpovedali detegovaným priamkam. Túto úlohu rieši nasledujúci algoritmus:

1. Prostredníctvom metódy Houghovej transformácie pre priamky v 2D priestore deteguj priamky v priestore XZ.
2. Prostredníctvom metódy Houghovej transformácie pre priamky v 2D priestore deteguj, k priamkam nájdeným v priestore XZ, priamky v ďalšom rozmere, t.j. v priestore XY alebo YZ.
3. Výsledné detegované priamky sú popísané štvoricou parametrov $((\rho_{xz},\ \theta_{xz}),\ (\rho_{xy},\ \theta_{xy}))$ alebo $((\rho_{xz},\ \theta_{xz}),\ (\rho_{yz},\ \theta_{yz}))$.

Pseudokód metódy Houghovej transformácie pre detekciu priamok v 3D priestore

$Points$ vstupné údaje 3D priestoru

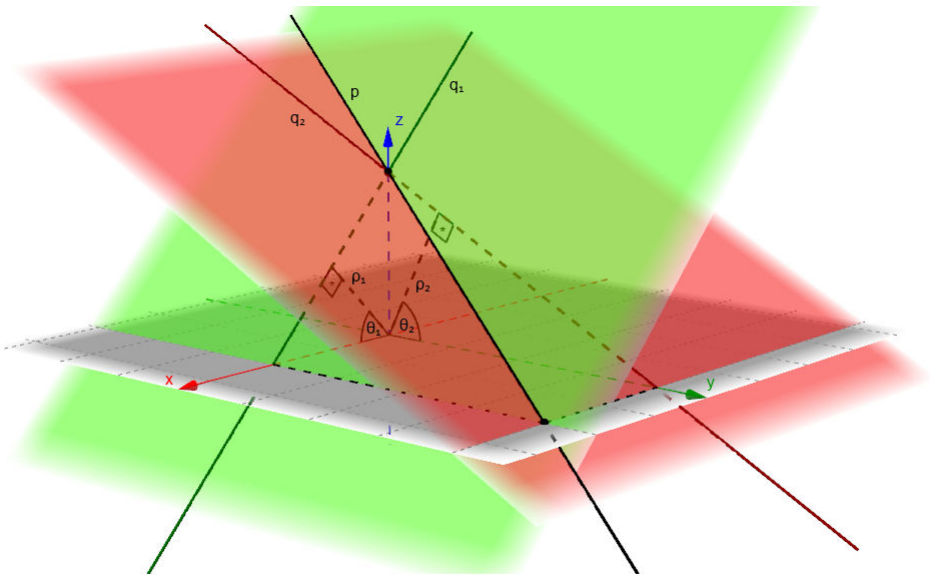
$HoughLines2D$ metóda Houghovej transformácie pre detekciu priamok v 2D priestore

$Lines_{xz}$ množina dvojíc parametrov detegovaných priamok v priestore XZ

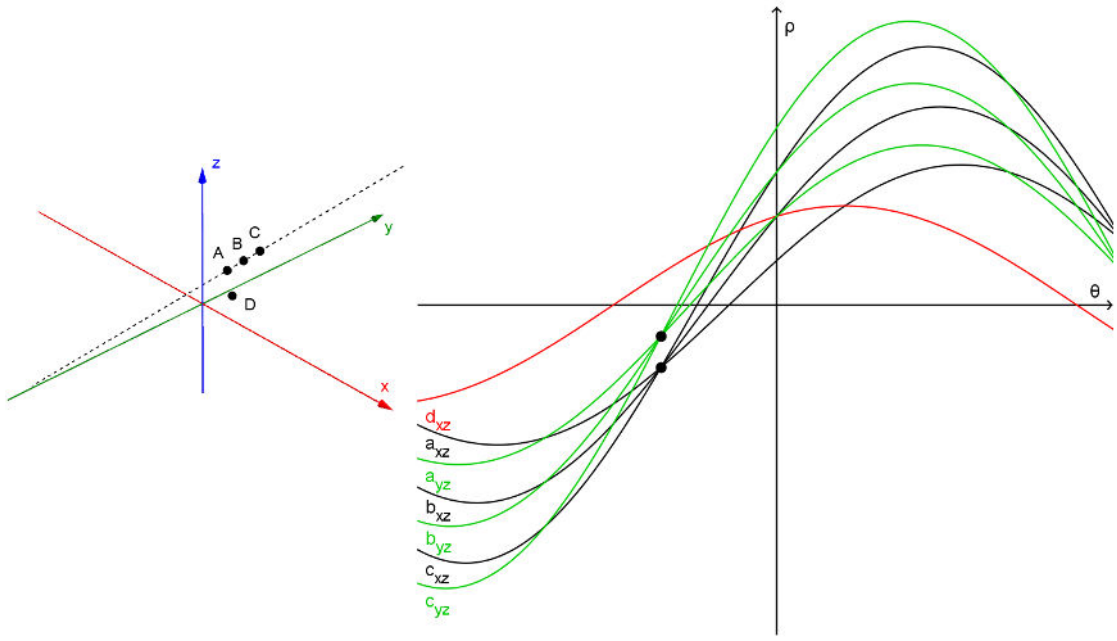
$Lines_{xyz}$ množina dvojíc dvojíc parametrov detegovaných priamok v priestore XYZ

1. $Lines_{xz} = HoughLines2D(Points_{xz})$
2. $Lines_{xyz} = \{(Lines_{xzi},\ HoughLines2D(Points_{Lines_{xzi}y}))\}$

Na Obrázku 4 – 9 je možné vidieť aplikáciu tohto algoritmu na množinu bodov v 3D priestore. Po prvej aplikácii metódy pre detekciu priamok v 2D priestore sa z počiatočnej množiny bodov vylúčil bod D. Po druhej aplikácii metódy pre detekciu priamok v 2D priestore vidíme, že riešenie určilo priamku, ktorá prechádza bodmi A, B a C.



Obr. 4 – 8 Grafické zobrazenie popisu priamky v 3D priestore.



Obr. 4 – 9 Aplikácia metódy Houghovej transformácie pre priamky v 3D priestore.

4.3 Houghova transformácia pre detekciu rovín

Princíp algoritmu Houghovej transformácie pre detekciu rovín je rovnaký ako princíp algoritmu prezentovaný pri Houghovej transformácii pre priamky v 2D priestore. Ak by sme sa pokúsili použiť pre popis roviny rovnicu

$$z_i = mx_i + ny_i + o \tag{4.8}$$

, kde x_i, y_i a z_i sú súradnice bodov ležiacich na rovine v priestore XYZ, m a n sú smernice a o je posunutie roviny po z-ovej osi, a teda definovali parametrický priestor MNO, narazili by sme na obdobný problém ako pri Houghovej technike. Vhodným riešením je popis roviny pomocou sférických súradníc

$$z_i = -\frac{\sin\phi\cos\theta}{\cos\phi}x_i - \frac{\sin\phi\sin\theta}{\cos\phi}y_i + \frac{\rho}{\cos\phi} \tag{4.9}$$

, kde x_i, y_i a z_i sú súradnice bodov ležiacich na rovine v priestore XYZ, ρ je kolmá vzdialenosť roviny od počiatku súradnicovej sústavy, θ je azimutový uhol, a ϕ je zenitový uhol (Obrázok 4 – 10). Pre popis roviny tak môžeme využiť parametrický

priestor určený kolmou vzdialenosťou roviny od počiatku súradnicovej sústavy, azimutovým a zenitovým uhlom. Týmto spôsobom je možné popísať každú rovinu jedinečným bodom $(\phi,\ \theta,\ \rho)$ parametrického priestoru, napr. práve vtedy, keď $\phi \in [-\frac{\pi}{2},\ \frac{\pi}{2}) \wedge \theta \in [-\frac{\pi}{2},\ \frac{\pi}{2}) \wedge \rho \in R$. Ak by sme predchádzajúcu rovnicu upravili na tvar

$$\rho_i = x \sin\phi_i \cos\theta_i + y \sin\phi_i \sin\theta_i + z \cos\phi_i \tag{4.10}$$

, kde ρ_i, ϕ_i a θ_i sú súradnice bodov plochy v priestore ΦΘP, x, y a z sú súradnice bodu v priestore XYZ, potom by plocha, popísaná touto rovnicou, bola reprezentáciou všetkých rovín prechádzajúcich bodom $(x,\ y,\ z)$ v priestore XYZ (Obrázok 4 – 11). Z toho vyplýva, že ak ľubovoľné body ležia na tej istej rovine, jej parametre je možné určiť ako priesečník plôch v parametrickom priestore, kde tieto plochy reprezentujú všetky roviny prechádzajúce danými bodmi v priestore XYZ (Obrázok 4 – 12). Algoritmus pre detekciu rovín sa tak od algoritmu prezentovaného pri detekcii priamok, okrem rovnice výpočtu, v ničom podstatnom nezmenil. Avšak so zmenou rovnice výpočtu máme jeden parameter naviac, preto je nutné použiť trojrozmernú akumulačnú maticu. Na základe popísaného algoritmu bol vytvorený nasledujúci pseudokód.

Pseudokód metódy Houghovej transformácie pre detekciu rovín

$Points$ vstupné údaje 3D priestoru

$Planes$ množina trojíc parametrov detegovaných rovín

$Accumulator$ akumulačné pole

$threshold_{point}$ prahová hodnota bodov

$threshold_{plane}$ prahová hodnota počtu bodov roviny

$[\theta_{min},\ \theta_{max}]$ rozsah azimutového uhla

θ_{array} jednorozmerné pole hodnôt azimutového uhla

θ_{count} veľkosť jednorozmerného poľa hodnôt azimutového uhla

θ_{step} krok diskretizácie priestoru azimutového uhla

$[\phi_{min},\ \phi_{max}]$ rozsah zenitového uhla

ϕ_{array} jednorozmerné pole hodnôt zenitového uhla

ϕ_{count} veľkosť jednorozmerného poľa hodnôt zenitového uhla

ϕ_{step} krok diskretizácie priestoru zenitového uhla

ρ_{max} absolútna hodnota maximálnej možnej kolmej vzdialenosti roviny

ρ_{array} jednorozmerné pole hodnôt kolmej vzdialenosti roviny

ρ_{count} veľkosť jednorozmerného poľa hodnôt kolmej vzdialenosti roviny

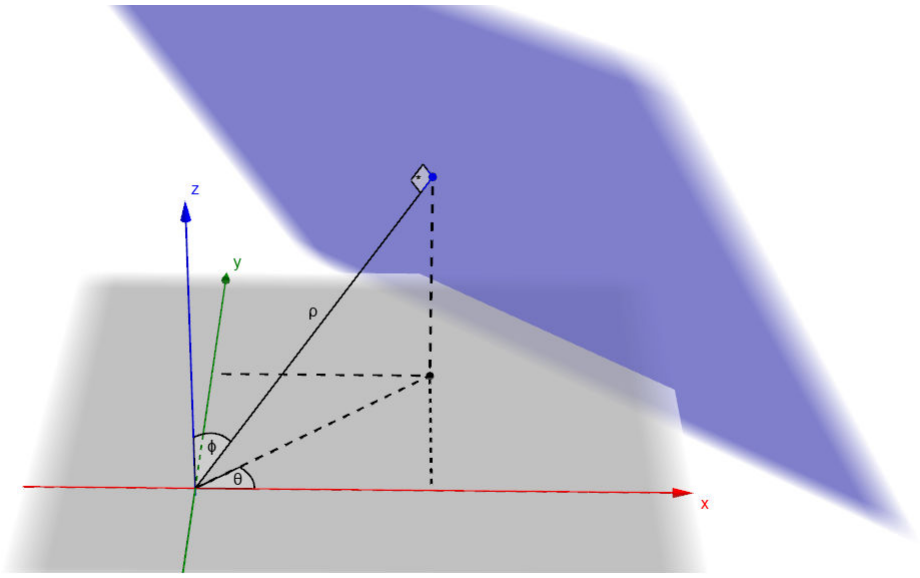
ρ_{step} krok diskretizácie priestoru kolmej vzdialenosti roviny

ρ_{value} vypočítaná hodnota kolmej vzdialenosti roviny

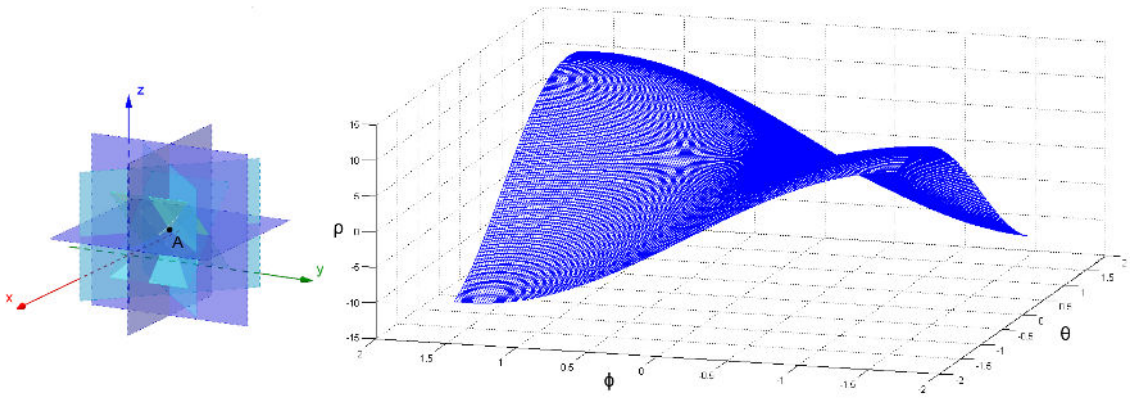
1. $\theta_{min} = -\frac{\pi}{2}$
2. $\theta_{max} = \frac{\pi}{2}$
3. $\phi_{min} = -\frac{\pi}{2}$
4. $\phi_{max} = \frac{\pi}{2}$
5. $X_{min} = min(Points_x)$
6. $X_{max} = max(Points_x)$
7. $Y_{min} = min(Points_y)$
8. $Y_{max} = max(Points_y)$
9. $Z_{min} = min(Points_z)$
10. $Z_{max} = max(Points_z)$

11. $\rho_{max} = \sqrt{X_{max}^2 + Y_{max}^2 + Z_{max}^2} > \sqrt{X_{min}^2 + Y_{min}^2 + Z_{min}^2}$
    $?\sqrt{X_{max}^2 + Y_{max}^2 + Z_{max}^2} : \sqrt{X_{min}^2 + Y_{min}^2 + Z_{min}^2}$
12. $\rho_{count} = ceil(\frac{2\rho_{max}}{\rho_{step}}) + 1$
13. $\rho_{array} = [-\rho_{max}, -\rho_{max} + \rho_{step}, ..., -\rho_{max} + \rho_{step}(\rho_{count} - 1)]_{\rho_{count}}$
14. $\theta_{count} = floor(\frac{\theta_{max} - \theta_{min}}{\theta_{step}}) + 1$
15. $\theta_{array} = [\theta_{min}, \theta_{min} + \theta_{step}, ..., \theta_{min} + \theta_{step}(\theta_{count} - 1)]_{\theta_{count}}$
16. $\phi_{count} = floor(\frac{\phi_{max} - \phi_{min}}{\phi_{step}}) + 1$
17. $\phi_{array} = [\phi_{min}, \phi_{min} + \phi_{step}, ..., \phi_{min} + \phi_{step}(\phi_{count} - 1)]_{\phi_{count}}$
18. $Accumulator[]_{\rho_{count}}[]_{\theta_{count}}[]_{\phi_{count}} = 0$
19. $for\ point : Points$
20. $if\ point_{value} > threshold_{point}$
21. $for\ \theta_{index} = 0 : (\theta_{count} - 1)$
22. $for\ \phi_{index} = 0 : (\phi_{count} - 1)$
23. $\rho_{value} = point_x \sin(\phi_{array}[\phi_{index}]) \cos(\theta_{array}[\theta_{index}])$
    $+ point_y \sin(\phi_{array}[\phi_{index}]) \sin(\theta_{array}[\theta_{index}])$
    $+ point_z \cos(\phi_{array}[\phi_{index}])$
24. $\rho_{index} = round(\frac{\rho_{value} - \rho_{array}[0]}{\rho_{step}})$
25. $Accumulator[\rho_{index}][\theta_{index}][\phi_{index}] += 1$
26. $end\ for$
27. $end\ if$
28. $end\ for$
29. $end\ for$

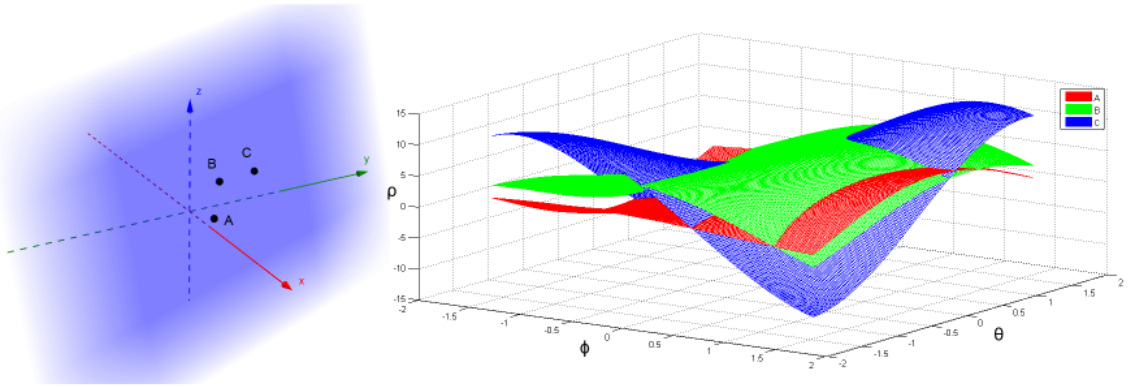
30. $Planes = \{(\rho_{array}[\rho_{index}],\ \theta_{array}[\theta_{index}],\ \phi_{array}[\phi_{index}])\ |$
$Accumulator[\rho_{index}][\theta_{index}][\phi_{index}] > threshold_{plane}\}$



Obr. 4 – 10 Grafické zobrazenie popisu roviny prostredníctvom sférických súradníc.



Obr. 4 – 11 Reprezentácia všetkých rovín, prechádzajúcich daným bodom, v parametrickom priestore ΦΘP.



Obr. 4 – 12 Aplikácia metódy Houghovej transformácie pre roviny.

5 Algoritmus rozpoznávania spŕšok

Ako bolo spomenuté, cieľom tejto práce bol vývoj alternatívneho algoritmu rozpoznávania vzorov spŕšok, založeného na metódach Houghovej transformácie. Výber vhodných metód Houghovej transformácie prebehol na základe vlastností spŕšok. Vývoj algoritmu prebehol v dvoch fázach, kedy v prvej fáze sa uskutočnili návrh, implementácia a analýza čo najjednoduchšieho algoritmu rozpoznávania vzorov spŕšok, ktorý slúžil ako zdroj informácií pre fázu dva, kedy prebehol vývoj finálneho algoritmu tejto práce. Počas vývoja sa pracovalo so simuláciou a rekonštrukciou spŕšok pomocou rámca ESAF. Algoritmy boli implementované ako v jazyku Java, tak aj, pre potreby začlenenia algoritmu do rámca ESAF, v jazyku C++. O prepis algoritmu do jazyka C++ a jeho integráciu s rámcom ESAF sa postaral Michal Vrábel (Vrábel, 2015).

5.1 Vlastnosti a údajová reprezentácia spŕšok

Údajová reprezentácia spŕšok je založená na informáciách o dopade fotónov na ohniskovú plochu detektora v určitých časových intervaloch. Detailnejšie o informáciách podstatných pre návrh algoritmu:

Pixel ohniskovej plochy Pixel ohniskovej plochy reprezentuje miesto dopadu fotónu. Pixely sú popísané indexmi v rôznych štruktúrach ohniskovej plochy (Obrázok 5 – 1, Obrázok5 – 2, Obrázok5 – 3), a taktiež priestorovými súradnicami vzhľadom na počiatok súradnej sústavy v strede ohniskovej plochy. K dispozícii sú:

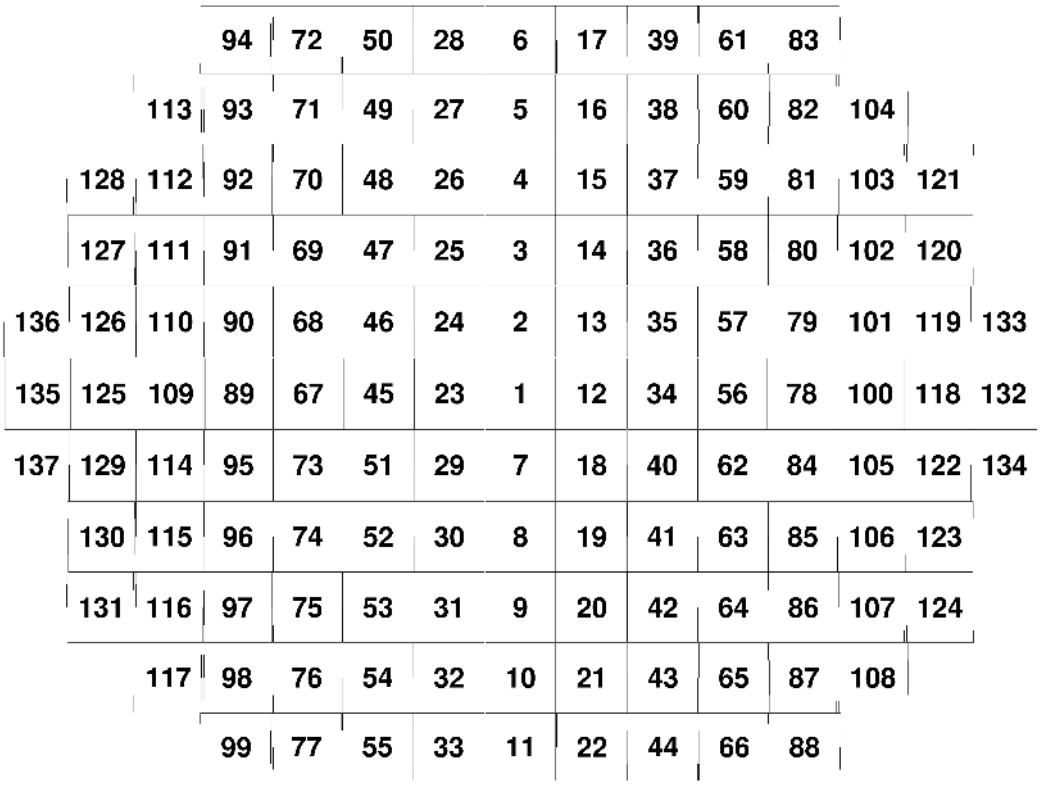
- X-ová súradnica pixela,
- Y-ová súradnica pixela,
- Z-ová súradnica pixela.

GTU GTU predstavuje relatívny časový interval, v ktorom sú ukladané informácie o dopade fotónov na pixely ohniskovej plochy. Tento interval má rozsah 2.5 μs (Santangelo, Picozza a Ebisuzaki, 2013).

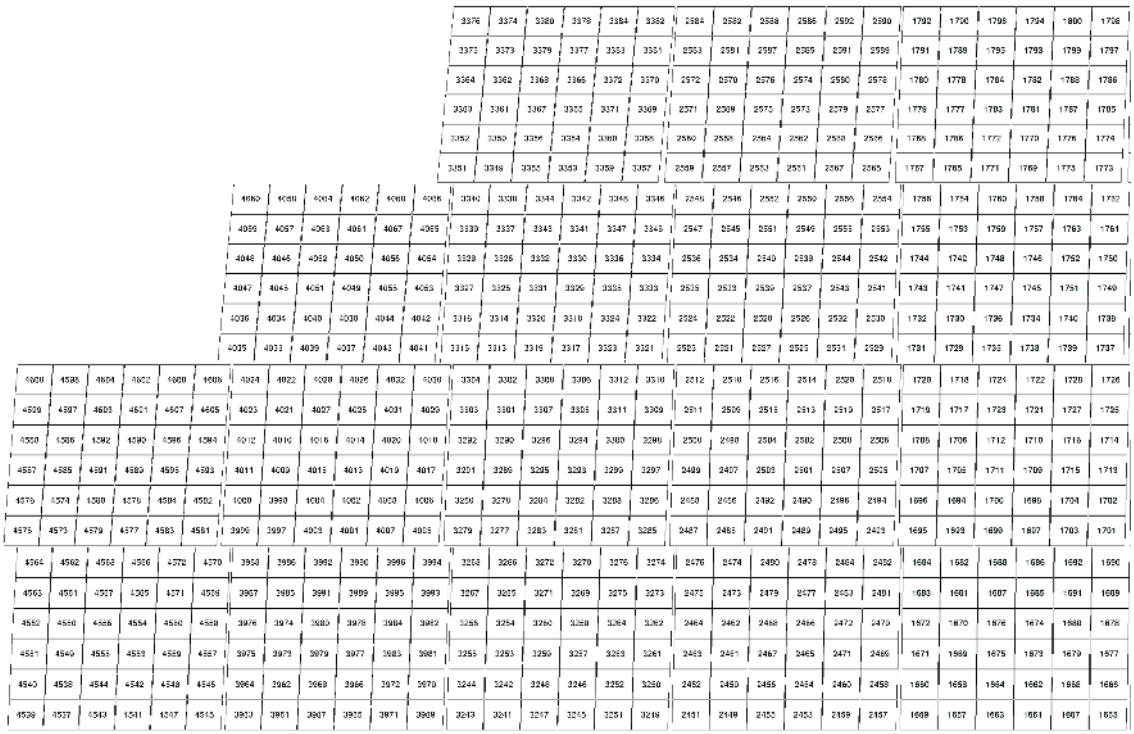
Počet signálov Informácia o počte fotónov, ktoré dopadli na pixel v intervale 2.5 μs.

Počet signálov bez šumu Informácia o počte fotónov, ktoré dopadli na pixel v intervale 2.5 μs a nemajú pôvod v UV pozadí. Táto informácia je dostupná iba pri simulovaných spŕškach a môže slúžiť pre vývojové a kontrolné účely.

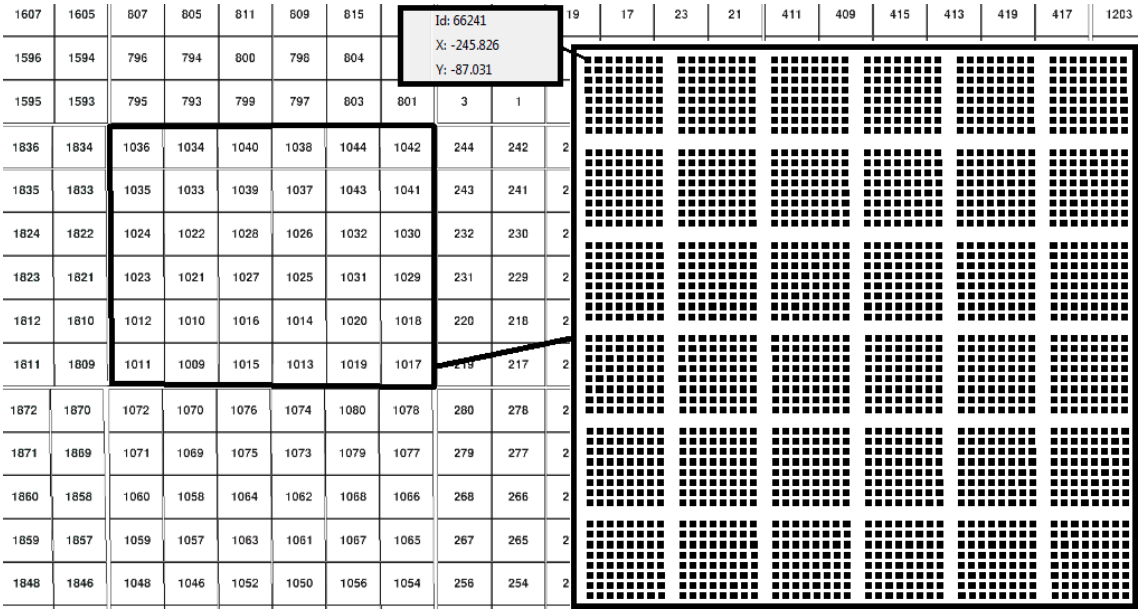
Príklad týchto údajov spŕšky je na Obrázku 5 – 4, kde vidieť časť údajov vygenerovaného súboru simulovanej spŕšky.



Obr. 5 – 1 PDM štruktúra ohniskovej plochy.



Obr. 5 – 2 PMT štruktúra PDM prvkov.



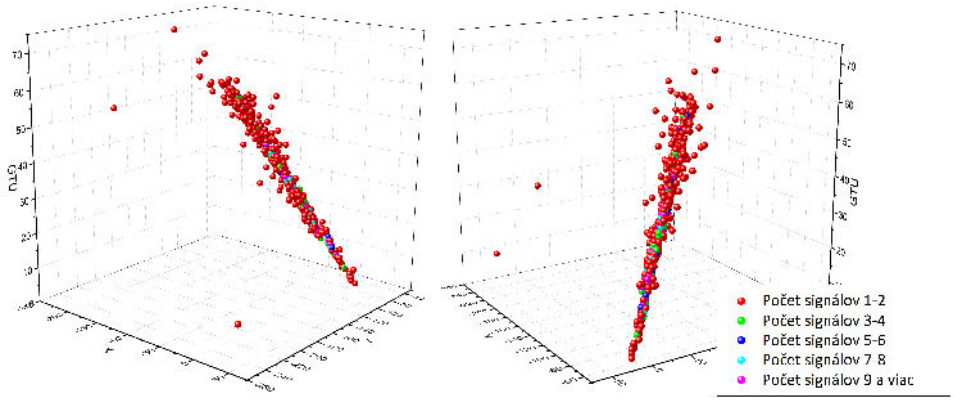
Obr. 5 – 3 Detail PMT štruktúry.

Energy	1.0000e+20 MeV
#theta	40.91 deg
#varphi	103.95 deg
X_{1}	40.43 g/cm^{2}
x_{init}	14.93 km
y_{init}	41.40 km
z_{init}	24.19 km
X_{max}	885.43 g/cm^{2}
x_{max}	19.24 km
y_{max}	24.05 km
z_{max}	3.56 km
EarthAge	1.22
x_{earth}	20.00 km
y_{earth}	21.00 km
z_{earth}	-0.07 km

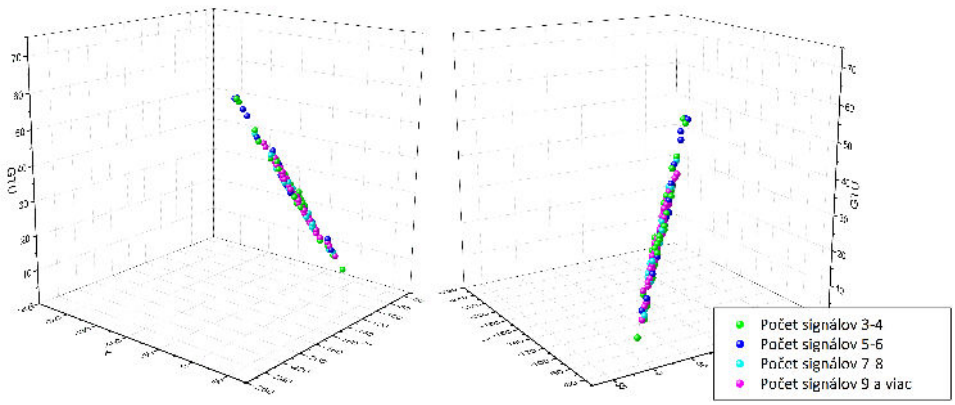
gtu	uid	pxX	pxY	pxZ	totalCount	signalCount	pmtId	pmtEC	pmtMC
0	64513	-217.891	-224.289	2.78248	1	0	1009	253	29
0	64517	-217.848	-236.269	2.07265	2	0	1009	253	29
0	64577	-217.987	-196.838	4.40912	1	0	1010	253	29
0	64578	-217.977	-199.833	4.23166	1	0	1010	253	29
0	64580	-217.956	-205.822	3.87674	1	0	1010	253	29
0	64581	-217.945	-208.817	3.69929	1	0	1010	253	29
0	64582	-217.935	-211.812	3.52183	1	0	1010	253	29
0	64583	-217.924	-214.807	3.34437	1	0	1010	253	29
0	64584	-217.913	-217.802	3.16691	1	0	1010	253	29
0	64522	-214.885	-227.284	2.78342	1	0	1009	253	29
0	64523	-214.875	-230.279	2.60597	1	0	1009	253	29
0	64524	-214.864	-233.274	2.42851	2	0	1009	253	29
0	64525	-214.853	-236.269	2.25105	6	0	1009	253	29
0	64527	-214.832	-242.259	1.89613	2	0	1009	253	29
0	64528	-214.822	-245.253	1.71868	2	0	1009	253	29

Obr. 5 – 4 Príklad údajov spŕšky.

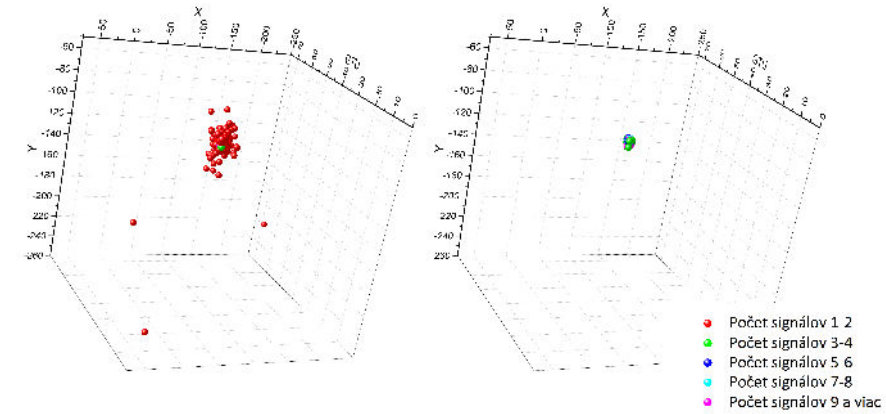
Z toho teda vidieť, že pre popis údajov máme k dispozícii tri priestorové súradnice, jednu časovú a k tomu informáciu o počte signálov. Vzhľadom na to, že priestorové zakrivenie ohniskovej plochy, popísané z-ovou súradnicou, nie je podstatne výrazné, bolo možné tento priestorový údaj zanedbať. S údajmi je tak možné pracovať v trojrozmernom priestore XYGtu. Zobrazenie údajov simulovanej spŕšky v tomto priestore je vidieť na Obrázku 5 – 5. To nás privádza k vlastnostiam spŕšok, ktoré by bolo možné využiť pri návrhu algoritmu. Zatiaľ čo na Obrázku 5 – 5 sú vykreslené všetky údaje spŕšky, a jej tvar pripomína kornút, na Obrázku 5 – 6 sú vykreslené už len údaje nad prahovou hodnotou počtu signálov dva. Je vidieť, že po tom, čo sa odstránili údaje s pôvodom v rozptýlených fotónoch spŕšky, sú zvyšné údaje v priestore usporiadané tesne okolo priamky, čo potvrdzuje aj Obrázok 5 – 7. Z tohto dôvodu bola ako vhodná metóda pre použitie zvolená metóda Houghovej transformácie pre detekciu priamok. Okrem nej bolo neskôr, pri návrhu druhého algoritmu, možné vďaka vhodným vlastnostiam využiť aj metódu Houghovej transformácie pre detekciu rovín.



Obr. 5 – 5 Zobrazenie údajov spŕšky v priestore XYGtu.



Obr. 5 – 6 Zobrazenie údajov spŕšky, nad prahovou hodnotou počtu signálov 2, v priestore XYGtu.



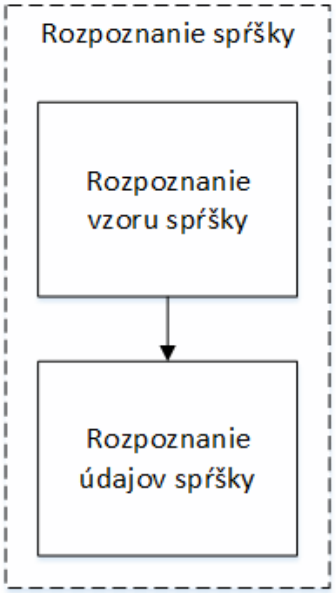
Obr. 5 – 7 Pohľad od koreňa spŕšky na zobrazenie jej údajov v priestore XYGtu.

5.2 Návrh algoritmu

Algoritmus pre rozpoznanie spŕšky pozostáva vo všeobecnosti z dvoch krokov (Obrázok 5 – 8):

1. Rozpoznanie vzoru spŕšky
2. Rozpoznanie údajov spŕšky

Aby sme si ujasnili terminológiu, pod pojmom algoritmus pre rozpoznanie spŕšky a algoritmus pre rozpoznanie vzoru, alebo vzoru spŕšky, sa myslí to isté. Výstupom algoritmu je vždy určitý nedokonalý objekt spŕšky - vzor spŕšky. Rozdiel medzi vzorom spŕšky v prvom kroku algoritmu a vzorom spŕšky ako výstupom algoritmu spočíva v tom, že na výstupe by mal byť objekt definovaný údajmi spŕšky, zatiaľ čo reprezentácia vzoru spŕšky po prvom kroku nemusí byť nutne rovnakého charakteru alebo kvality. Prvý krok rozpoznania vzoru spŕšky slúži ako predbežný výber na oddelenie čo najväčšieho množstva údajov s pôvodom v spŕške, od údajov s pôvodom v UV pozadí, a na lokalizáciu spŕšky. Druhý krok potom slúži na jemnejšie oddelenie údajov spŕšky od údajov pozadia. Túto štruktúru algoritmu si môžeme ukázať na algoritmoch rozpoznávania vzorov, popísaných v kapitole 3. Prvý krok pri metóde PWISE je realizovaný nastavením prahovej hodnoty počtu signálov a výberom pixelov vzhľadom na túto hodnotu, pri metóde Track finding method nastavením prahovej hodnoty počtu signálov a výberom údajov vzhľadom na túto hodnotu, a pri metóde LTT nastavením prahovej hodnoty počtu signálov, výberom údajov vzhľadom na túto hodnotu a následným získaním cesty s maximálnou akumuláciou signálu. V druhom kroku realizujú jednotlivé metódy svoje filtračné algoritmy pre výber údajov s pôvodom v spŕške, PWISE pracuje s výpočtami hodnôt UV pozadia a časových okien, Track finding method hľadá údaje, ktoré by boli usporiadané okolo priamej cesty maximalizujúcej hodnotu sumy počtu signálov, a LTT vyberá údaje na základe zadanej vzdialenosti od rozpoznanej cesty s maximálnou akumuláciou signálu.



Obr. 5 – 8 Štruktúra algoritmu rozpoznávania spŕšok.

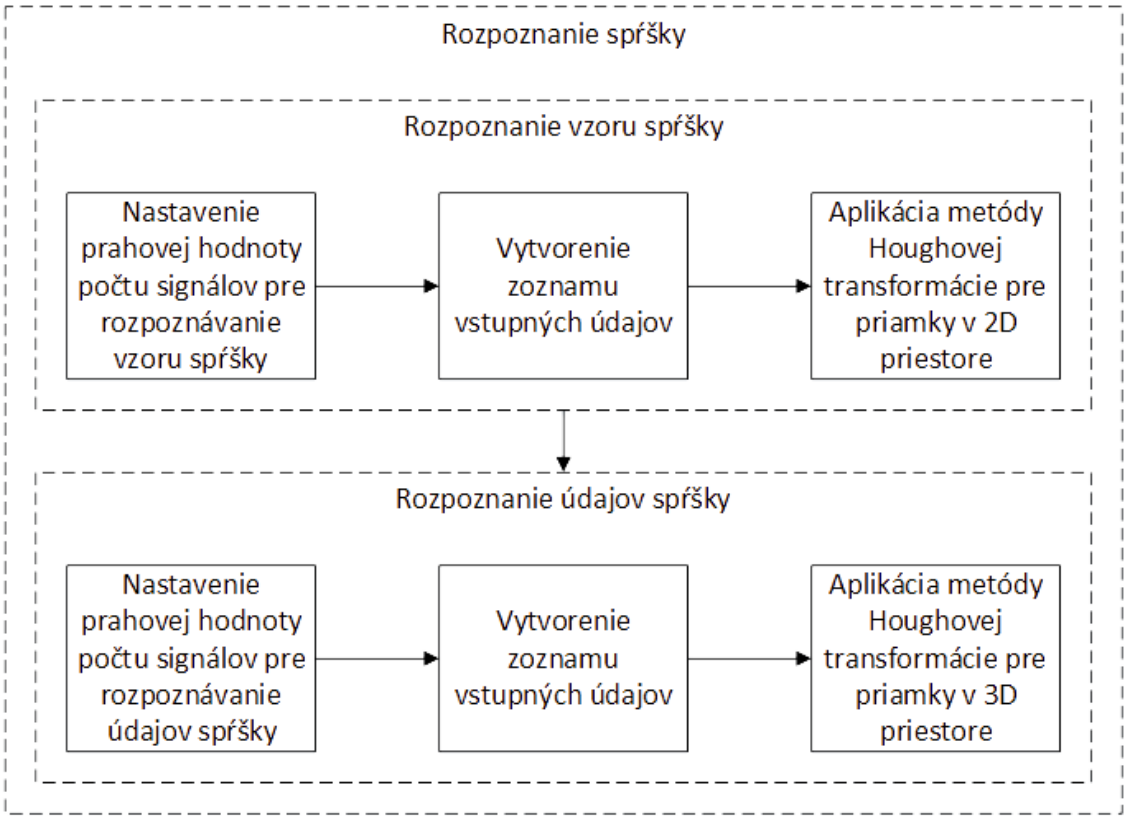
Touto štruktúrou sa riadil aj návrh algoritmov v tejto práci, ktorý je popísaný v nasledujúcich podkapitolách. Keďže cieľom pri vývoji algoritmu boli čo najlepšie výsledky uhlovej rekonštrukcie rozpoznaných spŕšok na simulovaných údajoch, kládol sa veľký dôraz na analýzu častí tejto štruktúry popísaných algoritmov, a na vhodné prevedenie a spôsob zvýšenia kvality týchto častí.

5.3 Algoritmus 1

Návrh prvého algoritmu pre rozpoznávanie spŕšok dáva dôraz na jednoduchosť. Využíva Houghovu transformáciu pre priamky v 2D a 3D priestore. Primárne mal tento algoritmus slúžiť pre získanie prvotných výsledkov z rozpoznávania a analýzy spŕšok a tiež ako zdroj informácií pre nasledovný komplexnejší návrh algoritmu.

5.3.1 Návrh algoritmu

Návrh prvého algoritmu je možné vidieť na Obrázku 5 – 9.



Obr. 5 – 9 Návrh prvého algoritmu rozpoznávania spŕšok.

Bolo rozhodnuté, že prvý algoritmus bude postavený na metódach Houghovej transformácie pre priamky v 2D a 3D priestore. Houghova transformácia pre priamky v 2D priestore mala poslúžiť v prvom kroku algoritmu k nájdeniu vzoru, pozostávajúceho z pixelov, z ktorého údajov by sa v druhom kroku prostredníctvom Houghovej transformácie pre priamky v 3D priestore rozhodlo, ktoré majú pôvod v spŕške a pôjdu ako vzor spŕšky na výstup. Detailnejšie:

Rozpoznanie vzoru spŕšky

1. Zadefinuje sa prahová hodnota počtu signálov.
2. Na každom pixeli sa nájde údaj s maximálnou hodnotou počtu signálov. Ak je táto hodnota nad zadefinovanou prahovou hodnotou z prvého kroku, údaj sa vyberie. Príklad zobrazenia pixelov vybraných údajov je na

Obrázku 5 – 10.

3. Na vybrané údaje sa v priestore XY aplikuje metóda Houghovej transformácie pre priamky v 2D priestore. Na základe výstupu metódy sa vytvorí vzor pozostávajúci z pixelov, ku ktorým sú jednotlivé výstupné údaje priradené. Parametrami metódy Houghovej transformácie pre priamky v 2D priestore sú, okrem vstupných údajov, zadefinované kroky diskretizácie parametrického priestoru a kolmá vzdialenosť od detegovanej priamky. Príklad zobrazenia pixelov nájdeného vzoru je na Obrázku 5 – 11.

Rozpoznanie údajov spŕšky

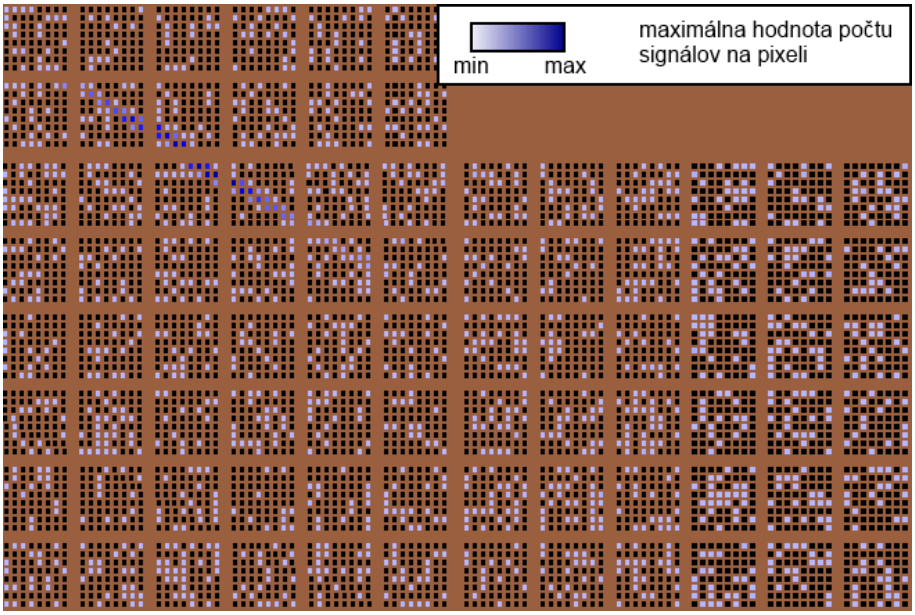
4. Zadefinuje sa prahová hodnota počtu signálov.

5. Vyberú sa údaje, ktoré patria pixelom vytvoreného vzoru a zároveň ich hodnota počtu signálov je nad zadefinovanou prahovou hodnotou zo štvrtého kroku.

