UNIVERSITATEA NATIONALA DE STIINTA SI TEHNOLOGIE
POLITEHNICA BUCURESTI
FACULTATEA DE AUTOMATICĂ ȘI CALCULATOARE
DEPARTAMENTUL DE CALCULATOARE

PROIECT DE DIPLOMA

Orientare in Spatiu folosind ORB-SLAM
BUCUREȘTI

Alfred Andrei Pietraru

Coordonator științific:

Prof. dr. ing. Anca Morar

2025

NATIONAL UNIVERSITY OF SCIENCE AND TECHNOLOGY
POLITEHNICA BUCHAREST
FACULTY OF AUTOMATIC CONTROL AND COMPUTERS
COMPUTER SCIENCE AND ENGINEERING DEPARTMENT

DIPLOMA PROJECT

Spatial Orientation using ORB-SLAM
BUCHAREST

Alfred Andrei Pietraru

Thesis advisor:

Prof. dr. ing. Anca Morar

2025

CUPRINS

SINOPSIS

Această lucrare reprezintă o reimplementare a algoritmului ORB-SLAM2, utilizat pentru localizarea și cartografierea unui mediu interior necunoscut. Sunt abordate concepte precum extragerea de trăsături folosind descriptorul ORB, realizarea sarcinilor de feature matching între cadre, utilizarea algoritmului Bundle Adjustment pentru optimizare, implementat prin biblioteca Ceres, precum și relocalizarea poziției camerei prin vectori obținuți cu metoda bag-of-words, implementată în DBOW2. Algoritmul este testat pe setul de date TUM RGB-D, iar rezultatele obținute sunt comparabile cu valorile de tip ground truth. Se explorează, de asemenea, integrarea unei rețele neuronale de tip FastDepth pentru estimarea matricii de adâncime în cadrul fluxului de procesare al algoritmului.Implementarea este realizată în C++, având un număr de linii de cod de aproximativ trei ori mai mic decât cel din implementarea oficială. De asemenea, algoritmul este optimizat pentru a funcționa pe perioade lungi de timp datorită unui management eficient al memoriei.

ABSTRACT

This work represents a reimplementation of the ORB-SLAM2 algorithm, used for localization and mapping of an unknown indoor environment. Concepts such as feature extraction using the ORB descriptor, performing feature matching tasks between frames, utilizing the Bundle Adjustment algorithm for optimization implemented through the Ceres library, as well as camera position relocalization through vectors obtained with the bag-of-words method implemented in DBOW2 are addressed. The algorithm is tested on the TUM RGB-D dataset, and the obtained results are comparable with ground truth values. The integration of a FastDepth neural network for depth matrix estimation within the algorithm's processing pipeline is also explored. The implementation is carried out in C++, having a number of lines of code approximately three times smaller than that of the official implementation. Additionally, the algorithm is optimized to function over long periods of time due to efficient memory management.

1 INTRODUCERE

1.1 Context

SLAM, Simultanious Localization and Mapping reprezintă o clasă de algoritmi de planificare și control a mișcării unui agent prin mediu pentru a construi un model al spațiului cât mai apropiat de realitate. Aceste clase au câștigat atenția publicului în ultimii ani, lucru care a condus la dezvoltarea numeroaselor variante prezente la momentul curent pe piață, fiecare adaptat pentru mediul și tipul de senzori folosiți. O atenție deosebită a fost acordată sistemelor de tip Visual SLAM din mai multe motive: camerele video sunt unul dintre cele mai comune tipuri de senzori, există o multitudine de tehnici de Computer Vision pentru procesarea imaginilor, iar filozofia pe care acești algoritmi o urmează este asemănătoare cu modul în care creierul uman interpretează mediul înconjurator. Astfel, sunt alese un set de puncte din spațiu care vor fi considerate referințe, iar unghiul din care acestea sunt observate poate oferi informații despre poziția agentului în mediu. Astăzi, cei mai populari algoritmi de tip Visual SLAM îmbină domenii precum Machine Learning, Computer Vision, Robotică și Matematică, pentru a crea sisteme robuste, capabile să îndeplinească o varietate de sarcini.

1.2 Problemă

Creați un sistem capabil să exploreze un mediu necunoscut din interior. Acesta trebuie să creeze o hartă a zonei parcurse, să reconstruiască traseul estimând poziția camerei pentru fiecare cadru citit, să fie tolerant la erori și să poată opera pentru perioade de timp îndelungate.