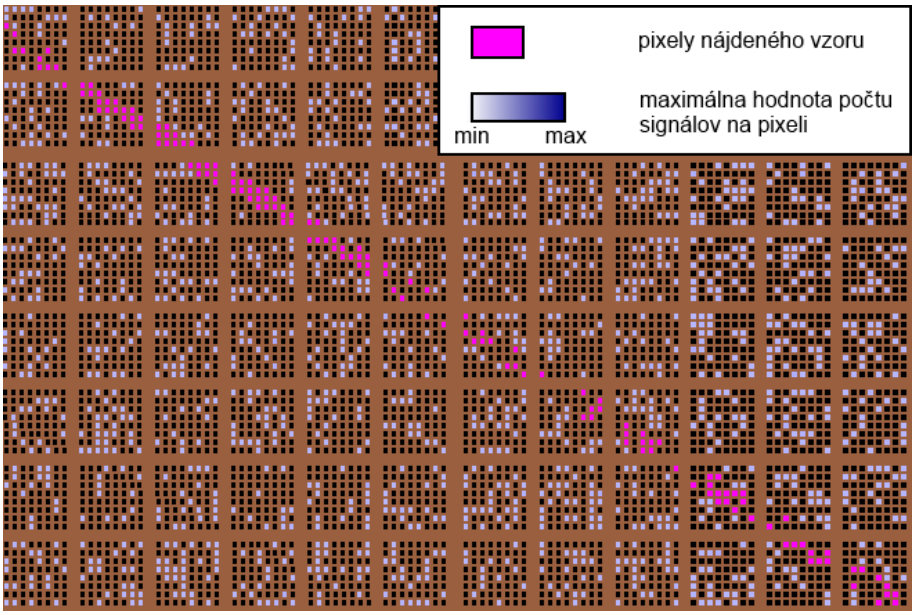
6. Na vybrané údaje sa aplikuje metóda Houghovej transformácie pre priamky v 3D priestore, pre ktorú platí, že jej subproblémy sa riešia v priestore XGtu a YGtu. Výstupom metódy sú údaje, ktoré tvoria finálny vzor spŕšky algoritmu. Parametrami metódy Houghovej transformácie pre priamky v 3D priestore sú, okrem vstupných údajov, zadefinované kroky diskretizácie parametrického priestoru a kolmá vzdialenosť od detegovaných priamok.

Pri metóde Houghovej transformácie pre priamky v 2D priestore je potrebné upresniť, že sa jedná o mierne upravenú verziu metódy Houghovej transformácie pre detekciu priamok v 2D priestore, prezentovanú v podkapitole 4.1. Tieto úpravy sú popísané v nasledujúcej podkapitole. Výsledkom prezentovaného algoritmu je vzor spŕšky, ktorého údaje sú v maximálnych zadefinovaných kolmých vzdialenostiach od priamky v priestoroch XY, XGtu a YGtu, pričom táto priamka bola určená na

základe akumulácie počtu signálov okolo nej, viď. nasledujúcu podkapitolu, s cieľom, aby prechádzala centrálnou líniou spŕšky.



Obr. 5 – 10 Zobrazenie pixelov vybraných údajov udalosti po nastavení prahovej hodnoty počtu signálov.



Obr. 5 – 11 Zobrazenie pixelov nájdeného vzoru.

5.3.2 Úprava metód Houghovej transformácie

Použitie metódy Houghovej transformácie pre detekciu priamok v 2D priestore pre účely algoritmu si vyžadovalo mierne úpravy:

Úprava metódy pre presné vyberanie údajov Štandardná metóda je postavená za účelom detekcie priamok na základe akumulácie bodov do určitej kolmej vzdialenosti, ktorá je spojená s krokom diskretizácie. To, pre potreby algoritmu rozpoznávania spŕšok, môže spôsobovať problémy. Problém ilustruje Obrázok 5 – 12. Máme skupinu bodov a chceli by sme ju detegovať a vybrať. Pre zjednodušenie problému sa uvažuje iba jeden uhol θ. Tiež poznáme kolmú vzdialenosť, vzhľadom na uhol θ, dvoch najvzdialenejších bodov. Predpokladáme preto, že po tom, čo detegujeme priamku, prechádzajúcu centrom skupiny pixelov, vyberieme túto skupinu bodov vzhľadom na kolmú vzdialenosť od detegovanej priamky. Ako ilustruje Obrázok 5 – 12 a), nemusí to byť vôbec jednoduché. Detegovaná priamka totižto centrom skupiny neprechádza, a teda nie je ani možné celú skupinu bodov vybrať. Ak by sme sa pokúsili zväčšiť krok diskretizácie tak, aby skupina bodov bola priamo súčasťou bodov detegovanej priamky, ako ukazuje Obrázok 5 – 12 b), opäť by sme sa požadovaného výsledku nedočkali. Problém teda spočíva v tom, že ak použijeme jemnejší krok diskretizácie, nie sme potom schopný detegovať priamky vzhľadom na body v požadovanej kolmej vzdialenosti od priamky, a naopak, ak krok diskretizácie zvýšime tak, aby sme dostali požadovanú hodnotu kolmej vzdialenosti bodov detegovanej priamky, ohromne stratíme presnosť pri detekcii tejto priamky. Riešenie tohto problému je triviálne - oddeliť krok diskretizácie kolmej vzdialenosti priamky od počiatku súradnicovej sústavy od maximálnej kolmej vzdialenosti bodov, vzhľadom na ktoré je priamka detegovaná. Výsledok takejto úpravy ilustruje Obrázok 5 – 12 c), kedy po zjemnení kroku diskretizácie a nastavení vhodnej maximálnej kolmej vzdialenosti bodov priamky, dostávame žiadaný výsledok. Čím menší krok diskretizácie, tým presnejšia de-

tekcia priamky, avšak je treba si uvedomiť, že to má za následok aj zvýšenie pamäťovej a výpočtovej náročnosti.

Zbavenie sa filtrácie údajov Nie je potrebné, aby v metóde prebiehala filtrácia údajov na základe zadaných prahových hodnôt.

Úprava inkrementácie Keďže údaje udalosti majú vzhľadom na počet signálov rôznu prioritu, malo by sa to prejaviť aj v metóde Houghovej transformácie. Rozhodlo sa, že ak je k nejakej priamke priradený údaj, jej hodnota sa inkrementuje o hodnotu počtu signálov tohto údaja.

Úprava výstupu Pre potreby algoritmu bolo vhodné, aby metóda dávala na výstup namiesto parametrov priamok priamo údaje k nim prislúchajúce.

Tieto úpravy metódy Houghovej transformácie pre detekciu priamok v 2D priestore boli zohľadnené v nasledujúcom pseudokóde.

Pseudokód upravenej metódy Houghovej transformácie pre detekciu priamok v 2D priestore

$Data$ vstupné údaje 2D priestoru

$OutputData$ výstupné údaje 2D priestoru

$Lines_{parametersIndexes}$ množina dvojíc indexov parametrov detegovaných priamok

$Accumulator$ akumulačné pole

$[\theta_{min},\ \theta_{max}]$ rozsah polárneho uhla

θ_{array} jednorozmerné pole hodnôt polárneho uhla

θ_{count} veľkosť jednorozmerného poľa hodnôt polárneho uhla

θ_{step} krok diskretizácie priestoru polárneho uhla

ρ_{max} absolútna hodnota maximálnej možnej kolmej vzdialenosti priamky

ρ_{array} jednorozmerné pole hodnôt kolmej vzdialenosti priamky

ρ_{count} veľkosť jednorozmerného poľa hodnôt kolmej vzdialenosti priamky

ρ_{step} krok diskretizácie priestoru kolmej vzdialenosti priamky

ρ_{value} vypočítaná hodnota kolmej vzdialenosti priamky

$size$ hodnota maximálnej kolmej vzdialenosti údajov detegovanej priamky

$[\rho_{lowerIndex},\ \rho_{upperIndex}]$ interval indexov kolmých vzdialeností priamok od počiatku súradnicovej sústavy, ktorý vyhovuje zadanej hodnote kolmej vzdialenosti údajov detegovanej priamky $size$

1. $\theta_{min} = -\frac{\pi}{2}$
2. $\theta_{max} = \frac{\pi}{2}$
3. $X_{min} = min(Data_x)$
4. $X_{max} = max(Data_x)$
5. $Y_{min} = min(Data_y)$
6. $Y_{max} = max(Data_y)$
7. $\rho_{max} = \sqrt{X_{max}^2 + Y_{max}^2} > \sqrt{X_{min}^2 + Y_{min}^2} ? \sqrt{X_{max}^2 + Y_{max}^2} : \sqrt{X_{min}^2 + Y_{min}^2}$
8. $\rho_{count} = ceil(\frac{2\rho_{max}}{\rho_{step}}) + 1$
9. $\rho_{array} = [-\rho_{max}, -\rho_{max} + \rho_{step}, ..., -\rho_{max} + \rho_{step}(\rho_{count} - 1)]_{\rho_{count}}$
10. $\theta_{count} = floor(\frac{\theta_{max} - \theta_{min}}{\theta_{step}}) + 1$
11. $\theta_{array} = [\theta_{min}, \theta_{min} + \theta_{step}, ..., \theta_{min} + \theta_{step}(\theta_{count} - 1)]_{\theta_{count}}$
12. $Accumulator[]_{\rho_{count}}[]_{\theta_{count}} = 0$
13. $for\ data : Data$
14. $for\ \theta_{index} = 0 : (\theta_{count} - 1)$

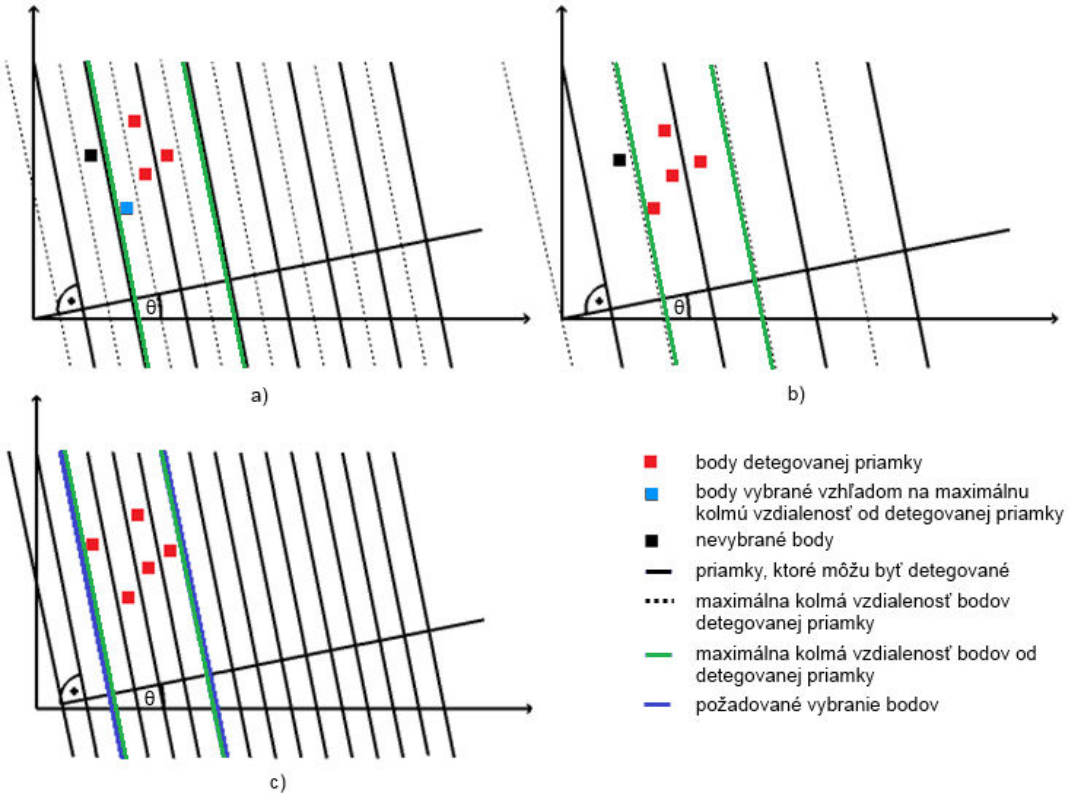
15. $\rho_{value} = data_x \cos(\theta_{array}[\theta_{index}]) + data_y \sin(\theta_{array}[\theta_{index}])$
16. $\rho_{lowerIndex} = round(\frac{\rho_{value} - size - \rho_{array}[0]}{\rho_{step}})$
17. $\rho_{upperIndex} = round(\frac{\rho_{value} + size - \rho_{array}[0]}{\rho_{step}})$
18. $if\ \rho_{lowerIndex} < 0$
19. $\rho_{lowerIndex} = 0$
20. $end\ if$
21. $if\ \rho_{upperIndex} > \rho_{count} - 1$
22. $\rho_{upperIndex} = \rho_{count} - 1$
23. $end\ if$
24. $for\ \rho_{index} = \rho_{lowerIndex} : (\rho_{upperIndex} - 1)$
25. $Accumulator[\rho_{index}][\theta_{index}] += data_{signalCount}$
26. $end\ for$
27. $end\ for$
28. $end\ for$
29. $lines_{parametersIndexes} =$
    $\{(\rho_{index},\ \theta_{index}) \mid Accumulator[\rho_{index}][\theta_{index}] = Accumulator_{max}\}$
30. $for\ (\rho_{index},\ \theta_{index}) : lines_{parametersIndexes}$
31. $for\ data : Data$
32. $\rho_{value} = data_x \cos(\theta_{array}[\theta_{index}]) + data_y \sin(\theta_{array}[\theta_{index}])$
33. $\rho_{lowerIndex} = round(\frac{\rho_{value} - size - \rho_{array}[0]}{\rho_{step}})$
34. $\rho_{upperIndex} = round(\frac{\rho_{value} + size - \rho_{array}[0]}{\rho_{step}})$
35. $if\ \rho_{lowerIndex} <= \rho_{index}\ \&\&\ \rho_{upperIndex} >= \rho_{index}$

36. $data \in OutputData$

37. $end\ if$

38. $end\ for$

39. $end\ for$



Obr. 5 – 12 Úprava metódy Houghovej transformácie pre priamky v 2D priestore.

Táto zmena tiež ovplyvní význam metódy Houghovej transformácie pre priamky v 3D priestore. Detegovanie priamky v 3D priestore sa, po tejto zmene, viaže k údajom, ktoré sú okolo priamky ohraničené štyrmi rovinami. Tvar takéhoto objektu môžme vidieť na Obrázku 5 – 13 a Obrázku 5 – 14. Hodnota vzdialenosti medzi dvoma rovinami, v pseudokóde označená ako *size*, nemusí byť pri oboch dvojiciach rovín rovnaká, čo bolo využité pri druhom návrhu algoritmu. Avšak v tejto verzii algoritmu bola použitá iba jedna spoločná hodnota premennej *size* pre obe dvojdi-

menzionálne podproblémy tejto metódy. Tomuto odpovedá nasledujúci pseudokód metódy Houghovej transformácie pre priamky v 3D priestore:

Pseudokód upravenej metódy Houghovej transformácie pre detekciu priamok v 3D priestore

$Data$ vstupné údaje 3D priestoru

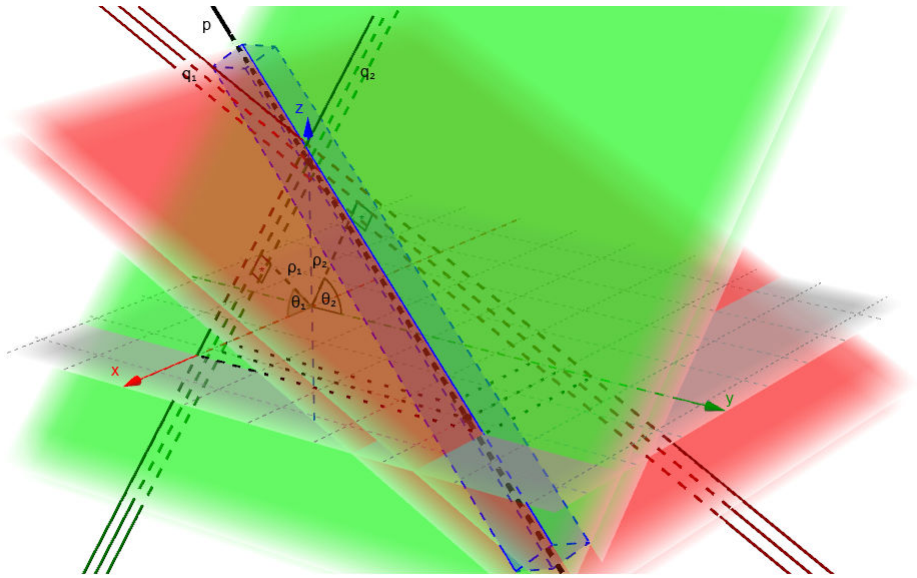
$OutputData$ výstupné údaje 3D priestoru

$HoughLines2D$ metóda Houghovej transformácie pre detekciu priamok v 2D priestore

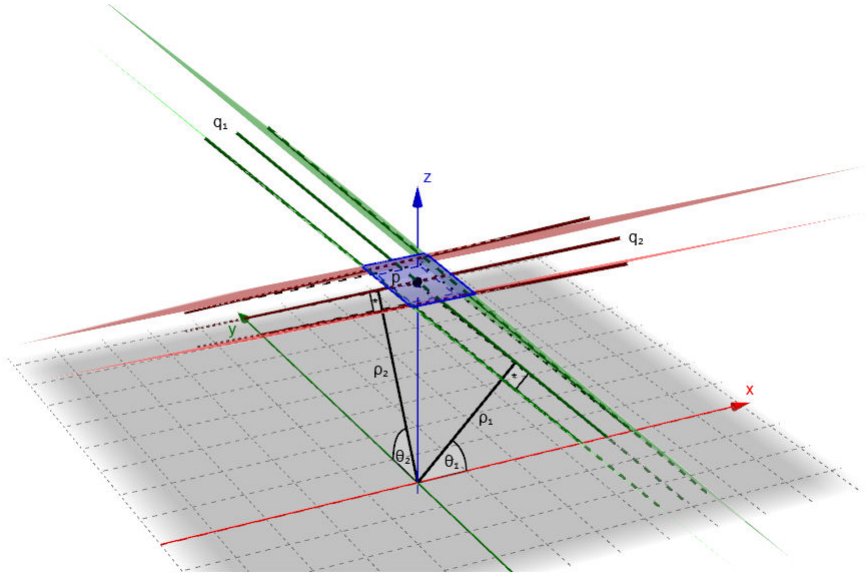
$Data_{L_{xz}}$ množina údajov detegovaných priamok v priestore XZ

$size$ hodnota maximálnej kolmej vzdialenosti údajov detegovanej priamky

1. $Data_{L_{xz}} = HoughLines2D(Data,\ size)$
2. $OutputData = HoughLines2D(Data_{L_{xz}},\ size)$



Obr. 5 – 13 Geometrický význam upravenej metódy Houghovej transformácie pre priamky v 3D priestore.



Obr. 5 – 14 Tvar detegovaného objektu upravenej metódy Houghovej transformácie pre priamky v 3D priestore.

5.3.3 Implementácia a výpočtové zložitosti

Pre potreby vývoja bol algoritmus implementovaný v jazyku Java, spoločne s jeho čiastočnou vizualizáciou. Po jeho dokončení bol Michalom Vrábelom (Vrábel, 2015) prepísaný tak, aby sa dal použiť v rámci ESAF. V nasledujúcom popise bude predstavený zdrojový kód Java triedy, ktorá obsahuje implementáciu algoritmu a je súčasťou programu, vytvoreného pre vývoj a vizualizáciu algoritmov tejto práce. Kompletný zdrojový kód je možné vidieť v prílohe A - Trieda Algorithm1.

Najprv sa bližšie pozrieme na implementáciu najdôležitejšej časti algoritmu - metódy Houghovej transformácie pre priamky. V algoritme sa pre volanie metód Houghovej transformácie pre priamku v 3D a 2D priestore používajú dve funkcie, ktoré využívajú funkciu s implementáciou algoritmu Houghovej transformácie pre priamky v 2D priestore.

Prvou funkciou je funkcia `HoughLine2D`, používaná pre volanie Houghovej transformácie pre priamky v 2D priestore pri hľadaní vzoru (Zdrojový kód 1).

Zdroj. kód 1 Funkcia `HoughLine2D`, použitá pri hľadaní vzoru.

```
/**
 * Houghova transformácia pre priamky v 2D priestore
 *
 * @param data - údaje
 * @param size - maximálna povolená kolmá vzdialenost údajov od
 * detegovanej priamky
 * @param indexFirstDim - index prvej dimenzie 2D priestoru
 * @param indexSecondDim - index druhej dimenzie 2D priestoru
 * @param thetaStep - krok diskretizácie priestoru uhla fi
 * [deg, priestor od -90 do 90 stupňov]
 * @param roStep - krok diskretizácie priestoru kolmej
 * vzdialenosti priamky od počiatku súradnicovej sústavy
 * @return - výstupné údaje Houghovej transformácie pre priamky
 * v 2D priestore
 */
```

```
    private List<Double[]>[] HoughLine2D(List<Double[]> data,
        double size, int indexFirstDim, int indexSecondDim, double
        thetaStep, double roStep) {
         //maximálna absolútna kolmá vzdialenost údajov od počiatku
            súradnicovej sústavy
         double ro = getMaxDist(data, indexFirstDim, indexSecondDim)
            ;
         return HoughLine2D(data, size, indexFirstDim,
            indexSecondDim, thetaStep, -Math.PI / 2, Math.PI / 2,
            roStep, -ro, ro);
    }
```

Táto funkcia okrem posunutia svojich parametrov, ktoré obsahujú údaje spŕšky, maximálnu kolmú vzdialenosť údajov detegovanej priamky, indexy dimenzií a kroky diskretizácie, ešte počíta maximálnu absolútnu kolmú vzdialenosť priamok od počiatku súradnicovej sústavy. Rozsahy polárneho uhla a maximálnej kolmej vzdialenosti priamok od počiatku súradnicovej sústavy tak odpovedajú rozsahom uvedeným v pseudokóde. S týmito parametrami sa volá funkcia s implementáciou algoritmu Houghovej transformácie pre priamky v 2D priestore.

Druhou funkciou je funkcia `HoughLine3D`, používaná pre volanie Houghovej transformácie pre priamky v 3D priestore pri rozpoznávaní údajov (Zdrojový kód 2).

Zdroj. kód 2 Funkcia `HoughLine3D`, použitá pri rozpoznávaní údajov.

```
    /**
     * Houghova transformácia pre priamky v 3D priestore
     *
     * @param data - údaje
     * @param size - maximálna povolená kolmá vzdialenost údajov od
     * detegovanej priamky
     * @param xIndex - index prvej dimenzie 3D priestoru
     * @param yIndex - index druhej dimenzie 3D priestoru
     * @param zIndex - index tretej dimenzie 3D priestoru
     * @param thetaStep - krok diskretizácie priestoru uhla fi
```

```
     * [deg, priestor od -90 do 90 stupňov]
     * @param roStep - krok diskretizácie priestoru kolmej
     * vzdialenosti priamky od počiatku súradnicovej sústavy
     * @return - výstupné údaje Houghovej transformácie pre priamky
     * v 3D priestore
     */
    private List<Double[]>[] HoughLine3D(List<Double[]> data,
        double size, int xIndex, int yIndex, int zIndex, double
        thetaStep, double roStep) {
         List<Double[]>[] firstData = HoughLine2D(data, size, xIndex
             , yIndex, thetaStep, roStep);
         List<Double[]>[] result = HoughLine2D(firstData[dataIndex],
              size, yIndex, zIndex, thetaStep, roStep);
         result[rangesIndex].add(firstData[rangesIndex].get(
            thetaIndexL));
         result[rangesIndex].add(firstData[rangesIndex].get(roIndexL
            ));
         return result;
    }
```

Význam parametrov tejto funkcie je zrejmý, keďže sa používajú ako parametre predchádzajúcej popísanej funkcie. Indexy dimenzií v algoritme pre rozpoznávanie sú: `xIndex` - X, `yIndex` - GTU, `zIndex` - Y. Najprv sa teda volá metóda Houghovej transformácie pre priamky v priestore XGtu, a na výsledných údajoch sa volá metóda Houghovej transformácie pre priamky v priestore GtuY. Zdrojový kód, ktorý ešte nasleduje, má využitie pri druhom algoritme, preto bude popísaný neskôr.

Nasledujúca funkcia `HoughLine2D`, s implementáciou algoritmu Houghovej transformácie pre priamky v 2D priestore, obsahuje aj kód použitý pre účely druhého algoritmu, preto sa týmto kódom zatiaľ nie je nutné zaoberať. Zdrojový kód je popisovaný postupne.

Zdroj. kód 3 Funkcia `HoughLine2D`, obsahujúca implementáciu Houghovej transformácie pre priamky v 2D priestore (1. časť).

```
    /**
     * Houghova transformácia pre priamky v 2D priestore
     *
     * @param data - údaje
     * @param size - maximálna povolená kolmá vzdialenost údajov od
     * detegovanej priamky
     * @param indexFirstDim - index prvej dimenzie 2D priestoru
     * @param indexSecondDim - index druhej dimenzie 2D priestoru
     * @param thetaStep - krok diskretizácie priestoru uhla
     * theta [deg]
     * @param thetaStart - začiatok priestoru uhla theta
     * v radiánoch (v intervale od -PI/2 do PI/2)
     * @param thetaEnd - koniec priestoru uhla theta v radiánoch
     * (v intervale od -PI/2 do PI/2)
     * @param roStep - krok diskretizácie priestoru kolmej
     * vzdialenosti priamky od počiatku súradnicovej sústavy
     * @param roStart - začiatok priestoru kolmej vzdialenosti
     * priamky od počiatku súradnicovej sústavy
     * @param roEnd - koniec priestoru kolmej vzdialenosti priamky
     * od počiatku súradnicovej sústavy
     * @return - výstupné údaje Houghovej transformácie pre priamky
     * v 2D priestore
     */
    private List<Double[]>[] HoughLine2D(List<Double[]> data,
        double size, Integer indexFirstDim, Integer indexSecondDim,
        double thetaStep, double thetaStart, double thetaEnd, double
         roStep, double roStart, double roEnd) {

        //definovanie pola hodnôt kolmej vzdialenosti priamky
        //od počiatku súradnicovej sústavy
        int nRo = (int) (Math.ceil(Math.abs(roEnd - roStart) /
            roStep)) + 1;
```

```
        double[] roArray = new double[nRo];
        int index = 0;
        double roValue = roStart;
        while (index < nRo) {
            roArray[index++] = roValue;
            roValue += roStep;
        }

        //definovanie pola hodnôt uhla theta
        thetaStep = thetaStep * Math.PI / 180;
        double[] thetaArray;
        int nTheta;
        if (thetaStart <= thetaEnd) {
            nTheta = (int) (Math.abs(thetaEnd - thetaStart) /
                thetaStep) + 1;
            if ((nTheta - 1) * thetaStep + thetaStart < thetaEnd &&
                 nTheta * thetaStep + thetaStart < Math.PI / 2) {
                 nTheta++;
            }
            thetaArray = new double[nTheta];
            index = 0;
            double thetaValue = thetaStart;
            while (index < nTheta) {
                 thetaArray[index++] = thetaValue;
                 thetaValue += thetaStep;
            }
        } else {
            int nTheta1 = (int) (Math.abs(thetaEnd - (-Math.PI / 2)
                ) / thetaStep) + 1;
            int nTheta2 = (int) (Math.abs(Math.PI / 2 - thetaStart)
                 / thetaStep) + 1;
            if ((nTheta1 - 1) * thetaStep + (-Math.PI / 2) <
                thetaEnd && nTheta1 * thetaStep + (-Math.PI / 2) <
                Math.PI / 2) {
                 nTheta1++;
```

```
            }
            nTheta = nTheta2 + nTheta1;
            thetaArray = new double[nTheta];
            index = 0;
            double thetaValue = -Math.PI / 2;
            while (index < nTheta1) {
                thetaArray[index++] = thetaValue;
                thetaValue += thetaStep;
            }
            thetaValue = thetaStart;
            while (index < nTheta) {
                thetaArray[index++] = thetaValue;
                thetaValue += thetaStep;
            }
        }
```

Zdrojový kód 3 slúži na diskretizáciu hodnôt maximálnej možnej kolmej vzdialenosti priamok od počiatku súradnicovej sústavy a hodnôt polárneho uhla. Pri polárnom uhle sa v tomto algoritme rozpoznávania vykoná iba časť kódu po splnení podmienky `(thetaStart <= thetaEnd)`. Podmienka `((nTheta1 - 1) * thetaStep + (-Math.PI / 2) < thetaEnd && nTheta1 * thetaStep + (-Math.PI / 2) < Math.PI / 2)` môže byť splnená iba pri druhom algoritme rozpoznávania.

Zdroj. kód 4 Funkcia `HoughLine2D`, obsahujúca implementáciu Houghovej transformácie pre priamky v 2D priestore (2. časť).

```
        //dvojrozmerná inkrementačná matica
        short[][] A = new short[nTheta][nRo];
        double lowerDistance = roArray[0] - 1.1 * size;
        double upperDistance = roArray[nRo - 1] + 1.1 * size;

        //výpočet kolmých vzdialeností priamky
        //od počiatku súradnicovej sústavy
        for (Double[] row : data) {
            for (int thetaIndex = 0; thetaIndex < nTheta;
```

```
                thetaIndex++) {
                double distance = row[indexFirstDim] * Math.cos(
                    thetaArray[thetaIndex])
                            + (row[indexSecondDim]) * Math.sin(
                                thetaArray[thetaIndex]);
                //vypočíta sa horný a dolný index pre hodnoty
                //kolmých vzdialeností priamky
                //od počiatku súradnicovej sústavy, vzhladom na
                    maximálnu povolenú kolmú vzdialenost údajov
                //od detegovanej priamky a vzorkovaciu frekvenciu
                    pola kolmých vzdialeností
                if (distance > lowerDistance && distance <
                    upperDistance) {
                    int lowerIndex = (int) Math.round((distance -
                        size - roArray[0]) / roStep);
                    int upperIndex = (int) Math.round((distance +
                        size - roArray[0]) / roStep);
                    //indexy musia byt v povolenom rozsahu
                    if (lowerIndex < 0) {
                        lowerIndex = 0;
                    }
                    if (upperIndex > nRo - 1) {
                        upperIndex = nRo - 1;
                    }
                    //inkrementácia polí priamok, v ktorých sa údaj
                        vyskytuje
                    for (int roIndex = lowerIndex; roIndex <
                        upperIndex + 1; roIndex++) {
                        A[thetaIndex][roIndex] += row[
                            totalCountIndex];
                    }
                }
            }
        }
```

V zdrojovom kóde 4 prebieha výpočet kolmej vzdialenosti a inkrementácia polí akumulátora. V podstatnej miere sa tu určuje výpočtová zložitosť algoritmu rozpoznávania. Ako je vidieť, počet operácií a pamäťová zložitosť závisia od dĺžky polí hodnôt kolmej vzdialenosti a polárneho uhla. Tie sa vypočítavajú na základe použitých rozsahov a krokov diskretizácie. Počet polí akumulátora je tak $nRo{\cdot}nTheta$. Ak sa vezme do úvahy, že v algoritme bol použitý rozsah polárneho uhla $[-\frac{\pi}{2}, \frac{\pi}{2}]$, tak to môžme upresniť na $(ceil(\frac{2{\cdot}roMax}{roStep}) + 1){\cdot}(floor(\frac{\pi}{thetaStep}) + 1)$. Ohraničenie vzhľadom na kolmú vzdialenosť možné nie je, keďže $roMax$ závisí od rozsahov údajov jednotlivých udalostí. Pre minimalizáciu dĺžky poľa hodnôt kolmej vzdialenosti je ale vhodné posunutím vycentrovať údaje vzhľadom na počiatok súradnicovej sústavy, a tým minimalizovať hodnotu maximálnej absolútnej kolmej vzdialenosti priamok od počiatku súradnicovej sústavy. Ako údajový typ bol pre polia akumulátora zvolený short, ktorý najlepšie odpovedal rozsahu hodnôt, ktoré polia nadobúdali. Pre počet operácií, ak berieme do úvahy iba výpočet kolmej vzdialenosti a konštantný počet inkrementácií pre každý údaj, platí, že sa vykoná $n{\cdot}nTheta{\cdot}2$ operácií násobenia a $n{\cdot}nTheta{\cdot}(1 + 2{\cdot}size/roStep)$ operácií sčítania, kde n je počet údajov, $size$ je daná maximálna kolmá vzdialenosť údajov detegovanej priamky a $2{\cdot}size/roStep$ je počet inkrementácií polí akumulátora. Táto časť kódu sa v algoritme rozpoznávania vykoná celkovo trikrát. Počet operácií tak vieme v prvom rade ovplyvniť zmenou nastavenia hodnoty kroku diskretizácie parametrického priestoru uhla θ. Ak tento krok zväčšíme na dvojnásobok, počet operácií, závislý od tohto parametera, sa zníži na polovicu. Rovnako to ovplyvní aj pamäťové nároky.

Zdroj. kód 5 Funkcia `HoughLine2D`, obsahujúca implementáciu Houghovej transformácie pre priamky v 2D priestore (3. časť).

```
        Set<Double[]> resultData = new HashSet<>();
        List<Double[]> resultRanges = new ArrayList<>();

        short max = 0;
        //zoznam indexov (fiIndex, roIndex) odpovedajúcich maximu
            inkrementačnej matice
```

```
        List<Integer[]> indexes = new ArrayList<>();

        for (int thetaIndex = 0; thetaIndex < nTheta; thetaIndex++)
            {
            for (int roIndex = 0; roIndex < nRo; roIndex++) {
                if (A[thetaIndex][roIndex] == max) {
                    indexes.add(new Integer[]{thetaIndex, roIndex})
                        ;
                } else if (A[thetaIndex][roIndex] > max) {
                    indexes.clear();
                    max = A[thetaIndex][roIndex];
                    indexes.add(new Integer[]{thetaIndex, roIndex})
                        ;
                }
            }
        }

        int maxRo = indexes.get(0)[roIndexL];
        int minRo = indexes.get(0)[roIndexL];
        //získavanie výsledných údajov a popisov priamok,
        //odpovedajúcich maximu inkrementačnej matice
        for (Integer[] indexRow : indexes) {
            if (indexRow[roIndexL] > maxRo) {
                maxRo = indexRow[roIndexL];
            } else if (indexRow[roIndexL] < minRo) {
                minRo = indexRow[roIndexL];
            }
            for (Double[] row : data) {
                double distance = row[indexFirstDim] * Math.cos(
                    thetaArray[indexRow[thetaIndexL]])
                        + row[indexSecondDim] * Math.sin(thetaArray
                            [indexRow[thetaIndexL]]);
                if (distance > lowerDistance && distance <
                    upperDistance) {
                    int lowerIndex = (int) Math.round((distance -
```

```
                    size - roArray[0]) / roStep);
                int upperIndex = (int) Math.round((distance +
                    size - roArray[0]) / roStep);
                if (lowerIndex <= indexRow[roIndexL] &&
                    upperIndex >= indexRow[roIndexL]) {
                    resultData.add(row);
                }
            }
        }
    }
```

V zdrojovom kóde 5 sa zisťuje maximálna hodnota akumulátora, indexy parametrov priamok s maximálnou hodnotou a údaje k nim prislúchajúce.

Zdroj. kód 6 Funkcia `HoughLine2D`, obsahujúca implementáciu Houghovej transformácie pre priamky v 2D priestore (4. časť).

```
        //výstupné hodnoty
        List<Double[]>[] result = new ArrayList[2];

        //zápis rozsahov parametrického priestoru, definovaných na
            základe výstupných údajov
        if (Math.abs(thetaArray[indexes.get(indexes.size() - 1)[
            thetaIndexL]] - thetaArray[indexes.get(0)[thetaIndexL]])
             < Math.PI / 2) {
             resultRanges.add(new Double[]{thetaArray[indexes.get(0)
                [thetaIndexL]], thetaArray[indexes.get(indexes.size
                () - 1)[thetaIndexL]]});
        } else {
             resultRanges.add(new Double[]{thetaArray[indexes.get(
                indexes.size() - 1)[thetaIndexL]], thetaArray[
                indexes.get(0)[thetaIndexL]]});
        }
        resultRanges.add(new Double[]{roArray[minRo], roArray[maxRo
            ]});
```

```
        result[dataIndex] = new ArrayList<>(resultData);
        result[rangesIndex] = resultRanges;

        return result;
    }
```

Záverečná časť zdrojového kódu 6 je, okrem výstupu údajov, podstatná len pre druhý algoritmus.

Okrem funkcií pre Houghove transformácie sa v implementácii použili:

- Funkcia `getCenter3D` (Zdrojový kód 7), ktorá vracia vektor pozície centra údajov v priestore XYGtu.
- Funkcia `shift` (Zdrojový kód 8), určená pre posunutie údajov v priestore XYGtu o zadaný vektor.
- Funkcia `getAbsMax` (Zdrojový kód 9), určená pre výpočet absolútneho maxima údajov v zadanej dimenzii.
- Funkcia `getMaxDist` (Zdrojový kód 10), určená pre výpočet maximálnej absolútnej kolmej vzdialenosti priamky od počiatku súradnicovej sústavy v 2D priestore.

Zdroj. kód 7 Funkcia `getCenter3D`, ktorá vracia vektor pozície centra údajov v priestore XYGtu.

```
    /**
     *
     * @param data - údaje
     * @return - vektor so začiatkom v počiatku súradnicovej
     * sústavy a koncom v centre údajov 3D priestoru XYGtu
     */
    private double[] getCenter3D(List<Double[]> data) {
        double maxX = data.get(0)[xIndex];
        double minX = data.get(0)[xIndex];
```

```
        double maxY = data.get(0)[yIndex];
        double minY = data.get(0)[yIndex];
        double maxGtu = data.get(0)[gtuIndex];
        double minGtu = data.get(0)[gtuIndex];
        for (int i = 1; i < data.size(); i++) {
            if (maxX < data.get(i)[xIndex]) {
                maxX = data.get(i)[xIndex];
            } else if (minX > data.get(i)[xIndex]) {
                minX = data.get(i)[xIndex];
            }
            if (maxY < data.get(i)[yIndex]) {
                maxY = data.get(i)[yIndex];
            } else if (minY > data.get(i)[yIndex]) {
                minY = data.get(i)[yIndex];
            }
            if (maxGtu < data.get(i)[gtuIndex]) {
                maxGtu = data.get(i)[gtuIndex];
            } else if (minGtu > data.get(i)[gtuIndex]) {
                minGtu = data.get(i)[gtuIndex];
            }
        }
        return new double[]{(minX + maxX) / 2, (minY + maxY) / 2, (
            minGtu + maxGtu) / 2};
    }
```

Zdroj. kód 8 Funkcia `shift`, určená pre posunutie údajov v priestore XYGtu o zadaný vektor.

```
    /**
     *
     * posun v 3D priestore XYgtu
     *
     * @param data - údaje
     * @param x - x-ová zložka vektora posunutia
     * @param y - y-ova zložka vektora posunutia
     * @param gtu - časová zložka vektora posunutia
     */
```

```
    private void shift(List<Double[]> data, double x, double y,
        double gtu) {
         for (Double[] row : data) {
             row[xIndex] = row[xIndex] + x;
             row[yIndex] = row[yIndex] + y;
             row[gtuIndex] = row[gtuIndex] + gtu;
         }
    }
```

Zdroj. kód 9 Funkcia `getAbsMax`, určená pre posunutie údajov v priestore XYGtu o zadaný vektor.

```
     /**
     *
     * @param data - údaje
     * @param index - index dimenzie priestoru
     * @return - maximálna absolútna vzdialenost údajov od počiatku
     * súradnicovej sústavy v jednorozmernom priestore
     */
    private double getAbsMax(List<Double[]> data, Integer index) {
         double max = Math.abs(data.get(0)[index]);
         for (int i = 1; i < data.size(); i++) {
             double value = Math.abs(data.get(i)[index]);
             if (max < value) {
                 max = value;
             }
         }
         return max;
    }
```

Zdroj. kód 10 Funkcia `getMaxDist`, určená pre posunutie údajov v priestore XYGtu o zadaný vektor.

```
    /**
     *
     * @param data - údaje
```

```
     * @param xIndex - index prvej dimenzie
     * @param yIndex - index druhej dimenzie
     * @return - maximálna absolútna vzdialenost od počiatku
     * súradnicovej sústavy v 2D priestore
     */
    private double getMaxDist(List<Double[]> data, int xIndex, int
        yIndex) {
        double xMax = getAbsMax(data, xIndex);
        double yMax = getAbsMax(data, yIndex);
        return Math.sqrt(Math.pow(xMax, 2) + Math.pow(yMax, 2));
    }
```

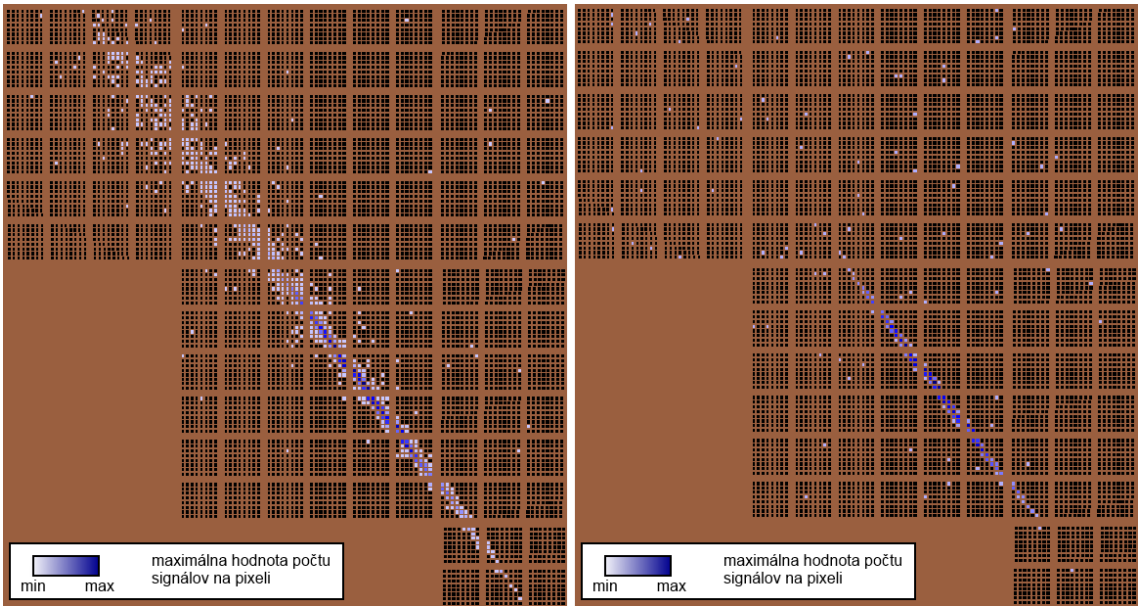
Vhodným použitím funkcií `getCenter3D` a `shift` je možné minimalizovať priestor kolmých vzdialeností priamky od počiatku súradnicovej sústavy a tým znížiť pamäťové nároky následne volanej metódy Houghovej transformácie (Zdrojový kód 11).

Zdroj. kód 11 Minimalizácia pamäťových nárokov.

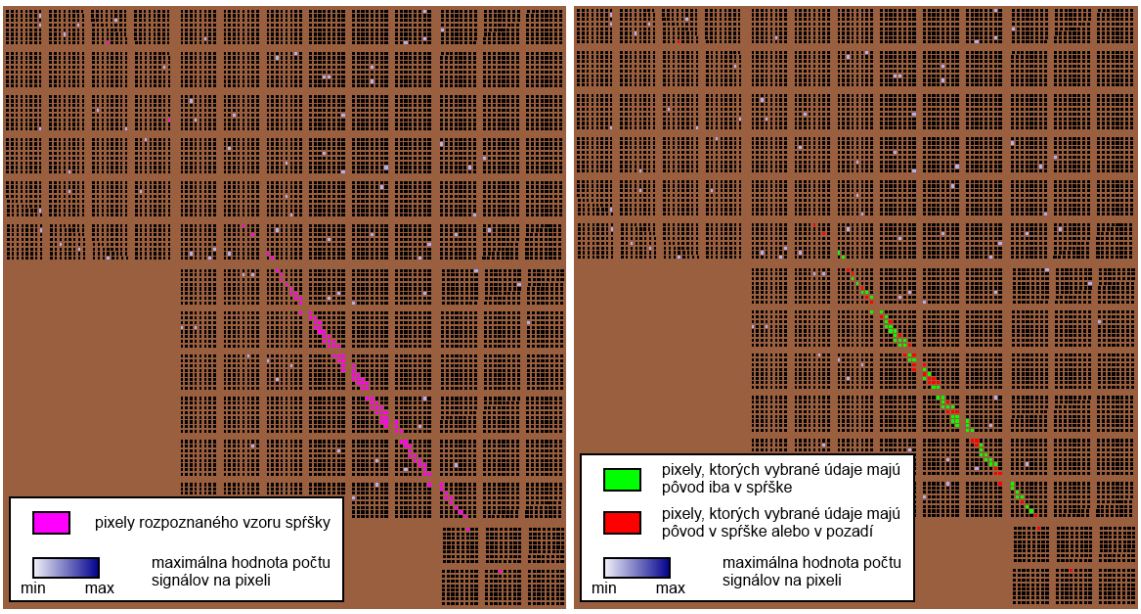
```
        //vypočíta sa posunutie údajov pre minimalizovanie
        //pamätového zaťaženia pri metóde Houghovej transformácie
        double[] vectorXYGtu = getCenter3D(allData);
        //posunutie údajov v priestore XYGtu
        shift(allData, -vectorXYGtu[0], -vectorXYGtu[1], -
            vectorXYGtu[2]);
```

Aplikácia implementácie algoritmu pre rozpoznávanie spŕšok na údajoch zvolenej udalosti je vizualizovaná v priestore XY a zobrazená na Obrázku 5 – 15 a Obrázku 5 – 16. Na Obrázku 5 – 15 vľavo sú zobrazené len pixely údajov s pôvodom v spŕške so 75 stupňovým zenitovým uhlom. Farba pixelov odpovedá maximálnym hodnotám počtu signálov na jednotlivých pixeloch. Na Obrázku 5 – 15 vpravo sú zobrazené pixely údajov s pôvodom v spŕške a pozadí po nastavení prahovej hodnoty počtu signálov pre rozpoznanie vzoru spŕšky. Na Obrázku 5 – 16 vľavo je vzor spŕšky získaný na základe výstupných údajov metódy Houghovej transformácie pre priamky

v 2D priestore.



Obr. 5 – 15 Vľavo: Zobrazenie spŕšky v priestore XY. Vpravo: Zobrazenie údajov udalosti, nad definovanou prahovou hodnotou počtu signálov, v priestore XY.



Obr. 5 – 16 Vľavo: Zobrazenie pixelov vzoru spŕšky. Vpravo: Zobrazenie údajov výsledného vzoru spŕšky v priestore XY.

Po nastavení prahovej hodnoty počtu signálov pre rozpoznanie údajov a aplikácii

metódy Houghovej transformácie pre priamky v 3D priestore bol získaný výsledný vzor spŕšky. Metóde Houghovej transformácie pre priamky v 3D priestore bolo posunutých 251 údajov s pôvodom v spŕške a 2083 údajov s pôvodom v pozadí. Výstup tejto metódy sa skladal z 228 údajov s pôvodom v spŕške a 43 údajov s pôvodom v pozadí. Pixely prislúchajúce k výsledným údajom sú zobrazené na Obrázku 5 – 16 vpravo. V prípade, že údaje pixela majú pôvod iba v spŕške, je tento pixel označený zelenou farbou, v opačnom prípade červenou.

5.3.4 Výsledky a analýza

Implementácia algoritmu bola aplikovaná na simulované údaje udalostí, pri UV pozadí 0.42 pe/(px μs), a hľadalo sa najlepšie nastavenie parametrov algoritmu z hľadiska štatistiky γ_{68} pre separačný uhol k skutočnému zenitovému uhlu spŕšky. Ako východzí zenitový uhol sa vzal uhol o približne 30 stupňoch a výsledky štatistiky na simulovaných údajoch udalostí s týmto uhlom je vidieť v Tabuľke 5 – 1, v Tabuľke 5 – 2 a v Tabuľke 5 – 3. Najlepší dosiahnutý výsledok pre separačný uhol, zobrazený v Tabuľke 5 – 2, bol 2.68 stupňa a to pri nastavení:

- Maximálna kolmá vzdialenosť od priamky (XGtu, YGtu) - 2.
- Maximálna kolmá vzdialenosť od priamky (XY) - 4.
- Prah počtu signálov pre vzor - 5.
- Prah počtu signálov pre údaje - 5.

Pre toto nastavenie parametrov algoritmu sa na simulovaných údajoch udalostí vykonalo porovnanie štatistiky γ_{68}, pre separačný uhol k skutočnému zenitovému uhlu spŕšky, s algoritmom PWISE. Ako ukazuje porovnanie na Obrázku 5 – 17, boli dosiahnuté lepšie (spŕšky so zenitovým uhlom o 30 až 65 stupňoch) alebo porovnateľné (spŕšky so zenitovým uhlom o 70 až 75 stupňoch) výsledky, pri väčšom počte zrekonštruovaných udalostí (Obrázok 5 – 18 a Obrázok 5 – 19). Taktiež bola overená

výkonnosť algoritmu pri narastajúcom UV pozadí pre simulované údaje udalostí, pre zenitový uhol 30, a jej výsledky sú zobrazené na Obrázku 5 – 20. Ukázalo sa, že síce boli výsledky pri rôznych UV pozadiach vcelku udržateľné, bolo to za cenu prudko klesajúceho počtu zrekonštruovaných udalostí, ako ukazuje Obrázok 5 – 21 a Obrázok 5 – 22.
Na základe analýzy výsledkov a správania sa algoritmu pri rôznych podmienkach boli formulované nasledujúce závery:

Rozpoznanie vzoru spŕšky

- Rozpoznanie vzoru spŕšky musí prebiehať pri dostatočne vysokej prahovej hodnote počtu signálov pre rozpoznávanie vzoru.
- S narastajúcou prahovou hodnotou počtu signálov pre rozpoznávanie vzoru obsahuje rozpoznaný vzor menej údajov z pozadia. Rovnako tak ale obsahuje aj menej údajov s pôvodom v spŕške, čo môže viesť k limitovaniu možností údajovej analýzy (Obrázok 5 – 30).

Rozpoznanie údajov spŕšky

- Prahová hodnota počtu signálov pre rozpoznávanie údajov bola pri dosiahnutí najlepších štatistických výsledkov na úrovni prahovej hodnoty počtu signálov pre rozpoznávanie vzoru. Pravdepodobne to pri porovnaní s algoritmom PWISE môže za slabšie výsledky pri udalostiach so zenitovým uhlom 65 až 75 stupňov, a pri overovaní výkonnosti algoritmu pri narastajúcom UV pozadí za možno až príliš prudký pokles počtu zrekonštruovaných udalostí.

Tabuľka 5 – 1 Výsledné hodnoty separačného uhla (v stupňoch) štatistiky γ_{68} pre jednotlivé nastavenia parametrov algoritmu (Maximálna kolmá vzdialenosť od priamky (XGtu, YGtu) - 1.5)

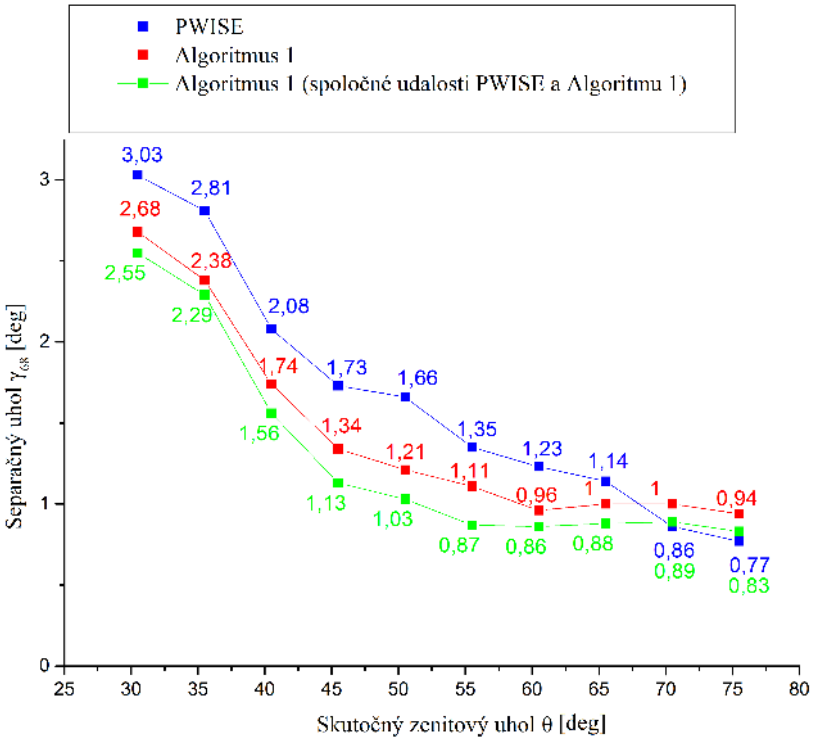
Maximálna kolmá vzdialenosť od priamky (XGtu, YGtu) - 1.5									
Maximálna kolmá vzdialenosť od priamky (XY) - 4									
* P. p. s. pre údaje / * P. p. s. pre vzor	0	1	2	3	4	5	6	7	8
5	5.45	4.24	3.43	2.99	2.77	2.77	2.77	2.77	2.77
6	5.34	4.2	3.56	3.09	2.88	2.77	2.75	2.75	2.75
* P. p. s. - Prah počtu signálov									

Tabuľka 5 – 2 Výsledné hodnoty separačného uhla (v stupňoch) štatistiky γ_{68} pre jednotlivé nastavenia parametrov algoritmu (Maximálna kolmá vzdialenosť od priamky (XGtu, YGtu) - 2)

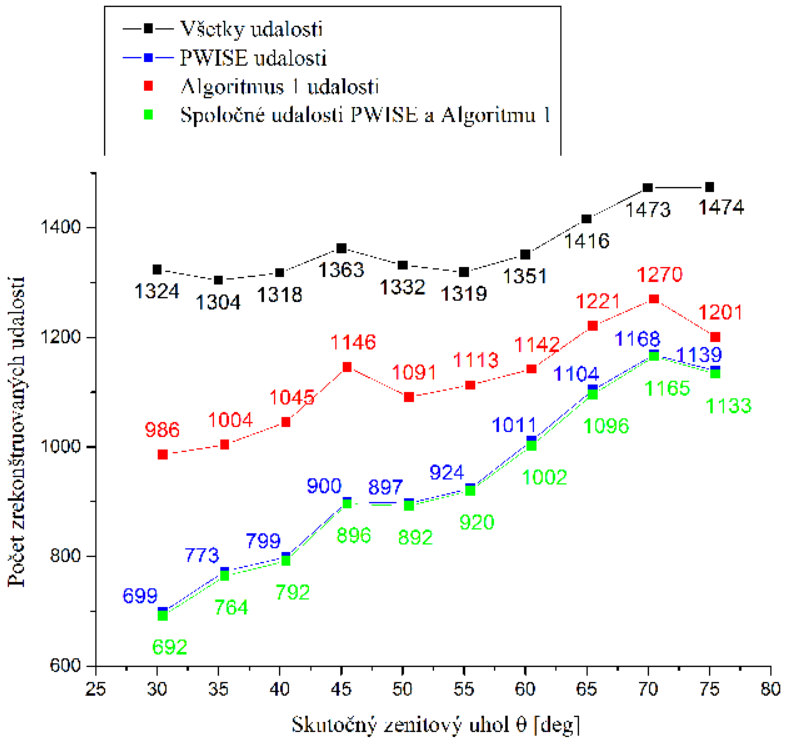
Maximálna kolmá vzdialenosť od priamky (XGtu, YGtu) - 2									
Maximálna kolmá vzdialenosť od priamky (XY) - 4									
* P. p. s. pre údaje / * P. p. s. pre vzor	0	1	2	3	4	5	6	7	8
5	5.6	4.17	3.35	2.95	2.74	2.68	2.68	2.68	2.68
6	5.42	4.24	3.45	3	2.8	2.75	2.75	2.75	2.75
* P. p. s. - Prah počtu signálov									

Tabuľka 5 – 3 Výsledné hodnoty separačného uhla (v stupňoch) štatistiky γ_{68} pre jednotlivé nastavenia parametrov algoritmu (Maximálna kolmá vzdialenosť od priamky (XGtu, YGtu) - 2.5)

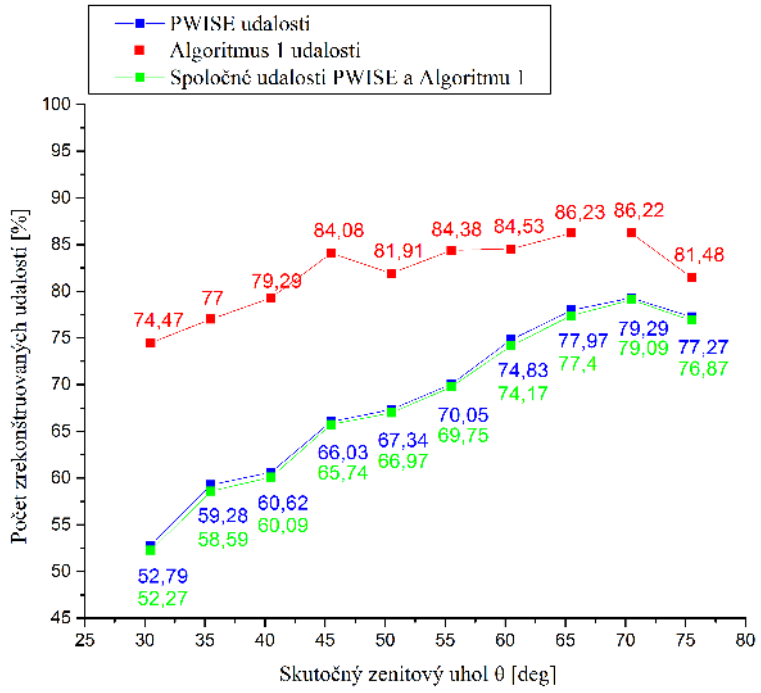
Maximálna kolmá vzdialenosť od priamky (XGtu, YGtu) - 2.5									
Maximálna kolmá vzdialenosť od priamky (XY) - 4									
* P. p. s. pre údaje / * P. p. s. pre vzor	0	1	2	3	4	5	6	7	8
5	6.31	4.79	3.68	3.2	2.85	2.81	2.81	2.81	2.81
6	6.1	4.85	3.81	3.18	2.93	2.77	2.8	2.8	2.8
* P. p. s. - Prah počtu signálov									



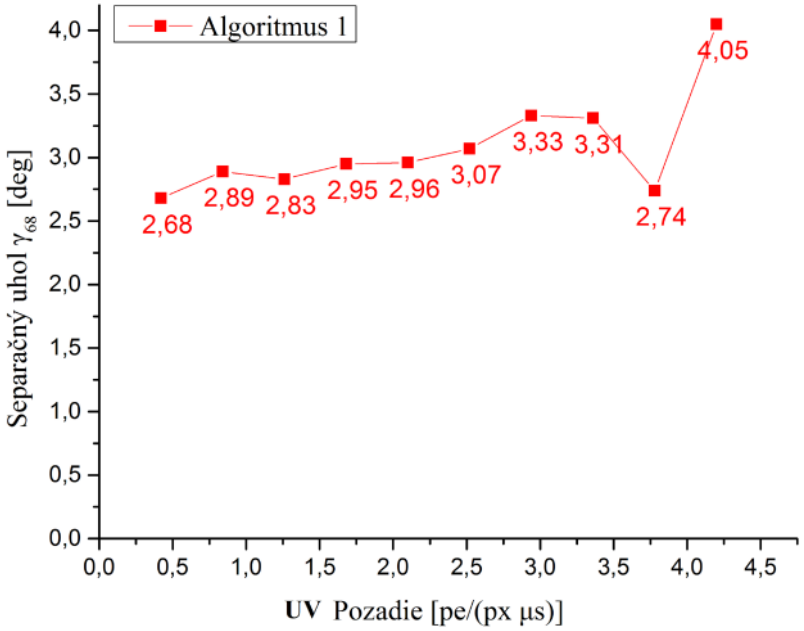
Obr. 5 – 17 Graf Skutočný zenitový uhol - Separačný uhol γ_{68} (Algoritmus 1).



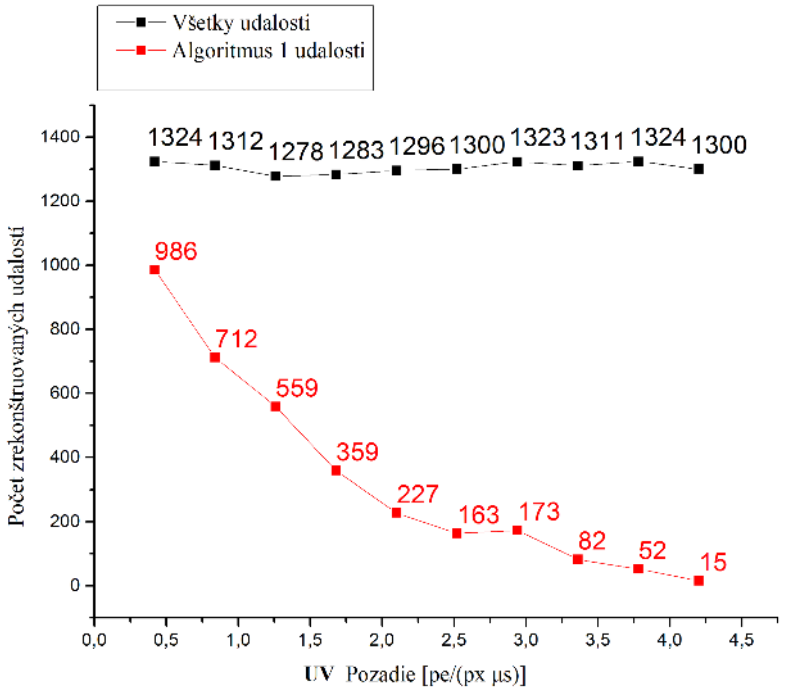
Obr. 5 – 18 Graf Skutočný zenitový uhol - Počet zrekonštruovaných udalostí (Algoritmus 1).



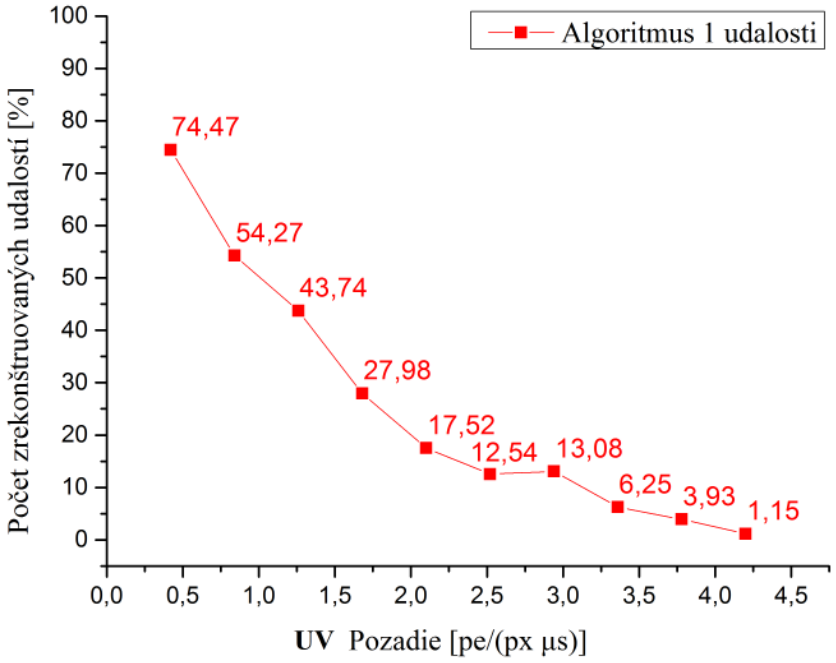
Obr. 5 – 19 Graf Skutočný zenitový uhol - Počet zrekonštruovaných udalostí [%] (Algoritmus 1).



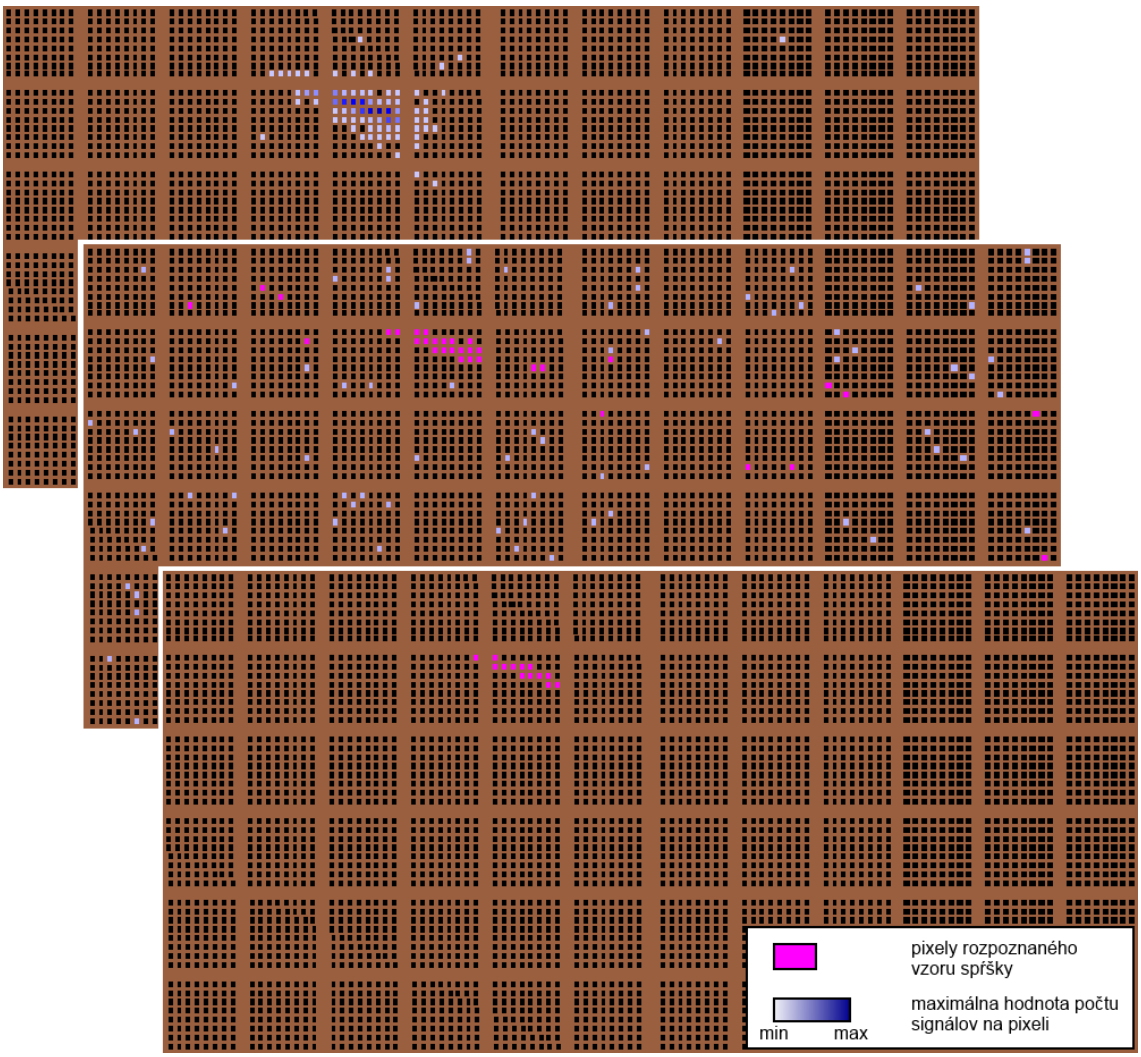
Obr. 5 – 20 Graf UV pozadie - Separačný uhol γ_{68} (Algoritmus 1, zenitový uhol 30 stupňov).



Obr. 5 – 21 Graf UV pozadie - Počet zrekonštruovaných udalostí (Algoritmus 1, zenitový uhol 30 stupňov).



Obr. 5 – 22 Graf UV pozadie - Počet zrekonštruovaných udalostí [%] (Algoritmus 1, zenitový uhol 30 stupňov).



Obr. 5 – 23 Vývoj nájdeného vzoru spŕšky s rastúcou prahovou hodnotou počtu signálov.

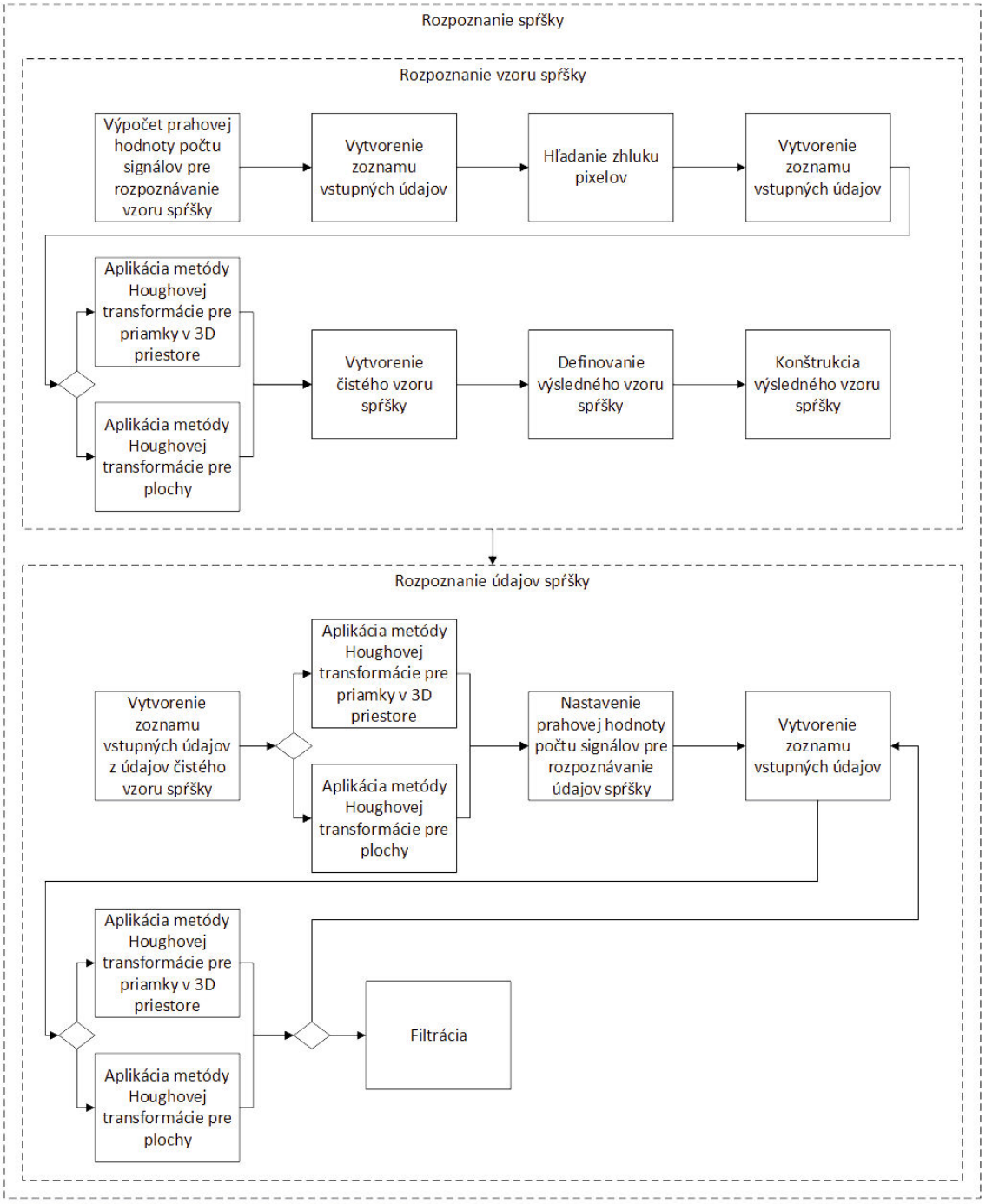
5.4 Algoritmus 2

Návrh druhého algoritmu pre rozpoznávanie spŕšok vznikal na základe analýzy prvého algoritmu. Mal za cieľ nielen v čo najväčšej miere potlačiť nedostatky prvého algoritmu, ale tiež overiť použiteľnosť nových nápadov pre zlepšenie efektivity rozpoznávania a pre dosiahnutie podstatne lepších výsledkov uhlovej rekonštrukcie. Okrem Houghových transformácií pre priamky v 2D a 3D priestore sa naviac použila Houghova transformácia pre roviny. V porovnaní s prvým algoritmom tiež prebehol pokus o dodatočnú filtráciu údajov na základe skupiny zadefinovaných jednoduchých pravidiel pre postupnosti údajov na pixeli.

5.4.1 Návrh algoritmu

Návrh druhého algoritmu je možné vidieť na Obrázku 5 – 24. Je dobré si všimnúť rozhodovania o použití metódy Houghovej transformácie pre priamky v 3D priestore a metódy Houghovej transformácie pre roviny. V tomto algoritme sa všetky riadia tým istým jedným rozhodnutím, ktoré určí, ktorá metóda sa využije. Pre ďalší vývoj algoritmu je ale možné použitie jednej metódy pri rozpoznávaní vzoru a druhej pri rozpoznávaní údajov spŕšky, v prípade, že sa potvrdí ich lepšia efektivita. V porovnaní s prvým algoritmom došlo k výraznejšiemu konceptuálnemu posunu pri návrhu algoritmu. Pri rozpoznávaní vzoru sa už nevyužíva nastavenie prahovej hodnoty počtu signálov a aplikácia metódy Houghovej transformácie pre priamky v 2D priestore. Namiesto toho sa prešlo k myšlienke hľadania zhluku pixelov, ktorý by indikoval spŕšku. Taktiež sa rozšírili možnosti definovania tvaru vzoru spŕšky. Pri rozpoznávaní údajov sa zase prešlo k postupnému spracovávaniu údajov vzoru spŕšky metódou Houghovej transformácie, čo zabezpečuje postupnú závislosť výsledného vzoru spŕšky na údajoch, od tých s najvyššou hodnotou počtu signálov až po tie s najnižšou, a teda aj väčšiu presnosť detekcie. Podstatným vylepšením je pri metódach Houghovej transformácie aj možnosť zadať hodnotu veľkosti kol-

mej vzdialenosti prostredníctvom hodnoty veľkosti časovej zložky vektora kolmej vzdialenosti.



Obr. 5 – 24 Návrh druhého algoritmu rozpoznávania spŕšok.

Ako voliteľná súčasť rozpoznávania údajov boli definované jednoduché filtračné pravidlá pre postupnosti údajov na pixeli, pri ktorých sa predpokladalo, že by mohli nájsť svoje uplatnenie.

Detailnejší popis krokov algoritmu:

Rozpoznanie vzoru spŕšky

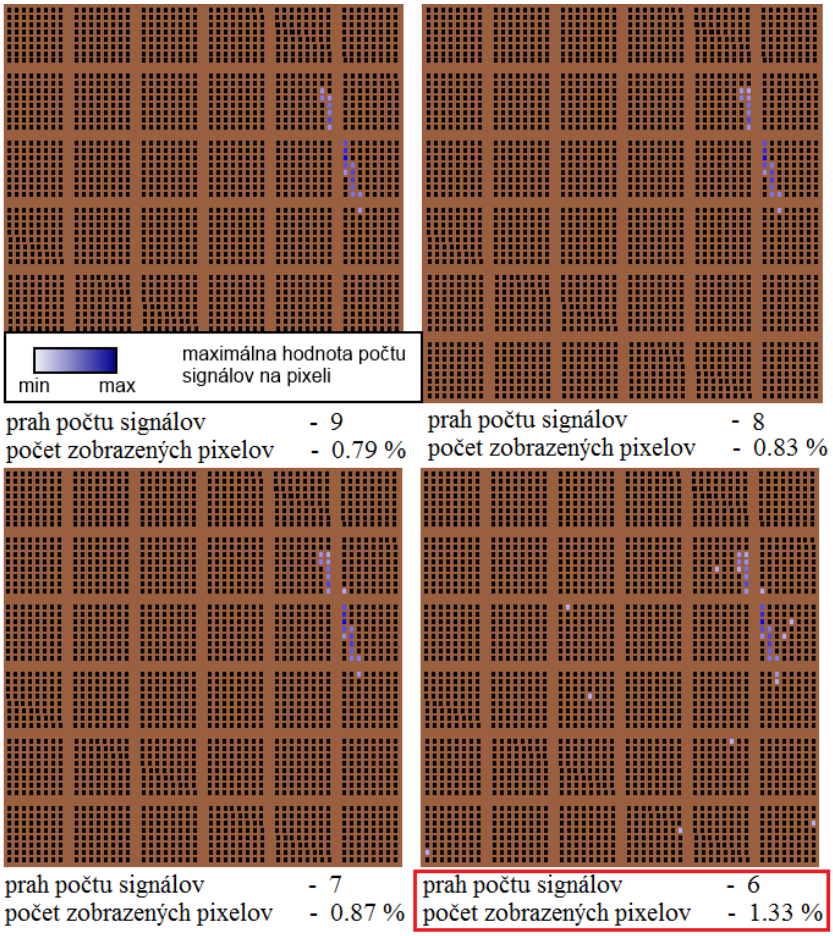
1. Nájdenie zhluku údajov, ktorý by reprezentoval časť obrazu spŕšky v priestore XY. Postup nájdenia tohto zhluku:

   I. Vyberú sa údaje s maximálnou hodnotou počtu signálov na danom pixeli. Tieto údaje sa následne rozdelia do niekoľkých skupín podľa hodnoty prahu počtu signálov, napr. sa určí, že budú 3 skupiny, v prvej skupine, s prahom počtu signálov 0, budú údaje s hodnotou počtu signálov 1, v druhej skupine, s prahom počtu signálov 1, údaje s hodnotou počtu signálov 2 a v tretej skupine, s prahom počtu signálov 2, údaje s hodnotou počtu signálov nad 2.

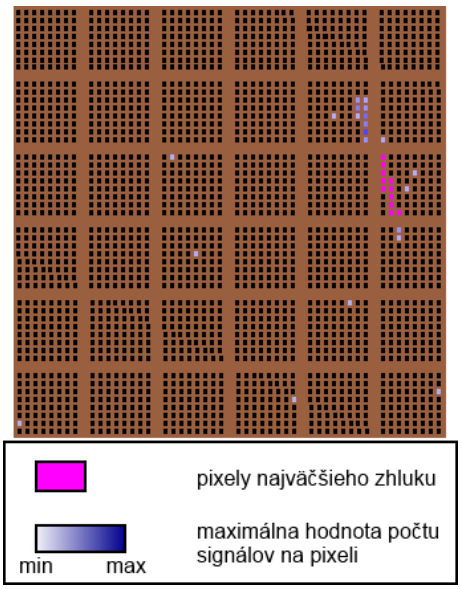
   II. Určí sa percentuálna hodnota počtu pixelov, v ktorých sa bude hľadať zhluk.

   III. Spracúvajú sa skupiny údajov, od tých s najvyššou hodnotou prahu počtu signálov, až po tie s najnižšou. Vezme sa skupina, vezmú sa jej údaje, skontroluje sa, či počet všetkých vybraných údajov prekročil predtým zadanú percentuálnu hodnotu počtu pixelov, v ktorých sa bude hľadať zhluk, ak áno, prejde sa k hľadaniu zhluku, ak nie, tak sa vezme ďalšia skupina v poradí, pridajú sa jej údaje a opäť sa prejde ku kontrole. Skupiny sa pridávajú až do chvíle kým nebude splnená podmienka alebo nedôjdu skupiny. Príklad tohto procesu, ktorého význam odpovedá znižovaniu prahovej hodnoty počtu signálov a tým zvyšovaniu počtu zobrazených pixelov a následnej kontrole počtu zobrazených pixelov, je na Obrázku 5 – 25. Po splnení

podmienky sa hľadá zhluk pixelov, na základe zadanej maximálnej vzdialenosti údajov od zhluku, v priestore XY. Zhluk pixelov sa vyberá na základe hodnoty počtu pixelov a na základe hodnoty sumy počtu signálov údajov zhluku. Touto podmienkou by sa malo zabezpečiť vybratie približne centrálnej časti spŕšky. Po nájdení zhluku sa skontroluje podmienka na minimálny počet pixelov zhluku, ak zhluk neobsahuje dostatočný počet pixelov, tak sa inkrementuje percentuálna hodnota počtu pixelov, v ktorých sa hľadá zhluk, a opäť sa pokračuje pridávaním skupín údajov a hľadaním zhluku. Príklad nájdenia najväčšieho zhluku pixelov je na Obrázku 5 – 26.



Obr. 5 – 25 Zobrazenie výpočtu prahovej hodnoty počtu signálov pre vzor, pri prahovej hodnote 1% zobrazených pixelov.



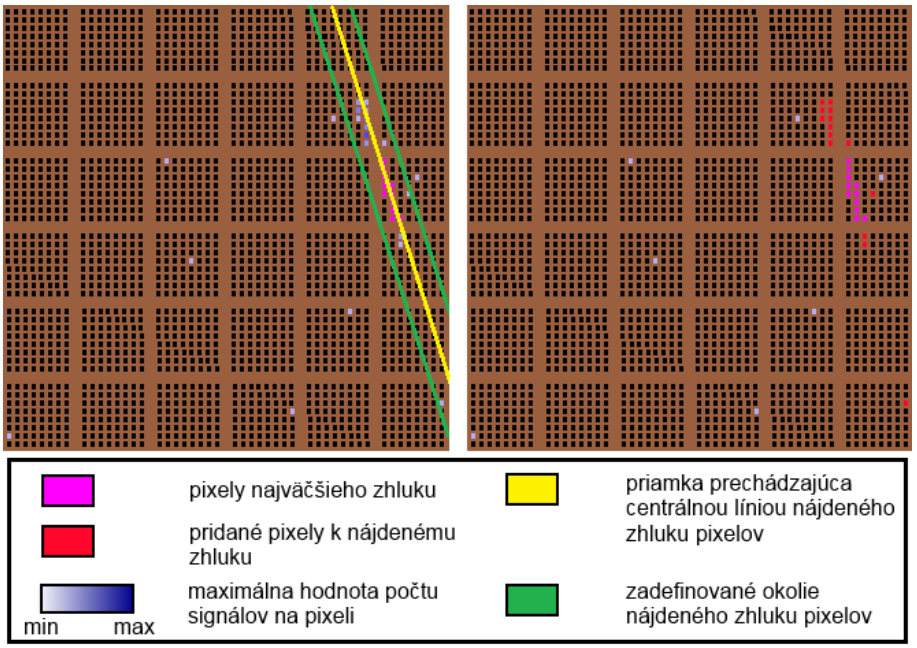
Obr. 5 – 26 Zobrazenie najväčšieho nájdeného zhluku pixelov.

2. Nájdenie zvyšku obrazu spŕšky v priestore XY. Postup:

   I. K dispozícii máme údaje, v ktorých sa našla časť spŕšky. Z týchto údajov sa pridajú k údajom časti spŕšky tie, ktoré sú v zadefinovanom okolí časti spŕšky v priestore XY. V implementácii tohto algoritmu sa okolie zadefinovalo na základe maximálnej povolenej x-ovej a y-ovej vzdialenosti údajov od časti spŕšky a na základe kolmej vzdialenosti údajov od priamky, ktorá prechádza pozdĺž nájdenej časti spŕšky. Táto priamka bola získaná aplikáciou metódy Houghovej transformácie pre priamky v priestore XY na údaje časti spŕšky. Podmienkou na okolie je, aby sa v ňom nachádzal kompletný obraz spŕšky a zároveň čo najmenší počet údajov, ktoré k spŕške nepatria. Príklad vytvoreného okolia je zobrazený na Obrázku 5 – 27.

   II. Na vybrané údaje sa aplikuje, podľa voľby, metóda Houghovej transformácie pre priamky v 3D priestore alebo metóda Houghovej transformácie pre roviny, čím sa zabezpečí, že budeme ďalej pracovať s údajmi lineárnymi v priestore XYGtu. Aby výstup obsahoval údaje

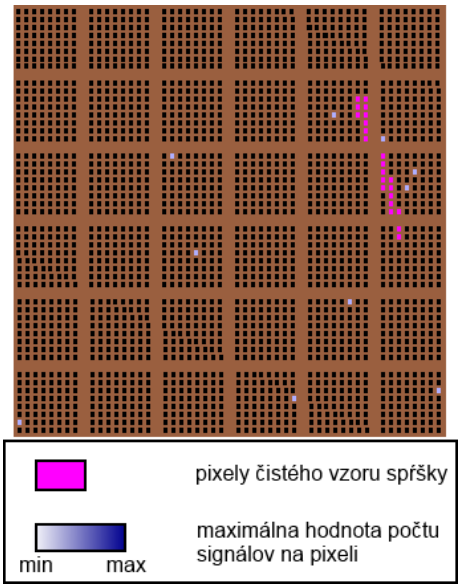
nájdenej časti spŕšky a zároveň aby sa podľa týchto údajov určoval lineárny priebeh, je nutné pred aplikáciou metódy Houghovej transformácie dostatočne zvýšiť hodnoty počtu signálov údajov časti spŕšky. Po skončení metódy Houghovej transformácie sa týmto údajov priradia pôvodné hodnoty počtu signálov. Je potrebné si dať pozor na prehnané zvýšenie hodnôt počtu signálov, mohlo by to viesť k neželanému výstupu metódy Houghovej transformácie. Výstupné údaje by mali obsahovať kompletný obraz spŕšky v priestore XY a k tomu treba počítať aj s nejakými nadbytočnými údajmi.



Obr. 5 – 27 Zobrazenie vytvoreného okolia okolo najväčšieho nájdeného zhluku pixelov.

III. Nadbytočné údaje v obraze spŕšky v priestore XY sa odstraňujú v dvoch krokoch. Prvý krok odstránenia nadbytočných údajov z obrazu predstavuje nájdenie zhluku údajov na základe hodnoty maximálnej povolenej vzdialenosti údajov od zhluku v priestore XY. Toto hľadanie sa líši od toho pri hľadaní časti spŕšky v zadaní hodnoty maximálnej povolenej vzdialenosti, pretože pre nájdenie zhluku,

ktorý by obsahoval kompletný obraz spŕšky, je nutné, aby bola hodnota dostatočne veľká na prekonanie medzier vytvorených mriežkou ohniskovej plochy. Druhý krok odstránenia nadbytočných údajov z obrazu predstavuje odstránenie údajov, v ktorých zadanom okolí sa nenachádza žiadny údaj. Odstránia sa tak vzdialené osamostatnené údaje od zhluku údajov spŕšky. Po ukončení týchto krokov máme k dispozícii údaje tvoriace tzv. čistý vzor spŕšky, na základe ktorého bude v ďalších krokoch prebiehať definovanie a konštrukcia finálneho vzoru spŕšky, a tiež inicializácia metód Houghovej transformácie pre rozpoznanie údajov spŕšky. Príklad čistého vzoru spŕšky, vzniknutého po aplikácii metódy Houghovej transformácie a odstránení nadbytočných pixelov, je zobrazený na Obrázku 5 – 28.



Obr. 5 – 28 Zobrazenie pixelov čistého vzoru spŕšky.

3. Definovanie a konštrukcia vzoru spŕšky. Je možné definovať nasledujúce vzory spŕšky:

   (a) Vzor na základe centrálnej línie a na základe dĺžky čistého vzoru spŕšky. Pomocou Houghovej transformácie pre priamky v priestore

XY sa zistí priamka prechádzajúca pozdĺž čistého vzoru spŕšky. Následne sa vezmú pixely, ktoré sa nachádzajú do zadanej maximálnej povolenej kolmej vzdialenosti od priamky v priestore XY. Tento vzor sa potom oreže na požadovanú dĺžku vzhľadom na dĺžku čistého vzoru spŕšky.

(b) Vzor definovaný čistým vzorom spŕšky a pridanými pixelmi okolo neho do zadanej maximálnej povolenej vzdialenosti.

Na základe týchto dvoch definovaných vzorov je možná konštrukcia:

(a) Skonštruuje sa iba prvý definovaný vzor.

(b) Skonštruuje sa iba druhý definovaný vzor.

(c) Skonštruuje sa vzor, ktorý je kombináciou oboch zadefinovaných vzorov.

Príklad možných vzorov spŕšky je zobrazený na Obrázku 5 – 29.Finálny vzor predstavujú pixely, ktorých údaje budú spracovávané vo fáze rozpoznávania údajov.

Rozpoznanie údajov spŕšky

4. Získanie inicializačných hodnôt rozsahov parametrického priestoru metód Houghovej transformácie. Kvôli čo najnižšiemu výpočtovému a pamäťovému zaťaženiu pri maximalizácii presnosti procesu výpočtov je vhodné obmedziť rozsah parametrického priestoru metód. To sa realizuje aplikáciou metód Houghovej transformácie na údaje čistého vzoru spŕšky. Pri týchto metódach sa nastavia maximálne rozsahy parametrického priestoru a vhodné kroky diskretizácie tohto priestoru, pričom na základe výstupu sa rozsahy parametrického priestoru obmedzia a pri následnom procese výpočtov bude možné nastaviť dostatočne jemné kroky diskretizácie parametrického priestoru pre dosiahnutie požadovanej presnosti. Pri

metóde Houghovej transformácie pre roviny sa inicializačné hodnoty rozsahu priestoru azimutového uhla nezisťujú, pretože je možné ešte pred volaním metódy vypočítať hodnotu tohto uhla. Túto hodnotu získame posunutím uhla θ, získaného aplikáciou metódy Houghovej transformácie pre priamky v priestore XY na údaje čistého vzoru spŕšky, o $\frac{\pi}{2}$.

5. Voľba zadania hodnoty kolmej vzdialenosti od detegovanej priamky alebo roviny. Kolmá vzdialenosť môže byť zadaná priamo alebo prostredníctvom hodnoty veľkosti časovej zložky vektora kolmej vzdialenosti. Umožňujú to získané inicializačné hodnoty rozsahov uhlov v metódach Houghovej transformácie.

6. Vytvorenie skupín údajov skonštruovaného vzoru podľa ich hodnoty počtu signálov. Toto zatriedenie údajov do skupín prebieha rovnako, ako pri kroku hľadania zhluku pixelov, s tým rozdielom, že spodná hranica pre prah počtu signálov je daná zadaným prahom počtu signálov pre údaje, a horná hranica je daná dekrementovanou hodnotou o jednotku vypočítaného prahu počtu signálov pre vzor spŕšky, ktorý odpovedá hodnote prahu počtu signálov poslednej pridanej skupiny pri hľadaní zhluku pixelov.

7. Na skupiny údajov, od skupiny s najvyššou hodnotou prahu počtu signálov až po tú s najnižšou, sa aplikuje vybraná metóda Houghovej transformácie. Výstupné údaje metódy sa postupne pridávajú do spoločnej skupiny, pričom zároveň sa na základe výstupu obmedzujú hodnoty rozsahov parametrického priestoru metód.

8. Voliteľná filtrácia údajov na základe ich postupnosti na pixeli z hľadiska času a z hľadiska počtu signálov. Algoritmus filtrácie z hľadiska času:

   I. Vstupom sú údaje vybraného pixela.

   II. Zadaná je hodnota prahu počtu signálov pre zahodenie postupnosti údajov. Táto hodnota môže byť zadaná absolútne alebo relatívne k

vypočítanému prahu počtu signálov pre vzor spŕšky.

III. Nájde sa údaj s maximálnou hodnotou počtu signálov.

IV. Ak údaj s maximálnou hodnotou počtu signálov nespĺňa podmienku prahu počtu signálov pre zahodenie postupnosti údajov, všetky údaje sa zahodia.

V. Ak existuje iba jeden údaj s maximálnou hodnotou počtu signálov, tak sa vyberie postupnosť časovo za sebou idúcich údajov, s časovou medzerou 1, obsahujúca tento údaj. V opačnom prípade sa pre každý takýto údaj vyberie postupnosť časovo za sebou idúcich údajov, s časovou medzerou 1, obsahujúca tento údaj. Údaje týchto postupností sa utriedia podľa hodnoty počtu signálov, od údaja s najvyššou po údaj s najnižšou hodnotou. Hodnoty údajov postupností na rovnakých miestach sa potom navzájom porovnávajú od najvyšších a podľa prvej rozdielnej hodnoty sa vyberie postupnosť s údajom s vyššou hodnotou, napr. postupnosti [6,5,4,4,2,1,1] a [6,5,4,3,2,2,1,1,1] - vyberie sa postupnosť [6,5,4,4,2,1,1] kvôli vyššej hodnote na štvrtom mieste.

VI. Zadaná je hodnota prahu počtu signálov pre postupnosť s dĺžkou 1.

VII. Ak má postupnosť iba jeden údaj, tak v prípade splnenia podmienky na hodnotu prahu počtu signálov pre postupnosť s dĺžkou 1, ide tento údaj na výstup, v opačnom prípade je výstup prázdny. V prípade dlhšej postupnosti, ide táto postupnosť priamo na výstup.

Algoritmus filtrácie z hľadiska počtu signálov:

I. Vstupom sú údaje vybraného pixela.

II. Zadaná je hodnota prahu počtu signálov pre zahodenie postupnosti údajov. Táto hodnota môže byť zadaná absolútne alebo relatívne k

vypočítanému prahu počtu signálov pre vzor spŕšky.

III. Nájde sa údaj s maximálnou hodnotou počtu signálov.

IV. Ak údaj s maximálnou hodnotou počtu signálov nespĺňa podmienku prahu počtu signálov pre zahodenie postupnosti údajov, všetky údaje sa zahodia.

V. Údaje sa časovo usporiadajú.

VI. Postupnosti údajov, ktoré idú časovo na jednú alebo druhú stranu od údaja s maximálnou hodnotou počtu signálov, musia spĺňať podmienku klesajúcej postupnosti, okrem prípadu, že má údaj maximálnu hodnotu počtu signálov. Údaj, pri ktorom sa klesajúca postupnosť preruší, sa odstráni, a spolu s ním aj zvyšné údaje postupnosti.