1.3 Obiective

Obiectivele principale ale lucrării sunt:

- studierea, configurarea și implementarea unui sistem de tip ORB-SLAM2[1]

- testarea performanțelor folosind seturi de date consacrate, utilizarea unui video realizat de mine pentru etapa de evaluare
- testarea unei implementări în care o cameră tip RGBD să fie înlocuită cu o cameră monoculară tradițională și o rețea neurală, această sarcină presupunând alegerea unei arhitecturi potrivite și compararea celor două metode
- prezentarea problemelor întâlnite în etapa de dezvoltare și a unor direcții de îmbunătățire

1.4 Soluția propusă

Lucrarea propune implementarea algoritmului ORB-SLAM2 adaptat pentru camerele de tip RGBD și testarea acestuia pe setul de date TUM RGBD[2]. Se vor analiza aspecte precum acuratețea traiectoriei și îndeplinirea condițiilor de funcționare în timp real. De asemenea, se va încerca înlocuirea matricei de adâncime creată de camera RGBD, cu o hartă de distanțe calculată de către rețeaua neurală FastDepth[3].

1.5 Rezultatele obținute

Rezultatele au arătat că implementarea algoritmului ORB-SLAM2, folosind o cameră tip RGBD prezintă rezultate bune în medii indoor și că poate funcționa în timp real cu viteze de aproximativ 10-15 cadre pe secundă. Utilizarea unei rețele neurale pentru estimarea distanței nu a avut rezultatele dorite, algoritmul fiind capabil să funcționeze pentru cel mult 50 de cadre.

1.6 Structura lucrării

Lucrarea este structurată în mai multe capitole. Introducerea oferă contextul lucrării și definește problema abordată, obiectivele și soluția propusă. Capitolul 2 descrie cerințele funcționale, nonfuncționale și motivația. Capitolul 3 realizează un studiu de piață asupra metodelor curente, separându-le in trei categorii. Capitolul 4 descrie la nivel conceptual soluția propusă: algoritmii folosiți și componentele logice. Capitolul 5 detaliază modul în care este realizată evaluarea și rezultatele obținute. Ultimul capitol prezintă impresii personale despre acest

proiect, lucruri pe care le-aș putea îmbunătăți, problemele întâlnite și direcțiile viitoare.

2 CERINȚE ȘI MOTIVAȚIE

2.1 Motivație

Filmele, cărțile și jocurile pe calculator ne arată un viitor al omenirii în care roboți inteligenți îndeplinesc sarcini complexe, sunt capabili să discute cu noi și să se adapteze mediului înconjurător. Deși în momentul de față suntem departe de a crea un framework suficient de complex pentru un asemenea agent, consider că algoritmii din categoria Visual SLAM sunt un pas spre această direcție. Îmi este greu să îmi imaginez un robot care să poată simula compartamentul uman și să nu fie capabil să se deplaseze și să înțeleagă mediul în care se află. Pentru noi aceste lucruri sunt adânc înrădăcinate în modul în care funcționeaza creierul, dar pentru un calculator, a fost nevoie de aproape 30 de ani de cercetare pentru a crea algoritmi suficient de complecși pentru a îndeplini sarcini minimale de orientare, cum ar fi capacitatea de învățare a mediului și de poziționare a agentului în spațiu. Prima dată, conceptul de SLAM a fost definit în anul 1995 în această lucrare[4], iar de atunci a avut parte de o dezvoltare continuă. În ziua de azi, această categorie de algoritmi are numeroase aplicații practice:

- realizarea sarcinilor din viața de zi cu zi: cazul roboților de curățenie sau a celor care transportă obiecte în interiorul unei clădiri
- în aplicații medicale, ca de exemplu asistența pentru persoanele nevăzătoare
- în aplicații militare: cartografierea zonelor necunoscute
- în interiorul clădirilor sau în medii ostile unde nu este acces la un sistem de coordonate globale cum ar fi poziția dată de un sistem GPS
- în aplicații industriale, inspecții asupra instalației sau depozitelor, detectarea unor erori și raportarea zonei în care au fost observate

Există numeroase aplicații pentru sistemele SLAM, întrucât toate pleacă de la faptul că agentul trebuie să creeze o hartă a mediului și să înțeleagă care este poziția acestuia. Pe măsură ce aceste sisteme vor evolua, va crește și complexitatea sarcinilor pe care le pot îndeplini.

2.2 Cerințe Funcționale și Nonfuncționale

Din punct de vedere al cerințelor funcționale, algoritmul ORB-SLAM va primi un video realizat cu o cameră de tip RGBD și va returna 2 fișiere text. Unul va conține estimarea poziției pentru fiecare cadru în parte, iar celălalt va avea salvată harta mediului înconjurator, alcătuită dintr-un nor de puncte în spațiu și cadrele cheie asociate acestora. Algoritmul va avea o interfață grafică minimală, reprezentată din 2 ferestre. Cea din stânga va conține o reprezentare pentru cadrul curent procesat. Acesta va avea culoarea albastru, iar celelalte cadre cheie salvate în harta vor avea cu verde și cu roșu punctele din spațiu. Toate acestea vor alcătui harta mediului înconjurător. În fereastra din dreapta va fi afișat fiecare cadru în format alb negru, iar cu roșu vor fi marcate feature-urile detectate de algoritmul ORB. Cadrele cheie consecutive sunt conectate între ele prin intermediul unei drepte de culoare neagră. Aceste segmente concatenate vor genera traseul realizat de camera în video.