VII. Zadaná je hodnota prahu počtu signálov pre odrezanie postupnosti údajov.

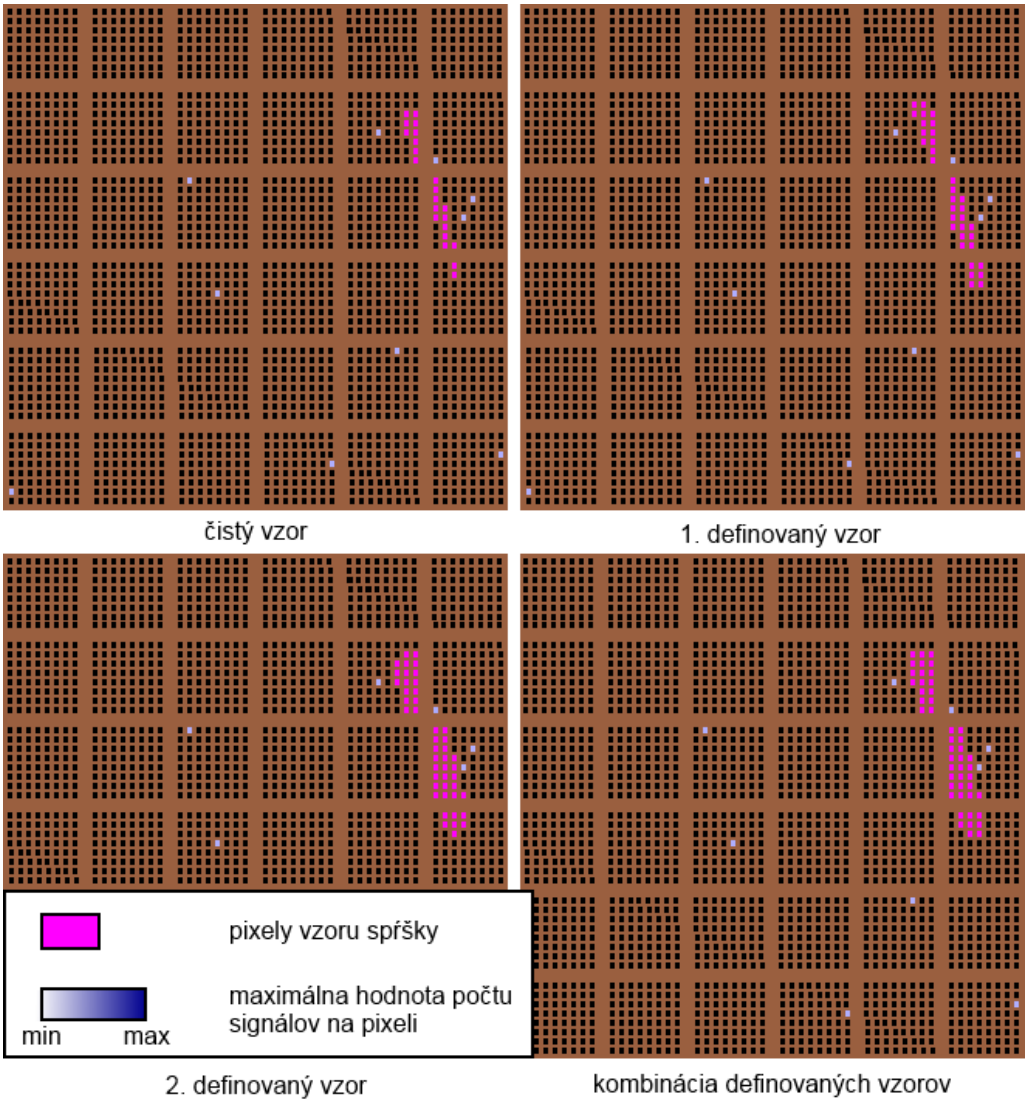
VIII. Ak údaj s maximálnou hodnotou počtu signálov nie je nad hodnotou prahu počtu signálov pre odrezanie postupnosti údajov, tak sa zvyšné údaje odstránia. V opačnom prípade sa postupuje k údajom časovo na jednej alebo druhej strane od tohto údaja, overuje sa rovnaká podmienka, a ak nie je splnená, tak sa zvyšné údaje časovej postupnosti odstránia.

IX. Zadaná je hodnota prahu počtu signálov pre postupnosť s dĺžkou 1.

X. Ak má postupnosť iba jeden údaj, tak v prípade splnenia podmienky na hodnotu prahu počtu signálov pre postupnosť s dĺžkou 1, ide tento údaj na výstup, v opačnom prípade je výstup prázdny. V prípade dlhšej postupnosti, ide táto postupnosť priamo na výstup.

Tak ako pri prvom algoritme, tak aj pri tomto bolo nutné pridať menšie úpravy metód Houghovej transformácie, ktoré sú popísané v nasledujúcej podkapitole.

Oproti prvému algoritmu pribudlo aj využitie, a teda aj nutné úpravy, metódy Houghovej transformácie pre roviny.



Obr. 5 – 29 Zobrazenie vzorov spŕšky.

5.4.2 Úprava metód Houghovej transformácie

S návrhom druhého algoritmu vyvstali pre upravené metódy Houghovej transformácie z prvého algoritmu a pre metódu Houghovej transformácie pre roviny nasledujúce požiadavky:

Úprava metódy Houghovej transformácie pre detekciu rovín Štandardná metóda Houghovej transformácie pre detekciu rovín si vyžaduje rovnaké úpravy, aké boli pri prvom algoritme prevedené na metóde Houghovej transformácie pre detekciu priamok v 2D priestore (Obrázok 5 – 30).

Zadávanie rozsahov parametrického priestoru Oproti metódam Houghovej transformácie pri prvom algoritme je pri tomto algoritme nutné mať aj možnosť prostredníctvom parametrov zadať hodnoty pre rozsahy parametrického priestoru metód Houghovej transformácie.

Úprava výstupu K výstupu údajov sa pridajú aj rozsahy parametrického priestoru na základe parametrov detegovaných objektov metód Houghovej transformácie.

Pseudokód upravenej metódy Houghovej transformácie pre detekciu priamok v 2D priestore

$Data$ vstupné údaje 2D priestoru

$OutputData$ výstupné údaje 2D priestoru

$OutputRanges$ výstupné intervaly hodnôt kolmej vzdialenosti od počiatku súradnicovej sústavy a polárneho uhla detegovaných priamok

$Accumulator$ akumulačné pole

$lines_{parametersIndexes}$ množina dvojíc indexov parametrov detegovaných priamok

$[\theta_{min},\ \theta_{max}]$ rozsah polárneho uhla

$[\theta_{start},\ \theta_{end}]$ interval prípustných hodnôt polárneho uhla

θ_{array} jednorozmerné pole hodnôt polárneho uhla

θ_{count} veľkosť jednorozmerného poľa hodnôt polárneho uhla

θ_{step} krok diskretizácie priestoru polárneho uhla

θ_{set} množina hodnôt polárneho uhla detegovaných priamok

$[\rho_{start},\ \rho_{end}]$ interval prípustných hodnôt kolmej vzdialenosti priamky od počiatku súradnicovej sústavy

ρ_{array} jednorozmerné pole hodnôt kolmej vzdialenosti priamky

ρ_{count} veľkosť jednorozmerného poľa hodnôt kolmej vzdialenosti priamky

ρ_{step} krok diskretizácie priestoru kolmej vzdialenosti priamky

ρ_{value} vypočítaná hodnota kolmej vzdialenosti priamky

ρ_{set} množina hodnôt kolmej vzdialenosti detegovaných priamok od počiatku súradnicovej sústavy

$size$ hodnota maximálnej kolmej vzdialenosti údajov detegovanej priamky

$[distance_{lower},\ distance_{upper}]$ interval prípustných hodnôt kolmej vzdialenosti údajov od priamky

$[\rho_{lowerIndex},\ \rho_{upperIndex}]$ interval indexov kolmých vzdialeností priamok od počiatku súradnicovej sústavy, ktorý vyhovuje zadanej hodnote kolmej vzdialenosti údajov detegovanej priamky $size$

1. $\theta_{min} = -\frac{\pi}{2}$
2. $\theta_{max} = \frac{\pi}{2}$
3. $\rho_{count} = ceil(abs(\frac{\rho_{end}-\rho_{start}}{\rho_{step}})) + 1$
4. $\rho_{array} = [\rho_{start}, \rho_{start} + \rho_{step}, ..., \rho_{start} + \rho_{step}(\rho_{count} - 1)]_{\rho_{count}}$
5. $if\ \theta_{start} <= \theta_{end}$
6. $\theta_{count} = floor(abs(\frac{\theta_{end}-\theta_{start}}{\theta_{step}})) + 1$
7. $\theta_{array} = [\theta_{start}, \theta_{start} + \theta_{step}, ..., \theta_{start} + \theta_{step}(\theta_{count} - 1)]_{\theta_{count}}$
8. $else$

9. $\theta_{count} = floor(abs(\frac{\theta_{end}-\theta_{min}}{\theta_{step}})) + floor(abs(\frac{\theta_{max}-\theta_{start}}{\theta_{step}})) + 2$
10. $\theta_{array} = [\theta_{start}, \theta_{start} + \theta_{step}, ..., \theta_{max}, \theta_{min}, ..., \theta_{end} - \theta_{step}, \theta_{end}]_{\theta_{count}}$
11. $end\ if$
12. $Accumulator[]_{\rho_{count}}[]_{\theta_{count}} = 0$
13. $distance_{lower} = \rho_{array}[0] - 1.1size$
14. $distance_{upper} = \rho_{array}[\rho_{count} - 1] + 1.1size$
15. $for\ data : Data$
16. $for\ \theta_{index} = 0 : (\theta_{count} - 1)$
17. $\rho_{value} = data_x \cos(\theta_{array}[\theta_{index}]) + data_y \sin(\theta_{array}[\theta_{index}])$
18. $if\ \rho_{value} > distance_{lower}\ \&\&\ \rho_{value} < distance_{upper}$
19. $\rho_{lowerIndex} = round(\frac{\rho_{value}-size-\rho_{array}[0]}{\rho_{step}})$
20. $\rho_{upperIndex} = round(\frac{\rho_{value}+size-\rho_{array}[0]}{\rho_{step}})$
21. $if\ \rho_{lowerIndex} < 0$
22. $\rho_{lowerIndex} = 0$
23. $end\ if$
24. $if\ \rho_{upperIndex} > \rho_{count} - 1$
25. $\rho_{upperIndex} = \rho_{count} - 1$
26. $end\ if$
27. $for\ \rho_{index} = \rho_{lowerIndex} : (\rho_{upperIndex} - 1)$
28. $Accumulator[\rho_{index}][\theta_{index}] += data_{signalCount}$
29. $end\ for$
30. $end\ if$

31. $end\ for$
32. $end\ for$
33. $lines_{parametersIndexes} =$
    $\{(\rho_{index},\ \theta_{index}) \mid Accumulator[\rho_{index}][\theta_{index}] = Accumulator_{max}\}$
34. $for\ (\rho_{index},\ \theta_{index}) : lines_{parametersIndexes}$
35. $for\ data : Data$
36. $\rho_{value} = data_x \cos(\theta_{array}[\theta_{index}]) + data_y \sin(\theta_{array}[\theta_{index}])$
37. $if\ \rho_{value} > distance_{lower}\ \&\&\ \rho_{value} < distance_{upper}$
38. $\rho_{lowerIndex} = round(\frac{\rho_{value} - size - \rho_{array}[0]}{\rho_{step}})$
39. $\rho_{upperIndex} = round(\frac{\rho_{value} + size - \rho_{array}[0]}{\rho_{step}})$
40. $if\ \rho_{lowerIndex} <= \rho_{index}\ \&\&\ \rho_{upperIndex} >= \rho_{index}$
41. $data \in OutputData$
42. $end\ if$
43. $end\ if$
44. $end\ for$
45. $end\ for$
46. $\theta_{set} = \{\theta_{array}[\theta_{index}] \mid Accumulator[\rho_{index}][\theta_{index}] = Accumulator_{max}\}$
47. $\rho_{set} = \{\rho_{array}[\rho_{index}] \mid Accumulator[\rho_{index}][\theta_{index}] = Accumulator_{max}\}$
48. $if\ abs(max(\theta_{set}) - min(\theta_{set})) < \frac{\pi}{2}$
49. $(min(\theta_{set}),\ max(\theta_{set})) \in OutputRanges$
50. $else$
51. $(max(\theta_{set}),\ min(\theta_{set})) \in OutputRanges$

52. $end\ if$

53. $(min(\rho_{set}),\ max(\rho_{set})) \in OutputRanges$

Pseudokód upravenej metódy Houghovej transformácie pre detekciu priamok v 3D priestore

$Data$ vstupné údaje 3D priestoru

$OutputData$ výstupné údaje 3D priestoru

$HoughLines2D$ metóda Houghovej transformácie pre detekciu priamok v 2D priestore

$Data_{L_{xz}}$ množina údajov detegovaných priamok v priestore XZ

$Ranges_{xz},\ Ranges_{L_{xzy}}$ vstupné množiny hodnôt rozsahov parametrického priestoru

$Ranges_{L_{xz}},\ Ranges_{LL_{xzy}}$ výstupné množiny hodnôt rozsahov parametrického priestoru na základe parametrov detegovaných priamok

$size_{xz},\ size_{L_{xzy}}$ hodnoty maximálnej kolmej vzdialenosti údajov detegovanej priamky

1. $[Data_{L_{xz}},\ Ranges_{L_{xz}}] = HoughLines2D(Data,\ size_{xz},\ Ranges_{xz})$
2. $[OutputData,\ Ranges_{LL_{xzy}}] =$
   $HoughLines2D(Data_{L_{xz}},\ size_{L_{xzy}},\ Ranges_{L_{xzy}})$

Pseudokód upravenej metódy Houghovej transformácie pre detekciu rovín

$Data$ vstupné údaje 3D priestoru

$OutputData$ výstupné údaje 3D priestoru

$OutputRanges$ výstupné intervaly hodnôt kolmej vzdialenosti od počiatku súradnicovej sústavy, zenitového a azimutového uhla detegovaných rovín

$Accumulator$ akumulačné pole

$planes_{parametersIndexes}$ množina trojíc indexov parametrov detegovaných rovín

$[\theta_{min},\ \theta_{max}]$ rozsah azimutového uhla

$[\theta_{start},\ \theta_{end}]$ interval prípustných hodnôt azimutového uhla

θ_{array} jednorozmerné pole hodnôt azimutového uhla

θ_{count} veľkosť jednorozmerného poľa hodnôt azimutového uhla

θ_{step} krok diskretizácie priestoru azimutového uhla

θ_{set} množina hodnôt azimutového uhla detegovaných rovín

$[\phi_{min},\ \phi_{max}]$ rozsah zenitového uhla

$[\phi_{start},\ \phi_{end}]$ interval prípustných hodnôt zenitového uhla

ϕ_{array} jednorozmerné pole hodnôt zenitového uhla

ϕ_{count} veľkosť jednorozmerného poľa hodnôt zenitového uhla

ϕ_{step} krok diskretizácie priestoru zenitového uhla

ϕ_{set} množina hodnôt zenitového uhla detegovaných rovín

$[\rho_{start},\ \rho_{end}]$ interval prípustných hodnôt kolmej vzdialenosti roviny od počiatku súradnicovej sústavy

ρ_{array} jednorozmerné pole hodnôt kolmej vzdialenosti roviny

ρ_{count} veľkosť jednorozmerného poľa hodnôt kolmej vzdialenosti roviny

ρ_{step} krok diskretizácie priestoru kolmej vzdialenosti roviny

ρ_{value} vypočítaná hodnota kolmej vzdialenosti roviny

ρ_{set} množina hodnôt kolmej vzdialenosti detegovaných rovín od počiatku súradnicovej sústavy

$size$ hodnota maximálnej kolmej vzdialenosti údajov detegovanej roviny

$[distance_{lower},\ distance_{upper}]$ interval prípustných hodnôt kolmej vzdialenosti údajov od priamky

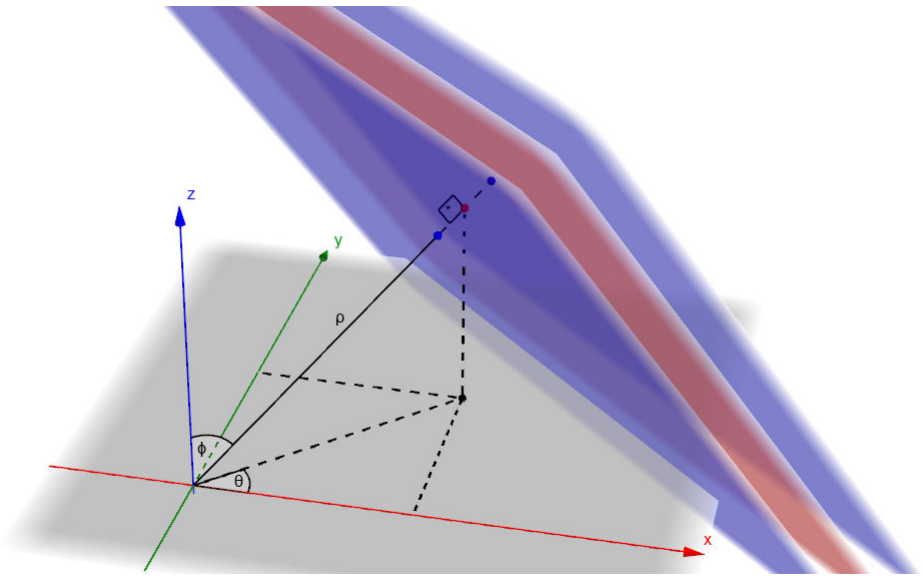
$[\rho_{lowerIndex},\ \rho_{upperIndex}]$ interval indexov kolmých vzdialeností rovín od počiatku súradnicovej sústavy, ktorý vyhovuje zadanej hodnote kolmej vzdialenosti údajov detegovanej roviny $size$

1. $\theta_{min} = -\frac{\pi}{2}$
2. $\theta_{max} = \frac{\pi}{2}$
3. $\phi_{min} = -\frac{\pi}{2}$
4. $\phi_{max} = \frac{\pi}{2}$
5. $\rho_{count} = ceil(abs(\frac{\rho_{end}-\rho_{start}}{\rho_{step}})) + 1$
6. $\rho_{array} = [\rho_{start}, \rho_{start} + \rho_{step}, ..., \rho_{start} + \rho_{step}(\rho_{count} - 1)]_{\rho_{count}}$
7. $if\ \theta_{start} <= \theta_{end}$
8. $\theta_{count} = floor(abs(\frac{\theta_{end}-\theta_{start}}{\theta_{step}})) + 1$
9. $\theta_{array} = [\theta_{start}, \theta_{start} + \theta_{step}, ..., \theta_{start} + \theta_{step}(\theta_{count} - 1)]_{\theta_{count}}$
10. $else$
11. $\theta_{count} = floor(abs(\frac{\theta_{end}-\theta_{min}}{\theta_{step}})) + floor(abs(\frac{\theta_{max}-\theta_{start}}{\theta_{step}})) + 2$
12. $\theta_{array} = [\theta_{start}, \theta_{start} + \theta_{step}, ..., \theta_{max}, \theta_{min}, ..., \theta_{end} - \theta_{step}, \theta_{end}]_{\theta_{count}}$
13. $end\ if$
14. $if\ \phi_{start} <= \phi_{end}$
15. $\phi_{count} = floor(abs(\frac{\phi_{end}-\phi_{start}}{\phi_{step}})) + 1$

16. $\phi_{array} = [\phi_{start}, \phi_{start} + \phi_{step}, ..., \phi_{start} + \phi_{step}(\phi_{count} - 1)]_{\phi_{count}}$
17. $else$
18. $\phi_{count} = floor(abs(\frac{\phi_{end} - \phi_{min}}{\phi_{step}})) + floor(abs(\frac{\phi_{max} - \phi_{start}}{\phi_{step}})) + 2$
19. $\phi_{array} = [\phi_{start}, \phi_{start} + \phi_{step}, ..., \phi_{max}, \phi_{min}, ..., \phi_{end} - \phi_{step}, \phi_{end}]_{\phi_{count}}$
20. $end\ if$
21. $distance_{lower} = \rho_{array}[0] - 1.1size$
22. $distance_{upper} = \rho_{array}[\rho_{count} - 1] + 1.1size$
23. $Accumulator[]_{\rho_{count}}[]_{\theta_{count}}[]_{\phi_{count}} = 0$
24. $for\ point : Data$
25. $for\ \theta_{index} = 0 : (\theta_{count} - 1)$
26. $for\ \phi_{index} = 0 : (\phi_{count} - 1)$
27. $\rho_{value} = data_x \sin(\phi_{array}[\phi_{index}]) \cos(\theta_{array}[\theta_{index}]) +$
    $data_y \sin(\phi_{array}[\phi_{index}]) \sin(\theta_{array}[\theta_{index}]) +$
    $data_z \cos(\phi_{array}[\phi_{index}])$
28. $if\ \rho_{value} > distance_{lower}\ \&\&\ \rho_{value} < distance_{upper}$
29. $\rho_{lowerIndex} = round(\frac{\rho_{value} - size - \rho_{array}[0]}{\rho_{step}})$
30. $\rho_{upperIndex} = round(\frac{\rho_{value} + size - \rho_{array}[0]}{\rho_{step}})$
31. $if\ \rho_{lowerIndex} < 0$
32. $\rho_{lowerIndex} = 0$
33. $end\ if$
34. $if\ \rho_{upperIndex} > \rho_{count} - 1$
35. $\rho_{upperIndex} = \rho_{count} - 1$

36. $end\ if$
37. $for\ \rho_{index} = \rho_{lowerIndex} : (\rho_{upperIndex} - 1)$
38. $Accumulator[\rho_{index}][\theta_{index}][\phi_{index}] += data_{signalCount}$
39. $end\ for$
40. $end\ if$
41. $end\ for$
42. $end\ for$
43. $end\ for$
44. $planes_{parametersIndexes} = \{(\rho_{index},\ \theta_{index},\ \phi_{index}) \mid Accumulator[\rho_{index}][\theta_{index}][\phi_{index}] = Accumulator_{max}\}$
45. $for\ (\rho_{index},\ \theta_{index},\ \phi_{index}) : planes_{parametersIndexes}$
46. $for\ data : Data$
47. $\rho_{value} = data_x \sin(\phi_{array}[\phi_{index}]) \cos(\theta_{array}[\theta_{index}]) + data_y \sin(\phi_{array}[\phi_{index}]) \sin(\theta_{array}[\theta_{index}]) + data_z \cos(\phi_{array}[\phi_{index}])$
48. $if\ \rho_{value} > distance_{lower}\ \&\&\ \rho_{value} < distance_{upper}$
49. $\rho_{lowerIndex} = round(\frac{\rho_{value} - size - \rho_{array}[0]}{\rho_{step}})$
50. $\rho_{upperIndex} = round(\frac{\rho_{value} + size - \rho_{array}[0]}{\rho_{step}})$
51. $if\ \rho_{lowerIndex} <= \rho_{index}\ \&\&\ \rho_{upperIndex} >= \rho_{index}$
52. $data \in OutputData$
53. $end\ if$
54. $end\ if$

55. $end\ for$
56. $end\ for$
57. $\theta_{set} = \{\theta_{array}[\theta_{index}] \mid Accumulator[\rho_{index}][\theta_{index}][\phi_{index}] = Accumulator_{max}\}$
58. $\phi_{set} = \{\phi_{array}[\phi_{index}] \mid Accumulator[\rho_{index}][\theta_{index}][\phi_{index}] = Accumulator_{max}\}$
59. $\rho_{set} = \{\rho_{array}[\rho_{index}] \mid Accumulator[\rho_{index}][\theta_{index}][\phi_{index}] = Accumulator_{max}\}$
60. $if\ abs(max(\theta_{set}) - min(\theta_{set})) < \frac{\pi}{2}$
61. $(min(\theta_{set}),\ max(\theta_{set})) \in OutputRanges$
62. $else$
63. $(max(\theta_{set}),\ min(\theta_{set})) \in OutputRanges$
64. $end\ if$
65. $if\ abs(max(\phi_{set}) - min(\phi_{set})) < \frac{\pi}{2}$
66. $(min(\phi_{set}),\ max(\phi_{set})) \in OutputRanges$
67. $else$
68. $(max(\phi_{set}),\ min(\phi_{set})) \in OutputRanges$
69. $end\ if$
70. $(min(\rho_{set}),\ max(\rho_{set})) \in OutputRanges$



Obr. 5 – 30 Geometrický význam upravenej metódy Houghovej transformácie pre roviny.

5.4.3 Implementácia a výpočtové zložitosti

Pre potreby vývoja bol algoritmus implementovaný v jazyku Java, spoločne s jeho čiastočnou vizualizáciou. Po jeho dokončení bol Michalom Vrábelom (Vrábel, 2015) prepísaný tak, aby sa dal použiť v rámci ESAF. V nasledujúcom popise bude predstavený zdrojový kód Java triedy, ktorá obsahuje implementáciu algoritmu a je súčasťou programu, vytvoreného pre vývoj a vizualizáciu algoritmov tejto práce. Kompletný zdrojový kód je možné vidieť v prílohe A - Trieda Algorithm2.

Uvedené budú podstatné súčasti implementácie, postupne popísané v súlade s algoritmom, uvedeným v podkapitole 5.4.1. Implementáciu prvých dvoch krokov algoritmu si môžte pozrieť na začiatku funkcie `run`, v prílohe A v triede Algorithm2. V treťom kroku uvedeného algoritmu sa pre hľadanie zhluku pixelov využíva funkcia `getShapedCore` (Zdrojový kód 12).

Zdroj. kód 12 Funkcia `getShapedCore`, určená pre hľadanie zhlukov pixelov.

```
/**
 *
```

```
     * @param data - údaje
     * @param maxGap - maximálna medzera medzi susediacimi údajmi
     * @param sizeLimit - velkostný limit skupiny
     * @return - vráti maximálnu skupinu susediacich údajov v rámci
     * povolenej medzery medzi nimi
     */
    List<Double[]> getShapedCore(List<Double[]> data, double maxGap
        , int sizeLimit) {

        //vytvorí sa mapa susediacich údajov s danou maximálnou
            medzerou
        int[][] neighboursMap = new int[data.size()][data.size()];
        for (int i = 0; i < data.size(); i++) {
            neighboursMap[i][i] = 2;
            double x = data.get(i)[xIndex];
            double y = data.get(i)[yIndex];
            for (int j = 0; j < data.size(); j++) {
                //0 indikuje, že o susednosti ešte nebolo
                //rozhodnuté
                if (neighboursMap[i][j] == 0) {
                    double nx = data.get(j)[xIndex];
                    double ny = data.get(j)[yIndex];
                    if (Math.sqrt((x - nx) * (x - nx) + (y - ny) *
                        (y - ny)) < maxGap) {
                        //susediace údaje majú v mape 1
                        neighboursMap[i][j] = 1;
                        neighboursMap[j][i] = 1;
                    } else {
                        //nesusediace údaje majú v mape 2
                        neighboursMap[i][j] = 2;
                        neighboursMap[j][i] = 2;
                    }
                }
            }
        }
```

```
        //rekurzívne sa vyhladá najväčšia skupina údajov, prípadne
            skupiny
        boolean[] neighboursChecked = new boolean[data.size()];
        List<List<Integer>> maxGroup = new ArrayList<>();
        maxGroup.add(new ArrayList<>());
        List<Integer> tempGroup;
        for (int i = 0; i < data.size(); i++) {
            if (!neighboursChecked[i]) {
                tempGroup = getNeighbours(i, neighboursMap,
                    neighboursChecked);
                if (tempGroup.size() > maxGroup.get(0).size()) {
                    maxGroup.clear();
                    maxGroup.add(tempGroup);
                } else if (tempGroup.size() == maxGroup.get(0).size
                    ()) {
                    maxGroup.add(tempGroup);
                }
            }
        }

        List<Double[]> resultGroup = new ArrayList<>();
        //overí sa, či má skupina dostatočný počet údajov
        if (maxGroup.get(0).size() >= sizeLimit) {
            //index výslednej skupiny
            int index = 0;
            //ak sa našlo viacero skupín s rovnakým množstvom
            //údajov, tak sa vezme tá, ktorá ma najväčšiu
            //hodnotu súčtu počtov signálov svojich údajov
            if (maxGroup.size() > 1) {
                double maxSignalCount = 0;
                for (int i = 0; i < maxGroup.size(); i++) {
                    double tempSignalCount = 0;
                    for (Integer ind : maxGroup.get(i)) {
                        tempSignalCount += data.get(ind)[
```

```
                        totalCountIndex];
                }
                if (tempSignalCount > maxSignalCount) {
                    maxSignalCount = tempSignalCount;
                    index = i;
                }
            }
        }
        //do výsledného zoznamu sa zapíšu údaje
        for (Integer idx : maxGroup.get(index)) {
            resultGroup.add(data.get(idx));
        }

    }
    return resultGroup;
}
```

Táto funkcia prijíma ako parametre údaje, maximálnu veľkosť vzdialenosti od zhluku a minimálny počet údajov zhluku. Funkcia si na začiatku vytvára dvojrozmernú mapu susediacich údajov. V tejto mape sú susediace údaje označené hodnotou 1, nesusediace hodnotou 2, a ak ešte nebolo o susednosti rozhodnuté, tak je v mape hodnota 0. Ak sa rozhoduje o susednosti údaja so sebou samým, tak je použitá hodnota 2. Po vytvorení mapy, sa pre nájdenie zhlukov použije rekurzívna funkcia `getNeighbours` (Zdrojový kód 13).

Zdroj. kód 13 Funkcia `getNeighbours`, použitá pre rekurzívne určenie susediacich pixelov.

```
/**
 *
 * @param row - riadok v mape susediacich údajov
 * @param neighboursMap - mapa susediacich údajov
 * @param neighboursChecked - pole pre označenie prejdených
 * riadkov v mape susediacich údajov
 * @return - zoznamy indexov skupín susediacich údajov
 */
```

```
    private List<Integer> getNeighbours(int row, int[][]
        neighboursMap, boolean[] neighboursChecked) {
         neighboursChecked[row] = true;
         List<Integer> result = new ArrayList<>();
         result.add(row);
         for (int i = 0; i < neighboursMap[row].length; i++) {
             if (neighboursMap[row][i] == 1) {
                 if (!neighboursChecked[i]) {
                     result.addAll(getNeighbours(i, neighboursMap,
                         neighboursChecked));
                 }
             }
         }
         return result;
    }
```

Jej parametrami sú index údaja v poli prejdených údajov, mapa susediacich údajov a pole prejdených údajov. Funkcia vracia indexy údajov v zozname všetkých údajov. Pre každý údaj sa rekurzívne hľadá zhluk, ktorého je údaj súčasťou. To, či bol údaj už predtým priradený k nejakému zhluku, indikuje pole prejdených údajov. Z nájdených zhlukov sa vyberie ten, ktorý obsahuje najviac údajov. V prípade rovnakého počtu údajov rozhoduje ich suma počtu signálov. Výstupom funkcie `getShapedCore` sú údaje výsledného zhluku. Po nájdení zhluku je vhodné pred hľadaním zvyšku spŕšky vykonať posunutie údajov, ktoré zredukuje pamäťové nároky pred volaním metódy Houghovej transformácie. V implementácii to realizuje zdrojový kód 14.

Zdroj. kód 14 Minimalizácia pamäťových nárokov.

```
        //centrovanie údajov pre čo najnižšie pamätové zataženie
        double[] vectorXYGtu = getCenter3D(maxGroup);
        //posunutie údajov
        shift(newData, -vectorXYGtu[0], -vectorXYGtu[1], -
            vectorXYGtu[2]);
```

Funkcie `getCenter3D` a `shift` už boli popísané pri implementácii prvého algoritmu a zostali nezmenené.

V algoritme nasleduje vybranie údajov okolo nájdeného zhluku, pri ktorom bola využitá metóda Houghovej transformácie pre priamky v 2D priestore pre nájdenie priamky prechádzajúcej centrálnou líniou nájdeného zhluku pixelov. Implementácia tejto metódy bola zobrazená a čiastočne popísaná pri implementácii prvého algoritmu, preto sa popíšu len časti, ktoré boli pridané pre potreby tohto algoritmu.

Prvou takouto časťou je zdrojový kód 15.

Zdroj. kód 15 Funkcia `HoughLine2D`, obsahujúca implementáciu Houghovej transformácie pre priamky v 2D priestore (1. popisovaná časť).

```
/**
 * Houghova transformácia pre priamky v 2D priestore
 *
 * @param data - údaje
 * @param size - maximálna povolená kolmá vzdialenost údajov od
 * detegovanej priamky
 * @param indexFirstDim - index prvej dimenzie 2D priestoru
 * @param indexSecondDim - index druhej dimenzie 2D priestoru
 * @param thetaStep - krok diskretizácie priestoru uhla
 * fi [deg]
 * @param thetaStart - začiatok priestoru uhla theta v
 * radiánoch (v intervale od -PI/2 do PI/2)
 * @param thetaEnd - koniec priestoru uhla theta v radiánoch
 * (v intervale od -PI/2 do PI/2)
 * @param roStep - krok diskretizácie priestoru kolmej
 * vzdialenosti priamky od počiatku súradnicovej sústavy
 * @param roStart - začiatok priestoru kolmej vzdialenosti
 * priamky od počiatku súradnicovej sústavy
 * @param roEnd - koniec priestoru kolmej vzdialenosti priamky
 * od počiatku súradnicovej sústavy
 * @return - výstupné údaje Houghovej transformácie pre priamky
 * v 2D priestore
```

```
     */
    private List<Double[]>[] HoughLine2D(List<Double[]> data,
        double size, Integer indexFirstDim, Integer indexSecondDim,
        double thetaStep, double thetaStart, double thetaEnd, double
         roStep, double roStart, double roEnd) {

        //definovanie pola hodnôt kolmej vzdialenosti priamky od
        //počiatku súradnicovej sústavy
        int nRo = (int) (Math.ceil(Math.abs(roEnd - roStart) /
            roStep)) + 1;
        double[] roArray = new double[nRo];
        int index = 0;
        double roValue = roStart;
        while (index < nRo) {
            roArray[index++] = roValue;
            roValue += roStep;
        }

        //definovanie pola hodnôt uhla theta
        thetaStep = thetaStep * Math.PI / 180;
        double[] thetaArray;
        int nTheta;
        if (thetaStart <= thetaEnd) {
            nTheta = (int) (Math.abs(thetaEnd - thetaStart) /
                thetaStep) + 1;
            if ((nTheta - 1) * thetaStep + thetaStart < thetaEnd &&
                 nTheta * thetaStep + thetaStart < Math.PI / 2) {
                nTheta++;
            }
            thetaArray = new double[nTheta];
            index = 0;
            double thetaValue = thetaStart;
            while (index < nTheta) {
                thetaArray[index++] = thetaValue;
                thetaValue += thetaStep;
```

```
            }
        } else {
            int nTheta1 = (int) (Math.abs(thetaEnd - (-Math.PI / 2)
                ) / thetaStep) + 1;
            int nTheta2 = (int) (Math.abs(Math.PI / 2 - thetaStart)
                 / thetaStep) + 1;
            if ((nTheta1 - 1) * thetaStep + (-Math.PI / 2) <
                thetaEnd && nTheta1 * thetaStep + (-Math.PI / 2) <
                Math.PI / 2) {
                 nTheta1++;
            }
            nTheta = nTheta2 + nTheta1;
            thetaArray = new double[nTheta];
            index = 0;
            double thetaValue = -Math.PI / 2;
            while (index < nTheta1) {
                 thetaArray[index++] = thetaValue;
                 thetaValue += thetaStep;
            }
            thetaValue = thetaStart;
            while (index < nTheta) {
                 thetaArray[index++] = thetaValue;
                 thetaValue += thetaStep;
            }
        }
```

Keďže si algoritmus vyžadoval kontrolu nad rozsahmi parametrického priestoru, tak boli použité parametre `thetaStart, thetaEnd, roStart` a `roEnd`. Zatiaľ čo použitie `roStart` a `roEnd` je očividné a nepotrebuje komentár, použitie parametrov `thetaStart` a `thetaEnd` má tú zvláštnosť, že `thetaStart` nemusí byť menšie ako `thetaEnd`. Je to spôsobené tým, že sa držíme rozsahu $[-\frac{\pi}{2}, \frac{\pi}{2}]$ a je nutné počítať s prechodom $\frac{\pi}{2} \rightarrow -\frac{\pi}{2}$. Po tom, čo sú vo funkcii detegované priamky, je potrebné dať na výstup rozsahy parametrického priestoru vzhľadom na hodnoty parametrov

týchto priamok (Zdrojový kód 16).

Zdroj. kód 16 Funkcia `HoughLine2D`, obsahujúca implementáciu Houghovej transformácie pre priamky v 2D priestore (2. popisovaná časť).

```
        //výstupné hodnoty
        List<Double[]>[] result = new ArrayList[2];

        //zápis rozsahov parametrického priestoru, definovaných na
            základe výstupných údajov
        if (Math.abs(thetaArray[indexes.get(indexes.size() - 1)[
            thetaIndexL]] - thetaArray[indexes.get(0)[thetaIndexL]])
             < Math.PI / 2) {
             resultRanges.add(new Double[]{thetaArray[indexes.get(0)
                 [thetaIndexL]], thetaArray[indexes.get(indexes.size
                 () - 1)[thetaIndexL]]});
        } else {
             resultRanges.add(new Double[]{thetaArray[indexes.get(
                 indexes.size() - 1)[thetaIndexL]], thetaArray[
                 indexes.get(0)[thetaIndexL]]});
        }
        resultRanges.add(new Double[]{roArray[minRo], roArray[maxRo
            ]});

        result[dataIndex] = new ArrayList<>(resultData);
        result[rangesIndex] = resultRanges;

        return result;
    }
```

Rozsah parametrického priestoru kolmej vzdialenosti priamky od počiatku súradnicovej sústavy sa definuje od minimálnej po maximálnu hodnotu množiny kolmých vzdialeností detegovaných priamok od počiatku súradnicovej sústavy. Pri rozsahu parametrického priestoru polárneho uhla θ je to rovnaké, s tým rozdielom, že ak je medzi maximálnou a minimálnou hodnotou rozdiel väčší ako $\frac{\pi}{2}$, tak je rozsah

na výstupe definovaný od maximálnej po minimálnu hodnotu množiny polárnych uhlov detegovaných priamok. Pomocou tejto funkcie sa hľadá priamka prechádzajúca centrálnou líniou nájdeného zhluku pixelov, zdrojový kód 17, kde volanie metódy Houghovej transformácie pre priamky v 2D priestore, ktoré bolo popísané už pri implementácii prvého algoritmu, je uvedené v zdrojovom kóde 18.

Zdroj. kód 17 Hľadanie priamky prechádzajúcej centrálnou líniou nájdeného zhluku pixelov.

```
        List<Double[]> lineRanges = HoughLine2D(topData, LINE_SIZE,
             xIndex, yIndex, THETA_STEP_LINE, RO_STEP_LINE)[
            rangesIndex];
        double thetaAngle = (lineRanges.get(thetaIndexL)[lowerRange
            ] + lineRanges.get(thetaIndexL)[upperRange]) / 2;
        if (lineRanges.get(thetaIndexL)[lowerRange] > lineRanges.
            get(thetaIndexL)[upperRange]) {
             thetaAngle += Math.PI / 2;
        }
        double roDistance = (lineRanges.get(roIndexL)[lowerRange] +
             lineRanges.get(roIndexL)[upperRange]) / 2;
```

Zdroj. kód 18 Funkcia `HoughLine2D`, použitá pri hľadaní priamky prechádzajúcej centrálnou líniou nájdeného zhluku pixelov.

```
    /**
     * Houghova transformácia pre priamky v 2D priestore
     *
     * @param data - údaje
     * @param size - maximálna povolená kolmá vzdialenost údajov od
     * detegovanej priamky
     * @param indexFirstDim - index prvej dimenzie 2D priestoru
     * @param indexSecondDim - index druhej dimenzie 2D priestoru
     * @param thetaStep - krok diskretizácie priestoru uhla
     * theta [deg]
     * @param roStep - krok diskretizácie priestoru kolmej
     * vzdialenosti priamky od počiatku súradnicovej sústavy
     * @return - výstupné údaje Houghovej transformácie pre priamky
```

```
     * v 2D priestore
     */
    private List<Double[]>[] HoughLine2D(List<Double[]> data,
        double size, int indexFirstDim, int indexSecondDim, double
        thetaStep, double roStep) {
         //maximalna absolutna vzdialenost udajov od pociatku
             suradnicovej sustavy
         double ro = getMaxDist(data, indexFirstDim, indexSecondDim)
             ;
         return HoughLine2D(data, size, indexFirstDim,
             indexSecondDim, thetaStep, -Math.PI / 2, Math.PI / 2,
             roStep, -ro, ro);
    }
```

Parametre priamky sú tak definované na základe rozsahov parametrického priestoru vrátených funkciou. Z rozsahov sa vezme stredná hodnota, poprípade, pri rozsahu polárneho uhla, ak je počiatočná hodnota rozsahu väčšia než koncová, je potrebné ešte získanú hodnotu posunúť o $\frac{\pi}{2}$, keďže platí

$$\frac{(\frac{\pi}{2} - a) + (-\frac{\pi}{2} + b)}{2} = \frac{b - a}{2} \tag{5.1}$$

, kde a a b sú kladné konštanty také, že

$$(\frac{\pi}{2} - a) - (-\frac{\pi}{2} + b) > \frac{\pi}{2} \tag{5.2}$$

, a teda k výsledku rovnice 5.1 je ešte potrebné pripočítať $\frac{\pi}{2}$ alebo $-\frac{\pi}{2}$. Keďže pôjde o popis tej istej priamky, rozdiel uhlov je 180 stupňov, tak na znamienku nezáleží. Po tom, čo sa našla priamka prechádzajúca centrálnou líniou nájdeného zhluku pixelov, je možné pridať k údajom zhluku údaje z okolia, tak ako je to popísané v návrhu algoritmu, a aplikovať vybranú metódu Houghovej transformácie (Zdrojový kód 19).

Zdroj. kód 19 Funkcia `Hough`, určená pre hľadanie vzoru.

```
    /**
     * metóda vybranej Houghovej transformácie pre hladanie vzoru
```

```
     *
     * @param data - údaje
     * @param indexHoughTransform - volba metódy Houghovej
     * transformácie
     * @return - výstupné údaje Houghovej transformácie pre
     * priamky/roviny
     */
    private List<Double[]>[] Hough(int indexHoughTransform, List<
        Double[]> data) {
         if (indexHoughTransform == 0) {
             return HoughLine3D(data, INITIAL_PATTERN_SIZE_LINE3D,
                 xIndex, gtuIndex, yIndex, THETA_STEP_LINE,
                 RO_STEP_LINE);
         } else {
             double ro = getMaxDist(data, xIndex, yIndex, gtuIndex);
             return HoughPlane(data, INITIAL_PATTERN_SIZE_PLANE,
                 INITIAL_PATTERN_FI_STEP_PLANE, -Math.PI / 2, Math.PI
                  / 2, INITIAL_PATTERN_THETA_STEP_PLANE, -Math.PI /
                 2, Math.PI / 2, INITIAL_PATTERN_RO_STEP_PLANE, -ro,
                 ro);
         }
    }
```

Voľba metódy sa deje na základe hodnoty `indexHoughTransform`. Parametre metód obsahujú maximálne rozsahy parametrických priestorov, vhodné maximálne kolmé vzdialenosti údajov od detegovaných objektov a vhodné kroky diskretizácie, vzhľadom na presnosť a pamäťové nároky. Volanie metódy Houghovej transformácie pre priamky v 3D je uvedené v zdrojovom kóde 20.

Zdroj. kód 20 Funkcia `HoughLine3D`, určená pre hľadanie vzoru.

```
    /**
     * Houghova transformácia pre priamky v 3D priestore pre
     * hladanie vzoru
     *
```

```
     * @param data - údaje
     * @param size - maximálna povolená kolmá vzdialenost údajov od
     * detegovanej priamky
     * @param xIndex - index prvej dimenzie 3D priestoru
     * @param yIndex - index druhej dimenzie 3D priestoru
     * @param zIndex - index tretej dimenzie 3D priestoru
     * @param thetaStep - krok diskretizácie priestoru uhla
     * theta [deg]
     * @param roStep - krok diskretizácie priestoru kolmej
     * vzdialenosti priamky od počiatku súradnicovej sústavy
     * @return - výstupné údaje Houghovej transformácie pre priamky
     * v 3D priestore
     */
    private List<Double[]>[] HoughLine3D(List<Double[]> data,
        double size, int xIndex, int yIndex, int zIndex, double
        thetaStep, double roStep) {
         List<Double[]>[] firstData = HoughLine2D(data, size, xIndex
             , yIndex, thetaStep, roStep);
         List<Double[]>[] result = HoughLine2D(firstData[dataIndex],
              size, yIndex, zIndex, thetaStep, roStep);
         result[rangesIndex].add(firstData[rangesIndex].get(
            thetaIndexL));
         result[rangesIndex].add(firstData[rangesIndex].get(roIndexL
            ));
         return result;
    }
```

Volanie metódy Houghovej transformácie pre priamky v 2D priestore odpovedá predtým popísanému kódu. Volaná metóda Houghovej transformácie pre roviny `HoughPlane` je v zdrojovom kóde 21.