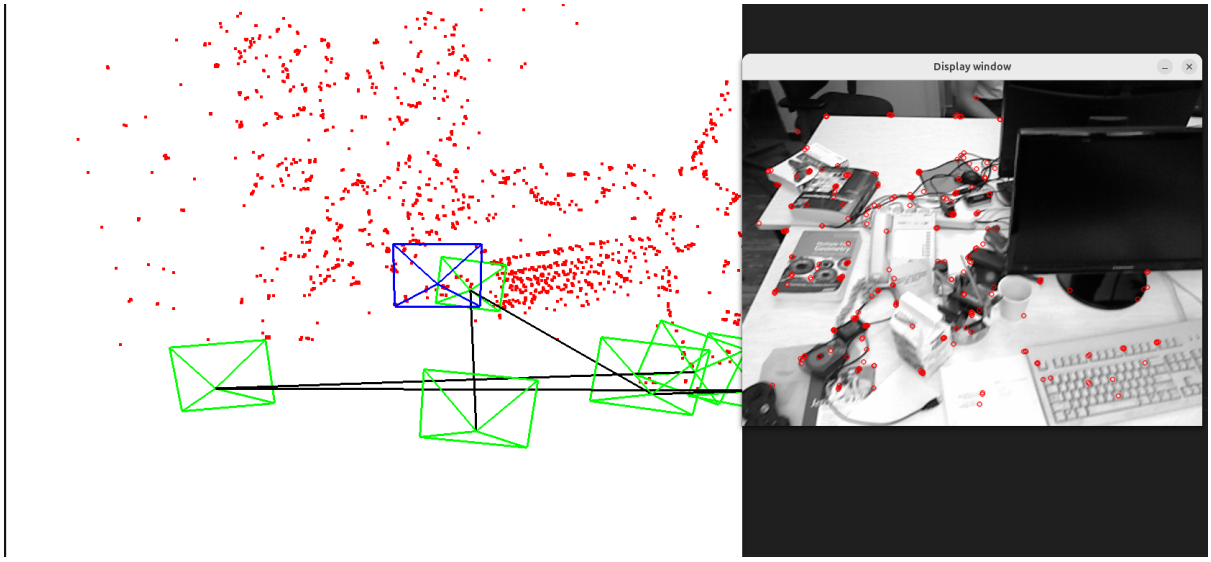

Figura 1: Interfața grafică, stânga reprezentarea hărții, dreapta extragere feature-uri cu ORB

Ca cerințe nonfuncționale, algoritmul trebuie să meargă în timp real, să proceseze între 10-15 cadre pe secundă și să poată fi folosit în sisteme embedded în care unitatea centrală de procesare are cel mult 2 core-uri și nu poate folosi GPU-ul. Sistemul trebuie să fie rezistent la erorile de estimare pentru fiecare imagine primită, să optimizeze atât harta, cât și traseul realizat și să aibă capacitatea de relocalizare în situația în care urmărirea cadru cu cadru eșuează. Mediul în care poate să opereze este unul static, de mici dimensiuni și nu poate accesa coordonatele globale ale poziției sale prin intermediul tehnologiilor precum GPS.

3 STUDIU DE PIAȚĂ

3.1 State of the art Visual SLAM

Cei mai noi algoritmi de Visual SLAM funcționează acum folosind tehnici de deep learning. În continuare vor fi prezentate lucrările care au reprezentat SOA, până la începutul anului 2025, grupate pe categorii în funcție de modul în care sunt folosite tehnicile de deep learning. Am considerat potrivită împărțirea algoritmilor pe 3 nivele, în funcție de gradul de utilizare a tehnicilor de deep learning pentru realizarea operațiilor specifice sistemelor SLAM:

1. Algoritmi care se bazează fundamental pe tehnici de deep learning pentru a funcționa, DPV-SLAM[5], ESLAM[6].
2. Algoritmi care sunt la granița dintre metodele clasice și cele deep learning, în care doar anumite componente sunt îmbunătățite cu ajutorul rețelelor neurale: Light-SLAM[7], HFNet-SLAM[8].
3. Algoritmii clasici care nu folosesc deloc rețele neurale: ORB-SLAM3[9], SVO[10].

Deep Patch Visual SLAM (DPV-SLAM), este un sistem SLAM care folosește deep neural networks. Acesta împarte operațiile care trebuie realizate în 2 categorii: frontend-ul care realizează sarcina de visual odometry cu ajutorul unui sistem derivat din Deep Patch Visual Odometry (DPVO)[11] și partea de backend alcătuită din 2 metode de loop closure: proximity loop closure și classical loop closure. Algoritmul are nevoie între 5-6 GB de memorie pe GPU pentru a putea rula. O altă problemă o reprezintă proximity loop closure. Aceasta funcționează cu ajutorul unei hărți foarte dense, de feature-uri obținute cu ajutorul metodei de optical flow, fiind imposibil de folosit în timp real fără a folosi GPU.