Zdroj. kód 21 Funkcia `HoughPlane`, obsahujúca implementáciu Houghovej transformácie pre roviny (1. časť).

```
    /**
     * Houghova transformácia pre roviny
```

```
     *
     * @param data - údaje
     * @param size - maximálna kolmá vzdialenost údajov od
     * detegovanej roviny
     * @param fiStep - krok diskretizácie priestoru uhla fi [deg]
     * @param fiStart - začiatok priestoru uhla fi v radiánoch
     * (v intervale od -PI/2 do PI/2)
     * @param fiEnd - koniec priestoru uhla fi v radiánoch
     * (v intervale od -PI/2 do PI/2)
     * @param thetaStep - krok diskretizácie priestoru uhla
     * theta [deg]
     * @param thetaStart - začiatok priestoru uhla theta v
     * radiánoch (v intervale od -PI/2 do PI/2)
     * @param thetaEnd - koniec priestoru uhla theta v radiánoch
     * (v intervale od -PI/2 do PI/2)
     * @param roStep - krok diskretizacie priestoru kolmej
     * vzdialenosti roviny od počiatku súradnicovej sústavy
     * @param roStart - začiatok priestoru kolmej vzdialenosti
     * roviny od počiatku súradnicovej sústavy
     * @param roEnd - koniec priestoru kolmej vzdialenosti roviny
     * od počiatku súradnicovej sústavy
     * @return - výstupné údaje Houghovej transformácie pre roviny
     */
    private List<Double[]>[] HoughPlane(List<Double[]> data, double
        size, double fiStep, double fiStart, double fiEnd, double
       thetaStep, double thetaStart, double thetaEnd, double roStep
       , double roStart, double roEnd) {
        //definovanie pola hodnôt uhla fi
        fiStep = fiStep * Math.PI / 180;
        double[] fiArray;
        int index;
        int nFi;
        if (fiStart <= fiEnd) {
            nFi = (int) (Math.abs(fiEnd - fiStart) / fiStep) + 1;
            if ((nFi - 1) * fiStep + fiStart < fiEnd && nFi *
```

```
            fiStep + fiStart < Math.PI / 2) {
                nFi++;
            }
            fiArray = new double[nFi];
            index = 0;
            double fiValue = fiStart;
            while (index < nFi) {
                fiArray[index++] = fiValue;
                fiValue += fiStep;
            }
        } else {
            int nFi1 = (int) (Math.abs(fiEnd - (-Math.PI / 2)) /
                fiStep) + 1;
            int nFi2 = (int) (Math.abs(Math.PI / 2 - fiStart) /
                fiStep) + 1;
            if ((nFi1 - 1) * fiStep + (-Math.PI / 2) < fiEnd &&
                nFi1 * fiStep + (-Math.PI / 2) < Math.PI / 2) {
                nFi1++;
            }
            nFi = nFi2 + nFi1;
            fiArray = new double[nFi];
            index = 0;
            double fiValue = -Math.PI / 2;
            while (index < nFi1) {
                fiArray[index++] = fiValue;
                fiValue += fiStep;
            }
            fiValue = fiStart;
            while (index < nFi) {
                fiArray[index++] = fiValue;
                fiValue += fiStep;
            }
        }

        //definovanie pola hodnôt uhla theta
```

```
        thetaStep = thetaStep * Math.PI / 180;
        double[] thetaArray;
        int nTheta;
        if (thetaStart <= thetaEnd) {
            nTheta = (int) (Math.abs(thetaEnd - thetaStart) /
                thetaStep) + 1;
            if ((nTheta - 1) * thetaStep + thetaStart < thetaEnd &&
                 nTheta * thetaStep + thetaStart < Math.PI / 2) {
                 nTheta++;
            }
            thetaArray = new double[nTheta];
            index = 0;
            double thetaValue = thetaStart;
            while (index < nTheta) {
                 thetaArray[index++] = thetaValue;
                 thetaValue += thetaStep;
            }
        } else {
            int nTheta1 = (int) (Math.abs(thetaEnd - (-Math.PI / 2)
                ) / thetaStep) + 1;
            int nTheta2 = (int) (Math.abs(Math.PI / 2 - thetaStart)
                 / thetaStep) + 1;
            if ((nTheta1 - 1) * thetaStep + (-Math.PI / 2) <
                thetaEnd && nTheta1 * thetaStep + (-Math.PI / 2) <
                Math.PI / 2) {
                 nTheta1++;
            }
            nTheta = nTheta2 + nTheta1;
            thetaArray = new double[nTheta];
            index = 0;
            double thetaValue = -Math.PI / 2;
            while (index < nTheta1) {
                 thetaArray[index++] = thetaValue;
                 thetaValue += thetaStep;
            }
```

```
            thetaValue = thetaStart;
            while (index < nTheta) {
                thetaArray[index++] = thetaValue;
                thetaValue += thetaStep;
            }
        }

        //definovanie pola hodnôt kolmej vzdialenosti roviny od
        //počiatku súradnicovej sústavy
        int nRo = (int) (Math.ceil(Math.abs(roEnd - roStart) /
           roStep)) + 1;
        double[] roArray = new double[nRo];
        index = 0;
        double roValue = roStart;
        while (index < nRo) {
            roArray[index++] = roValue;
            roValue += roStep;
        }
```

Tento zdrojový kód je rovnaký, ako ten pri metóde Houghovej transformácie pre priamky v 2D priestore s tým, že pribudol uhol ϕ, pre ktorý platí presne to isté, čo pre uhol *theta*. S jedným parametrom naviac rovnako pribudol aj jeden rozmer akumulačného poľa a namiesto rovnice priamky sa používa rovnica roviny. Zdrojový kód pokračuje v 22.

Zdroj. kód 22 Funkcia `HoughPlane`, obsahujúca implementáciu Houghovej transformácie pre roviny (2. časť).

```
        //trojrozmerná inkrementačná matica
        short[][][] H = new short[nFi][nTheta][nRo];

        double lowerDistance = roArray[0] - 1.1 * size;
        double upperDistance = roArray[nRo - 1] + 1.1 * size;

        //výpočet kolmých vzdialeností roviny od počiatku
```

```
        //súradnicovej sústavy
        for (Double[] row : data) {
            for (int thetaIndex = 0; thetaIndex < nTheta;
               thetaIndex++) {
                for (int fiIndex = 0; fiIndex < nFi; fiIndex++) {
                    //výpočet
                    double distance = Math.sin(fiArray[fiIndex]) *
                       Math.cos(thetaArray[thetaIndex]) * row[
                       xIndex]
                            + Math.sin(fiArray[fiIndex]) * Math.sin
                               (thetaArray[thetaIndex]) * row[
                               yIndex]
                            + Math.cos(fiArray[fiIndex]) * row[
                               gtuIndex];
                    //kontrola, či sa bude inkrementovat
                    //inkrementačná matica
                    if (distance > lowerDistance && distance <
                       upperDistance) {
                        //vypočíta sa horný a dolný index pre
                           hodnoty kolmých vzdialeností roviny
                        //od počiatku súradnicovej sústavy vzhladom
                            na maximálnu povolenú kolmú vzdialenost
                        //údajov od nájdenej roviny a vzorkovaciu
                           frekvenciu pola kolmých vzdialeností
                        int lowerIndex = (int) Math.round((distance
                            - size - roArray[0]) / roStep);
                        int upperIndex = (int) Math.round((distance
                            + size - roArray[0]) / roStep);
                        //indexy musia byt v povolenom rozsahu
                        if (lowerIndex < 0) {
                            lowerIndex = 0;
                        }
                        if (upperIndex > nRo - 1) {
                            upperIndex = nRo - 1;
                        }
```

```
                    //inkrementácia polí rovín, v ktorých
                    //sa údaj vyskytuje
                    for (int roIndex = lowerIndex; roIndex <
                        upperIndex + 1; roIndex++) {
                        H[fiIndex][thetaIndex][roIndex] += row[
                            totalCountIndex];
                    }
                }
            }
        }
    }
```

S rozmerom naviac sa mení aj výpočtová zložitosť. Počet polí akumulátora oproti metóde Houghovej transformácie pre priamky v 2D priestore narastá na $nRo{\cdot}nTheta{\cdot}nFi$. Keďže v algoritme bol použitý pre uhly maximálny rozsah $[-\frac{\pi}{2}, \frac{\pi}{2}]$, tak to môžme čiastočne ohraničiť ako $(ceil(\frac{2{\cdot}roMax}{roStep})+1){\cdot}(floor(\frac{\pi}{thetaStep})+1){\cdot}(floor(\frac{\pi}{fiStep})$ $+ 1)$. Pre počet operácií, ak berieme do úvahy iba výpočet kolmej vzdialenosti a konštantný počet inkrementácií pre každý údaj, platí, že sa vykoná $n{\cdot}nTheta{\cdot}nFi{\cdot}5$ operácií násobenia a $n{\cdot}nTheta{\cdot}nFi{\cdot}(2 + 2{\cdot}size/roStep)$ operácií sčítania, kde n je počet údajov, $size$ je daná maximálna kolmá vzdialenosť údajov detegovanej roviny a $2{\cdot}size/roStep$ je počet inkrementácií polí akumulátora. Táto funkcia je v algoritme volaná s maximálnymi rozsahmi parametrického priestoru iba jedenkrát, a aj to s vhodnou voľbou hodnôt ďalších parametrov. Jej použitie je problematické hlavne z pohľadu pamäťových nárokov, preto bolo nutné pri následných volaniach mať vypočítaný uhol θ a tým pádom zmenšiť rozmer problému. Počet operácií a pamäťové zaťaženie tak vieme pri tejto metóde a rovnako tak aj pri metóde Houghovej transformácie pre priamky v 2D priestore kontrolovať vhodnými nastaveniami hodnôt krokov diskretizácie parametrického priestoru a nastaveniami rozsahov parametrického priestoru.

Zdrojový kód 23 je rovnakého významu ako pri metóde Houghovej transformácie pre priamky v 2D priestore, s kódom pre uhol ϕ naviac.

Zdroj. kód 23 Funkcia `HoughPlane`, obsahujúca implementáciu Houghovej transformácie pre roviny (3. časť).

```
        //hladanie maxima v inkrementačnej matici
        short max = 0;
        //zoznam indexov (fiIndex, thetaIndex, roIndex)
        //odpovedajúcich maximu inkrementačnej matice
        List<Integer[]> indexes = new ArrayList<>();
        for (int thetaIndex = 0; thetaIndex < nTheta; thetaIndex++)
            {
            for (int fiIndex = 0; fiIndex < nFi; fiIndex++) {
                for (int roIndex = 0; roIndex < nRo; roIndex++) {
                    if (H[fiIndex][thetaIndex][roIndex] == max) {
                        indexes.add(new Integer[]{fiIndex,
                            thetaIndex, roIndex});
                    } else if (H[fiIndex][thetaIndex][roIndex] >
                        max) {
                        indexes.clear();
                        max = H[fiIndex][thetaIndex][roIndex];
                        indexes.add(new Integer[]{fiIndex,
                            thetaIndex, roIndex});
                    }
                }
            }
        }

        Set<Double[]> resultPixels = new HashSet<>();

        //získavanie rozsahov
        int minFi = indexes.get(0)[fiIndexP];
        int maxFi = indexes.get(0)[fiIndexP];
        int maxRo = indexes.get(0)[roIndexP];
        int minRo = indexes.get(0)[roIndexP];
```

```
        for (Integer[] indexRow : indexes) {
            if (indexRow[fiIndexP] > maxFi) {
                maxFi = indexRow[fiIndexP];
            } else if (indexRow[fiIndexP] < minFi) {
                minFi = indexRow[fiIndexP];
            }
            if (indexRow[roIndexP] > maxRo) {
                maxRo = indexRow[roIndexP];
            } else if (indexRow[roIndexP] < minRo) {
                minRo = indexRow[roIndexP];
            }

            //získavanie údajov pre výstup
            for (Double[] row : data) {
                double distance = Math.sin(fiArray[indexRow[
                    fiIndexP]]) * Math.cos(thetaArray[indexRow[
                    thetaIndexP]]) * row[xIndex]
                         + Math.sin(fiArray[indexRow[fiIndexP]]) *
                             Math.sin(thetaArray[indexRow[thetaIndexP
                             ]]) * row[yIndex]
                         + Math.cos(fiArray[indexRow[fiIndexP]]) *
                             row[gtuIndex];
                if (distance > lowerDistance && distance <
                    upperDistance) {
                    int lowerIndex = (int) Math.round((distance -
                        size - roArray[0]) / roStep);
                    int upperIndex = (int) Math.round((distance +
                        size - roArray[0]) / roStep);
                    if (lowerIndex <= indexRow[roIndexP] &&
                        upperIndex >= indexRow[roIndexP]) {
                        resultPixels.add(row);
                    }
                }
            }
```

```
        }

        //zápis rozsahov
        List<Double[]> resultRanges = new ArrayList<>();
        if (fiArray[maxFi] - fiArray[minFi] < Math.PI / 2) {
            resultRanges.add(new Double[]{fiArray[minFi], fiArray[
                maxFi]});
        } else {
            resultRanges.add(new Double[]{fiArray[maxFi], fiArray[
                minFi]});
        }
        if (thetaArray[indexes.get(indexes.size() - 1)[thetaIndexP
            ]] - thetaArray[indexes.get(0)[thetaIndexP]] < Math.PI /
             2) {
            resultRanges.add(new Double[]{thetaArray[indexes.get(0)
                [thetaIndexP]], thetaArray[indexes.get(indexes.size
                () - 1)[thetaIndexP]]});
        } else {
            resultRanges.add(new Double[]{thetaArray[indexes.get(
                indexes.size() - 1)[thetaIndexP]], thetaArray[
                indexes.get(0)[thetaIndexP]]});
        }
        resultRanges.add(new Double[]{roArray[minRo], roArray[maxRo
            ]});

        List<Double[]>[] results = new ArrayList[2];
        results[dataIndex] = new ArrayList<>(resultPixels);
        results[rangesIndex] = resultRanges;
        return results;
    }
```

Po tom, čo sa získal výstup z metódy Houghovej transformácie, je pre získanie čistého vzoru spŕšky potrebné ešte odstrániť údaje, ktoré k vzoru spŕšky nepatria. Tieto údaje sa odstraňujú v priestore XY. Najprv sa na údaje použije vyššie

popísaná funkcia `getShapedCore`, ktorá nájde v týchto údajov zhluk, pričom parameter maximálnej veľkosti vzdialenosti od zhluku musí mať dostatočnú hodnotu k prekonaniu medzier spôsobených mriežkou ohniskovej plochy. Následne sa ešte odstránia osamotené údaje funkciou rovnakého mena, ktorej parametrami sú údaje a maximálna veľkosť vzdialenosti od údaja, v ktorej sa má nejaký údaj vyskytovať (Zdrojový kód 24).

Zdroj. kód 24 Funkcia `getShapedCore`, určená pre odstránenie osamotených pixelov.

```
/**
 * odstránia sa určité osamotené pixely
 *
 * @param data - údaje
 * @param gap - maximálna vzdialenost, v ktorej sa hladá nejaký
 * susediaci pixel
 * @return - skupina pixelov
 */
List<Double[]> getShapedCore(List<Double[]> data, double gap) {

    List<Double[]> resultGroup = new ArrayList<>();

    for (int i = 0; i < data.size(); i++) {
        double x = data.get(i)[xIndex];
        double y = data.get(i)[yIndex];
        for (int j = 0; j < data.size(); j++) {
            if (i != j) {
                double nx = data.get(j)[xIndex];
                double ny = data.get(j)[yIndex];
                if (Math.sqrt((x - nx) * (x - nx) + (y - ny) *
                    (y - ny)) < gap) {
                    resultGroup.add(data.get(i));
                    break;
                }
            }
        }
```

```
        }

        return resultGroup;
    }
```

Po tom, čo sa získal čistý vzor spŕšky, sa na základe tohto vzoru vytvára vzor, s ktorým sa bude ďalej pracovať pri aplikácii metód Houghovej transformácie. V návrhu algoritmu boli definované tri vzory - vzor na základe centrálnej línie čistého vzoru spŕšky, vzor na základe pixelov čistého vzoru spŕšky a pridaných pixelov okolo neho, a ich kombinácia. Nájdenie priamky, ktorá by prechádzala centrálnou líniou vzoru, pomocou metódy Houghovej transformácie pre priamky v 2D, už bolo popísané, a vybranie pixelov do určitej kolmej vzdialenosti od nej je potom triviálne. Preto je z tohto pohľadu zaujímavejšia implementácia funkcie `getWrap`, určená pre získanie druhého definovaného vzoru (Zdrojový kód 25).

Zdroj. kód 25 Funkcia `getWrap`, určená pre pridanie pixelov z okolia čistého vzoru.

```
    /**
     * pridanie pixelov okolo jadra spŕšky
     *
     * @param cleanCoreData - údaje pixelov jadra
     * @param rectangleData - údaje pixelov obdĺžnikového
     * ohraničenia spŕšky
     * @param gap - maximálna vzdialenost, v ktorej sa hladá
     * susediaci pixel
     * @return - skupina pixelov
     */
    private List<Double[]> getWrap(List<Double[]> cleanCoreData,
        List<Double[]> rectangleData, double gap) {
        if (gap == 0) {
            return cleanCoreData;
        }
        Set<Double[]> result = new HashSet<>();
        for (Double[] row : cleanCoreData) {
```

```
            double x = row[xIndex];
            double y = row[yIndex];
            for (Double[] row1 : rectangleData) {
                double nx = row1[xIndex];
                double ny = row1[yIndex];
                if (Math.sqrt((x - nx) * (x - nx) + (y - ny) * (y -
                    ny)) < gap) {
                    result.add(row1);
                }
            }
        }
        return new ArrayList<>(result);
    }
```

Parametrami tejto funkcie sú údaje čistého vzoru spŕšky, údaje z okolia čistého vzoru spŕšky a maximálna povolená vzdialenosť pixela od čistého vzoru spŕšky. Pri kombinácii vzorov potom ide iba o zjednotenie množiny pixelov jednotlivých vzorov. Vytvorením finálneho vzoru spŕšky sa končí prvá fáza algoritmu. Druhá fáza začína získaním inicializačných hodnôt rozsahov parametrického priestoru metód Houghovej transformácie. To sa realizuje aplikáciou metód Houghovej transformácie, funkcia Hough (Zdrojový kód 26), na údaje čistého vzoru spŕšky.

Zdroj. kód 26 Funkcia `Hough`, určená pre inicializáciu parametrického priestoru.

```
    /**
     * metóda vybranej Houghovej transformácie pre inicializáciu
     * parametrického priestoru
     *
     * @param data - údaje
     * @param indexHouhTransform - volba metódy Houghovej
     * transformácie
     * @param thetaAngleLine - uhol theta priamky idúcej pozdĺž
     * vzoru spŕšky
     * @return - výstupné údaje Houghovej transformácie pre
     * priamky/roviny
```

```
     */
    private List<Double[]>[] Hough(int indexHoughTransform, List<
        Double[]> data, double thetaAngleLine) {
        if (indexHoughTransform == 0) {
            return HoughLine3D(data, INITIAL_DATA_SIZE_LINE3D,
                xIndex, gtuIndex, yIndex, THETA_STEP_LINE,
                RO_STEP_LINE);
        } else {
            return HoughPlane(data, INITIAL_DATA_SIZE_PLANE,
                thetaAngleLine, FI_STEP_PLANE, THETA_STEP_PLANE,
                RO_STEP_PLANE);
        }
    }
```

Parametrami tejto funkcie sú voľba metódy, údaje čistého vzoru spŕšky a uhol θ, priamky prechádzajúcej centrálnou líniou čistého vzoru spŕšky v priestore XY. Uhol θ je použitý ako parameter pri volaní funkcie Houghovej transformácie pre roviny (Zdrojový kód 27).

Zdroj. kód 27 Funkcia `HoughPlane`, určená pre inicializáciu parametrického priestoru.

```
    /**
     * Houghova transformácia pre roviny pre inicializáciu
     * parametrického priestoru
     *
     * @param data - údaje
     * @param size - maximálna kolmá vzdialenost údajov od
     * detegovanej roviny
     * @param thetaAngleLine - uhol theta priamky idúcej pozdĺž
     * vzoru spŕšky v radiánoch
     * @param fiStep - krok diskretizácie priestoru uhla fi [deg]
     * @param thetaStep - krok diskretizácie priestoru uhla
     * theta [deg]
     * @param roStep - krok diskretizacie priestoru kolmej
     * vzdialenosti roviny od počiatku súradnicovej sústavy
```

```
     * @return - výstupné údaje Houghovej transformácie pre roviny
     */
    private List<Double[]>[] HoughPlane(List<Double[]> data, double
        size, double thetaAngleLine, double fiStep, double
       thetaStep, double roStep) {
        double ro = getMaxDist(data, xIndex, yIndex, gtuIndex);
        //posunutie uhla o PI / 2 tak, aby odpovedal uhlu theta v
           popise roviny
        thetaAngleLine += Math.PI / 2;
        return HoughPlane(data, size, fiStep, -Math.PI / 2, Math.PI
            / 2, thetaStep, thetaAngleLine, thetaAngleLine, roStep,
            -ro, ro);
    }
```

Keďže je tento uhol vo vzťahu s uhlom θ v popise roviny, je oproti nemu posunutý o $\frac{\pi}{2}$, tak ho je možné použiť na zníženie rozmeru problému detegenovania roviny. Pre metódy Houghovej transformácie je pri týchto volaniach potrebné nastaviť čo najmenšie vhodné kroky diskretizácie parametrického priestoru a hodnoty maximálnej kolmej vzdialenosti údajov detegovaných objektov. Výsledné získané rozsahy by mali zabezpečiť, že sa budú následnými aplikáciami metód Houghovej transformácie vyberať údaje stabilne okolo centra spŕšky, a že bude možné použiť dostatočne malé kroky diskretizácie. Ako bolo uvedené pri návrhu algoritmu, jedným z vylepšení je aj voľba maximálnej kolmej vzdialenosti údajov detegovaných objektov prostredníctvom časovej zložky. Je tým umožnené pracovať pri výbere údajov s vhodným časovým oknom. Prepočet hodnoty maximálnej kolmej vzdialenosti údajov zabezpečuje funkcia `getTimeSize` (Zdrojový kód 28).

Zdroj. kód 28 Funkcia `getTimeSize`, ktorá vracia hodnotu maximálnej kolmej vzdialenosti údajov od roviny/priamky vzhľadom na zadanú veľkosť časovej zložky.

```
    /**
     *
     * @param indexHoughTransform - volba metódy Houghovej
     * transformácie
```

```
     * @param ranges - rozsahy priamky/roviny
     * (fi, ro / fi, theta, ro)
     * @param size - časová zložka maximálnej kolmej vzdialenosti
     * údajov od detegovanej priamky/roviny
     * @return - vracia hodnotu maximálnej kolmej vzdialenosti
     * údajov od roviny/priamky vzhladom na zadanú velkost
     * časovej zložky
     */
    double getTimeSize(int indexHoughTransform, List<Double[]>
        ranges, double size) {
         if (indexHoughTransform == 0) {
             double thetaAngle1 = (ranges.get(thetaIndexL2)[
                 lowerRange] + ranges.get(thetaIndexL2)[upperRange])
                 / 2;
             if (ranges.get(thetaIndexL2)[lowerRange] > ranges.get(
                 thetaIndexL2)[upperRange]) {
                  thetaAngle1 += Math.PI / 2;
             }
             double thetaAngle2 = (ranges.get(thetaIndexL1)[
                 lowerRange] + ranges.get(thetaIndexL1)[upperRange])
                 / 2;
             if (ranges.get(thetaIndexL1)[lowerRange] > ranges.get(
                 thetaIndexL1)[upperRange]) {
                  thetaAngle2 += Math.PI / 2;
             }
             //size * Math.sin(fiAngle1) pretože v HoughLine3D je
             //HoughLine2D(x, Gtu),size * Math.cos(fiAngle2)
             //pretože v HoughLine3D je HoughLine2D(Gtu, y)
             double size1 = Math.abs(size * Math.sin(thetaAngle1));
             double size2 = Math.abs(size * Math.cos(thetaAngle2));
             return size1 > size2 ? size1 : size2;
         } else {
             double fiAngle = (ranges.get(fiIndexP)[lowerRange] +
                 ranges.get(fiIndexP)[upperRange]) / 2;
             if (ranges.get(fiIndexP)[lowerRange] > ranges.get(
```

```
                fiIndexP)[upperRange]) {
                 fiAngle += Math.PI / 2;
            }
            return Math.abs(size * Math.cos(fiAngle));
        }
    }
```

Jej parametrami sú voľba metódy, inicializačné rozsahy metód a hodnota veľkosti časovej zložky vektora maximálnej kolmej vzdialenosti údajov detegovaných objektov. Funkcia vracia hodnotu maximálnej kolmej vzdialenosti údajov detegovaných objektov, ktorá sa má pre danú metódu použiť, aby sa zabezpečilo použitie obmedzenia na časové okno. Po získaní potrebných hodnôt parametrov metód Houghovej transformácie je ešte potrebné zatriediť údaje pixelov vzoru spŕšky do skupín, tak ako je to popísané pri návrhu algoritmu, a následne sa v cykle (Zdrojový kód 29) na tieto údaje aplikuje vybraná metóda Houghovej transformácie.

Zdroj. kód 29 Cyklus rozpoznávania údajov.

```
        //výpočet
        //outputData - množina výstupných údajov
        List<Double[]> outputData = new ArrayList<>();
        //threshold - prahová hodnota počtu signálov
        //thresholdLowerLimit - spodná hranica pre prahovú hodnotu
            počtu signálov
        while (--threshold >= thresholdLowerLimit) {
            //sortedInputData - množina skupín vstupných údajov
            //data[rangesIndex] - množina rozsahov parametrického
                priestoru
            //size - maximálna kolmá vzdialenost údajov
            //indexHoughTransform - volba metódy
            data = Hough(indexHoughTransform, sortedInputData[top
                --], data[rangesIndex], size);
            printer.printInfo("Run " + threshold + " :", data[
                dataIndex]);
```

```
            outputData.addAll(data[dataIndex]);
        }
```

Výstup údajov tohto cyklu je buď konečným riešením problému rozpoznania vzoru spŕšky, alebo sa na údaje môžu použiť zadefinované algoritmy pre filtráciu, popísané vo 8. kroku návrhu algoritmu. Implementáciou algoritmu filtrácie z hľadiska času je funkcia `timeSeqFilter` (Zdrojový kód 30).

Zdroj. kód 30 Funkcia `timeSeqFilter`, určená pre filtráciu z hľadiska času.

```
    /**
     * filtrácia údajov na základe ich postupnosti na pixeli
     * z hladiska času
     *
     * @param data - údaje
     * @param dropSeqThresholdType - volba typu prahovej hodnoty
     * počtu signálov pre zahodenie postupnosti údajov
     * @param dropSeqThreshold - prahová hodnota počtu signálov pre
     * zahodenie postupnosti údajov
     * @param singleDataThreshold - prahová hodnota počtu signálov
     * pre postupnost s dĺžkou 1
     * @return - prefiltrované údaje
     */
    private List<Double[]> timeSeqFilter(List<Double[]> data, int
        dropSeqThresholdType, int dropSeqThreshold, int
        singleDataThreshold) {
        //ak nie sú dodané údaje, tak sa vracia prázdny zoznam
        if (data.isEmpty()) {
            return new ArrayList<>();
        }

        if (dropSeqThresholdType == 1) {
            dropSeqThreshold += patternThreshold;
        }
```

```
        //údaje sa časovo usporiadajú, od najvyššej hodnoty po
            najnižšiu
        data.sort(new DataComparator(new Integer[]{gtuIndex}));

        //vytvorí sa retazec, ktorý indikuje, či sú časové medzery
            medzi údajmi
        //menšie alebo rovné danej hodnote časovej medzery, a či je
             splnená podmienka prahu počtu signálov
        int[] neighboursMap = new int[data.size() - 1];
        int max = data.get(0)[totalCountIndex].intValue();
        List<Integer> maxCount = new ArrayList<>();
        maxCount.add(0);
        for (int i = 1; i < data.size(); i++) {
            if (data.get(i - 1)[gtuIndex] - data.get(i)[gtuIndex]
                <= 1) {
                neighboursMap[i - 1] = 1;
            }
            if (data.get(i)[totalCountIndex] > max) {
                max = data.get(i)[totalCountIndex].intValue();
                maxCount.clear();
                maxCount.add(i);
            } else if (data.get(i)[totalCountIndex] == max) {
                maxCount.add(i);
            }
        }

        if (max <= dropSeqThreshold) {
            return new ArrayList<>();
        }

        List<Double[]> maxSeq;
        if (maxCount.size() == 1) {
            maxSeq = getNeighbours(data, neighboursMap, maxCount.
                get(0));
        } else {
```

```
        List<List<Double[]>> sequences = new ArrayList<>();
        for (int index : maxCount) {
            List<Double[]> tempList = getNeighbours(data,
                neighboursMap, index);
            tempList.sort(new DataComparator(new Integer[]{
                totalCountIndex}));
            sequences.add(tempList);
        }

        boolean changed = false;
        maxSeq = sequences.get(0);
        for (int i = 1; i < sequences.size(); i++) {
            List<Double[]> nextSeq = sequences.get(i);
            int length = maxSeq.size() < nextSeq.size() ?
                maxSeq.size() : nextSeq.size();
            for (int j = 1; j < length; j++) {
                if (nextSeq.get(j)[totalCountIndex] > maxSeq.
                    get(j)[totalCountIndex]) {
                    maxSeq = nextSeq;
                    changed = true;
                    break;
                } else if (nextSeq.get(j)[totalCountIndex] <
                    maxSeq.get(j)[totalCountIndex]) {
                    changed = true;
                    break;
                }
            }
            if (!changed) {
                maxSeq = maxSeq.size() > nextSeq.size() ?
                    maxSeq : nextSeq;
            }
            changed = false;
        }
    }
```

```
        if (maxSeq.size() == 1 && !(max > singleDataThreshold)) {
            return new ArrayList<>();
        }

        return maxSeq;
    }
```

Parametrami tejto funkcie sú údaje pixela, typ prahu počtu signálov pre zahodenie postupnosti údajov, prah počtu signálov pre zahodenie postupnosti údajov a prah počtu signálov pre postupnosť údajov dĺžky 1. Výstupom sú prefiltrované údaje pixela. Pre získanie časových postupností údajov sa využíva funkcia `getNeighbours` (Zdrojový kód 31).

Zdroj. kód 31 Funkcia `getNeighbours`, určená pre získanie časových postupností údajov.

```
    /**
     *
     * @param data - údaje
     * @param neighboursMap - mapa časovo susediacich údajov
     * @param index - index údaja
     * @return - skupina časovo susediacich údajov
     */
    private List<Double[]> getNeighbours(List<Double[]> data, int[]
        neighboursMap, int index) {
        List<Double[]> result = new ArrayList<>();
        result.add(data.get(index));
        for (int i = index; i < neighboursMap.length; i++) {
            if (neighboursMap[i] == 1) {
                result.add(data.get(i + 1));
            } else {
                break;
            }
        }
        for (int i = index - 1; i > - 1; i--) {
            if (neighboursMap[i] == 1) {
```

```
                result.add(data.get(i));
            } else {
                break;
            }
        }
        return result;
    }
```

Parametrami funkcie sú údaje pixela, mapa časovo susediacich údajov a index údaja v zozname údajov pixela. Implementáciou algoritmu filtrácie z hľadiska počtu signálov je funkcia `signalSeqFilter` (Zdrojový kód 32).

Zdroj. kód 32 Funkcia `signalSeqFilter`, určená pre filtráciu z hľadiska počtu signálov.

```
    /**
     * filtrácia údajov na základe ich postupnosti na pixeli
     * z hladiska počtu signálov
     *
     * @param data - údaje
     * @param dropSeqThresholdType - volba typu prahovej
     * hodnoty  počtu signálov pre zahodenie postupnosti údajov
     * @param dropSeqThreshold - prahová hodnota počtu signálov
     * pre zahodenie postupnosti údajov
     * @param cutSeqThreshold - prahová hodnota počtu signálov
     * pre odrezanie postupnosti údajov
     * @param singleDataThreshold - prahová hodnota počtu
     * signálov pre postupnost s dĺžkou 1
     * @return - prefiltrované údaje
     */
    private List<Double[]> signalSeqFilter(List<Double[]> data, int
        dropSeqThresholdType, int dropSeqThreshold, int
       cutSeqThreshold, int singleDataThreshold) {

        //ak nie sú dodané údaje, tak sa vracia prázdny zoznam
        if (data.isEmpty()) {
            return new ArrayList<>();
```

```
        }

        if (dropSeqThresholdType == 1) {
            dropSeqThreshold += patternThreshold;
        }

        //hladá sa maximálna hodnota počtu signálov
        double max = 0;
        List<Double[]> maxList = new ArrayList<>();
        for (Double[] row : data) {
            if (row[totalCountIndex] > max) {
                max = row[totalCountIndex];
                maxList.clear();
                maxList.add(row);
            } else if (row[totalCountIndex] == max) {
                maxList.add(row);
            }
        }

        //ak údaj s maximálnym počtom signálov nie je nad prahom
        //počtu signálov pre zahodenie postupnosti údajov
        if (max <= dropSeqThreshold) {
            return new ArrayList<>();
        }

        //údaje sa časovo usporiadajú, od najvyššej hodnoty po
            najnižšiu
        data.sort(new DataComparator(new Integer[]{gtuIndex}));

        //údaje sa rozdelia do dvoch skupín
        //- v prvej sú údaje skoršie ako údaj s maximálnou hodnotou
            počtu signálov
        //- v druhej je údaj s maximálnou hodnotou počtu signálov a
            údaje neskoršie ako
        //údaj s maximálnou hodnotou počtu signálov
```

```
        Double[] maxRow = maxList.get(0);
        List<Double[]> rightList = new ArrayList<>();
        List<Double[]> leftList = new ArrayList<>();
        for (Double[] row : data) {
            if (row[gtuIndex] < maxRow[gtuIndex]) {
                leftList.add(row);
            } else {
                rightList.add(row);
            }
        }

        //zoznam s výstupom
        List<Double[]> resultList = new ArrayList<>();

        if (!leftList.isEmpty()) {
            if (maxRow[totalCountIndex] > cutSeqThreshold) {
                resultList.add(leftList.get(0));
                if (leftList.get(0)[totalCountIndex] >
                    cutSeqThreshold) {
                    //inak sa pridá prvý údaj do výsledného zoznamu
                        a zvyšné sa pridávajú iba ak je splnená
                       podmienka klesajúcej postupnosti
                    for (int i = 1; i < leftList.size(); i++) {
                        if (leftList.get(i)[totalCountIndex] == max
                            || leftList.get(i)[totalCountIndex] <
                           leftList.get(i - 1)[totalCountIndex]) {
                            resultList.add(leftList.get(i));
                            if (leftList.get(i)[totalCountIndex] <=
                                cutSeqThreshold) {
                                break;
                            }
                        } else {
                            break;
                        }
                    }
```

```
            }
        }
    }

    //ak má druhá skupina iba 1 údaj, tak sa pridá do
    //výsledného zoznamu
    if (rightList.size() < 2) {
        resultList.addAll(rightList);
    } else {
        //inak sa pridá údaj s maximálnou hodnotou počtu
        //signálov do výsledného zoznamu a zvyšné
        //sa pridávajú iba ak je splnená podmienka klesajúcej
            postupnosti
        resultList.add(rightList.get(rightList.size() - 1));
        if (rightList.get(rightList.size() - 1)[totalCountIndex
            ] > cutSeqThreshold) {
            if (rightList.get(rightList.size() - 1)[
                totalCountIndex] > cutSeqThreshold) {
                for (int i = rightList.size() - 2; i > -1; i--)
                    {
                    if (rightList.get(i)[totalCountIndex] ==
                        max || rightList.get(i)[totalCountIndex]
                        < rightList.get(i + 1)[totalCountIndex
                        ]) {
                        resultList.add(rightList.get(i));
                        if (rightList.get(i)[totalCountIndex]
                            <= cutSeqThreshold) {
                            break;
                        }
                    } else {
                        break;
                    }
                }
            }
        }
```

```
        }

        if (resultList.size() == 1 && resultList.get(0)[
            totalCountIndex] <= singleDataThreshold) {
             return new ArrayList<>();
        }
        return resultList;
    }
```

Parametrami tejto funkcie sú údaje pixela, typ prahu počtu signálov pre zahodenie postupnosti údajov, prah počtu signálov pre zahodenie postupnosti údajov, prah počtu signálov pre odrezanie postupnosti údajov a prah počtu signálov pre postupnosť údajov dĺžky 1. Výstupom sú prefiltrované údaje pixela. Aplikácia tejto implementácie algoritmu, pre rozpoznávanie spŕšok, na údajoch zvolenej udalosti, Obrázok 5 – 31 vľavo, bola vizualizovaná v priestore XY. Použitá bola metóda Houghovej transformácie pre priamky v 3D priestore a zobrazené sú tieto fázy:

Určenie prahu počtu signálov pre vzor spŕšky Na Obrázku 5 – 31 vpravo sú zobrazené údaje nad vypočítaným prahom počtu signálov pre vzor spŕšky. Na týchto údajoch prebiehalo hľadanie vzoru spŕšky.

Nájdenie zhluku pixelov Na Obrázku 5 – 32 vľavo je zobrazený najväčší nájdený zhluk pixelov.

Pridanie údajov z okolia nájdeného zhluku pixelov Na Obrázku 5 – 32 vpravo sú zobrazené pixely vybraných údajov, ktoré sú vstupnými údajmi pre metódu Houghovej transformácie pre priamky v 3D priestore.

Nájdenie zvyšku obrazu spŕšky Na Obrázku 5 – 33 vľavo sú zobrazené pixely výstupných údajov metódy Houghovej transformácie pre priamky v 3D priestore, použitej pre nájdenie zvyšku obrazu spŕšky.

Odstránenie nadbytočných údajov Na Obrázku 5 – 33 vpravo sú zobrazené pi-

xely čistého vzoru spŕšky, ktorý vznikol po odstránení nadbytočných údajov.

Vytvorenie vzoru spŕšky Na Obrázku 5 – 34 vľavo sú zobrazené pixely vzoru, ktorý vznikol z čistého vzoru spŕšky a pridaných pixelov okolo neho do zadanej maximálnej povolenej vzdialenosti. Na Obrázku 5 – 34 vpravo sú zobrazené pixely vzoru, ktorý vznikol na základe centrálnej línie a na základe dĺžky čistého vzoru spŕšky. Na Obrázku 5 – 35 vľavo sú zobrazené pixely finálneho vzoru spŕšky, ktorý vznikol kombináciou vzorov na Obrázku 5 – 34.

Rozpoznávanie údajov (prahová hodnota počtu signálov 0) Údaje vzoru spŕšky boli rozdelené do skupín, pri nastavení prahovej hodnoty počtu signálov pre výber údajov na hodnotu 0, Tabuľka 5 – 4, a bola na nich aplikovaná metóda Houghovej transformácie pre priamky v 3D priestore. Počet vybraných údajov z jednotlivých skupín je uvedený v Tabuľke 5 – 5 a celková suma údajov v Tabuľke 5 – 6. Na Obrázku 5 – 35 vpravo sú zobrazené pixely výsledných údajov.

Filtrácia údajov Na Obrázku 5 – 36 vľavo sú zobrazené pixely výsledných údajov, ktoré prešli definovanými filtračnými podmienkami. Celková suma týchto údajov je uvedená v Tabuľke 5 – 7.

Rozpoznávanie údajov (prahová hodnota počtu signálov 1) Metóda Houghovej transformácie pre priamky v 3D priestore bola aplikovaná len na skupiny 1 až 5, uvedené v Tabuľke 5 – 4, čo odpovedá nastaveniu prahovej hodnoty počtu signálov pre výber údajov na hodnotu 1, a preto aj vybrané údaje odpovedajú údajom v Tabuľke 5 – 5 bez skupiny 0. Celková suma údajov je uvedená v Tabuľke 5 – 8. Pixely výsledných údajov sú zobrazené na Obrázku 5 – 36 vpravo.

Tabuľka 5 – 4 Rozdelenie údajov do skupín

Skupina	Pôvod v spŕške	Pôvod v pozadí
5	196	10
4	40	75
3	64	385
2	87	1502
1	119	4193
0	62	8079

Tabuľka 5 – 5 Výber údajov zo skupín

Skupina	Pôvod v spŕške	Pôvod v pozadí
5	192	0
4	33	1
3	52	3
2	61	16
1	73	50
0	36	77

Tabuľka 5 – 6 Výsledok výberu údajov (prahová hodnota počtu signálov 0)

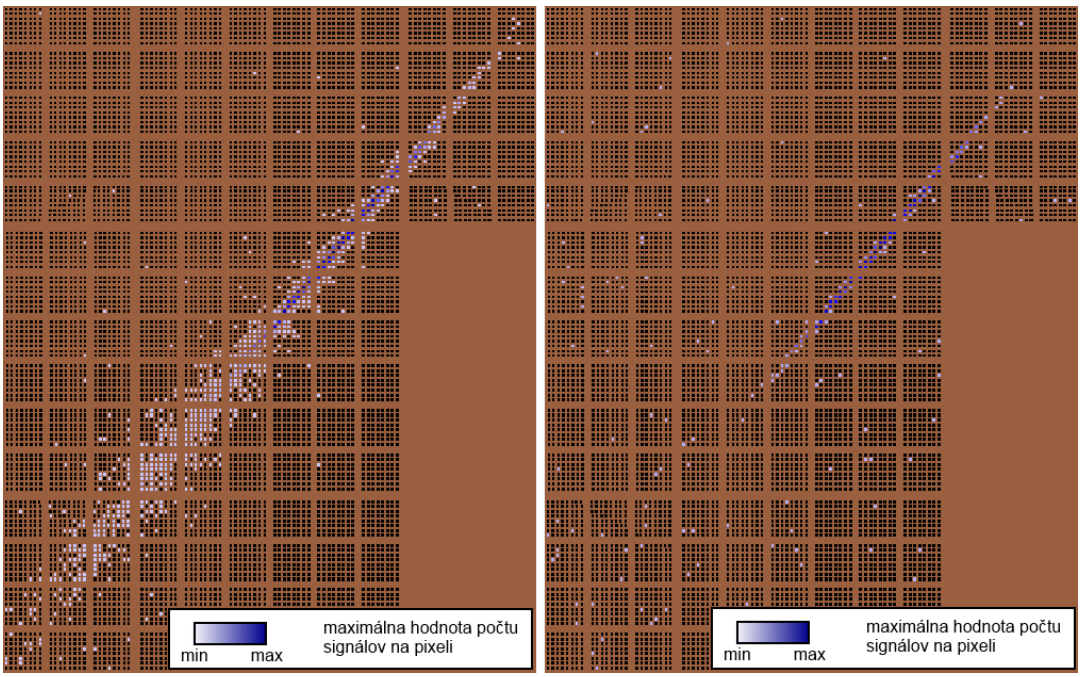
	Pôvod v spŕške	Pôvod v pozadí
Spolu	447	147

Tabuľka 5 – 7 Výsledok filtrácie údajov

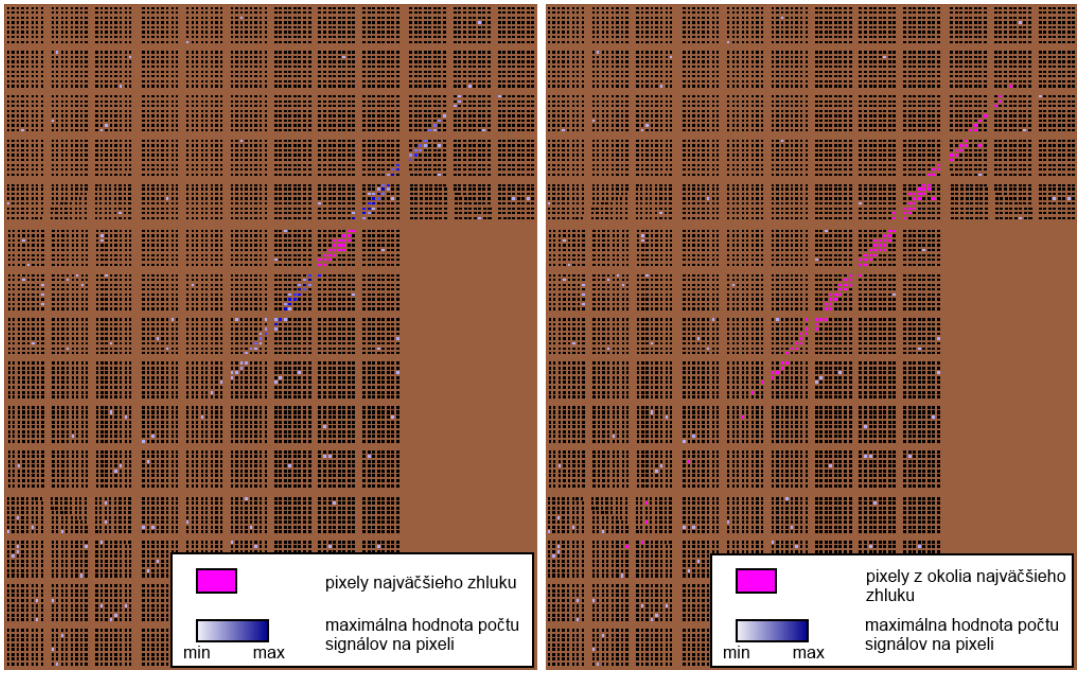
	Pôvod v spŕške	Pôvod v pozadí
Spolu	399	85

Tabuľka 5 – 8 Výsledok výberu údajov (prahová hodnota počtu signálov 1)

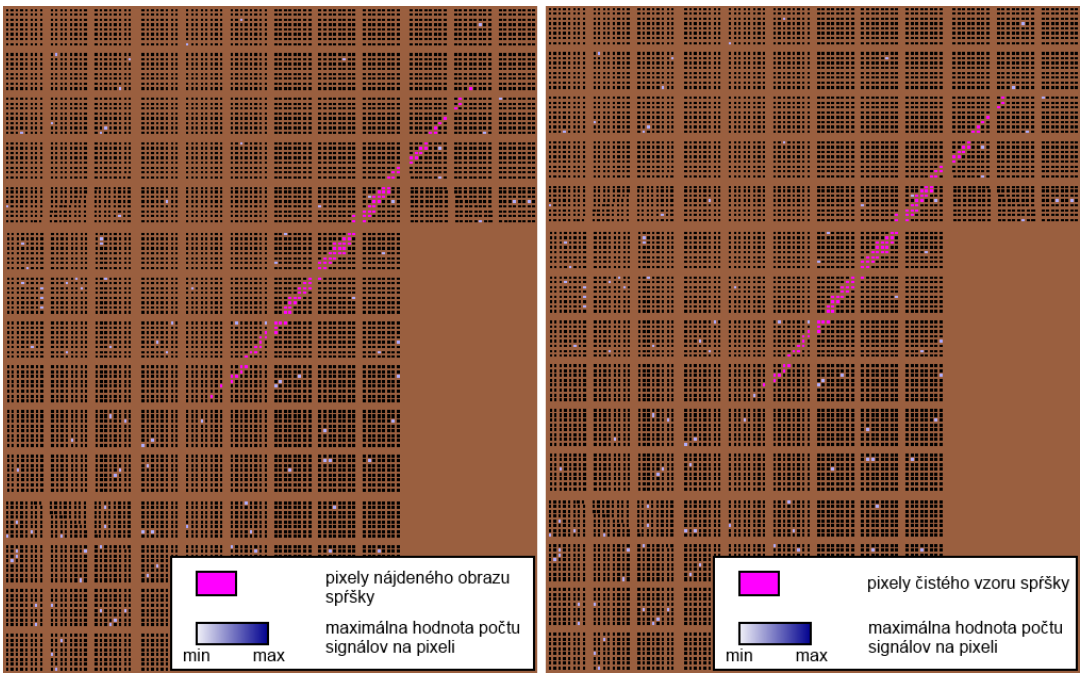
	Pôvod v spŕške	Pôvod v pozadí
Spolu	411	70



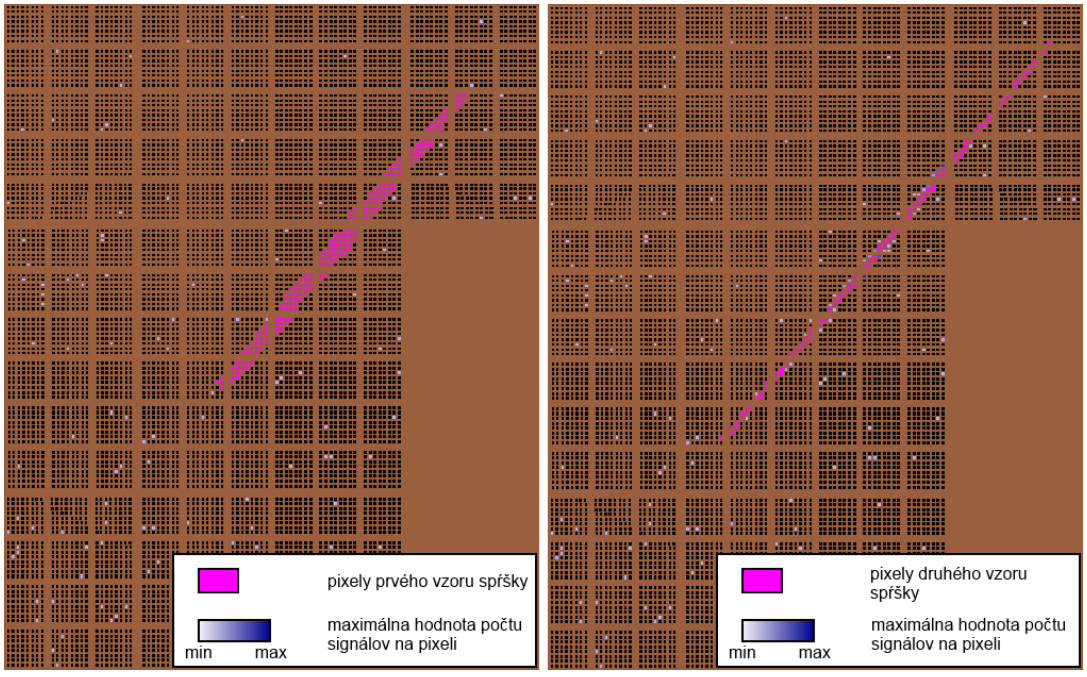
Obr. 5 – 31 Vľavo: Zobrazenie spŕšky v priestore XY. Vpravo: Zobrazenie údajov udalosti, nad vypočítanou prahovou hodnotou počtu signálov, v priestore XY.