ESLAM sau Efficient Dense Visual SLAM using Neural Implicit Maps este un sistem de SLAM monocameră RGB-D care folosește o combinație între o hartă densă 3D, reprezentată de o rețea neurală implicită și un backend optimizat geometric pentru estimarea matricei de poziție

a camerei. Avantajele acestei implementări sunt faptul că produce o hartă densă și detaliată, poate reconstrui detalii chiar și în zone parțial observate. Problema acestei implementări este că necesită un GPU și resurse mari de calcul, nefiind potrivită pentru dispozitivele embedded.

Implementarea Light-SLAM folosește ORB-SLAM2 la bază. Partea de backend, alcătuită din local mapping și loop closure, folosește în continuare metode clasice. Local mapping este responsabil de optimizarea hărții, iar loop closure de recunoașterea zonelor prin care a mai trecut algoritmul și de închiderea buclelor traiectoriei. Acestea sunt realizate folosind metode clasice. Extragerea de keypoint-uri, descriptori și sarcina de matching între descriptorii a două imagini consecutive este realizată de 2 rețele neurale. Acest sistem poate funcționa în timp real dacă se poate folosi un GPU, dar cea mai mare problemă o reprezintă faptul că rețelele neurale nu sunt capabile să găsească feature-uri cu acuratețe suficient de bună în zone care nu seamănă cu ceea ce au întâlnit în setul de date de antrenare. Astfel, algoritmul nu are garanția că va funcționa în situații critice.

HFNet-SLAM este o metodă construită pe baza ORB-SLAM3 și se folosește de arhitectura HF-Net, având straturile de convoluție separate în depthwise convolution și pointwise convolution, asemănător cu modul în care este gândit Mobile_Net. În loc să folosească 2 rețele neurale, precum Light-SLAM, aceasta folosește una singură, atât pentru extragerea keypoint-urilor și a descriptorilor, cât și pentru feature-urile globale, folosite în sarcinile de loop closure. Pe lângă problema rețelei neurale care trebuie să ruleze pe GPU și a feature-urilor instabile extrase din imagini pentru zone care nu au fost întâlnite în setul de antrenare, algoritmul calculează pentru fiecare cadru în parte feature-urile sale globale, lucru care adaugă un overhead computațional inutil. De asemenea, rețeaua neurală nu extrage keypoint-urile pe mai multe nivele, astfel, obținându-se prea puține puncte pentru a menține sistemul stabil în mediile slab texturate.

ORB-SLAM3 este continuarea implementării algoritmului ORB-SLAM2 pe care l-am ales eu. Acesta a apărut în 2021 și până în acest moment este cea mai complexă și completă metodă de a estima traiectoria camerei și de a reconstrui o hartă de puncte a mediului înconjurător folosind doar metode clasice. În comparație cu precedesorul acestuia, implementarea de

ORB-SLAM3 folosește datele obținute de la Inertial Measurement Unit (IMU) și optimizează rezultatele primite folosind tehnica de Maximum a Posteriori Estimation (MAP). Față de versiunea anterioară a algoritmului, cea curentă lucrează cu multiple hărți locale, respectiv noruri de puncte în spațiu. În momentul în care ORB-SLAM3 pierde orientarea și trebuie să execute o relocalizare, sistemul generează o nouă hartă pentru a menține fluidă procesarea cadru cu cadru. În situația în care tracker-ul recunoaște zona în care a ajuns, acesta încearcă să unească hărțile între ele pentru a reconstrui întreg mediul. Am considerat că zona pe care o parcurge agentul nostru este de mici dimensiuni și nu ar avea nevoie de un sistem foarte de complex de interconectare a hărților generate, iar utlizarea acestuia ar adăuga un overhead nejustificat. În plus, nu avem acces la datele ce aparțin componentei IMU.

SVO sau Semi-Direct Visual Odometry for Monocular and Multi-Camera Systems este un exemplu de algoritm de tip SLAM care folosește doar 2 thread-uri: unul responsabil de urmărirea cadru cu cadru și unul pentru optimizarea hărții. Acesta folosește gradienții pixelilor în imagini pentru a crea feature-uri, nu doar contururile obiectelor. În cazul algoritmilor din familia ORB-SLAM care încearcă să optimizeze eroarea de proiecție a punctelor din spațiu, aici se folosesc metode directe și trebuie minimizată eroarea fotometrică a pixelilor aflați în apropierea contururilor obiectelor. Este printre cei mai rapizi algoritmi de SLAM, procesând peste 100 de cadre pe secundă pe un CPU. Totuși, acesta generează o hartă densă, dar cu puncte de slabă calitate care nu pot fi refolosite, nu există capacitate de relocalizare și este dificil de extins. Așadar, acesta ar putea fi considerat a doua cea mai bună opțiune după ORB-SLAM2.