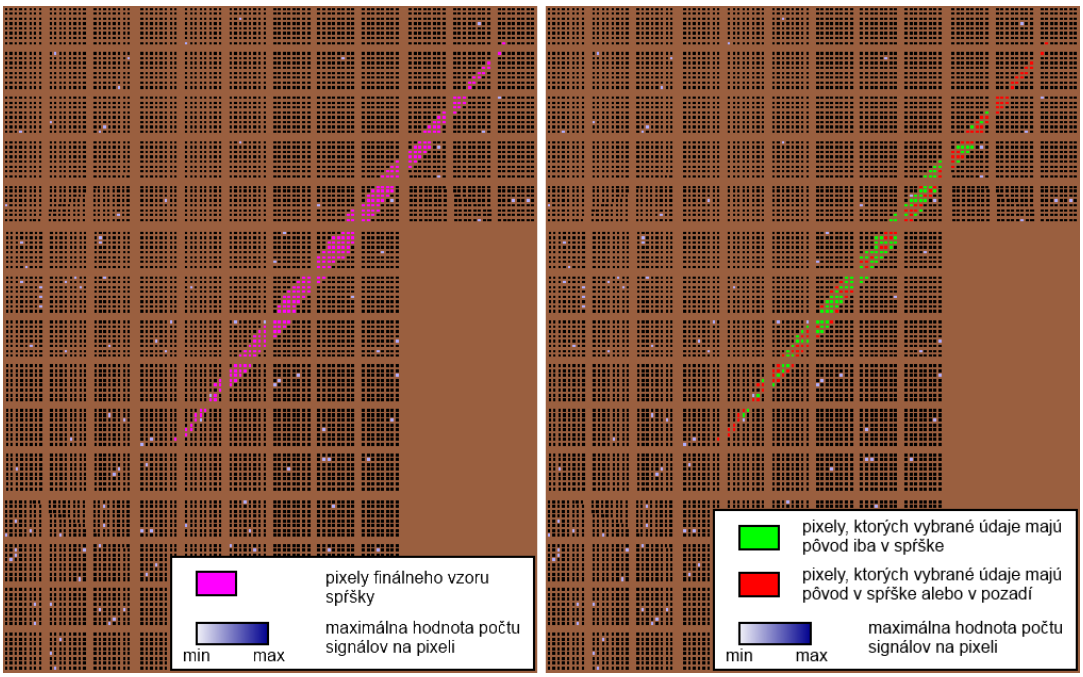
Obr. 5 – 32 Vľavo: Zobrazenie najväčšieho nájdeného zhluku pixelov spŕšky. Vpravo: Zobrazenie najväčšieho nájdeného zhluku pixelov spŕšky a pixelov z vypočítaného okolia zhluku.



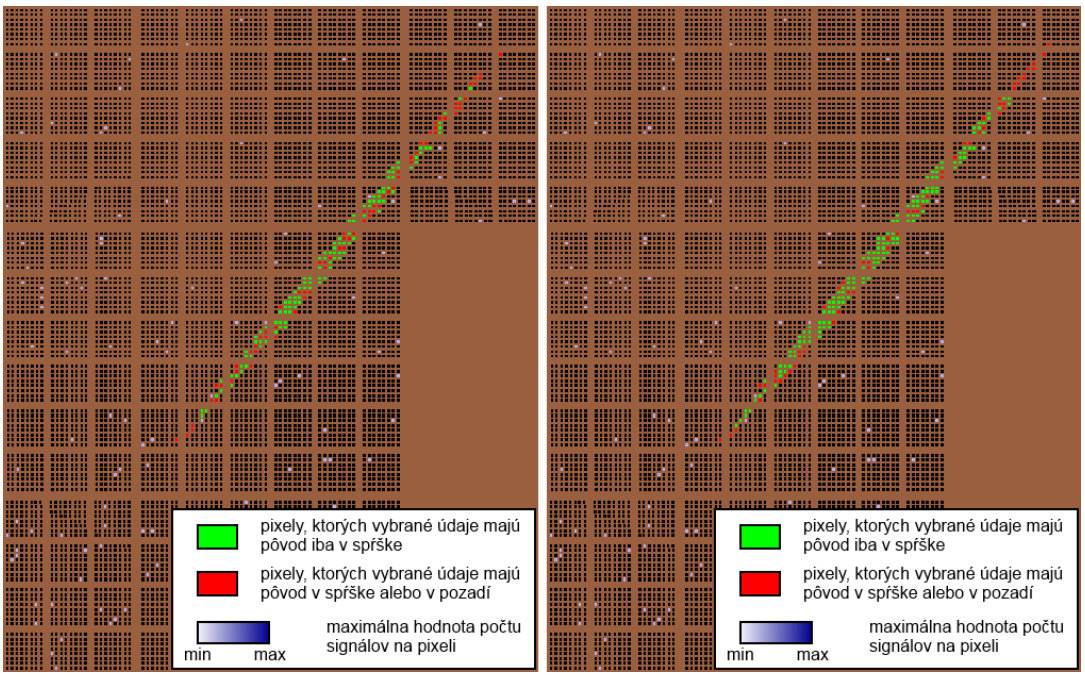
Obr. 5 – 33 Vľavo: Zobrazenie pixelov nájdeného obrazu spŕšky. Vpravo: Zobrazenie pixelov čistého vzoru spŕšky.



Obr. 5 – 34 Vľavo: Zobrazenie pixelov prvého vzoru spŕšky. Vpravo: Zobrazenie pixelov druhého vzoru spŕšky.



Obr. 5 – 35 Vľavo: Zobrazenie pixelov finálneho vzoru spŕšky. Vpravo: Zobrazenie údajov výsledného vzoru spŕšky v priestore XY, pri prahovej hodnote počtu signálov pre údaje 0.



Obr. 5 – 36 Vľavo: Zobrazenie údajov výsledného vzoru spŕšky v priestore XY, pri prahovej hodnote počtu signálov pre údaje 0 a po filtrácii. Vpravo: Zobrazenie údajov výsledného vzoru spŕšky v priestore XY, pri prahovej hodnote počtu signálov pre údaje 1.

5.4.4 Výsledky a analýza

Rovnako ako pri prvom algoritme, bola implementácia druhého algoritmu aplikovaná na simulované údaje udalostí a hľadalo sa najlepšie nastavenie parametrov algoritmu z hľadiska štatistiky γ_{68} pre separačný uhol k skutočnému zenitovému uhlu spŕšky. Po nájdení vhodného nastavenia parametrov algoritmu sa vykonalo porovnanie štatistiky γ_{68}, pre separačný uhol k skutočnému zenitovému uhlu spŕšky, s algoritmom PWISE a s prvým algoritmom, pri nastavení UV pozadia udalostí na hodnotu 0.42 pe/(px μs). Ako ukazuje porovnanie na Obrázku 5 – 37, pri počte zrekonštruovaných udalostí zobrazených na Obrázku 5 – 38 a Obrázku 5 – 39, opäť boli dosiahnuté o čosi lepšie výsledky v porovnaní s algoritmom PWISE. V porovnaní s prvým algoritmom k väčšiemu prielomu nedošlo, avšak očividne sa podarilo odstrániť problém pri rozpoznávaní na udalostiach so zenitovým uhlom 65 až 75 stupňov. Rovnako bola overená výkonnosť algoritmu pri narastajúcom UV pozadí pre simulované údaje udalostí, pre zenitový uhol 30, a jej výsledky v porovnaní s prvým algoritmom sú zobrazené na Obrázku 5 – 40. Ukázalo sa, že výsledky sú pri rôznych UV pozadiach porovnateľné s prvým algoritmom, avšak zdá sa, že nastal značný posun v množstve zrekonštruovaných udalostí (Obrázok 5 – 41 a Obrázok 5 – 42).

Na základe analýzy výsledkov a správania sa algoritmu pri rôznych podmienkach boli formulované nasledujúce závery:

Rozpoznanie vzoru spŕšky

- Rozpoznávanie vzoru spŕšky nevykazovalo nejaké značné nedostatky, napriek tomu by si vyžadovalo o čosi precíznejší prístup a zrejme aj používanie iba jednej vybranej metódy Houghovej transformácie, ktorá by na to bola najvhodnejšia. Ak sa pozrieme na počet spoločných udalostí s algoritmom PWISE (Obrázok 5 – 38), pre zenitový uhol 30, tak vidíme, že je tu rozdiel 50 udalostí. Tento rozdiel mohli spôsobiť spŕšky, ktorých stopa

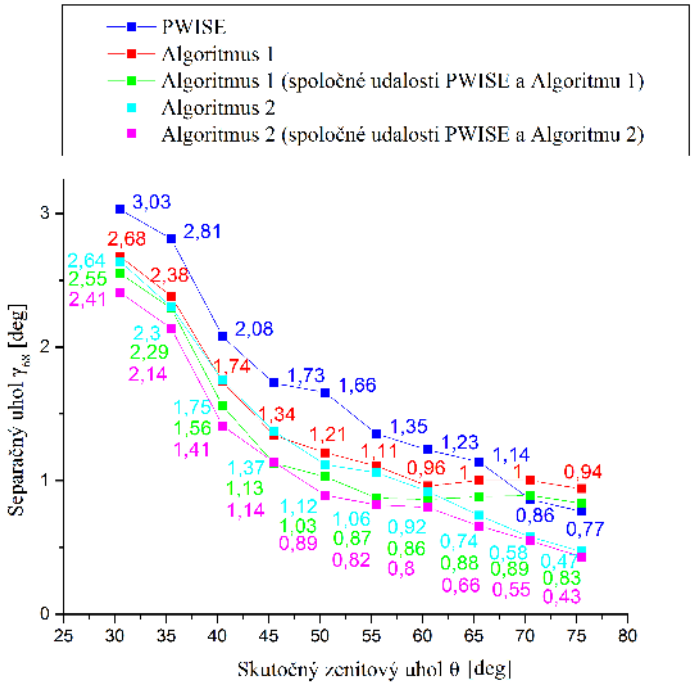
značne zanikla na mriežke ohniskovej plochy, alebo udalosti, pri ktorých algoritmus jednoducho vybral málo údajov pre uhlovú rekonštrukciu. Ak by šlo v podstatnej miere o prvý prípad, tak pri použití kritéria algoritmu PWISE, prahovej hodnoty počtu signálov pre výber pixelov s hodnotou 8, by bolo zrejme možné vybrať dostatok pixelov a následne ich údajov. Preto je možné uvažovať o budúcom vylepšení hľadania zhlukov pixelov práve takýmto spôsobom - prvotným určením pixelov, ktoré by sa po nastavení dostatočne vysokej prahovej hodnoty počtu signálov, s najväčšou pravdepodobnosťou mali v nájdenom zhluku vyskytnúť.

Rozpoznanie údajov spŕšky

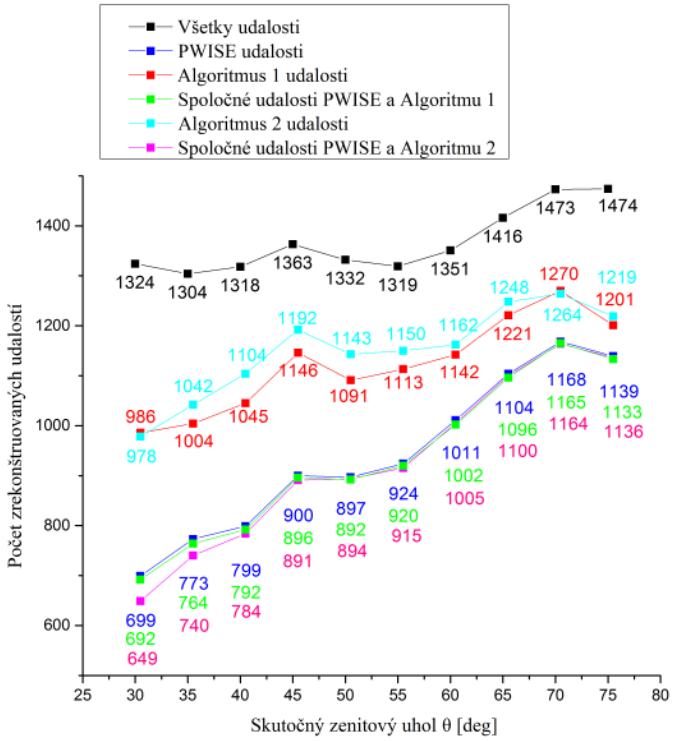
- Dosiahnutie čo najlepších výsledkov si, rovnako ako pri prvom algoritme, vyžadovalo vhodné zvýšenie prahovej hodnoty počtu signálov pre údaje. V porovnaní s prvým algoritmom bolo ale možné použiť nižšie hodnoty.
- V kľúčovej miere závisia výsledky od inicializácie parametrických priestorov metód Houghovej transformácie. Bol pri nej využitý čistý vzor spŕšky, ktorý ku každému svojmu pixelu obsahuje iba údaj s maximálnou hodnotou počtu signálu, preto je do budúcna potrebné overiť vplyv pridania aj ďalších údajov pixelov s hodnotami nad prahom počtu signálov, vypočítaným pri hľadaní vzoru. Rovnako by bolo dobré vyskúšať pridanie nastavenia prahovej hodnoty počtu signálov pre inicializáciu parametrických priestorov, momentálne je inicializácia závislá od vypočítanej hodnoty počtu signálov pri rozpoznaní vzoru, z dôvodu, aby bolo možné korektne porovnať vylepšenia druhého algoritmu oproti prvému, a taktiež využiť vlastnosť, ktorá sa ukázala pri prvom algoritme, a to že najlepšie výsledky uhlovej analýzy boli dosiahnuté pri vyšších prahových hodnotách počtu signálov pre údaje a vzor.
- Pri dosiahnutí najlepšieho výsledku, pri nastavení štandardného UV po-

zadia, boli použité aj definované filtračné algoritmy. Pre zenitový uhol 30 bol pri rovnakom nastavení algoritmu ako pri najlepšom dosiahnutom výsledku, ale bez filtrácie, dosiahnutý výsledok separačného uhla 2.81 stupňov pri 1010 zrekonštruovaných udalostiach. To je v porovnaní s najlepším výsledkom separačného uhla 2.64 stupňov pri 978 zrekonštruovaných udalostiach o čosi horší výsledok. Posunutie výsledkov separačného uhla nadol by tak mohlo byť v budúcnosti závislé na vylepšení filtračných algoritmov.

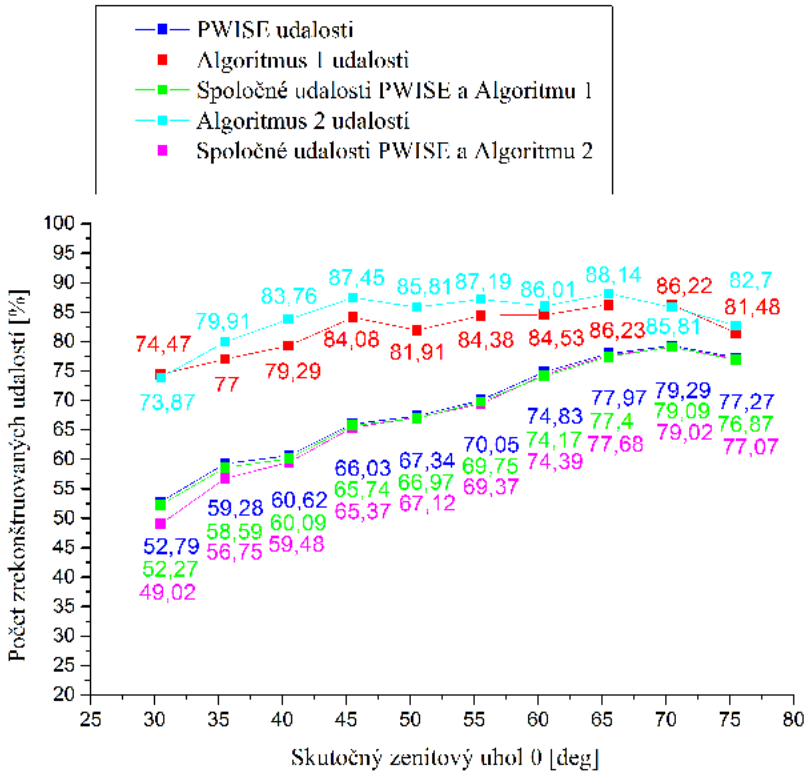
- Pri najnižšom používanom UV pozadí sa rovnako dobre osvedčili obe používané metódy Houghovej transformácie.
- Najlepšie výsledky uhlovej analýzy boli dosiahnuté pri použití prvého definovaného vzoru spŕšky.
- Pri narastajúcom UV pozadí bolo nutné nastaviť čo najstriktnejšiu inicializáciu parametrických priestorov. Jej vyššie navrhnuté úpravy by mohli viesť k lepším výsledkom pri narastajúcom UV pozadí, než aké boli momentálne dosiahnuté.
- Spracovanie údajov po skupinách a kontrola rozsahov parametrických priestorov metód Houghovej transformácie prispeli k zvýšeniu stability algoritmu, čo sa pri porovnaní s prvým algoritmom, najviac prejavilo pri porovnaní výsledkov z rozpoznávania udalostí pre vyššie zenitové uhly a pri porovnaní počtu zrekonštruovaných udalostí s rastúcim UV pozadím.



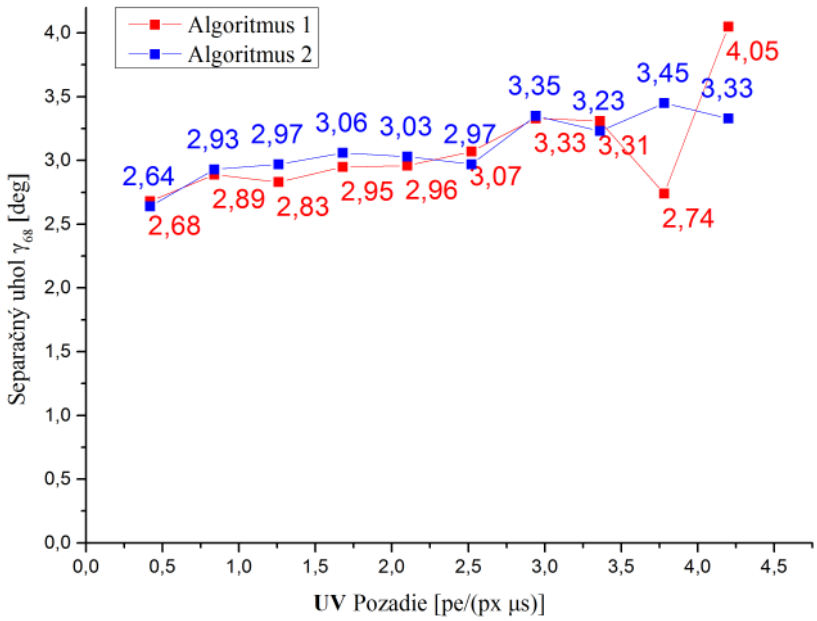
Obr. 5 – 37 Graf Skutočný zenitový uhol - Separačný uhol γ_{68} (Algoritmus 2).



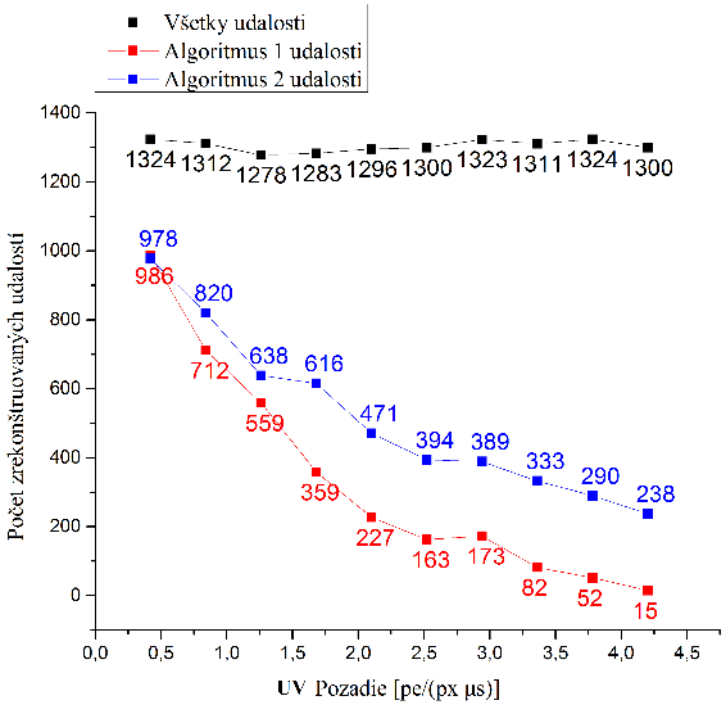
Obr. 5 – 38 Graf Skutočný zenitový uhol - Počet zrekonštruovaných udalostí (Algoritmus 2).



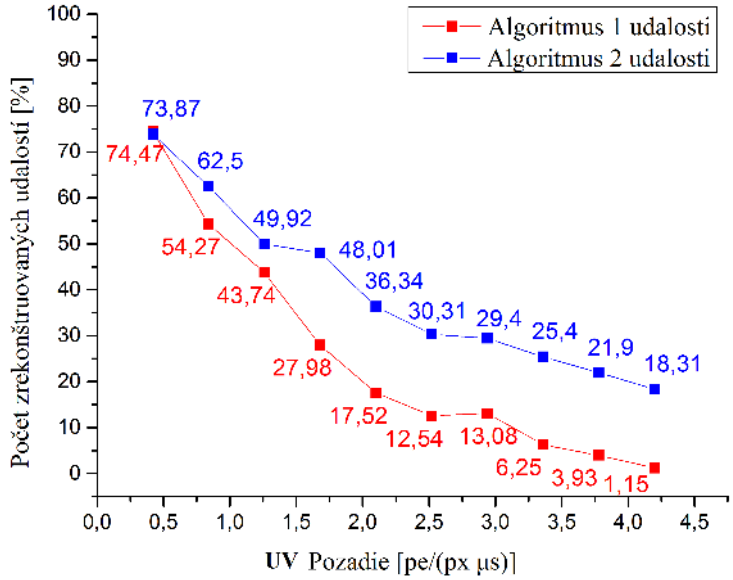
Obr. 5 – 39 Graf Skutočný zenitový uhol - Počet zrekonštruovaných udalostí [%] (Algoritmus 2).



Obr. 5 – 40 Graf UV pozadie - Separačný uhol γ_{68} (Algoritmus 2, zenitový uhol 30 stupňov).



Obr. 5 – 41 Graf UV pozadie - Počet zrekonštruovaných udalostí (Algoritmus 2, zenitový uhol 30 stupňov).



Obr. 5 – 42 Graf UV pozadie - Počet zrekonštruovaných udalostí [%] (Algoritmus 2, zenitový uhol 30 stupňov).

6 Záver

Cieľom tejto práce bol návrh algoritmu pre rozpoznávanie UHECR spŕšok, so základom vo využití Houghovej transformácie, pre spŕšky generované simulačnou časťou rámca ESAF. Výsledkom práce sú dva algoritmy využívajúce dohromady tri rôzne metódy Houghovej transformácie. Výkonnosť týchto algoritmov bola overená na simulovaných spŕškach pre rozsah zenitového uhla 30-75 stupňov, pri UV pozadí 0.42 pe/(px μs), a porovnaná s výkonnosťou algoritmu PWISE. Taktiež bola overená výkonnosť oboch algoritmov na simulovaných spŕškach pre zenitový uhol s hodnotou 30 stupňov, pri rozsahu UV pozadia 0.42-4.2 pe/(px μs). Keďže aktuálne nie sú známe úspešné pokusy o overenie výkonnosti iných algoritmov, vyvinutých v rámci kolaborácie JEM-EUSO, pri rastúcom UV pozadí, nebolo možné výsledky porovnať inak než pre obe vyvinuté algoritmy navzájom.

Porovnanie výsledkov separačného uhla γ_{68} prvého algoritmu ukázalo, v porovnaní s algoritmom PWISE, zlepšenie skoro na celom rozsahu zenitového uhla, pri rovnako väčšom počte zrekonštruovaných udalostí. Oblasť výsledkov, v ktorej nedošlo k očakávaným výsledkom, sa stala jednou z hlavných otázok pri návrhu druhého algoritmu. Ukázalo sa, že optimálne nastavenia oboch prahových hodnôt počtu signálov algoritmu boli späté s hodnotou, pri ktorej sa oddeľuje signál spŕšok od signálu UV pozadia. Pri nastavení parametrov algoritmu, pri ktorom bol dosiahnutý najlepší výsledok pri UV pozadí 0.42 pe/(px μs), sa pre overenie výkonnosti pri rastúcom UV pozadí menili len hodnoty prahových hodnôt počtu signálov. Výsledky ukázali, že výkonnosť pri rastúcom UV pozadí je v porovnaní s výsledkami pre UV pozadie 0.42 pe/(px μs) vcelku udržateľná, avšak len za cenu prudko klesajúceho počtu zrekonštruovaných udalostí s rastom hodnôt UV pozadia.

Porovnanie výsledkov separačného uhla γ_{68} druhého algoritmu ukázalo v porovnaní s algoritmom PWISE ešte o niečo výraznejšie zlepšenie ako pri prvom algoritme, a to na celom rozsahu zenitového uhla. V porovnaní s prvým algoritmom nebolo

zlepšenie nijak výrazné, avšak tentoraz sa už podarilo dosiahnuť želané výsledky na, pre prvý algoritmus, kritickom rozsahu zenitového uhla. Hodnoty počtov zrekonštruovaných udalostí sa v porovnaní pohybovali okolo alebo nad hodnotami počtu zrekonštruovaných udalostí prvého algoritmu. Optimálne nastavenie druhého algoritmu, pri ktorom boli dosiahnuté najlepšie výsledky, bolo pri porovnateľných častiach podobné ako pri prvom algoritme s tým, že bolo možné použiť nižšie hodnoty prahu počtu signálov pre údaje. Pre overenie výkonnosti druhého algoritmu pri rastúcom UV pozadí už ale bolo nutné nastavenia algoritmu trochu zmeniť. V porovnaní s prvým algoritmom neboli vo väčšine prípadov dosiahnuté, pri rastúcom UV pozadí, lepšie výsledky separačného uhla γ_{68}, avšak stalo sa tak za vyššieho počtu zrekonštruovaných udalostí druhého algoritmu. Napriek tomu by si tieto výsledky zaslúžili dôslednejšiu analýzu vzhľadom na nastavenia algoritmu a prípadne potrebné vylepšenia. Rast počtu zrekonštruovaných udalostí oproti prvému algoritmu poukazuje na správnosť ideí, uplatnených pri návrhu druhého algoritmu.

V celkovom výsledku tak boli úspešne vyvinuté dva algoritmy pre rozpoznávanie UHECR spŕšok, so základom vo využití Houghovej transformácie, ktoré by mali spĺňať nároky kolaborácie, a prostredníctvom ktorých bolo možné prvé nám známe overenie možností rozpoznávania, pri rastúcom UV pozadí, v rámci kolaborácie JEM-EUSO.

Literatúra

SANTANGELO, A. - PICOZZA, P. - EBISUZAKI, T. 2013. Status of the JEM-EUSO Mission. V: ADAMS, J.H. a iní. The JEM-EUSO Mission: Contributions to the ICRC 2013. Rio de Janeiro : The JEM-EUSO Collaboration, 2013. 7 s.

GAJDOŠ, E. 2014. Algoritmy pre rozpoznávanie obrazcov pri štúdiu kozmického žiarenia ultravysokých energií v rámci JEM-EUSO experimentu : diplomová práca. Košice : Technická univerzita v Košiciach, 2014.

BLASCHKE, F. 2009. Analýza korelovaných sprškek kosmického záření: diplomová práca. Opava: SLU FPF, 2009.

STAROŇ, M. 2013. Algoritmy pre rozpoznávanie obrazcov pri štúdiu kozmického žiarenia ultravysokých energií v rámci JEM-EUSO experiment : diplomová práca. Košice : Technická univerzita v Košiciach, 2013.

KAJINO, F. a iní. 2013. The JEM-EUSO instruments. V: ADAMS, J.H. a iní. The JEM-EUSO Mission: Contributions to the ICRC 2013. Rio de Janeiro : The JEM-EUSO Collaboration, 2013. 11 s.

RICCI, M. - FRANCESCHI, M. A. – NAPOLITANO, T. 2011. The JEM-EUSO Focal Surface Mechanical Structure. V: ADAMS JR., J.H. a iní. The JEM-EUSO Mission: Status and Prospects in 2011. Beijing : The JEM-EUSO Collaboration, 2011. 35 s.

KAWASAKI, Y. a iní. 2011. The focal surface of the JEM-EUSO instrument: angular reconstruction. V:Astrophysics and Space Sciences Transactions. ISSN: 1810-6536, 2011, Ročník 7, Číslo 2, s. 167-169. Doi: 10.5194/astra-7-167-2011.

MACARONE, M.C a iní. 2010. The JEM-EUSO Purple Book: Report on the Phase A Study 2010.

MERNIK, T. 2013. ESAF-Simulation of the EUSO-Balloon. V: ADAMS, J.H. a iní.

The JEM-EUSO Mission: Contributions to the ICRC 2013. Rio de Janeiro : The JEM-EUSO Collaboration, 2013. 51 s.

BERAT, C. a iní. 2010. Full simulation of space-based extensive air showers detectors with ESAF. V: Astroparticle Physics. ISSN: 09276505, 2010, Ročník 33, Číslo 4, s. 221-247. DOI: 10.1016/j.astropartphys.2010.02.005.

BIKTEMEROVA, S. a iní. 2013. Performances of JEM-EUSO: Angular reconstruction. V: Experimental Astronomy. ISSN: 0922-6435, 2013, Ročník 409. DOI: 10.1007/s10686-013-9371-0.

MERNIK, T. a iní. 2013. Simulating the JEM-EUSO Mission: Expected Reconstruction Performance. V: ADAMS, J.H. a iní. The JEM-EUSO Mission: Contributions to the ICRC 2013. Rio de Janeiro : The JEM-EUSO Collaboration, 2013. 55 s.

GUZMAN, A. a iní. 2013. The Peak and Window Searching Technique for the EUSO Simulation and Analysis Framework: Impact on the Angular Reconstruction of EAS. V: Journal of Physics: Conference Series. ISSN: 1742-6596, 2013, Ročník 409. DOI:10.1088/1742-6596/409/1/012104.

FENU, F. 2013. A simulation study of the JEM-EUSO mission for the detection of ultra-high energy cosmic rays : dizertačná práca. Tübingen : Universität Tübingen, 2013.

BIKTEMEROVA, S. - GONCHAR, M. – SHARAKIN, S. 2013. Pattern recognition and direction reconstruction for JEM-EUSO experiment. V: ADAMS, J.H. a iní. The JEM-EUSO Mission: Contributions to the ICRC 2013. Rio de Janeiro : The JEM-EUSO Collaboration, 2013. 63 s.

HOUGH, P. V. C. 1962. Method and means for recognizing complex patterns : U.S. Patent 3 069 654. Dec. 18, 1962.

DUDA, R. O. – HART, P. E. 1972. Use of the Hough Transformation to Detect

Lines and Curves in Pictures. V: Communications of the ACM. ISSN:0001-0782, 1972, Ročník 15, s. 11–15.

FERNANDES, L. A. F. - OLIVEIRA, M.M. 2012. A general framework for subspace detection in unordered multidimensional data. V: Pattern Recognition. ISSN: 0031-3203, 2012, Ročník 45, Číslo 9, s. 3566–3579.

BALLARD, D.H. 1981.Generalizing the Houghtransform to detectarbitraryshapes. V: Pattern Recognition. ISSN: 0031-3203, 1981, Ročník 13, Číslo 2, s. 111–122.

ROSENFELD, A. 1969. Picture Processing by Computer. New York: Academic, 1969. s. 335–336.

VRÁBEL, M. 2015. Programovanie modulov pre softvérový rámec ESAF experimentu JEM-EUSO zameraného na detekciu častíc s ultravysokou energiou : diplomová práca. Košice : Technická univerzita v Košiciach, 2013.

Zoznam príloh

Príloha A Zdrojové kódy implementácie navrhnutých algoritmov

Príloha B Používateľská príručka

Príloha C Systémová príručka

Príloha D Odborný článok v anglickom jazyku

Príloha E CD médium

Príloha A

Zdrojové kódy implementácie navrhnutých algoritmov

Navrhnuté algoritmy boli implementované v jazyku Java a sú súčasťou nasledujúcich tried:

- Trieda Algorithm1 - implementácia prvého navrhnutého algoritmu.
- Trieda Algorithm2 - implementácia druhého navrhnutého algoritmu.

Trieda Algorithm1

```
public class Algorithm1 {

    //objekt s údajmi udalosti
    private final EventDataHolder dataHolder;
    //objekt pre výpisy
    private final Printer printer;
    //výstupné údaje
    private List<Double[]> printData;
    //údaje jednotlivých fáz rozpoznávania
    private final StageData stageData;

    //indexy údajov
    private final int totalCountIndex;
    private final int xIndex;
    private final int yIndex;
    private final int idIndex;
    private final int gtuIndex;
    private final int signalCountIndex;

    //definovanie indexov
    private final int dataIndex = 0;
```

```
    private final int rangesIndex = 1;
    private final int thetaIndexL = 0;
    private final int roIndexL = 1;

    //krok diskretizácie priestoru hodnôt uhla theta v h.t. pre
       priamky
    private final double THETA_STEP;
    //krok diskretizácie priestoru hodnôt kolmej vzdialenosti ro v
       h.t. pre priamky
    private final double RO_STEP;

    /**
     * konštruktor
     *
     * @param dataHolder - objekt s údajmi udalosti
     * @param columns - objekt s indexmi na údaje udalosti
     */
    public Algorithm1(EventDataHolder dataHolder, Columns columns)
       {

        this.dataHolder = dataHolder;

        printer = new Printer(columns);

        //definovanie fáz rozpoznávania
        stageData = new StageData(new String[]{
            "Line pattern",
            "Result"});

        //indexy na údaje
        totalCountIndex = columns.getIndex("totalCount");
        xIndex = columns.getIndex("x");
        yIndex = columns.getIndex("y");
        idIndex = columns.getIndex("id");
        gtuIndex = columns.getIndex("gtu");
```

```
        signalCountIndex = columns.getIndex("signalCount");

        //načítanie parametrov z konfiguračného súboru
        THETA_STEP = Double.valueOf(AlgorithmProperties.
            getAlgorithm1Prop("THETA_STEP"));
        RO_STEP = Double.valueOf(AlgorithmProperties.
            getAlgorithm1Prop("RO_STEP"));

    }

    /**
     *
     * @return - zoznam údajov udalosti
     */
    private List<Double[]> getEventData() {
        List<Double[]> eventData = new ArrayList<>();
        for (int i = 0; i < dataHolder.getValuesSize(); i++) {
            eventData.add(dataHolder.getValues(i));
        }
        return eventData;
    }

    /**
     *
     * @param patternThreshold - prahová hodnota počtu signálov
     * pre vzor
     * @param dataThreshold - prahová hodnota počtu signálov
     * pre údaje
     * @param line2DSize - maximálna kolmá vzdialenost od
     * detegovanej priamky v priestore XY
     * @param line3DSize - maximálna kolmá vzdialenost od
     * detegovanej priamky v priestore XGtu a GtuY
     * @return - identifikátory pixelov výstupných údajov
     */
```

```
public Integer[] run(int patternThreshold, int dataThreshold,
    double line2DSize, double line3DSize) {
     line2DSize = line2DSize / 10;
     line3DSize = line3DSize / 10;
     stageData.clearStageData();

     //údaje nad prahovou hodnotou počtu signálov pre vzor
     List<Double[]> patternData = new ArrayList<>();
     //zvyšné údaje nad prahovou hodnotou počtu
     //signálov pre údaje
     List<Double[]> additionalData = new ArrayList<>();
     for (Double[] row : getEventData()) {
         if (row[totalCountIndex] > patternThreshold) {
             patternData.add(row);
         } else if (row[totalCountIndex] > dataThreshold) {
             additionalData.add(row);
         }
     }
     patternData.sort(new DataComparator(new Integer[]{idIndex,
        totalCountIndex}));

     //údaje maximálnych hodnôt počtu signálov na danom pixeli
     List<Double[]> uniqueData = new ArrayList<>();
     //všetky údaje
     List<Double[]> allData = new ArrayList<>();
     int currentId = patternData.get(0)[idIndex].intValue();
     int rowCount = 0;

     //výber údajov maximálnych hodnôt počtu signálov na danom
        pixeli
     while (rowCount < patternData.size()) {
         int lastId = currentId;
         uniqueData.add(patternData.get(rowCount));
         while (currentId == lastId) {
             if (!(++rowCount < patternData.size())) {
```

```
                break;
            }
            currentId = patternData.get(rowCount)[idIndex].
                intValue();
        }
    }

    //pridajú sa údaje
    allData.addAll(patternData);
    allData.addAll(additionalData);

    //vypočíta sa posunutie údajov pre minimalizovanie
    //pamätového zataženia pri metóde Houghovej transformácie
    double[] vectorXYGtu = getCenter3D(allData);
    //posunutie údajov v priestore XYGtu
    shift(allData, -vectorXYGtu[0], -vectorXYGtu[1], -
        vectorXYGtu[2]);

    //hladanie vzoru
    List<Double[]> data = HoughLine2D(uniqueData, line2DSize,
        xIndex, yIndex, THETA_STEP, RO_STEP)[dataIndex];
    //uloženie pixelov výsledných údajov vzoru
    stageData.saveStageData(0, getIndexes(data).toArray(new
        Integer[data.size()]));

    //usporiadanie údajov podla identifikátorov
    allData.sort(new DataComparator(new Integer[]{idIndex}));
    //identifikátory pixelov vzoru
    Set<Integer> indexes = getIndexes(data);

    //vstupné údaje pre rozpoznávanie údajov
    List<Double[]> inputData = new ArrayList<>();
    currentId = allData.get(0)[idIndex].intValue();
    rowCount = 0;
```

```
//výber vstupných údajov
while (rowCount < allData.size()) {
    int lastId = currentId;
    if (indexes.contains(currentId)) {
        while (currentId == lastId) {
            inputData.add(allData.get(rowCount));
            if (!(++rowCount < allData.size())) {
                break;
            }
            currentId = allData.get(rowCount)[idIndex].
                intValue();
        }
    } else {
        while (currentId == lastId) {
            if (!(++rowCount < allData.size())) {
                break;
            }
            currentId = allData.get(rowCount)[idIndex].
                intValue();
        }
    }
}

//výpis informácie o vstupných údajoch
printer.printInfo("Data :", inputData);
//rozpoznávanie údajov
data = HoughLine3D(inputData, line3DSize, xIndex, gtuIndex,
    yIndex, THETA_STEP, RO_STEP)[dataIndex];
//uloženie pixelov výsledných údajov rozpoznávania
stageData.saveStageData(1, returnData(data));

//spätné posunutie výsledných údajov v priestore XYGtu
shift(data, vectorXYGtu[0], vectorXYGtu[1], vectorXYGtu[2])
    ;
```

```
        //výstupné údaje
        printData = data;
        //výstupné súbory
        printer.write(data, "output0", 0);
        printer.write(data, "output1", 1);
        //informácia o výstupných údajoch
        printer.printInfo("Result :", data);
        return returnData(data);

    }

    /**
     * Houghova transformácia pre priamky v 2D priestore
     *
     * @param data - údaje
     * @param size - maximálna povolená kolmá vzdialenost údajov od
     * detegovanej priamky
     * @param indexFirstDim - index prvej dimenzie 2D priestoru
     * @param indexSecondDim - index druhej dimenzie 2D priestoru
     * @param thetaStep - krok diskretizácie priestoru uhla fi
     * [deg, priestor od -90 do 90 stupňov]
     * @param roStep - krok diskretizácie priestoru kolmej
     * vzdialenosti priamky od počiatku súradnicovej sústavy
     * @return - výstupné údaje Houghovej transformácie pre priamky
     * v 2D priestore
     */
    private List<Double[]>[] HoughLine2D(List<Double[]> data,
        double size, int indexFirstDim, int indexSecondDim, double
        thetaStep, double roStep) {
        //maximálna absolútna kolmá vzdialenost údajov od počiatku
            súradnicovej sústavy
        double ro = getMaxDist(data, indexFirstDim, indexSecondDim)
            ;
        return HoughLine2D(data, size, indexFirstDim,
            indexSecondDim, thetaStep, -Math.PI / 2, Math.PI / 2,
```

```
            roStep, -ro, ro);
    }

    /**
     * Houghova transformácia pre priamky v 3D priestore
     *
     * @param data - údaje
     * @param size - maximálna povolená kolmá vzdialenost údajov od
     * detegovanej priamky
     * @param xIndex - index prvej dimenzie 3D priestoru
     * @param yIndex - index druhej dimenzie 3D priestoru
     * @param zIndex - index tretej dimenzie 3D priestoru
     * @param thetaStep - krok diskretizácie priestoru uhla fi
     * [deg, priestor od -90 do 90 stupňov]
     * @param roStep - krok diskretizácie priestoru kolmej
     * vzdialenosti priamky od počiatku súradnicovej sústavy
     * @return - výstupné údaje Houghovej transformácie pre priamky
     * v 3D priestore
     */
    private List<Double[]>[] HoughLine3D(List<Double[]> data,
        double size, int xIndex, int yIndex, int zIndex, double
        thetaStep, double roStep) {
        List<Double[]>[] firstData = HoughLine2D(data, size, xIndex
            , yIndex, thetaStep, roStep);
        List<Double[]>[] result = HoughLine2D(firstData[dataIndex],
             size, yIndex, zIndex, thetaStep, roStep);
        result[rangesIndex].add(firstData[rangesIndex].get(
            thetaIndexL));
        result[rangesIndex].add(firstData[rangesIndex].get(roIndexL
            ));
        return result;
    }

    /**
     * Houghova transformácia pre priamky v 2D priestore
```

```
     *
     * @param data - údaje
     * @param size - maximálna povolená kolmá vzdialenost údajov od
     * detegovanej priamky
     * @param indexFirstDim - index prvej dimenzie 2D priestoru
     * @param indexSecondDim - index druhej dimenzie 2D priestoru
     * @param thetaStep - krok diskretizácie priestoru uhla
     * theta [deg]
     * @param thetaStart - začiatok priestoru uhla theta
     * v radiánoch (v intervale od -PI/2 do PI/2)
     * @param thetaEnd - koniec priestoru uhla theta v radiánoch
     * (v intervale od -PI/2 do PI/2)
     * @param roStep - krok diskretizácie priestoru kolmej
     * vzdialenosti priamky od počiatku súradnicovej sústavy
     * @param roStart - začiatok priestoru kolmej vzdialenosti
     * priamky od počiatku súradnicovej sústavy
     * @param roEnd - koniec priestoru kolmej vzdialenosti priamky
     * od počiatku súradnicovej sústavy
     * @return - výstupné údaje Houghovej transformácie pre priamky
     * v 2D priestore
     */
    private List<Double[]>[] HoughLine2D(List<Double[]> data,
        double size, Integer indexFirstDim, Integer indexSecondDim,
        double thetaStep, double thetaStart, double thetaEnd, double
         roStep, double roStart, double roEnd) {

        //definovanie pola hodnôt kolmej vzdialenosti priamky
        //od počiatku súradnicovej sústavy
        int nRo = (int) (Math.ceil(Math.abs(roEnd - roStart) /
            roStep)) + 1;
        double[] roArray = new double[nRo];
        int index = 0;
        double roValue = roStart;
        while (index < nRo) {
            roArray[index++] = roValue;
```

```
            roValue += roStep;
        }

        //definovanie pola hodnôt uhla theta
        thetaStep = thetaStep * Math.PI / 180;
        double[] thetaArray;
        int nTheta;
        if (thetaStart <= thetaEnd) {
            nTheta = (int) (Math.abs(thetaEnd - thetaStart) /
               thetaStep) + 1;
            if ((nTheta - 1) * thetaStep + thetaStart < thetaEnd &&
                nTheta * thetaStep + thetaStart < Math.PI / 2) {
                nTheta++;
            }
            thetaArray = new double[nTheta];
            index = 0;
            double thetaValue = thetaStart;
            while (index < nTheta) {
                thetaArray[index++] = thetaValue;
                thetaValue += thetaStep;
            }
        } else {
            int nTheta1 = (int) (Math.abs(thetaEnd - (-Math.PI / 2)
               ) / thetaStep) + 1;
            int nTheta2 = (int) (Math.abs(Math.PI / 2 - thetaStart)
                / thetaStep) + 1;
            if ((nTheta1 - 1) * thetaStep + (-Math.PI / 2) <
               thetaEnd && nTheta1 * thetaStep + (-Math.PI / 2) <
               Math.PI / 2) {
                nTheta1++;
            }
            nTheta = nTheta2 + nTheta1;
            thetaArray = new double[nTheta];
            index = 0;
            double thetaValue = -Math.PI / 2;
```

```
        while (index < nTheta1) {
            thetaArray[index++] = thetaValue;
            thetaValue += thetaStep;
        }
        thetaValue = thetaStart;
        while (index < nTheta) {
            thetaArray[index++] = thetaValue;
            thetaValue += thetaStep;
        }
    }

    //dvojrozmerná inkrementačná matica
    short[][] A = new short[nTheta][nRo];
    double lowerDistance = roArray[0] - 1.1 * size;
    double upperDistance = roArray[nRo - 1] + 1.1 * size;

    //výpočet kolmých vzdialeností priamky
    //od počiatku súradnicovej sústavy
    for (Double[] row : data) {
        for (int thetaIndex = 0; thetaIndex < nTheta;
           thetaIndex++) {
            double distance = row[indexFirstDim] * Math.cos(
               thetaArray[thetaIndex])
                    + (row[indexSecondDim]) * Math.sin(
                       thetaArray[thetaIndex]);
            //vypočíta sa horný a dolný index pre hodnoty
            //kolmých vzdialeností priamky
            //od počiatku súradnicovej sústavy, vzhladom na
               maximálnu povolenú kolmú vzdialenost údajov
            //od detegovanej priamky a vzorkovaciu frekvenciu
               pola kolmých vzdialeností
            if (distance > lowerDistance && distance <
               upperDistance) {
                int lowerIndex = (int) Math.round((distance -
                   size - roArray[0]) / roStep);
```

```
                int upperIndex = (int) Math.round((distance +
                    size - roArray[0]) / roStep);
                //indexy musia byt v povolenom rozsahu
                if (lowerIndex < 0) {
                    lowerIndex = 0;
                }
                if (upperIndex > nRo - 1) {
                    upperIndex = nRo - 1;
                }
                //inkrementácia polí priamok, v ktorých sa údaj
                    vyskytuje
                for (int roIndex = lowerIndex; roIndex <
                    upperIndex + 1; roIndex++) {
                    A[thetaIndex][roIndex] += row[
                        totalCountIndex];
                }
            }
        }
    }

    Set<Double[]> resultData = new HashSet<>();
    List<Double[]> resultRanges = new ArrayList<>();

    short max = 0;
    //zoznam indexov (fiIndex, roIndex) odpovedajúcich maximu
        inkrementačnej matice
    List<Integer[]> indexes = new ArrayList<>();

    for (int thetaIndex = 0; thetaIndex < nTheta; thetaIndex++)
        {
        for (int roIndex = 0; roIndex < nRo; roIndex++) {
            if (A[thetaIndex][roIndex] == max) {
                indexes.add(new Integer[]{thetaIndex, roIndex})
                    ;
            } else if (A[thetaIndex][roIndex] > max) {
```

```
                indexes.clear();
                max = A[thetaIndex][roIndex];
                indexes.add(new Integer[]{thetaIndex, roIndex})
                    ;
            }
        }
    }

    int maxRo = indexes.get(0)[roIndexL];
    int minRo = indexes.get(0)[roIndexL];
    //získavanie výsledných údajov a popisov priamok,
    //odpovedajúcich maximu inkrementačnej matice
    for (Integer[] indexRow : indexes) {
        if (indexRow[roIndexL] > maxRo) {
            maxRo = indexRow[roIndexL];
        } else if (indexRow[roIndexL] < minRo) {
            minRo = indexRow[roIndexL];
        }
        for (Double[] row : data) {
            double distance = row[indexFirstDim] * Math.cos(
                thetaArray[indexRow[thetaIndexL]])
                     + row[indexSecondDim] * Math.sin(thetaArray
                         [indexRow[thetaIndexL]]);
            if (distance > lowerDistance && distance <
                upperDistance) {
                int lowerIndex = (int) Math.round((distance -
                    size - roArray[0]) / roStep);
                int upperIndex = (int) Math.round((distance +
                    size - roArray[0]) / roStep);
                if (lowerIndex <= indexRow[roIndexL] &&
                    upperIndex >= indexRow[roIndexL]) {
                    resultData.add(row);
                }
            }
        }
```

```
        }

        //výstupné hodnoty
        List<Double[]>[] result = new ArrayList[2];

        //zápis rozsahov parametrického priestoru, definovaných na
            základe výstupných údajov
        if (Math.abs(thetaArray[indexes.get(indexes.size() - 1)[
            thetaIndexL]] - thetaArray[indexes.get(0)[thetaIndexL]])
             < Math.PI / 2) {
             resultRanges.add(new Double[]{thetaArray[indexes.get(0)
                [thetaIndexL]], thetaArray[indexes.get(indexes.size
                () - 1)[thetaIndexL]]});
        } else {
             resultRanges.add(new Double[]{thetaArray[indexes.get(
                indexes.size() - 1)[thetaIndexL]], thetaArray[
                indexes.get(0)[thetaIndexL]]});
        }
        resultRanges.add(new Double[]{roArray[minRo], roArray[maxRo
            ]});

        result[dataIndex] = new ArrayList<>(resultData);
        result[rangesIndex] = resultRanges;

        return result;
    }

    /**
     *
     * @param data - údaje
     * @return - vektor so začiatkom v počiatku súradnicovej
     * sústavy a koncom v centre údajov 3D priestoru XYGtu
     */
    private double[] getCenter3D(List<Double[]> data) {
        double maxX = data.get(0)[xIndex];
```

```
        double minX = data.get(0)[xIndex];
        double maxY = data.get(0)[yIndex];
        double minY = data.get(0)[yIndex];
        double maxGtu = data.get(0)[gtuIndex];
        double minGtu = data.get(0)[gtuIndex];
        for (int i = 1; i < data.size(); i++) {
            if (maxX < data.get(i)[xIndex]) {
                maxX = data.get(i)[xIndex];
            } else if (minX > data.get(i)[xIndex]) {
                minX = data.get(i)[xIndex];
            }
            if (maxY < data.get(i)[yIndex]) {
                maxY = data.get(i)[yIndex];
            } else if (minY > data.get(i)[yIndex]) {
                minY = data.get(i)[yIndex];
            }
            if (maxGtu < data.get(i)[gtuIndex]) {
                maxGtu = data.get(i)[gtuIndex];
            } else if (minGtu > data.get(i)[gtuIndex]) {
                minGtu = data.get(i)[gtuIndex];
            }
        }
        return new double[]{(minX + maxX) / 2, (minY + maxY) / 2, (
            minGtu + maxGtu) / 2};
    }

    /**
     *
     * posun v 3D priestore XYgtu
     *
     * @param data - údaje
     * @param x - x-ová zložka vektora posunutia
     * @param y - y-ova zložka vektora posunutia
     * @param gtu - časová zložka vektora posunutia
     */
```

```
    private void shift(List<Double[]> data, double x, double y,
       double gtu) {
        for (Double[] row : data) {
            row[xIndex] = row[xIndex] + x;
            row[yIndex] = row[yIndex] + y;
            row[gtuIndex] = row[gtuIndex] + gtu;
        }
    }

     /**
     *
     * @param data - údaje
     * @param index - index dimenzie priestoru
     * @return - maximálna absolútna vzdialenost údajov od počiatku
     * súradnicovej sústavy v jednorozmernom priestore
     */
    private double getAbsMax(List<Double[]> data, Integer index) {
        double max = Math.abs(data.get(0)[index]);
        for (int i = 1; i < data.size(); i++) {
            double value = Math.abs(data.get(i)[index]);
            if (max < value) {
                max = value;
            }
        }
        return max;
    }

    /**
     *
     * @param data - údaje
     * @param xIndex - index prvej dimenzie
     * @param yIndex - index druhej dimenzie
     * @return - maximálna absolútna vzdialenost od počiatku
     * súradnicovej sústavy v 2D priestore
     */
```

```
    private double getMaxDist(List<Double[]> data, int xIndex, int
        yIndex) {
         double xMax = getAbsMax(data, xIndex);
         double yMax = getAbsMax(data, yIndex);
         return Math.sqrt(Math.pow(xMax, 2) + Math.pow(yMax, 2));
    }

    /**
     *
     * @param data - údaje
     * @return - jedinečné identifikátory pixelov
     */
    private Set<Integer> getIndexes(List<Double[]> data) {
         Set<Integer> indexes = new HashSet<>();
         for (Double[] row : data) {
             indexes.add(row[idIndex].intValue());
         }
         return indexes;
    }

    /**
     *
     * @param data - údaje
     * @return - jedinečné identifikátory pixelov
     */
    Integer[] returnData(List<Double[]> outputData) {
         Set<Integer> indexes = new HashSet<>();
         for (Double[] row : outputData) {
             if (row[signalCountIndex] > 0) {
                 indexes.add(-row[idIndex].intValue());
             } else {
                 indexes.add(row[idIndex].intValue());

             }
         }
```

```
        return indexes.toArray(new Integer[indexes.size()]);
    }

    /**
     * výpis výstupných údajov
     */
    public void printData() {
        if (printData != null) {
            printData.sort(new DataComparator(new Integer[]{idIndex
                , gtuIndex}));
            printer.printData(printData);
        }
    }

    /**
     *
     * @param i - poradové číslo fázy rozpoznávania
     * @return - identifikátory pixelov výstupných údajov danej
     * fázy rozpoznávania
     */
    public Integer[] getStageData(int i) {
        return stageData.getStageData(i);
    }

    /**
     *
     * @param i - poradové číslo fázy rozpoznávania
     * @return - názov fázy rozpoznávania
     */
    public String getStageName(int i) {
        return stageData.getStageName(i);
    }

    /**
     *
```

```
     * @param i - poradové číslo fázy rozpoznávania
     * @return - nasledujúce poradové číslo fázy rozpoznávania
     */
    public int getNext(int i) {
        return stageData.getNext(i);
    }

    /**
     *
     * @param i - poradové číslo fázy rozpoznávania
     * @return - predchádzajúce poradové číslo fázy rozpoznávania
     */
    public int getPrevious(int i) {
        return stageData.getPrevious(i);
    }

}
```