În ciuda faptului că algoritmul ORB-SLAM2 a aparut în 2017, acesta rămâne în continuare un exemplu de sistem bine gândit, cu multe posibilități de extindere și capacitatea de a fi adaptat la cerințele din zilele noastre. Îndeplinește cu succes toate cerințele funcționale și nonfuncționale pe care sistemul ar trebui să le aibă: poate fi folosit în real time, implementarea procesând aproximativ 15 cadre pe secundă, creează o hartă a mediului înconjurător pe care o poate optimiza, are capacitate de relocalizare și corectează erorile de estimare care apar în timp prin mecanismul de loop closure. Acesta poate rula exclusiv pe CPU, fiind potrivit atât pentru vehicule la sol, cât și pentru drone. Nu are nevoie de o estimare a poziției globale,

putând fi folosit în medii ostile unde terenul este complet necunoscut.

4 SOLUȚIE PROPUSĂ

Soluția mea propune implementarea algoritmului ORB-SLAM2. Acesta are 2 scopuri fundamentale:

- să estimeze pentru fiecare cadru matricea de poziție și orientarea camerei, reconstruind astfel traseul parcurs în timpul funcționării algoritmului
- să creeze o hartă locală a mediului înconjurător pentru a memora zonele prin care a mai trecut și pentru a îmbunătăți estimarea traiectoriei

Matricea de poziție și orientare a camerei (pose matrix) are dimensiuni 4×4 și are formatul prezentat mai jos, unde R reprezintă matricea de rotație 3×3, iar t este vectorul coloană de dimensiune 3, reprezentând translația față de punctul de origine $(0, 0, 0)$. Aceasta mai este denumită și matricea de conversie din sistemul de coordonate global (world space) în sistemul de coordonate al camerei (camera space), fiind notată în implementarea mea cu T_{cw}. Inversa acestei matrice realizează operația de conversie dintre cele 2 sisteme de coordonate în sens opus.

$$T_{cw} = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix}, \quad T_{wc} = \begin{bmatrix} R & -R^t t \\ 0 & 1 \end{bmatrix} \tag{1}$$

Algoritmul primește ca date de intrare sursa de la care se vor obține imaginile de tip RGB, matricile de adancime pentru fiecare cadru, parametrii de distorsiune si matricea parametrilor interni ai camerei, având dimensiunea 3×3, notată în mod tradițional cu K. Această matrice trebuie modificată de fiecare dată când este schimbată camera cu care se realizează filmarea. Dacă se execută operații de modificare a dimensiunii imaginilor față de modul în care acestea sunt obținute natural, atât K, cât și parametrii de distorsiune nu o să mai fie valizi. Matricea parametrilor camerei are următorul format:

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{2}$$

Algoritmul va returna un fișier text în care se vor afla estimările matricilor de poziție împreună cu timestamp-ul asociat pentru fiecare cadru în parte în ordine cronologică. Rezultatul poate fi comparat cu fișiere care conțin valorile reale și care respectă același format pentru a verifica corectitudinea algoritmului. Diagrama UML prezintă etapele principale ale algoritmului. O componentă reprezintă o funcție care se va executa pentru fiecare cadru procesat. În continuare, o să detaliez logica fiecărui bloc din punctul de vedere al algoritmilor folosiți și al valorilor de intrare și de ieșire asociate acestora.

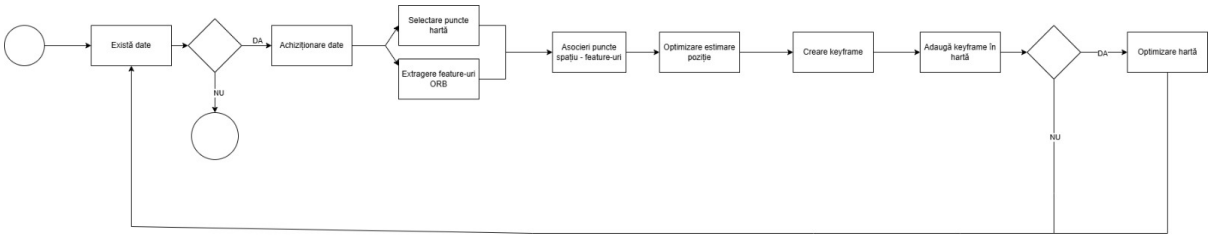

Figura 2: Diagrama UML de activități a întregului sistem

4.1 Achiziția datelor

Scopul acestei componente este citirea imaginii de tip RGB, a matricei de adâncime și estimarea poziției curente a camerei pe baza măsurătorilor anterioare. În viitor, o altă funcție a acestei componente ar putea fi extragerea datelor de la alți senzori precum un giroscop sau accelerometru, pentru a obține informații suplimentare cu privire la orientarea și distanța efectuată de către cameră. Acest lucru ar putea duce la o îmbunătățire considerabilă a estimării inițiale a poziției. Imaginile pot proveni de la cameră în timp real, dintr-un video sau dintr-un set de date. Matricea de adâncime poate fi obținută de la o cameră RGBD/Stereo sau prin utilizarea unei rețele neurale pentru estimarea distanțelor.