Trieda Algorithm2

```
public class Algorithm2 {

    //výsledné údaje po poslednom behu
    private List<Double[]> lastOutputData;
    //údaje pre výpis
    private List<Double[]> printData;
    //objekt s údajmi udalosti
    private final EventDataHolder dataHolder;
    //objekt pre výpisy
    private final Printer printer;
    //údaje jednotlivých fáz rozpoznávania
    private final StageData stageData;

    //počiatočná prahová hodnota počtu signálov pre údaje
```

```
    private int patternThreshold = 0;

    //indexy údajov
    private final int totalCountIndex;
    //x-ová súradnica pixela
    private final int xIndex;
    //y-ová súradnica pixela
    private final int yIndex;
    //id pixela
    private final int idIndex;
    //gtu čas údajov
    private final int gtuIndex;
    //simulované počty signálov spŕšky
    private final int signalCountIndex;

    //definovanie indexov
    //index údajov
    private final int dataIndex = 0;
    //index rozsahov
    private final int rangesIndex = 1;
    //index spodnej hodnoty rozsahu
    private final int lowerRange = 0;
    //index hornej hodnoty rozsahu
    private final int upperRange = 1;
    //index pre rozsahy a hodnoty uhla fi v h.t. pre roviny
    private final int fiIndexP = 0;
    //index pre rozsahy a hodnoty uhla theta v h.t. pre roviny
    private final int thetaIndexP = 1;
    //index pre rozsahy a hodnoty kolmej vzdialenosti ro v h.t. pre
        roviny
    private final int roIndexP = 2;
    //index pre rozsahy a hodnoty uhla fi v h.t. pre priamky
    private final int thetaIndexL = 0;
    //index pre rozsahy a hodnoty kolmej vzdialenosti ro v h.t. pre
        priamky
```

```
    private final int roIndexL = 1;
    //index pre rozsahy uhla fi v h.t. pre priamky v 3D priestore
    private final int thetaIndexL1 = 0;
    //index pre rozsahy kolmej vzdialenosti ro v h.t. pre priamky v
        3D priestore
    private final int roIndexL1 = 1;
    //index pre rozsahy uhla fi v h.t. pre priamky v 3D priestore
    private final int thetaIndexL2 = 2;
    //index pre rozsahy kolmej vzdialenosti ro v h.t. pre priamky v
        3D priestore
    private final int roIndexL2 = 3;

    //parametre algoritmu
    //krok diskretizácie priestoru hodnôt kolmej vzdialenosti ro v
       h.t. pre roviny / pri hladaní vzoru
    private final double INITIAL_PATTERN_RO_STEP_PLANE;
    //krok diskretizácie priestoru hodnôt uhla fi v h.t. pre roviny
        / pri hladaní vzoru
    private final double INITIAL_PATTERN_FI_STEP_PLANE;
    //krok diskretizácie priestoru hodnôt uhla theta v h.t. pre
       roviny / pri hladaní vzoru
    private final double INITIAL_PATTERN_THETA_STEP_PLANE;
    //krok diskretizácie priestoru hodnôt kolmej vzdialenosti ro v
       h.t. pre roviny
    private final double RO_STEP_PLANE;
    //krok diskretizácie priestoru hodnôt uhla fi v h.t. pre roviny
    private final double FI_STEP_PLANE;
    //krok diskretizácie priestoru hodnôt uhla theta v h.t. pre
       roviny
    private final double THETA_STEP_PLANE;

    //krok diskretizácie priestoru hodnôt kolmej vzdialenosti ro v
       h.t. pre priamky
    private final double RO_STEP_LINE;
```

```
    //krok diskretizácie priestoru hodnôt uhla theta v h.t. pre
        priamky
    private final double THETA_STEP_LINE;

    //najväčšia diagonálna vzdialenost stredov dvoch pixelov
    //zaokrúhlená nahor (tzn. použi reálnu vzdialenost + malá
    //hodnota, tak aby bolo možné pomocou tejto hodnoty vybrat
    //len priamo susediace pixely)
    private final int PIXELS_DIAGONAL_DISTANCE;
    //vhodná hodnota, pomocou ktorej je možné spojit jednotlivé
    //časti jadra spŕšky oddelené medzerami mriežky ohniskovej
    //plochy
    private final int MAX_CORE_PIXEL_DISTANCE;
    //vhodná hodnota, pre odstránenie osamotených pixelov od jadra
        spŕšky
    private final double SINGLE_PIXEL_DISTANCE;
    //hodnota size v h.t. pre priamky v 3D priestore / pri hladaní
        vzoru
    private final double INITIAL_PATTERN_SIZE_LINE3D;
    //hodnota size v h.t. pre priamky v 3D priestore / pri
    //rozpoznávaní údajov
    private final double INITIAL_DATA_SIZE_LINE3D;
    //hodnota size v h.t. pre roviny / pri hladaní vzoru
    private final double INITIAL_PATTERN_SIZE_PLANE;
    //hodnota size v h.t. pre roviny / pri rozpoznávaní údajov
    private final double INITIAL_DATA_SIZE_PLANE;
    //hodnota size v h.t. pre priamky / pri hladaní priamok idúcich
         pozdĺž spŕšky
    private final int LINE_SIZE;
    //násobok dĺžky nájdeného vzoru spŕšky
    private final double MULT_SIZE;
    //počet košov, do ktorých sa rozdelia pixely, pri hladaní
    //prahu počtu signálov pre vzor, nutné nastavit s ohladom
    //na UV pozadie (problém by spôsobila iba nízka hodnota,
    //vyššie hodnoty problémy nespôsobujú a sú istotou)
```

```
    private final int DATA_LEVEL_BINS;
    //počiatočná percentuálna hodnota počtu zobrazených pixelov pri
    //hladaní prahu počtu signálov pre vzor
    private final int DISPLAYED_DATA_LIMIT;
    // minimálny počet pixelov pri hladaní počiatočnej skupiny
       pixelov jadra spŕšky
    private final int MIN_GROUP;
    //polovica dĺžky oblasti, v ktorej sa hladá vzor spŕšky
    private final int CORE_DATA_LENGTH;
    //polovica šírky oblasti, v ktorej sa hladá vzor spŕšky
    private final int CORE_DATA_WIDTH;
    //vzdialenost okolia pixela, ktoré k nemu bude pričlenené pri
       obalovaní čistého vzoru spŕšky
    private final double WRAP_DISTANCE;

    /**
     *
     * @param dataHolder - objekt s údajmi udalosti
     * @param columns - objekt s indexmi na údaje udalosti
     */
    public Algorithm2(EventDataHolder dataHolder, Columns columns)
       {

        this.dataHolder = dataHolder;

        printer = new Printer(columns);

        //definovanie fáz rozpoznávania
        stageData = new StageData(new String[]{
            "First group",
            "Core data",
            "Core",
            "Shaped core 1",
            "Shaped core 2",
            "Wrapped core",
```

```
        "Pattern",
        "Result",
        "Filtered result"});

    //indexy na údaje
    totalCountIndex = columns.getIndex("totalCount");
    xIndex = columns.getIndex("x");
    yIndex = columns.getIndex("y");
    idIndex = columns.getIndex("id");
    gtuIndex = columns.getIndex("gtu");
    signalCountIndex = columns.getIndex("signalCount");

    //načítanie parametrov z konfiguračného súboru
    INITIAL_PATTERN_RO_STEP_PLANE = Double.valueOf(
       AlgorithmProperties.getAlgorithm2Prop("
       INITIAL_PATTERN_RO_STEP_PLANE"));
    INITIAL_PATTERN_FI_STEP_PLANE = Double.valueOf(
       AlgorithmProperties.getAlgorithm2Prop("
       INITIAL_PATTERN_FI_STEP_PLANE"));
    INITIAL_PATTERN_THETA_STEP_PLANE = Double.valueOf(
       AlgorithmProperties.getAlgorithm2Prop("
       INITIAL_PATTERN_THETA_STEP_PLANE"));
    RO_STEP_PLANE = Double.valueOf(AlgorithmProperties.
       getAlgorithm2Prop("RO_STEP_PLANE"));
    FI_STEP_PLANE = Double.valueOf(AlgorithmProperties.
       getAlgorithm2Prop("FI_STEP_PLANE"));
    THETA_STEP_PLANE = Double.valueOf(AlgorithmProperties.
       getAlgorithm2Prop("THETA_STEP_PLANE"));
    RO_STEP_LINE = Double.valueOf(AlgorithmProperties.
       getAlgorithm2Prop("RO_STEP_LINE"));
    THETA_STEP_LINE = Double.valueOf(AlgorithmProperties.
       getAlgorithm2Prop("THETA_STEP_LINE"));
    PIXELS_DIAGONAL_DISTANCE = Integer.valueOf(
       AlgorithmProperties.getAlgorithm2Prop("
       PIXELS_DIAGONAL_DISTANCE"));
```

```
        MAX_CORE_PIXEL_DISTANCE = Integer.valueOf(
            AlgorithmProperties.getAlgorithm2Prop("
            MAX_CORE_PIXEL_DISTANCE"));
        SINGLE_PIXEL_DISTANCE = Double.valueOf(AlgorithmProperties.
            getAlgorithm2Prop("SINGLE_PIXEL_DISTANCE"));
        INITIAL_PATTERN_SIZE_LINE3D = Double.valueOf(
            AlgorithmProperties.getAlgorithm2Prop("
            INITIAL_PATTERN_SIZE_LINE3D"));
        INITIAL_DATA_SIZE_LINE3D = Double.valueOf(
            AlgorithmProperties.getAlgorithm2Prop("
            INITIAL_DATA_SIZE_LINE3D"));
        INITIAL_PATTERN_SIZE_PLANE = Double.valueOf(
            AlgorithmProperties.getAlgorithm2Prop("
            INITIAL_PATTERN_SIZE_PLANE"));
        INITIAL_DATA_SIZE_PLANE = Double.valueOf(
            AlgorithmProperties.getAlgorithm2Prop("
            INITIAL_DATA_SIZE_PLANE"));
        LINE_SIZE = Integer.valueOf(AlgorithmProperties.
            getAlgorithm2Prop("LINE_SIZE"));
        MULT_SIZE = Double.valueOf(AlgorithmProperties.
            getAlgorithm2Prop("MULT_SIZE"));
        DATA_LEVEL_BINS = Integer.valueOf(AlgorithmProperties.
            getAlgorithm2Prop("DATA_LEVEL_BINS"));
        DISPLAYED_DATA_LIMIT = Integer.valueOf(AlgorithmProperties.
            getAlgorithm2Prop("DISPLAYED_DATA_LIMIT"));
        MIN_GROUP = Integer.valueOf(AlgorithmProperties.
            getAlgorithm2Prop("MIN_GROUP"));
        CORE_DATA_LENGTH = Integer.valueOf(AlgorithmProperties.
            getAlgorithm2Prop("CORE_DATA_LENGTH"));
        CORE_DATA_WIDTH = Integer.valueOf(AlgorithmProperties.
            getAlgorithm2Prop("CORE_DATA_WIDTH"));
        WRAP_DISTANCE = Double.valueOf(AlgorithmProperties.
            getAlgorithm2Prop("WRAP_DISTANCE"));
    }
```

```
    /**
     *
     * @return - zoznam údajov udalosti
     */
    private List<Double[]> getEventData() {
        List<Double[]> eventData = new ArrayList<>();
        for (int i = 0; i < dataHolder.getValuesSize(); i++) {
            eventData.add(dataHolder.getValues(i));
        }
        return eventData;
    }

    /**
     *
     * @param indexHoughTransform - volba metódy Houghovej
     * transformácie
     * @param linePattern - volba vzoru založeného na priamke
     * @param linePatternSize - maximálna kolmá vzdialenosť údajov
     * od detegovanej priamky prechádzajúcej pozdĺž vzoru
     * @param size - maximálna kolmá vzdialenosť údajov od
     * detegovanej priamky/roviny
     * @param timeSizeEnabled - volba zadania maximálnej kolmej
     * vzdialenosti údajov od detegovanej priamky/roviny
     * prostredníctvom časovej zložky
     * @param thresholdLowerLimit - prahová hodnota počtu signálov
     * pre údaje
     * @param enableFilter - volba filtrácie
     * @param enableTimeFilter - volba filtrácie založenej na čase
     * @param dropSeqThresholdTypeTF - volba typu prahovej hodnoty
     * počtu signálov pre zahodenie postupnosti údajov
     * @param dropSeqThresholdTF - prahová hodnota počtu signálov
     * pre zahodenie postupnosti údajov
     * @param singleDataThresholdTF - prahová hodnota počtu
     * signálov pre postupnosť s dĺžkou 1
     * @param enableSignalFilter - volba filtrácie založenej na
```

```
     * počte signálov
     * @param dropSeqThresholdTypeSF - volba typu prahovej hodnoty
     * počtu signálov pre zahodenie postupnosti údajov
     * @param dropSeqThresholdSF - prahová hodnota počtu signálov
     * pre zahodenie postupnosti údajov
     * @param cutSeqThresholdSF - prahová hodnota počtu signálov
     * pre odrezanie postupnosti údajov
     * @param singleDataThresholdSF - prahová hodnota počtu
     * signálov pre postupnost s dĺžkou 1
     * @return - identifikátory pixelov výstupných údajov
     */
    public Integer[] run(
            int indexHoughTransform,
            boolean linePattern,
            int linePatternSize,
            double size,
            boolean timeSizeEnabled,
            int thresholdLowerLimit,
            boolean enableFilter,
            boolean enableTimeFilter,
            int dropSeqThresholdTypeTF,
            int dropSeqThresholdTF,
            int singleDataThresholdTF,
            boolean enableSignalFilter,
            int dropSeqThresholdTypeSF,
            int dropSeqThresholdSF,
            int cutSeqThresholdSF,
            int singleDataThresholdSF) {

        size /= 10;
        linePatternSize /= 10;
        int threshold = 0;
        stageData.clearStageData();

        //kopírovanie údajov
```

```
System.out.println("Data copying");
List<Double[]> newData = getEventData();

//zadefinovanie košov pre prerozdelenie údajov podla ich
   maximálnej hodnoty počtu signálov
List<Double[]>[] sortedData = new ArrayList[DATA_LEVEL_BINS
   ];
for (int i = 0; i < sortedData.length; i++) {
    sortedData[i] = new ArrayList<>();
}

//triedenie podla id a počtu signálov
newData.sort(new DataComparator(new Integer[]{idIndex,
   totalCountIndex}));

//údaje maximálnych hodnôt počtu signálov na danom pixeli
List<Double[]> uniqueData = new ArrayList<>();

//prerozdelenie údajov podla ich maximálnej hodnoty počtu
   signálov
double idCount = 0;
int rowCount = 0;
int currentId = newData.get(0)[idIndex].intValue();
while (rowCount < newData.size()) {
    int lastId = currentId;
    idCount++;
    Integer maxTotalCount = newData.get(rowCount)[
       totalCountIndex].intValue();
    if (maxTotalCount >= sortedData.length) {
        sortedData[sortedData.length - 1].add(newData.get(
           rowCount));
    } else {
        sortedData[maxTotalCount - 1].add(newData.get(
           rowCount));
    }
```

```
            uniqueData.add(newData.get(rowCount));
            while (currentId == lastId) {
                if (!(++rowCount < newData.size())) {
                    break;
                }
                currentId = newData.get(rowCount)[idIndex].intValue
                    ();
            }
        }

        //nastavenie minimálnej hranice zobrazených pixelov
        int displayedLimit = DISPLAYED_DATA_LIMIT;
        //jedno percento zobrazených pixelov
        int onePercent = (int) Math.ceil(idCount / 100);
        //zoznam s výslednou skupinou susediacich údajov
        List<Double[]> maxGroup = new ArrayList<>();
        //zoznam postupne zozbieravaných údajov
        List<Double[]> topData = new ArrayList<>();
        //štartovací kôš
        int topBin = sortedData.length - 1;
        boolean emptyBins = false;

        //hladanie počiatočnej maximálnej skupiny susediacich
        //údajov, ktorá spĺňa podmienku minimálneho počtu
        System.out.println("Group finding");
        while (maxGroup.isEmpty() && !emptyBins) {

            for (int i = topBin; i > -1; i--) {
                topData.addAll(sortedData[i]);
                if (i == 0) {
                    emptyBins = true;
                }
                if (topData.size() / onePercent >= displayedLimit)
                    {
                    threshold = i;
```

```
                topBin = i - 1;
                break;
            }

        }
        //pokus o naplnenie zoznamu maximálnej skupiny
            susediacich údajov,
        //ktorá spĺňa podmienku minimálneho počtu
        maxGroup.addAll(getShapedCore(topData, 5, MIN_GROUP));
        //inkrementácia percenta zobrazených pixelov
        displayedLimit++;
    }

    if (maxGroup.isEmpty()) {
        return null;
    }

    //získaná prahová hodnota počtu signálov pre vzor
    patternThreshold = threshold;

    //uloženie pixelov fázy s menom First group
    stageData.saveStageData(0, getIndexes(maxGroup).toArray(new
         Integer[maxGroup.size()]));

    //skupina susediacich údajov naberie na váhe pred volaním
       HoughLine()
    for (Double[] row : maxGroup) {
        row[totalCountIndex] *= 5;
    }

    //centrovanie údajov pre čo najnižšie pamäťové zaťaženie
    double[] vectorXYGtu = getCenter3D(maxGroup);
    //posunutie údajov
    shift(newData, -vectorXYGtu[0], -vectorXYGtu[1], -
       vectorXYGtu[2]);
```

```
//nájdenie priamky, ktorá prechádza pozdĺž skupiny
   susediacich údajov, čo bolo zaručené zvýšením ich váhy
System.out.println("HoughLine(topData)");

List<Double[]> lineRanges = HoughLine2D(topData, LINE_SIZE,
    xIndex, yIndex, THETA_STEP_LINE, RO_STEP_LINE)[
   rangesIndex];
double thetaAngle = (lineRanges.get(thetaIndexL)[lowerRange
   ] + lineRanges.get(thetaIndexL)[upperRange]) / 2;
if (lineRanges.get(thetaIndexL)[lowerRange] > lineRanges.
   get(thetaIndexL)[upperRange]) {
    thetaAngle += Math.PI / 2;
}
double roDistance = (lineRanges.get(roIndexL)[lowerRange] +
    lineRanges.get(roIndexL)[upperRange]) / 2;

//údaje, v ktorých sa bude hladat jadro vzoru spŕšky
System.out.println("CoreData");
List<Double[]> coreData = new ArrayList<>();
for (Double[] row : topData) {
    //right, left, upper, lower border / údaje sú
    //už vzhladom na xy maxGroup vycentrované
    if (row[xIndex] < CORE_DATA_LENGTH && row[xIndex] > -
       CORE_DATA_LENGTH && row[yIndex] < CORE_DATA_LENGTH
       && row[yIndex] > -CORE_DATA_LENGTH) {
        double rValue = row[xIndex] * Math.cos(thetaAngle)
           + row[yIndex] * Math.sin(thetaAngle);
        if (Math.abs(roDistance - rValue) < CORE_DATA_WIDTH
           ) {
            coreData.add(row);
        }
    }
}
```

```
//uloženie pixelov fázy s menom Core data
stageData.saveStageData(1, getIndexes(coreData).toArray(new
    Integer[coreData.size()]));

//prvé volanie s obmedzeným počtom údajov za účelom
//získania rozsahov pre
//fi, theta, ro / fi1, fi2, ro1, ro2
System.out.println("Hough(coreData)");
List<Double[]>[] data = Hough(indexHoughTransform, coreData
   );

//uloženie pixelov fázy s menom Core
stageData.saveStageData(2, getIndexes(data[dataIndex]).
   toArray(new Integer[data[dataIndex].size()]));

//zníženie predtým pridanej váhy skupine susediacich údajov
for (Double[] row : maxGroup) {
    row[totalCountIndex] /= 5;
}

//očistenie jadra o údaje, ktoré sú príliš daleko,
//získa sa relatívne čisté jadro vzoru spŕšky
System.out.println("CleanCoreData");
List<Double[]> cleanCoreData = getShapedCore(data[dataIndex
   ], MAX_CORE_PIXEL_DISTANCE, MIN_GROUP);

//uloženie pixelov fázy s menom Shaped core 1
stageData.saveStageData(3, getIndexes(cleanCoreData).
   toArray(new Integer[cleanCoreData.size()]));

//odstránia sa osamotené pixely, t.j. pixely, od ktorých do
    určitej vzdialenosti nie je možné nájst iný pixel,
//inak by tieto pixely ovplyvnili další krok obdĺžnikového
   ohraničenia spŕšky
```

```
cleanCoreData = getShapedCore(cleanCoreData,
    SINGLE_PIXEL_DISTANCE);

//uloženie pixelov fázy s menom Shaped core 2
stageData.saveStageData(4, getIndexes(cleanCoreData).
    toArray(new Integer[cleanCoreData.size()]));

//získanie obdĺžnikového ohraničenia spŕšky
Double[] border = getRectangle(cleanCoreData, MULT_SIZE,
    PIXELS_DIAGONAL_DISTANCE);

//získanie údajov z obdĺžnikového ohraničenia spŕšky
System.out.println("RectangleData");
List<Double[]> rectangleData = new ArrayList<>();
for (Double[] row : uniqueData) {
    //right, left, upper, lower border
    if (row[xIndex] < border[1] && row[xIndex] > border[0]
        && row[yIndex] < border[3] && row[yIndex] > border
        [2]) {
        rectangleData.add(row);
    }
}

//nájdenie priamky, ktorá prechádza pozdĺž očisteného jadra
    spŕšky
System.out.println("HoughLine(cleanCoreData)");
lineRanges = HoughLine2D(cleanCoreData, LINE_SIZE, xIndex,
    yIndex, THETA_STEP_LINE, RO_STEP_LINE)[rangesIndex];
thetaAngle = (lineRanges.get(thetaIndexL)[lowerRange] +
    lineRanges.get(thetaIndexL)[upperRange]) / 2;
if (lineRanges.get(thetaIndexL)[lowerRange] > lineRanges.
    get(thetaIndexL)[upperRange]) {
    thetaAngle += Math.PI / 2;
}
```

```
roDistance = (lineRanges.get(roIndexL)[lowerRange] +
    lineRanges.get(roIndexL)[upperRange]) / 2;

//vyberú sa indexy pixelov vzoru spŕšky
System.out.println("PatternIndexes");
//indexy pixelov vzoru spŕšky
Set<Integer> patternIndexes = new HashSet<>();
//určí sa vzor výberu pixelov, bud je to len priamka s
    danou šírkou, alebo
//sa k nej pridajú aj pixely z nájdeného jadra spŕšky a
    jeho blízkeho okolia
if (!linePattern) {
    //k pixelom jadra sa pridaju pixely v danej maximálnej
        vzdialenosti okolo neho
    patternIndexes.addAll(getIndexes(getWrap(cleanCoreData,
        rectangleData, WRAP_DISTANCE)));
    //uloženie pixelov fázy s menom Wrapped core
    stageData.saveStageData(5, patternIndexes.toArray(new
        Integer[patternIndexes.size()]));
}

if (linePatternSize > 0) {
    for (Double[] pixelRow : rectangleData) {
        double distance = pixelRow[xIndex] * Math.cos(
            thetaAngle) + pixelRow[yIndex] * Math.sin(
            thetaAngle);
        if (Math.abs(roDistance - distance) <
            linePatternSize) {
            //pridávajú sa indexy pixelov okolo priamky,
                ktorá prechádza pozdĺž jadra,
            //kedze jadro bolo ocistene a mohol by sa
                stratit cerenkovou pixel
            patternIndexes.add(pixelRow[idIndex].intValue()
                );
        }
```

```
        }
    }

    //uloženie pixelov fázy s nazvom Pattern
    stageData.saveStageData(6, patternIndexes.toArray(new
        Integer[patternIndexes.size()]));

    //vyberanie údajov pixelov spŕšky
    System.out.println("PatternData");
    List<Double[]> dirtyShowerData = getAllData(patternIndexes,
        newData);

    //zadefinujú sa koše pre údaje podla počtu signálov
    if (threshold - thresholdLowerLimit < 1) {
        thresholdLowerLimit = threshold - 1;
    }
    int count = threshold - thresholdLowerLimit;
    List<Double[]>[] sortedInputData = new ArrayList[count];
    for (int i = 0; i < count; i++) {
        sortedInputData[i] = new ArrayList<>();
    }

    //zatriedenie údajov do košov podla počtu signálov
    int index;
    int top = count - 1;
    int topThresh = threshold - 1;
    for (Double[] pixelRow : dirtyShowerData) {
        if (pixelRow[totalCountIndex] > topThresh) {
            sortedInputData[top].add(pixelRow);
        } else if ((index = (pixelRow[totalCountIndex].intValue
            () % threshold) - thresholdLowerLimit - 1) > -1) {
            sortedInputData[index].add(pixelRow);
        }
    }
```

```
        //informačný výpis
        printer.printLine();
        for (int i = sortedInputData.length - 1; i > -1; i--) {
            printer.printInfo("Level " + (i + thresholdLowerLimit)
                + " :", sortedInputData[i]);
        }
        printer.printLine();

        //inicializácia rozsahov
        data = Hough(indexHoughTransform, cleanCoreData, thetaAngle
            );

        if (timeSizeEnabled) {
            size = getTimeSize(indexHoughTransform, data[
                rangesIndex], size);
        }

        //výpočet
        //outputData - množina výstupných údajov
        List<Double[]> outputData = new ArrayList<>();
        //threshold - prahová hodnota počtu signálov
        //thresholdLowerLimit - spodná hranica pre prahovú hodnotu
            počtu signálov
        while (--threshold >= thresholdLowerLimit) {
            //sortedInputData - množina skupín vstupných údajov
            //data[rangesIndex] - množina rozsahov parametrického
                priestoru
            //size - maximálna kolmá vzdialenosť údajov
            //indexHoughTransform - volba metódy
            data = Hough(indexHoughTransform, sortedInputData[top
                --], data[rangesIndex], size);
            printer.printInfo("Run " + threshold + " :", data[
                dataIndex]);
            outputData.addAll(data[dataIndex]);
        }
```

```
        //spätné posunutie výsledných údajov v priestore XYGtu
        shift(outputData, vectorXYGtu[0], vectorXYGtu[1],
            vectorXYGtu[2]);

        //uloženie údajov pre možné budúce filtrovanie
        lastOutputData = outputData;
        //uloženie pixelov fázy s menom Result
        stageData.saveStageData(7, returnData(outputData));

        //výpis
        printer.printLine();
        printer.printInfo("Result :", outputData);

        //filtrovanie údajov
        if (enableFilter) {
            outputData = filter(outputData, enableTimeFilter,
                dropSeqThresholdTypeTF, dropSeqThresholdTF,
                singleDataThresholdTF, enableSignalFilter,
                dropSeqThresholdTypeSF, dropSeqThresholdSF,
                cutSeqThresholdSF, singleDataThresholdSF);
            //výpis
            printer.printInfo("Filtered result :", outputData);
        } else {
            //uloženie výslednych údajov pre ich možný výpis
            printData = outputData;
        }

        printer.write(outputData, "output0", 0);
        printer.write(outputData, "output1", 1);
        //identifikátory pixelov výstupných údajov
        return returnData(outputData);

    }
```

```
/**
 * metóda vybranej Houghovej transformácie pre hladanie vzoru
 *
 * @param data - údaje
 * @param indexHoughTransform - volba metódy Houghovej
 * transformácie
 * @return - výstupné údaje Houghovej transformácie pre
 * priamky/roviny
 */
private List<Double[]>[] Hough(int indexHoughTransform, List<
    Double[]> data) {
    if (indexHoughTransform == 0) {
        return HoughLine3D(data, INITIAL_PATTERN_SIZE_LINE3D,
            xIndex, gtuIndex, yIndex, THETA_STEP_LINE,
            RO_STEP_LINE);
    } else {
        double ro = getMaxDist(data, xIndex, yIndex, gtuIndex);
        return HoughPlane(data, INITIAL_PATTERN_SIZE_PLANE,
            INITIAL_PATTERN_FI_STEP_PLANE, -Math.PI / 2, Math.PI
             / 2, INITIAL_PATTERN_THETA_STEP_PLANE, -Math.PI /
            2, Math.PI / 2, INITIAL_PATTERN_RO_STEP_PLANE, -ro,
            ro);
    }
}

/**
 * metóda vybranej Houghovej transformácie pre inicializáciu
 * parametrického priestoru
 *
 * @param data - údaje
 * @param indexHouhTransform - volba metódy Houghovej
 * transformácie
 * @param thetaAngleLine - uhol theta priamky idúcej pozdĺž
 * vzoru spŕšky
 * @return - výstupné údaje Houghovej transformácie pre
```

```
     * priamky/roviny
     */
    private List<Double[]>[] Hough(int indexHoughTransform, List<
        Double[]> data, double thetaAngleLine) {
        if (indexHoughTransform == 0) {
            return HoughLine3D(data, INITIAL_DATA_SIZE_LINE3D,
                xIndex, gtuIndex, yIndex, THETA_STEP_LINE,
                RO_STEP_LINE);
        } else {
            return HoughPlane(data, INITIAL_DATA_SIZE_PLANE,
                thetaAngleLine, FI_STEP_PLANE, THETA_STEP_PLANE,
                RO_STEP_PLANE);
        }
    }

    /**
     * metóda vybranej Houghovej transformácie pre rozpoznávanie
     * údajov
     *
     * @param data - údaje
     * @param indexHoughTransform - volba metódy Houghovej
     * transformácie
     * @param ranges - rozsahy priamky/roviny
     * (fi, ro / fi, theta, ro)
     * @param size - maximálna kolmá vzdialenost údajov od
     * detegovanej priamky/roviny
     * @return - výstupné údaje Houghovej transformácie pre
     * priamky/roviny
     */
    private List<Double[]>[] Hough(int indexHoughTransform, List<
        Double[]> data, List<Double[]> ranges, double size) {
        if (indexHoughTransform == 0) {
            return HoughLine3D(data, ranges, size, xIndex, gtuIndex
                , yIndex, THETA_STEP_LINE, RO_STEP_LINE);
        } else {
```

```
            return HoughPlane(data, ranges, size, FI_STEP_PLANE,
                THETA_STEP_PLANE, RO_STEP_PLANE);
        }
    }

    /**
     * Houghova transformácia pre roviny pre inicializáciu
     * parametrického priestoru
     *
     * @param data - údaje
     * @param size - maximálna kolmá vzdialenost údajov od
     * detegovanej roviny
     * @param thetaAngleLine - uhol theta priamky idúcej pozdĺž
     * vzoru spŕšky v radiánoch
     * @param fiStep - krok diskretizácie priestoru uhla fi [deg]
     * @param thetaStep - krok diskretizácie priestoru uhla
     * theta [deg]
     * @param roStep - krok diskretizacie priestoru kolmej
     * vzdialenosti roviny od počiatku súradnicovej sústavy
     * @return - výstupné údaje Houghovej transformácie pre roviny
     */
    private List<Double[]>[] HoughPlane(List<Double[]> data, double
        size, double thetaAngleLine, double fiStep, double
       thetaStep, double roStep) {
        double ro = getMaxDist(data, xIndex, yIndex, gtuIndex);
        //posunutie uhla o PI / 2 tak, aby odpovedal uhlu theta v
            popise roviny
        thetaAngleLine += Math.PI / 2;
        return HoughPlane(data, size, fiStep, -Math.PI / 2, Math.PI
            / 2, thetaStep, thetaAngleLine, thetaAngleLine, roStep,
            -ro, ro);
    }

    /**
     * Houghova transformácia pre roviny pre rozpoznávanie údajov
```

```
     *
     * @param data - údaje
     * @param ranges - rozsahy parametrického priestoru
     * fi, theta, ro
     * @param size - maximálna kolmá vzdialenost údajov od
     * detegovanej roviny
     * @param fiStep - krok diskretizácie priestoru uhla fi [deg]
     * @param thetaStep - krok diskretizácie priestoru uhla
     * theta [deg]
     * @param roStep - krok diskretizacie priestoru kolmej
     * vzdialenosti roviny od počiatku súradnicovej sústavy
     * @return - výstupné údaje Houghovej transformácie pre roviny
     */
    private List<Double[]>[] HoughPlane(List<Double[]> data, List<
        Double[]> ranges, double size, double fiStep, double
        thetaStep, double roStep) {
         return HoughPlane(data, size, fiStep, ranges.get(fiIndexP)[
             lowerRange], ranges.get(fiIndexP)[upperRange], thetaStep
             , ranges.get(thetaIndexP)[lowerRange], ranges.get(
             thetaIndexP)[upperRange], roStep, ranges.get(roIndexP)[
             lowerRange], ranges.get(roIndexP)[upperRange]);
    }

    /**
     * Houghova transformácia pre roviny
     *
     * @param data - údaje
     * @param size - maximálna kolmá vzdialenost údajov od
     * detegovanej roviny
     * @param fiStep - krok diskretizácie priestoru uhla fi [deg]
     * @param fiStart - začiatok priestoru uhla fi v radiánoch
     * (v intervale od -PI/2 do PI/2)
     * @param fiEnd - koniec priestoru uhla fi v radiánoch
     * (v intervale od -PI/2 do PI/2)
     * @param thetaStep - krok diskretizácie priestoru uhla
```

```
     * theta [deg]
     * @param thetaStart - začiatok priestoru uhla theta v
     * radiánoch (v intervale od -PI/2 do PI/2)
     * @param thetaEnd - koniec priestoru uhla theta v radiánoch
     * (v intervale od -PI/2 do PI/2)
     * @param roStep - krok diskretizacie priestoru kolmej
     * vzdialenosti roviny od počiatku súradnicovej sústavy
     * @param roStart - začiatok priestoru kolmej vzdialenosti
     * roviny od počiatku súradnicovej sústavy
     * @param roEnd - koniec priestoru kolmej vzdialenosti roviny
     * od počiatku súradnicovej sústavy
     * @return - výstupné údaje Houghovej transformácie pre roviny
     */
    private List<Double[]>[] HoughPlane(List<Double[]> data, double
        size, double fiStep, double fiStart, double fiEnd, double
       thetaStep, double thetaStart, double thetaEnd, double roStep
       , double roStart, double roEnd) {
        //definovanie pola hodnôt uhla fi
        fiStep = fiStep * Math.PI / 180;
        double[] fiArray;
        int index;
        int nFi;
        if (fiStart <= fiEnd) {
            nFi = (int) (Math.abs(fiEnd - fiStart) / fiStep) + 1;
            if ((nFi - 1) * fiStep + fiStart < fiEnd && nFi *
               fiStep + fiStart < Math.PI / 2) {
                nFi++;
            }
            fiArray = new double[nFi];
            index = 0;
            double fiValue = fiStart;
            while (index < nFi) {
                fiArray[index++] = fiValue;
                fiValue += fiStep;
            }
```

```
        } else {
            int nFi1 = (int) (Math.abs(fiEnd - (-Math.PI / 2)) /
               fiStep) + 1;
            int nFi2 = (int) (Math.abs(Math.PI / 2 - fiStart) /
               fiStep) + 1;
            if ((nFi1 - 1) * fiStep + (-Math.PI / 2) < fiEnd &&
               nFi1 * fiStep + (-Math.PI / 2) < Math.PI / 2) {
                nFi1++;
            }
            nFi = nFi2 + nFi1;
            fiArray = new double[nFi];
            index = 0;
            double fiValue = -Math.PI / 2;
            while (index < nFi1) {
                fiArray[index++] = fiValue;
                fiValue += fiStep;
            }
            fiValue = fiStart;
            while (index < nFi) {
                fiArray[index++] = fiValue;
                fiValue += fiStep;
            }
        }

        //definovanie pola hodnôt uhla theta
        thetaStep = thetaStep * Math.PI / 180;
        double[] thetaArray;
        int nTheta;
        if (thetaStart <= thetaEnd) {
            nTheta = (int) (Math.abs(thetaEnd - thetaStart) /
               thetaStep) + 1;
            if ((nTheta - 1) * thetaStep + thetaStart < thetaEnd &&
                nTheta * thetaStep + thetaStart < Math.PI / 2) {
                nTheta++;
            }
```

```
            thetaArray = new double[nTheta];
            index = 0;
            double thetaValue = thetaStart;
            while (index < nTheta) {
                thetaArray[index++] = thetaValue;
                thetaValue += thetaStep;
            }
        } else {
            int nTheta1 = (int) (Math.abs(thetaEnd - (-Math.PI / 2)
               ) / thetaStep) + 1;
            int nTheta2 = (int) (Math.abs(Math.PI / 2 - thetaStart)
                / thetaStep) + 1;
            if ((nTheta1 - 1) * thetaStep + (-Math.PI / 2) <
               thetaEnd && nTheta1 * thetaStep + (-Math.PI / 2) <
               Math.PI / 2) {
                nTheta1++;
            }
            nTheta = nTheta2 + nTheta1;
            thetaArray = new double[nTheta];
            index = 0;
            double thetaValue = -Math.PI / 2;
            while (index < nTheta1) {
                thetaArray[index++] = thetaValue;
                thetaValue += thetaStep;
            }
            thetaValue = thetaStart;
            while (index < nTheta) {
                thetaArray[index++] = thetaValue;
                thetaValue += thetaStep;
            }
        }

        //definovanie pola hodnôt kolmej vzdialenosti roviny od
        //počiatku súradnicovej sústavy
```

```
int nRo = (int) (Math.ceil(Math.abs(roEnd - roStart) /
   roStep)) + 1;
double[] roArray = new double[nRo];
index = 0;
double roValue = roStart;
while (index < nRo) {
    roArray[index++] = roValue;
    roValue += roStep;
}

//trojrozmerná inkrementačná matica
short[][][] H = new short[nFi][nTheta][nRo];

double lowerDistance = roArray[0] - 1.1 * size;
double upperDistance = roArray[nRo - 1] + 1.1 * size;

//výpočet kolmých vzdialeností roviny od počiatku
//súradnicovej sústavy
for (Double[] row : data) {
    for (int thetaIndex = 0; thetaIndex < nTheta;
       thetaIndex++) {
        for (int fiIndex = 0; fiIndex < nFi; fiIndex++) {
            //výpočet
            double distance = Math.sin(fiArray[fiIndex]) *
               Math.cos(thetaArray[thetaIndex]) * row[
               xIndex]
                    + Math.sin(fiArray[fiIndex]) * Math.sin
                       (thetaArray[thetaIndex]) * row[
                       yIndex]
                    + Math.cos(fiArray[fiIndex]) * row[
                       gtuIndex];
            //kontrola, či sa bude inkrementovat
            //inkrementačná matica
            if (distance > lowerDistance && distance <
               upperDistance) {
```

```
                    //vypočíta sa horný a dolný index pre
                        hodnoty kolmých vzdialeností roviny
                    //od počiatku súradnicovej sústavy vzhladom
                         na maximálnu povolenú kolmú vzdialenost
                    //údajov od nájdenej roviny a vzorkovaciu
                        frekvenciu pola kolmých vzdialeností
                    int lowerIndex = (int) Math.round((distance
                         - size - roArray[0]) / roStep);
                    int upperIndex = (int) Math.round((distance
                         + size - roArray[0]) / roStep);
                    //indexy musia byt v povolenom rozsahu
                    if (lowerIndex < 0) {
                         lowerIndex = 0;
                    }
                    if (upperIndex > nRo - 1) {
                         upperIndex = nRo - 1;
                    }

                    //inkrementácia polí rovín, v ktorých
                    //sa údaj vyskytuje
                    for (int roIndex = lowerIndex; roIndex <
                        upperIndex + 1; roIndex++) {
                         H[fiIndex][thetaIndex][roIndex] += row[
                             totalCountIndex];
                    }
               }
          }
     }
}

//hladanie maxima v inkrementačnej matici
short max = 0;
//zoznam indexov (fiIndex, thetaIndex, roIndex)
//odpovedajúcich maximu inkrementačnej matice
List<Integer[]> indexes = new ArrayList<>();
```

```
        for (int thetaIndex = 0; thetaIndex < nTheta; thetaIndex++)
            {
            for (int fiIndex = 0; fiIndex < nFi; fiIndex++) {
                for (int roIndex = 0; roIndex < nRo; roIndex++) {
                    if (H[fiIndex][thetaIndex][roIndex] == max) {
                        indexes.add(new Integer[]{fiIndex,
                            thetaIndex, roIndex});
                    } else if (H[fiIndex][thetaIndex][roIndex] >
                        max) {
                        indexes.clear();
                        max = H[fiIndex][thetaIndex][roIndex];
                        indexes.add(new Integer[]{fiIndex,
                            thetaIndex, roIndex});
                    }
                }
            }
        }

        Set<Double[]> resultPixels = new HashSet<>();

        //získavanie rozsahov
        int minFi = indexes.get(0)[fiIndexP];
        int maxFi = indexes.get(0)[fiIndexP];
        int maxRo = indexes.get(0)[roIndexP];
        int minRo = indexes.get(0)[roIndexP];

        for (Integer[] indexRow : indexes) {
            if (indexRow[fiIndexP] > maxFi) {
                maxFi = indexRow[fiIndexP];
            } else if (indexRow[fiIndexP] < minFi) {
                minFi = indexRow[fiIndexP];
            }
            if (indexRow[roIndexP] > maxRo) {
                maxRo = indexRow[roIndexP];
            } else if (indexRow[roIndexP] < minRo) {
```

```
            minRo = indexRow[roIndexP];
        }

        //získavanie údajov pre výstup
        for (Double[] row : data) {
            double distance = Math.sin(fiArray[indexRow[
                fiIndexP]]) * Math.cos(thetaArray[indexRow[
                thetaIndexP]]) * row[xIndex]
                     + Math.sin(fiArray[indexRow[fiIndexP]]) *
                         Math.sin(thetaArray[indexRow[thetaIndexP
                         ]]) * row[yIndex]
                     + Math.cos(fiArray[indexRow[fiIndexP]]) *
                         row[gtuIndex];
            if (distance > lowerDistance && distance <
                upperDistance) {
                int lowerIndex = (int) Math.round((distance -
                    size - roArray[0]) / roStep);
                int upperIndex = (int) Math.round((distance +
                    size - roArray[0]) / roStep);
                if (lowerIndex <= indexRow[roIndexP] &&
                    upperIndex >= indexRow[roIndexP]) {
                    resultPixels.add(row);
                }
            }
        }
    }

    //zápis rozsahov
    List<Double[]> resultRanges = new ArrayList<>();
    if (fiArray[maxFi] - fiArray[minFi] < Math.PI / 2) {
        resultRanges.add(new Double[]{fiArray[minFi], fiArray[
            maxFi]});
    } else {
        resultRanges.add(new Double[]{fiArray[maxFi], fiArray[
            minFi]});
```

```
        }
        if (thetaArray[indexes.get(indexes.size() - 1)[thetaIndexP
            ]] - thetaArray[indexes.get(0)[thetaIndexP]] < Math.PI /
             2) {
            resultRanges.add(new Double[]{thetaArray[indexes.get(0)
                [thetaIndexP]], thetaArray[indexes.get(indexes.size
                () - 1)[thetaIndexP]]});
        } else {
            resultRanges.add(new Double[]{thetaArray[indexes.get(
                indexes.size() - 1)[thetaIndexP]], thetaArray[
                indexes.get(0)[thetaIndexP]]});
        }
        resultRanges.add(new Double[]{roArray[minRo], roArray[maxRo
            ]});

        List<Double[]>[] results = new ArrayList[2];
        results[dataIndex] = new ArrayList<>(resultPixels);
        results[rangesIndex] = resultRanges;
        return results;
    }

    /**
     * Houghova transformácia pre priamky v 3D priestore pre
     * hladanie vzoru
     *
     * @param data - údaje
     * @param size - maximálna povolená kolmá vzdialenost údajov od
     * detegovanej priamky
     * @param xIndex - index prvej dimenzie 3D priestoru
     * @param yIndex - index druhej dimenzie 3D priestoru
     * @param zIndex - index tretej dimenzie 3D priestoru
     * @param thetaStep - krok diskretizácie priestoru uhla
     * theta [deg]
     * @param roStep - krok diskretizácie priestoru kolmej
     * vzdialenosti priamky od počiatku súradnicovej sústavy
```

```
     * @return - výstupné údaje Houghovej transformácie pre priamky
     * v 3D priestore
     */
    private List<Double[]>[] HoughLine3D(List<Double[]> data,
       double size, int xIndex, int yIndex, int zIndex, double
       thetaStep, double roStep) {
        List<Double[]>[] firstData = HoughLine2D(data, size, xIndex
            , yIndex, thetaStep, roStep);
        List<Double[]>[] result = HoughLine2D(firstData[dataIndex],
             size, yIndex, zIndex, thetaStep, roStep);
        result[rangesIndex].add(firstData[rangesIndex].get(
           thetaIndexL));
        result[rangesIndex].add(firstData[rangesIndex].get(roIndexL
           ));
        return result;
    }

    /**
     * Houghova transformácia pre priamky v 3D priestore pre
     * rozpoznávanie údajov
     *
     * @param data - údaje
     * @param ranges - rozsahy fi, ro