4.2 Extragerea trăsăturilor

Ca date de intrare această componentă primeste doar imaginea de tip RGB și extrage aproximativ 1000 de trăsături și descriptori asociați acestora. Trăsăturile sunt zone de interes în imagine care pot fi folosite pentru a detecta obiecte sau găsi asocieri între cadrele consecutive. Acestea sunt numite și keypoint-uri în literatura de specialitate iar librării precum OpenCV au structuri de date dedicate pentru acestea. O trăsătură poate fi interpretată matematic ca o zonă în care apare o schimbare bruscă a gradientului culorii. De cele mai multe ori, astfel de variații se regăsesc în zonele de frontieră dintre obiecte, deoarece apare o diferență de culoare și implicit una de intensitate luminoasă. Zonele slab texturate, cum ar fi cerul sau pereții în interiorul unei clădiri au valori asemanatoare pentru toti pixeli de pe suprafata. Dacă s-ar folosi un keypoint dintr-o astfel de zonă, ar fi dificil de spus cu exactitate de unde a fost extras. Acesta ar putea fi asociat mai multe coordonate în imagine. În schimb, o cameră complet mobilată ar fi o zonă puternic texturată iar un algoritm de detectie de keypoint-uri ar putea să gasească ușor 1000 de trăsături pe care să le folosească. Dacă algoritmul nu reușește să găsească suficiente keypoint-uri pentru a face urmărirea intre cadre, de obicei minim 500, operația ar eșua. Din această cauză algoritmul de ORB-SLAM2 dă rezultate eronate în zonele slab texturate. Dacă algoritmul de extragere funcționează corect iar traiectoria camerei este una stabilă, fără schimbări bruște ale direcției de deplasare, trăsături similare ar trebui să fie observate în ambele imagini. Asocierile dintre ele, ne pot da informații despre modul în care s-a deplasat camera între 2 cadre. Problema este că aceste keypoint-uri nu pot fi comparate direct între ele, din aceasta cauză ne folosim de descriptori. Aceștia sunt vectori de diferite dimensiuni care trebuie să sintetizeze informația esențială observată în zona respectivă din imagine, în mod ideal descriptorii ar trebui să rămână invariabili la operațiile de redimensionare și rotație aplicate pe keypoint-uri. Algoritmului Oriented Fast and Rotated Brief (ORB)[12] este folosit pentru extragerea de keypoint-uri și descriptori. A fost creat în anul 2011 ca alternativă pentru alți algoritmi de extragere de feature-uri precum SIFT[13] și SURF[14]. Motivul pentru care acesta a ajuns atât de popular se datorează mai multor factori:

- Este mult mai rapid decat SIFT și SURF, fiind mult mai potrivit pentru sisteme în timp real și pentru dispozitive embedded[15].
- La momentul realizării lucrării științifice ORB-SLAM2, atât SIFT cât și SURF se aflau

sub protectia drepturilor de autor pe când ORB nu avea o astfel de restrictie.

- ORB este invariant din punct de vedere al rotației și are o toleranță bună la variația distanței.
- Folosește descriptori binari, care pot fi ușor de comparat folosind distanța Hamming[16],

Implementarea algoritmului ORB poate fi separată în 2 componente, calcularea keypoint-urilor și cea a descriptorilor. Pașii pe care îi urmează algoritmul sunt realizați într-o structură for loop. ORB extrage feature-uri la diferite dimensiuni ale imaginii, pentru a crea trăsături mai robuste la modificarea distanței. Numărul de execuții ale buclei for, este același cu numărul de resize-uri pe care trebuie să le aplice algoritmul. Etapele realizate la fiecare iterație sunt următoarele:

1. Calcularea keypoint-urilor folosind algoritmul FAST-9[17].
2. Selectarea celor mai potrivite keypoint-uri folosind Harris Corner Measure[18], Trăsăturile sunt sortate în ordine descrescătoare și sunt selectate primele N cele mai potrivite
3. Pentru fiecare keypoint se calculează orientarea acestuia folosind intensitatea centroidului.
4. Înainte de a calcula descriptorii, se aplică o operație de smoothing Gaussian pentru fiecare zonă selectată de un keypoint, aceasta având o dimensiune prestabilită de 31×31 de pixeli. Kernel-ul folosit este de 5×5.
5. Pentru fiecare keypoint se calculează un descriptor binar de tip BRIEF.
6. Fiecare descriptor va fi transformat folosind o matrice de rotație, unghiul fiind dat de orientarea keypoint-ului corespondent. În acest fel se obțin descriptorii de tip steer BRIEF[19].
7. Se obtine rBRIEF, rotated BRIEF, o variantă optimizată a steer BRIEF, prin alegerea bitilor despre care se știe ca au varianță mare și grad de corelație scăzut între ei.