     * @param size - maximálna kolmá vzdialenost údajov od
     * detegovanej priamky v 2D priestore
     * @param xIndex - index prvej dimenzie 3D priestoru
     * @param yIndex - index druhej dimenzie 3D priestoru
     * @param zIndex - index tretej dimenzie 3D priestoru
     * @param thetaStep - krok diskretizácie priestoru uhla
     * theta [deg]
     * @param roStep - krok diskretizácie priestoru kolmej
     * vzdialenosti priamky od počiatku súradnicovej sústavy
     * @return - výstupné údaje Houghovej transformácie pre priamky
     * v 3D priestore
     */
```

```
private List<Double[]>[] HoughLine3D(List<Double[]> data, List<
    Double[]> ranges, double size, int xIndex, int yIndex, int
    zIndex, double thetaStep, double roStep) {
     List<Double[]>[] firstData = HoughLine2D(data, size, xIndex
         , yIndex, thetaStep, ranges.get(thetaIndexL2)[lowerRange
         ], ranges.get(thetaIndexL2)[upperRange], roStep, ranges.
         get(roIndexL2)[lowerRange], ranges.get(roIndexL2)[
         upperRange]);
     List<Double[]>[] result = HoughLine2D(firstData[dataIndex],
          size, yIndex, zIndex, thetaStep, ranges.get(
         thetaIndexL1)[lowerRange], ranges.get(thetaIndexL1)[
         upperRange], roStep, ranges.get(roIndexL1)[lowerRange],
         ranges.get(roIndexL1)[upperRange]);
     result[rangesIndex].add(firstData[rangesIndex].get(
         thetaIndexL));
     result[rangesIndex].add(firstData[rangesIndex].get(roIndexL
         ));
     return result;
}

/**
 * Houghova transformácia pre priamky v 2D priestore
 *
 * @param data - údaje
 * @param size - maximálna povolená kolmá vzdialenost údajov od
 * detegovanej priamky
 * @param indexFirstDim - index prvej dimenzie 2D priestoru
 * @param indexSecondDim - index druhej dimenzie 2D priestoru
 * @param thetaStep - krok diskretizácie priestoru uhla
 * theta [deg]
 * @param roStep - krok diskretizácie priestoru kolmej
 * vzdialenosti priamky od počiatku súradnicovej sústavy
 * @return - výstupné údaje Houghovej transformácie pre priamky
 * v 2D priestore
 */
```

```
    private List<Double[]>[] HoughLine2D(List<Double[]> data,
        double size, int indexFirstDim, int indexSecondDim, double
        thetaStep, double roStep) {
        //maximalna absolutna vzdialenost udajov od pociatku
            suradnicovej sustavy
        double ro = getMaxDist(data, indexFirstDim, indexSecondDim)
            ;
        return HoughLine2D(data, size, indexFirstDim,
            indexSecondDim, thetaStep, -Math.PI / 2, Math.PI / 2,
            roStep, -ro, ro);
    }

    /**
     * Houghova transformácia pre priamky v 2D priestore
     *
     * @param data - údaje
     * @param size - maximálna povolená kolmá vzdialenost údajov od
     * detegovanej priamky
     * @param indexFirstDim - index prvej dimenzie 2D priestoru
     * @param indexSecondDim - index druhej dimenzie 2D priestoru
     * @param thetaStep - krok diskretizácie priestoru uhla
     * fi [deg]
     * @param thetaStart - začiatok priestoru uhla theta v
     * radiánoch (v intervale od -PI/2 do PI/2)
     * @param thetaEnd - koniec priestoru uhla theta v radiánoch
     * (v intervale od -PI/2 do PI/2)
     * @param roStep - krok diskretizácie priestoru kolmej
     * vzdialenosti priamky od počiatku súradnicovej sústavy
     * @param roStart - začiatok priestoru kolmej vzdialenosti
     * priamky od počiatku súradnicovej sústavy
     * @param roEnd - koniec priestoru kolmej vzdialenosti priamky
     * od počiatku súradnicovej sústavy
     * @return - výstupné údaje Houghovej transformácie pre priamky
     * v 2D priestore
     */
```

```
private List<Double[]>[] HoughLine2D(List<Double[]> data,
    double size, Integer indexFirstDim, Integer indexSecondDim,
    double thetaStep, double thetaStart, double thetaEnd, double
     roStep, double roStart, double roEnd) {

    //definovanie pola hodnôt kolmej vzdialenosti priamky od
    //počiatku súradnicovej sústavy
    int nRo = (int) (Math.ceil(Math.abs(roEnd - roStart) /
       roStep)) + 1;
    double[] roArray = new double[nRo];
    int index = 0;
    double roValue = roStart;
    while (index < nRo) {
        roArray[index++] = roValue;
        roValue += roStep;
    }

    //definovanie pola hodnôt uhla theta
    thetaStep = thetaStep * Math.PI / 180;
    double[] thetaArray;
    int nTheta;
    if (thetaStart <= thetaEnd) {
        nTheta = (int) (Math.abs(thetaEnd - thetaStart) /
           thetaStep) + 1;
        if ((nTheta - 1) * thetaStep + thetaStart < thetaEnd &&
            nTheta * thetaStep + thetaStart < Math.PI / 2) {
            nTheta++;
        }
        thetaArray = new double[nTheta];
        index = 0;
        double thetaValue = thetaStart;
        while (index < nTheta) {
            thetaArray[index++] = thetaValue;
            thetaValue += thetaStep;
        }
```

```
        } else {
            int nTheta1 = (int) (Math.abs(thetaEnd - (-Math.PI / 2)
               ) / thetaStep) + 1;
            int nTheta2 = (int) (Math.abs(Math.PI / 2 - thetaStart)
                / thetaStep) + 1;
            if ((nTheta1 - 1) * thetaStep + (-Math.PI / 2) <
               thetaEnd && nTheta1 * thetaStep + (-Math.PI / 2) <
               Math.PI / 2) {
                nTheta1++;
            }
            nTheta = nTheta2 + nTheta1;
            thetaArray = new double[nTheta];
            index = 0;
            double thetaValue = -Math.PI / 2;
            while (index < nTheta1) {
                thetaArray[index++] = thetaValue;
                thetaValue += thetaStep;
            }
            thetaValue = thetaStart;
            while (index < nTheta) {
                thetaArray[index++] = thetaValue;
                thetaValue += thetaStep;
            }
        }

//dvojrozmerná inkrementačná matica
        short[][] A = new short[nTheta][nRo];
        double lowerDistance = roArray[0] - 1.1 * size;
        double upperDistance = roArray[nRo - 1] + 1.1 * size;

        //výpočet kolmých vzdialeností priamky
        //od počiatku súradnicovej sústavy
        for (Double[] row : data) {
            for (int thetaIndex = 0; thetaIndex < nTheta;
               thetaIndex++) {
```

```
            double distance = row[indexFirstDim] * Math.cos(
                thetaArray[thetaIndex])
                        + (row[indexSecondDim]) * Math.sin(
                            thetaArray[thetaIndex]);
            //vypočíta sa horný a dolný index pre hodnoty
            //kolmých vzdialeností priamky
            //od počiatku súradnicovej sústavy, vzhladom na
                maximálnu povolenú kolmú vzdialenost údajov
            //od detegovanej priamky a vzorkovaciu frekvenciu
                pola kolmých vzdialeností
            if (distance > lowerDistance && distance <
                upperDistance) {
                int lowerIndex = (int) Math.round((distance -
                    size - roArray[0]) / roStep);
                int upperIndex = (int) Math.round((distance +
                    size - roArray[0]) / roStep);
                //indexy musia byt v povolenom rozsahu
                if (lowerIndex < 0) {
                    lowerIndex = 0;
                }
                if (upperIndex > nRo - 1) {
                    upperIndex = nRo - 1;
                }
                //inkrementácia polí priamok, v ktorých sa údaj
                    vyskytuje
                for (int roIndex = lowerIndex; roIndex <
                    upperIndex + 1; roIndex++) {
                    A[thetaIndex][roIndex] += row[
                        totalCountIndex];
                }
            }
        }
    }

    Set<Double[]> resultData = new HashSet<>();
```

```
        List<Double[]> resultRanges = new ArrayList<>();

        short max = 0;
        //zoznam indexov (fiIndex, roIndex) odpovedajúcich maximu
            inkrementačnej matice
        List<Integer[]> indexes = new ArrayList<>();

        for (int thetaIndex = 0; thetaIndex < nTheta; thetaIndex++)
            {
            for (int roIndex = 0; roIndex < nRo; roIndex++) {
                if (A[thetaIndex][roIndex] == max) {
                    indexes.add(new Integer[]{thetaIndex, roIndex})
                        ;
                } else if (A[thetaIndex][roIndex] > max) {
                    indexes.clear();
                    max = A[thetaIndex][roIndex];
                    indexes.add(new Integer[]{thetaIndex, roIndex})
                        ;
                }
            }
        }

        int maxRo = indexes.get(0)[roIndexL];
        int minRo = indexes.get(0)[roIndexL];
        //získavanie výsledných údajov a popisov priamok,
        //odpovedajúcich maximu inkrementačnej matice
        for (Integer[] indexRow : indexes) {
            if (indexRow[roIndexL] > maxRo) {
                maxRo = indexRow[roIndexL];
            } else if (indexRow[roIndexL] < minRo) {
                minRo = indexRow[roIndexL];
            }
            for (Double[] row : data) {
                double distance = row[indexFirstDim] * Math.cos(
                    thetaArray[indexRow[thetaIndexL]])
```

```
                    + row[indexSecondDim] * Math.sin(thetaArray
                        [indexRow[thetaIndexL]]);
            if (distance > lowerDistance && distance <
                upperDistance) {
                int lowerIndex = (int) Math.round((distance -
                    size - roArray[0]) / roStep);
                int upperIndex = (int) Math.round((distance +
                    size - roArray[0]) / roStep);
                if (lowerIndex <= indexRow[roIndexL] &&
                    upperIndex >= indexRow[roIndexL]) {
                    resultData.add(row);
                }
            }
        }
    }

    //výstupné hodnoty
    List<Double[]>[] result = new ArrayList[2];

    //zápis rozsahov parametrického priestoru, definovaných na
        základe výstupných údajov
    if (Math.abs(thetaArray[indexes.get(indexes.size() - 1)[
        thetaIndexL]] - thetaArray[indexes.get(0)[thetaIndexL]])
         < Math.PI / 2) {
        resultRanges.add(new Double[]{thetaArray[indexes.get(0)
            [thetaIndexL]], thetaArray[indexes.get(indexes.size
            () - 1)[thetaIndexL]]});
    } else {
        resultRanges.add(new Double[]{thetaArray[indexes.get(
            indexes.size() - 1)[thetaIndexL]], thetaArray[
            indexes.get(0)[thetaIndexL]]});
    }
    resultRanges.add(new Double[]{roArray[minRo], roArray[maxRo
        ]});
```

```
        result[dataIndex] = new ArrayList<>(resultData);
        result[rangesIndex] = resultRanges;

        return result;
    }

    /**
     *
     * @param data - údaje
     * @param xIndex - index prvej dimenzie
     * @param yIndex - index druhej dimenzie
     * @param zIndex - index druhej dimenzie
     * @return - maximálna absolútna vzdialenost od počiatku
     * súradnicovej sústavy v 3D priestore
     */
    double getMaxDist(List<Double[]> data, int xIndex, int yIndex,
        int zIndex) {
        double xMax = getAbsMax(data, xIndex);
        double yMax = getAbsMax(data, yIndex);
        double zMax = getAbsMax(data, zIndex);
        return Math.sqrt(Math.pow(xMax, 2) + Math.pow(yMax, 2) +
            Math.pow(zMax, 2));
    }

    /**
     *
     * @param data - údaje
     * @param xIndex - index prvej dimenzie
     * @param yIndex - index druhej dimenzie
     * @return - maximálna absolútna vzdialenost od počiatku
     * súradnicovej sústavy v 2D priestore
     */
    double getMaxDist(List<Double[]> data, int xIndex, int yIndex)
        {
        double xMax = getAbsMax(data, xIndex);
```

```
        double yMax = getAbsMax(data, yIndex);
        return Math.sqrt(Math.pow(xMax, 2) + Math.pow(yMax, 2));
    }

    /**
     *
     * @param data - údaje
     * @param index - index dimenzie priestoru
     * @return - maximálna absolútna vzdialenost údajov od počiatku
     * súradnicovej sústavy v jednorozmernom priestore
     */
    private double getAbsMax(List<Double[]> data, Integer index) {
        double max = Math.abs(data.get(0)[index]);
        for (int i = 1; i < data.size(); i++) {
            double value = Math.abs(data.get(i)[index]);
            if (max < value) {
                max = value;
            }
        }
        return max;
    }

    /**
     *
     * posun v 3D priestore XYgtu
     *
     * @param data - údaje
     * @param x - x-ová zložka vektora posunutia
     * @param y - y-ova zložka vektora posunutia
     * @param gtu - časová zložka vektora posunutia
     */
    private void shift(List<Double[]> data, double x, double y,
       double gtu) {
        for (Double[] row : data) {
            row[xIndex] = row[xIndex] + x;
```

```
            row[yIndex] = row[yIndex] + y;
            row[gtuIndex] = row[gtuIndex] + gtu;
        }
    }

    /**
     *
     * @param data - údaje
     * @return - vektor so začiatkom v počiatku súradnicovej
     * sústavy a koncom v centre údajov 3D priestoru XYGtu
     */
    private double[] getCenter3D(List<Double[]> data) {
        double maxX = data.get(0)[xIndex];
        double minX = data.get(0)[xIndex];
        double maxY = data.get(0)[yIndex];
        double minY = data.get(0)[yIndex];
        double maxGtu = data.get(0)[gtuIndex];
        double minGtu = data.get(0)[gtuIndex];
        for (int i = 1; i < data.size(); i++) {
            if (maxX < data.get(i)[xIndex]) {
                maxX = data.get(i)[xIndex];
            } else if (minX > data.get(i)[xIndex]) {
                minX = data.get(i)[xIndex];
            }
            if (maxY < data.get(i)[yIndex]) {
                maxY = data.get(i)[yIndex];
            } else if (minY > data.get(i)[yIndex]) {
                minY = data.get(i)[yIndex];
            }
            if (maxGtu < data.get(i)[gtuIndex]) {
                maxGtu = data.get(i)[gtuIndex];
            } else if (minGtu > data.get(i)[gtuIndex]) {
                minGtu = data.get(i)[gtuIndex];
            }
        }
```

```
        return new double[]{(minX + maxX) / 2, (minY + maxY) / 2, (
            minGtu + maxGtu) / 2};
    }

    /**
     *
     * @param data - údaje
     * @param mult - miera multiplikácie základnej velkosti
     * obdĺžnika ohraničujúceho údaje
     * @param additionalSize - hodnota, ktorá sa pripočíta k
     * výslednému zmultiplikovanému obdĺžniku ohraničujúceho
     * údaje
     * @return - obdĺžnik ohraničujúci údaje v 2D priestore XY
     */
    private Double[] getRectangle(List<Double[]> data, double mult,
        int additionalSize) {

        double maxX = data.get(0)[xIndex];
        double minX = data.get(0)[xIndex];
        double maxY = data.get(0)[yIndex];
        double minY = data.get(0)[yIndex];

        for (int i = 1; i < data.size(); i++) {
            if (maxX < data.get(i)[xIndex]) {
                maxX = data.get(i)[xIndex];
            } else if (minX > data.get(i)[xIndex]) {
                minX = data.get(i)[xIndex];
            }
            if (maxY < data.get(i)[yIndex]) {
                maxY = data.get(i)[yIndex];
            } else if (minY > data.get(i)[yIndex]) {
                minY = data.get(i)[yIndex];
            }
        }
```

```
        double distX = Math.abs(maxX - minX) * (mult - 1) / 2;
        double distY = Math.abs(maxY - minY) * (mult - 1) / 2;

        if (distX < additionalSize) {
            distX = additionalSize;
        }
        if (distY < additionalSize) {
            distY = additionalSize;
        }

        return new Double[]{minX - distX, maxX + distX, minY -
            distY, maxY + distY};
    }

    /**
     *
     * @param data - údaje
     * @param maxGap - maximálna medzera medzi susediacimi údajmi
     * @param sizeLimit - velkostný limit skupiny
     * @return - vráti maximálnu skupinu susediacich údajov v rámci
     * povolenej medzery medzi nimi
     */
    List<Double[]> getShapedCore(List<Double[]> data, double maxGap
        , int sizeLimit) {

        //vytvorí sa mapa susediacich údajov s danou maximálnou
            medzerou
        int[][] neighboursMap = new int[data.size()][data.size()];
        for (int i = 0; i < data.size(); i++) {
            neighboursMap[i][i] = 2;
            double x = data.get(i)[xIndex];
            double y = data.get(i)[yIndex];
            for (int j = 0; j < data.size(); j++) {
                //0 indikuje, že o susednosti ešte nebolo
                //rozhodnuté
```

```
            if (neighboursMap[i][j] == 0) {
                double nx = data.get(j)[xIndex];
                double ny = data.get(j)[yIndex];
                if (Math.sqrt((x - nx) * (x - nx) + (y - ny) *
                    (y - ny)) < maxGap) {
                    //susediace údaje majú v mape 1
                    neighboursMap[i][j] = 1;
                    neighboursMap[j][i] = 1;
                } else {
                    //nesusediace údaje majú v mape 2
                    neighboursMap[i][j] = 2;
                    neighboursMap[j][i] = 2;
                }
            }
        }
    }

    //rekurzívne sa vyhladá najväčšia skupina údajov, prípadne
        skupiny
    boolean[] neighboursChecked = new boolean[data.size()];
    List<List<Integer>> maxGroup = new ArrayList<>();
    maxGroup.add(new ArrayList<>());
    List<Integer> tempGroup;
    for (int i = 0; i < data.size(); i++) {
        if (!neighboursChecked[i]) {
            tempGroup = getNeighbours(i, neighboursMap,
                neighboursChecked);
            if (tempGroup.size() > maxGroup.get(0).size()) {
                maxGroup.clear();
                maxGroup.add(tempGroup);
            } else if (tempGroup.size() == maxGroup.get(0).size
                ()) {
                maxGroup.add(tempGroup);
            }
        }
```

```
        }

        List<Double[]> resultGroup = new ArrayList<>();
        //overí sa, či má skupina dostatočný počet údajov
        if (maxGroup.get(0).size() >= sizeLimit) {
            //index výslednej skupiny
            int index = 0;
            //ak sa našlo viacero skupín s rovnakým množstvom
            //údajov, tak sa vezme tá, ktorá ma najväčšiu
            //hodnotu súčtu počtov signálov svojich údajov
            if (maxGroup.size() > 1) {
                double maxSignalCount = 0;
                for (int i = 0; i < maxGroup.size(); i++) {
                    double tempSignalCount = 0;
                    for (Integer ind : maxGroup.get(i)) {
                        tempSignalCount += data.get(ind)[
                            totalCountIndex];
                    }
                    if (tempSignalCount > maxSignalCount) {
                        maxSignalCount = tempSignalCount;
                        index = i;
                    }
                }
            }
            //do výsledného zoznamu sa zapíšu údaje
            for (Integer idx : maxGroup.get(index)) {
                resultGroup.add(data.get(idx));
            }

        }
        return resultGroup;
    }

    /**
     *
```

```
     * @param row - riadok v mape susediacich údajov
     * @param neighboursMap - mapa susediacich údajov
     * @param neighboursChecked - pole pre označenie prejdených
     * riadkov v mape susediacich údajov
     * @return - zoznamy indexov skupín susediacich údajov
     */
    private List<Integer> getNeighbours(int row, int[][]
        neighboursMap, boolean[] neighboursChecked) {
        neighboursChecked[row] = true;
        List<Integer> result = new ArrayList<>();
        result.add(row);
        for (int i = 0; i < neighboursMap[row].length; i++) {
            if (neighboursMap[row][i] == 1) {
                if (!neighboursChecked[i]) {
                    result.addAll(getNeighbours(i, neighboursMap,
                        neighboursChecked));
                }
            }
        }
        return result;
    }

    /**
     *
     * @param indexHoughTransform - volba metódy Houghovej
     * transformácie
     * @param ranges - rozsahy priamky/roviny
     * (fi, ro / fi, theta, ro)
     * @param size - časová zložka maximálnej kolmej vzdialenosti
     * údajov od detegovanej priamky/roviny
     * @return - vracia hodnotu maximálnej kolmej vzdialenosti
     * údajov od roviny/priamky vzhladom na zadanú velkost
     * časovej zložky
     */
```

```
double getTimeSize(int indexHoughTransform, List<Double[]>
    ranges, double size) {
    if (indexHoughTransform == 0) {
        double thetaAngle1 = (ranges.get(thetaIndexL2)[
            lowerRange] + ranges.get(thetaIndexL2)[upperRange])
            / 2;
        if (ranges.get(thetaIndexL2)[lowerRange] > ranges.get(
            thetaIndexL2)[upperRange]) {
            thetaAngle1 += Math.PI / 2;
        }
        double thetaAngle2 = (ranges.get(thetaIndexL1)[
            lowerRange] + ranges.get(thetaIndexL1)[upperRange])
            / 2;
        if (ranges.get(thetaIndexL1)[lowerRange] > ranges.get(
            thetaIndexL1)[upperRange]) {
            thetaAngle2 += Math.PI / 2;
        }
        //size * Math.sin(fiAngle1) pretože v HoughLine3D je
        //HoughLine2D(x, Gtu),size * Math.cos(fiAngle2)
        //pretože v HoughLine3D je HoughLine2D(Gtu, y)
        double size1 = Math.abs(size * Math.sin(thetaAngle1));
        double size2 = Math.abs(size * Math.cos(thetaAngle2));
        return size1 > size2 ? size1 : size2;
    } else {
        double fiAngle = (ranges.get(fiIndexP)[lowerRange] +
            ranges.get(fiIndexP)[upperRange]) / 2;
        if (ranges.get(fiIndexP)[lowerRange] > ranges.get(
            fiIndexP)[upperRange]) {
            fiAngle += Math.PI / 2;
        }
        return Math.abs(size * Math.cos(fiAngle));
    }
}

/**
```

```
     *
     * @param data - údaje
     * @param enableTimeFilter - volba filtrácie založenej na čase
     * @param dropSeqThresholdTypeTF - volba typu prahovej hodnoty
     * počtu signálov pre zahodenie postupnosti údajov
     * @param dropSeqThresholdTF - prahová hodnota počtu signálov
     * pre zahodenie postupnosti údajov
     * @param singleDataThresholdTF - prahová hodnota počtu
     * signálov pre postupnost s dĺžkou 1
     * @param enableSignalFilter - volba filtrácie založenej
     * na počte signálov
     * @param dropSeqThresholdTypeSF - volba typu prahovej hodnoty
     * počtu signálov pre zahodenie postupnosti údajov
     * @param dropSeqThresholdSF - prahová hodnota počtu signálov
     * pre zahodenie postupnosti údajov
     * @param cutSeqThresholdSF - prahová hodnota počtu signálov
     * pre odrezanie postupnosti údajov
     * @param singleDataThresholdSF - prahová hodnota počtu
     * signálov pre postupnost s dĺžkou 1
     * @return - prefiltrované údaje
     */
    private List<Double[]> filter(List<Double[]> data, boolean
        enableTimeFilter, int dropSeqThresholdTypeTF, int
        dropSeqThresholdTF, int singleDataThresholdTF, boolean
        enableSignalFilter, int dropSeqThresholdTypeSF, int
        dropSeqThresholdSF, int cutSeqThresholdSF, int
        singleDataThresholdSF) {
        //usporiadanie údajov podla ich id
        data.sort(new DataComparator(new Integer[]{idIndex}));
        List<Double[]> tempData = new ArrayList<>();
        List<Double[]> resultData = new ArrayList<>();
        //rozdelenie údajov a ich filtrácia
        int rowCount = 0;
        int currentId = data.get(0)[idIndex].intValue();
        while (rowCount < data.size()) {
```

```
            tempData.clear();
            double lastId = currentId;
            while (currentId == lastId) {
                tempData.add(data.get(rowCount));
                if (!(++rowCount < data.size())) {
                    break;
                }
                currentId = data.get(rowCount)[idIndex].intValue();
            }

            //filter pre časové sekvencie
            if (enableTimeFilter) {
                tempData = timeSeqFilter(tempData,
                    dropSeqThresholdTypeTF, dropSeqThresholdTF,
                    singleDataThresholdTF);
            }
            //filter pre sekvencie hodnôt počtu signálov
            if (enableSignalFilter) {
                tempData = signalSeqFilter(tempData,
                    dropSeqThresholdTypeSF, dropSeqThresholdSF,
                    cutSeqThresholdSF, singleDataThresholdSF);
            }
            resultData.addAll(tempData);
        }
        printData = resultData;
        //uloženie pixelov fázy s menom Filtered result
        stageData.saveStageData(8, returnData(resultData));

        return resultData;
    }

    /**
     * filtrácia údajov na základe ich postupnosti na pixeli
     * z hladiska času
     *
```

```
     * @param data - údaje
     * @param dropSeqThresholdType - volba typu prahovej hodnoty
     * počtu signálov pre zahodenie postupnosti údajov
     * @param dropSeqThreshold - prahová hodnota počtu signálov pre
     * zahodenie postupnosti údajov
     * @param singleDataThreshold - prahová hodnota počtu signálov
     * pre postupnost s dĺžkou 1
     * @return - prefiltrované údaje
     */
    private List<Double[]> timeSeqFilter(List<Double[]> data, int
        dropSeqThresholdType, int dropSeqThreshold, int
        singleDataThreshold) {
        //ak nie sú dodané údaje, tak sa vracia prázdny zoznam
        if (data.isEmpty()) {
            return new ArrayList<>();
        }

        if (dropSeqThresholdType == 1) {
            dropSeqThreshold += patternThreshold;
        }

        //údaje sa časovo usporiadajú, od najvyššej hodnoty po
            najnižšiu
        data.sort(new DataComparator(new Integer[]{gtuIndex}));

        //vytvorí sa retazec, ktorý indikuje, či sú časové medzery
            medzi údajmi
        //menšie alebo rovné danej hodnote časovej medzery, a či je
             splnená podmienka prahu počtu signálov
        int[] neighboursMap = new int[data.size() - 1];
        int max = data.get(0)[totalCountIndex].intValue();
        List<Integer> maxCount = new ArrayList<>();
        maxCount.add(0);
        for (int i = 1; i < data.size(); i++) {
```

```
            if (data.get(i - 1)[gtuIndex] - data.get(i)[gtuIndex]
                <= 1) {
                neighboursMap[i - 1] = 1;
            }
            if (data.get(i)[totalCountIndex] > max) {
                max = data.get(i)[totalCountIndex].intValue();
                maxCount.clear();
                maxCount.add(i);
            } else if (data.get(i)[totalCountIndex] == max) {
                maxCount.add(i);
            }
        }

        if (max <= dropSeqThreshold) {
            return new ArrayList<>();
        }

        List<Double[]> maxSeq;
        if (maxCount.size() == 1) {
            maxSeq = getNeighbours(data, neighboursMap, maxCount.
                get(0));
        } else {

            List<List<Double[]>> sequences = new ArrayList<>();
            for (int index : maxCount) {
                List<Double[]> tempList = getNeighbours(data,
                    neighboursMap, index);
                tempList.sort(new DataComparator(new Integer[]{
                    totalCountIndex}));
                sequences.add(tempList);
            }

            boolean changed = false;
            maxSeq = sequences.get(0);
            for (int i = 1; i < sequences.size(); i++) {
```

```
                List<Double[]> nextSeq = sequences.get(i);
                int length = maxSeq.size() < nextSeq.size() ?
                   maxSeq.size() : nextSeq.size();
                for (int j = 1; j < length; j++) {
                    if (nextSeq.get(j)[totalCountIndex] > maxSeq.
                       get(j)[totalCountIndex]) {
                        maxSeq = nextSeq;
                        changed = true;
                        break;
                    } else if (nextSeq.get(j)[totalCountIndex] <
                       maxSeq.get(j)[totalCountIndex]) {
                        changed = true;
                        break;
                    }
                }
                if (!changed) {
                    maxSeq = maxSeq.size() > nextSeq.size() ?
                       maxSeq : nextSeq;
                }
                changed = false;
            }
        }
        if (maxSeq.size() == 1 && !(max > singleDataThreshold)) {
            return new ArrayList<>();
        }

        return maxSeq;
    }

    /**
     *
     * @param data - údaje
     * @param neighboursMap - mapa časovo susediacich údajov
     * @param index - index údaja
     * @return - skupina časovo susediacich údajov
```

```
     */
    private List<Double[]> getNeighbours(List<Double[]> data, int[]
        neighboursMap, int index) {
        List<Double[]> result = new ArrayList<>();
        result.add(data.get(index));
        for (int i = index; i < neighboursMap.length; i++) {
            if (neighboursMap[i] == 1) {
                result.add(data.get(i + 1));
            } else {
                break;
            }
        }
        for (int i = index - 1; i > - 1; i--) {
            if (neighboursMap[i] == 1) {
                result.add(data.get(i));
            } else {
                break;
            }
        }
        return result;
    }

    /**
     * filtrácia údajov na základe ich postupnosti na pixeli
     * z hladiska počtu signálov
     *
     * @param data - údaje
     * @param dropSeqThresholdType - volba typu prahovej
     * hodnoty  počtu signálov pre zahodenie postupnosti údajov
     * @param dropSeqThreshold - prahová hodnota počtu signálov
     * pre zahodenie postupnosti údajov
     * @param cutSeqThreshold - prahová hodnota počtu signálov
     * pre odrezanie postupnosti údajov
     * @param singleDataThreshold - prahová hodnota počtu
     * signálov pre postupnost s dĺžkou 1
```

```
     * @return - prefiltrované údaje
     */
    private List<Double[]> signalSeqFilter(List<Double[]> data, int
        dropSeqThresholdType, int dropSeqThreshold, int
       cutSeqThreshold, int singleDataThreshold) {

        //ak nie sú dodané údaje, tak sa vracia prázdny zoznam
        if (data.isEmpty()) {
            return new ArrayList<>();
        }

        if (dropSeqThresholdType == 1) {
            dropSeqThreshold += patternThreshold;
        }

        //hladá sa maximálna hodnota počtu signálov
        double max = 0;
        List<Double[]> maxList = new ArrayList<>();
        for (Double[] row : data) {
            if (row[totalCountIndex] > max) {
                max = row[totalCountIndex];
                maxList.clear();
                maxList.add(row);
            } else if (row[totalCountIndex] == max) {
                maxList.add(row);
            }
        }

        //ak údaj s maximálnym počtom signálov nie je nad prahom
        //počtu signálov pre zahodenie postupnosti údajov
        if (max <= dropSeqThreshold) {
            return new ArrayList<>();
        }
```

```
        //údaje sa časovo usporiadajú, od najvyššej hodnoty po
            najnižšiu
        data.sort(new DataComparator(new Integer[]{gtuIndex}));

        //údaje sa rozdelia do dvoch skupín
        //- v prvej sú údaje skoršie ako údaj s maximálnou hodnotou
            počtu signálov
        //- v druhej je údaj s maximálnou hodnotou počtu signálov a
            údaje neskoršie ako
        //údaj s maximálnou hodnotou počtu signálov
        Double[] maxRow = maxList.get(0);
        List<Double[]> rightList = new ArrayList<>();
        List<Double[]> leftList = new ArrayList<>();
        for (Double[] row : data) {
            if (row[gtuIndex] < maxRow[gtuIndex]) {
                leftList.add(row);
            } else {
                rightList.add(row);
            }
        }

        //zoznam s výstupom
        List<Double[]> resultList = new ArrayList<>();

        if (!leftList.isEmpty()) {
            if (maxRow[totalCountIndex] > cutSeqThreshold) {
                resultList.add(leftList.get(0));
                if (leftList.get(0)[totalCountIndex] >
                    cutSeqThreshold) {
                    //inak sa pridá prvý údaj do výsledného zoznamu
                        a zvyšné sa pridávajú iba ak je splnená
                        podmienka klesajúcej postupnosti
                    for (int i = 1; i < leftList.size(); i++) {
                        if (leftList.get(i)[totalCountIndex] == max
                            || leftList.get(i)[totalCountIndex] <
```

```
                        leftList.get(i - 1)[totalCountIndex]) {
                        resultList.add(leftList.get(i));
                        if (leftList.get(i)[totalCountIndex] <=
                            cutSeqThreshold) {
                            break;
                        }
                    } else {
                        break;
                    }
                }
            }
        }
    }

    //ak má druhá skupina iba 1 údaj, tak sa pridá do
    //výsledného zoznamu
    if (rightList.size() < 2) {
        resultList.addAll(rightList);
    } else {
        //inak sa pridá údaj s maximálnou hodnotou počtu
        //signálov do výsledného zoznamu a zvyšné
        //sa pridávajú iba ak je splnená podmienka klesajúcej
            postupnosti
        resultList.add(rightList.get(rightList.size() - 1));
        if (rightList.get(rightList.size() - 1)[totalCountIndex
            ] > cutSeqThreshold) {
            if (rightList.get(rightList.size() - 1)[
                totalCountIndex] > cutSeqThreshold) {
                for (int i = rightList.size() - 2; i > -1; i--)
                    {
                    if (rightList.get(i)[totalCountIndex] ==
                        max || rightList.get(i)[totalCountIndex]
                        < rightList.get(i + 1)[totalCountIndex
                        ]) {
                        resultList.add(rightList.get(i));
```

```
                        if (rightList.get(i)[totalCountIndex]
                            <= cutSeqThreshold) {
                            break;
                        }
                    } else {
                        break;
                    }
                }
            }
        }

        if (resultList.size() == 1 && resultList.get(0)[
            totalCountIndex] <= singleDataThreshold) {
            return new ArrayList<>();
        }
        return resultList;
    }

    /**
     * odstránia sa určité osamotené pixely
     *
     * @param data - údaje
     * @param gap - maximálna vzdialenost, v ktorej sa hladá nejaký
     * susediaci pixel
     * @return - skupina pixelov
     */
    List<Double[]> getShapedCore(List<Double[]> data, double gap) {

        List<Double[]> resultGroup = new ArrayList<>();

        for (int i = 0; i < data.size(); i++) {
            double x = data.get(i)[xIndex];
            double y = data.get(i)[yIndex];
            for (int j = 0; j < data.size(); j++) {
```

```
                if (i != j) {
                    double nx = data.get(j)[xIndex];
                    double ny = data.get(j)[yIndex];
                    if (Math.sqrt((x - nx) * (x - nx) + (y - ny) *
                       (y - ny)) < gap) {
                        resultGroup.add(data.get(i));
                        break;
                    }
                }
            }
        }

        return resultGroup;
    }

    /**
     * pridanie pixelov okolo jadra spŕšky
     *
     * @param cleanCoreData - údaje pixelov jadra
     * @param rectangleData - údaje pixelov obdĺžnikového
     * ohraničenia spŕšky
     * @param gap - maximálna vzdialenost, v ktorej sa hladá
     * susediaci pixel
     * @return - skupina pixelov
     */
    private List<Double[]> getWrap(List<Double[]> cleanCoreData,
       List<Double[]> rectangleData, double gap) {
        if (gap == 0) {
            return cleanCoreData;
        }
        Set<Double[]> result = new HashSet<>();
        for (Double[] row : cleanCoreData) {
            double x = row[xIndex];
            double y = row[yIndex];
            for (Double[] row1 : rectangleData) {
```

```
                double nx = row1[xIndex];
                double ny = row1[yIndex];
                if (Math.sqrt((x - nx) * (x - nx) + (y - ny) * (y -
                    ny)) < gap) {
                    result.add(row1);
                }
            }
        }
        return new ArrayList<>(result);
    }

    /**
     * k jednotlivým indexom sa priradia vsetky ich údaje
     *
     * @param indexes - indexy pixelov spŕšky
     * @param allSortedData - údaje zotriedené podla indexov
     * pixelov
     * @return - údaje zadaných pixelov
     */
    private List<Double[]> getAllData(Set<Integer> indexes, List<
        Double[]> allSortedData) {
        List<Double[]> result = new ArrayList<>();
        int rowCount = 0;
        int currentId = allSortedData.get(0)[idIndex].intValue();
        while (rowCount < allSortedData.size()) {
            int lastId = currentId;
            if (indexes.contains(currentId)) {
                while (currentId == lastId) {
                    result.add(allSortedData.get(rowCount));
                    if (!(++rowCount < allSortedData.size())) {
                        break;
                    }
                    currentId = allSortedData.get(rowCount)[idIndex
                        ].intValue();
                }
```

```
            } else {
                while (currentId == lastId) {
                    if (!(++rowCount < allSortedData.size())) {
                        break;
                    }
                    currentId = allSortedData.get(rowCount)[idIndex
                        ].intValue();
                }
            }
        }
        return result;
    }

    /**
     *
     * @return - prahová hodnota počtu signálov pre vzor
     */
    public int getPatternThreshold() {
        return patternThreshold;
    }

    /**
     *
     * @param enableTimeFilter - volba filtrácie založenej na čase
     * @param dropSeqThresholdTypeTF - volba typu prahovej hodnoty
     * počtu signálov pre zahodenie postupnosti údajov
     * @param dropSeqThresholdTF - prahová hodnota počtu signálov
     * pre zahodenie postupnosti údajov
     * @param singleDataThresholdTF - prahová hodnota počtu
     * signálov pre postupnost s dĺžkou 1
     * @param enableSignalFilter - volba filtrácie založenej na
     * počte signálov
     * @param dropSeqThresholdTypeSF - volba typu prahovej hodnoty
     * počtu signálov pre zahodenie postupnosti údajov
     * @param dropSeqThresholdSF - prahová hodnota počtu signálov
```

```
     * pre zahodenie postupnosti údajov
     * @param cutSeqThresholdSF - prahová hodnota počtu signálov
     * pre odrezanie postupnosti údajov
     * @param singleDataThresholdSF - prahová hodnota počtu
     * signálov pre postupnost s dĺžkou 1
     * @return - pixely prefiltrovaných údajov
     */
    public Integer[] returnFiltered(boolean enableTimeFilter, int
        dropSeqThresholdTypeTF, int dropSeqThresholdTF, int
        singleDataThresholdTF, boolean enableSignalFilter, int
        dropSeqThresholdTypeSF, int dropSeqThresholdSF, int
        cutSeqThresholdSF, int singleDataThresholdSF) {
        if (lastOutputData != null) {
            List<Double[]> data = filter(lastOutputData,
                enableTimeFilter, dropSeqThresholdTypeTF,
                dropSeqThresholdTF, singleDataThresholdTF,
                enableSignalFilter, dropSeqThresholdTypeSF,
                dropSeqThresholdSF, cutSeqThresholdSF,
                singleDataThresholdSF);
            printer.printInfo("Filtered result :", data);
            printer.write(data, "output0", 0);
            printer.write(data, "output1", 1);
            return returnData(data);
        } else {
            return null;
        }
    }

    /**
     *
     * @param data - údaje
     * @return - jedinečné identifikátory pixelov
     */
    private Integer[] returnData(List<Double[]> outputData) {
        Set<Integer> indexes = new HashSet<>();
```

```
        for (Double[] row : outputData) {
            if (row[signalCountIndex] > 0) {
                indexes.add(-row[idIndex].intValue());
            } else {
                indexes.add(row[idIndex].intValue());

            }
        }
        return indexes.toArray(new Integer[indexes.size()]);
    }

    /**
     *
     * @param data - údaje
     * @return - jedinečné identifikátory pixelov
     */
    private Set<Integer> getIndexes(List<Double[]> data) {
        Set<Integer> indexes = new HashSet<>();
        for (Double[] row : data) {
            indexes.add(row[idIndex].intValue());
        }
        return indexes;
    }

    /**
     * výpis výstupných údajov
     */
    public void printData() {
        if (printData != null) {
            printData.sort(new DataComparator(new Integer[]{idIndex
                , gtuIndex}));
            printer.printData(printData);
        }
    }
```

```
    /**
     *
     * @param i - poradové číslo fázy rozpoznávania
     * @return - identifikátory pixelov výstupných údajov
     * danej fázy rozpoznávania
     */
    public Integer[] getStageData(int i) {
        return stageData.getStageData(i);
    }

    /**
     * @param i - poradové číslo fázy rozpoznávania
     * @return - názov fázy rozpoznávania
     */
    public String getStageName(int i) {
        return stageData.getStageName(i);
    }

    /**
     * @param i - poradové číslo fázy rozpoznávania
     * @return - nasledujúce poradové číslo fázy rozpoznávania
     */
    public int getNext(int i) {
        return stageData.getNext(i);
    }

    /**
     * @param i - poradové číslo fázy rozpoznávania
     * @return - predchádzajúce poradové číslo fázy rozpoznávania
     */
    public int getPrevious(int i) {
        return stageData.getPrevious(i);
    }
}
```