În cazul algoritmul FAST-9, cifra 9 vine de la diametrul ferestrei circulare în care se face compararea între valoarea intensității pixelului și centru. Acest algoritm primește ca parametru imaginea și pragul pe care trebuie să îl depășească diferența de intensitate între pixeli pentru a fi considerat un keypoint. De cele mai multe ori, informația dată de keypoint-uri este redundantă, pentru a selecta un număr restrâns de trăsături, de preferat cele mai expresive, se folosește Harris Corner Measure. Pentru a calcula orientarea unui keypoint, vom defini

noțiunea de centroid C care este diferit de centrul zonei de 31×31 pixeli din imagine, O. Vectorul $\vec{OC}$ va fi cel care va da unghiul θ al keypoint-ului pe care îl vom obține direct din următoarea formulă, unde $I(x, y)$ reprezintă intensitatea luminoasă a pixelului cu coordonate (x, y).

$$m_{pq} = \sum_{x,y} x^p y^q I(x, y), \quad \theta = \text{atan2}(m_{01}, m_{10}) \tag{3}$$

În etapele 5 și 6 se realizează calcularea descriptorilor: aceștia vor avea formă binară și o lungime finală de 256 de biti. Compararea lor se va realiza folosind distanța Hamming. Cu cât 2 descriptori au o valoarea mai mică a acestei distanțe, cu atat mai similari sunt. Valorile fiecărui bit ai descriptorilor sunt asociate pe baza unui test binar în care este comparată intensitatea a 2 puncte din planul imaginii. Problema este că descriptorii BRIEF sunt sensibili la schimbările de rotație, din această cauză, prin rotirea coordonatelor pixelilor cu unghiul θ al orientării se obține steered BRIEF. Pentru a obține rBRIEF, au fost învățate în offline prin aplicarea unui algoritm de tip Greedy, care teste de verificare a intensității au cea mai mare variantă, și primele 256 dintre acestea au fost alese pentru a alcătui descriptorul.

4.3 Harta punctelor din spatiu

Unul dintre scopurile fundamentale ale algoritmului de ORB-SLAM2, pe lângă cel de estimare a traiectoriei camerei, este cel de creare a hărții mediului înconjurător. În comparație cu versiuni mai avansate ale acestui algoritm, special modificate pentru o reconstrucție cât mai fidelă a mediului, implementarea trebuie să funcționeze pentru un sistem embedded care are capacitate de procesare minimala. Din această cauză, harta creată de către algoritm trebuie să simuleze mediul printr-un nor de puncte cu o densitate redusă (sparse), pentru a putea fi ușor de interogat și optimizat. Cele 2 sarcini ale algoritmului de tip SLAM sunt interdependente, fiecare element din norul de puncte actionează ca o referință, o caracteristică a mediului care ar trebui să fie observată de fiecare dată când punctul se află în frustum-ul camerei. De exemplu: presupunem ca avem o imagine în care este observată în totalitate o masă în interiorul unei încăperi. ORB va identifica aproape instantaneu feature-urile (colțurile mesei) și indiferent de modul în care camera s-ar roti în jurul piesei de mobilier, aceleași feature-uri ar trebui să fie observate de fiecare dată. Considerând că mediul este static, feature-urile observate în fiecare cadru ar trebui să fie asociate cu aceleași puncte din spatiu, devenind astfel o referință pe baza

căreia putem estima modul în care s-a deplasat camera. În literatura de specialitate, aceste puncte din spațiu sunt denumite MapPoint-uri, iar funcționalitatea corectă a algoritmului depinde strict de modul în care aceste entitați sunt observate cadru cu cadru. Un astfel de punct este creat prin proiectarea în spațiu a unui keypoint. Considerăm coordonatele în imagine ale centrului zonei cheie ca fiind x și y, pentru cele 2 axe și distanța față de cameră, extrasă din harta de adâncime, notată cu d. Ne vom folosi de matricea transformării din coordonatele camerei în coordonatele globale și de parametrii interni ai camerei f_x, f_y distanța focală, și c_x, c_y coordonatele centrului imaginii. Vectorul coloană cu 3 dimensiuni reprezintă poziția în spațiu a MapPoint-ului creat. Ca alternativă, pentru a nu lucra cu matrici de dimensiuni 4×4 putem folosi R_{wc} reprezentând matricea de rotație și t_{wc} vectorul de translație.

$$\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = T_{wc} * \begin{bmatrix} \frac{x-c_x}{f_x} * d \\ \frac{y-c_y}{f_y} * d \\ d \\ 1 \end{bmatrix}, \quad \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = R_{wc} * \begin{bmatrix} \frac{x-c_x}{f_x} * d \\ \frac{y-c_y}{f_y} * d \\ d \end{bmatrix} + t_{wc} \tag{4}$$

În literatura de specialitate MapPoint-urile sunt considerate niște ancore (landmark), poziționate dinamic de către algoritm. Acestea sunt asociate cu un anumit cadru cheie și ne vor ajuta în optimizarea matricei de poziție dar și pentru sarcina de relocalizare și de memorare a zonelor cunoscute.

4.4 Asociere puncte din spațiu cu feature-uri ORB

Ca date de intrare avem feature-urile și descriptorii extrași din imagine, matricea de adâncime și harta de MapPoint-uri. Scopul acestei componente este să gasească cât mai multe asocieri de 1:1 între feature-uri și MapPoint-uri. Într-un caz ideal, fiecare feature găsit ar trebui să aibă asociat un MapPoint. În pratică, această situație nu poate avea loc din 2 motive: imperfecțiuni ale algoritmului de detecție ale feature-urilor și probleme cauzate de traiectorie. Algoritmul ORB nu garantează că pentru cadre consecutive, vor fi observate mereu aceleași feature-uri. Iar în cazul deplasării, dacă agentul își schimbă constant direcția sau apar frecvent operații de rotație, algoritmul nu poate observa punctele vechi din spațiu, fiind nevoit să creeze altele noi.

Un MapPoint este un feature al unui cadru anterior, proiectat în spațiu. În final, această componentă realizează tot o comparare de feature-uri între cadrul curent, și multiple cadre anterioare folosind descriptorii asociati si calculand distanta Hamming. Cu cât valoarea acestei distanțe este mai mică, cu atât cele 2 feature-uri sunt mai asemănătoare. Există mai multe tipuri de algoritmi folosiți pentru feature matching. În implementarea curentă singurul utilizat este Brute Force Feature Matching optimizat. Acest algoritm primește ca date de intrare 2 seturi de feature-uri și încearcă să găsească asocieri între ele. Asocierele sunt făcute cu ajutorul descriptorilor. Se calculează distanța Hamming iar dacă valoarea obtinută este minimă, perechea respectivă de feature-uri a fost corect asociată. Pentru ORB-SLAM2 o potrivire între 2 keypoint-uri arată că ele fac referire la exact acelasi punct din spatiu, observat din cadre diferite. Daca N este numărul de feature-uri din primul set, M numărul de feature-uri din al doilea set si D, dimensiunea descriptorului, în cazul nostru 32, complexitatea algoritmului devine $O(N * M * D)$. Destul de costisitor de folosit pentru un sistem in timp real, mai mult de atât este predispus la erori, compararea feature-urilor nu ține cont de locația acestora în imagine, obținându-se astfel asocieri care matematic par corecte, dar ele nu au sens din punct de vedere logic. Pentru a rezolva această problemă și a reduce complexitatea temporală, se stabilește o fereastră circulară de dimensiune prestabilită în jurul punctului de proiecție, unde se pot căuta feature-uri. O dată ce 2 keypoint-uri au fost considerate ca făcând referintă la acelasi punct din spatiu, cadrului curent ii este asociat un nou MapPoint.

4.5 Optimizarea Estimării Poziției Inițiale

Această componenta primește ca date de intrare estimarea poziției curente a camerei T_{cw} și o asociere bijectivă intre feature-urile găsite în imagine și punctele care există la momentul respectiv în spațiu. Ca date de ieșire, vom avea doar matricea T_{cw} optimizată. Dacă asocierile între feature-uri și MapPoint-uri sunt perfecte, ar trebui ca proiecția punctului din spatiu pe imagine să se suprapună pe centrul keypoint-ului pentru fiecare pereche în parte. Rareori se petrece acest lucru în practică, iar distanța dintre proiecția unui MapPoint și coordonotale centrului feature-ului reprezintă eroarea de asociere. Pentru a minimiza această eroare, există 2 optimizari care se pot face: prima este modificarea valorilor matricei de poziție, iar cea de-a doua este modificarea coordonatelor MapPoint-ului. Înainte de a prezenta algoritmul de optimizare folosit, voi arăta modul în care se proiectează un MapPoint în plan.

4.5.1 Proiectarea unui MapPoint în planul imaginii

Această operație de proiecție poate fi vazută ca aplicarea unui funcții $\pi(\cdot)$ ce primește ca date de intrare coordonatele globale ale punctului, iar ca rezultat va returna coordonatele omogene în planul imaginii. Această transformare se petrece in 2 etape:

1. conversia din sistemul de coordonate globale în sistemul de coordonate al camerei
2. conversia din sistemul de coordonate al camerei în sistemul de coordonate al imaginii

În prima etapă putem folosi coordonatele omogene, pentru a face conversia în mod direct. Alternativ, putem extrage din matricea de pozitie T_{cw} atât matricea de rotație R_{cw} cât și vectorul coloană de translatie t_{cw}.

$$\mathbf{X}_{camera} = \mathbf{T}_{cw} \cdot \begin{bmatrix} \mathbf{X}_w \\ 1 \end{bmatrix}, \quad \mathbf{T}_{cw} = \begin{bmatrix} \mathbf{R_{cw}} & \mathbf{t_{cw}} \\ \mathbf{0}^T & 1 \end{bmatrix}, \quad \mathbf{X}_{camera} = \mathbf{R_{cw}} \cdot \mathbf{X_w} + \mathbf{t_{cw}} \tag{5}$$

Matricea T_{cw} este utilizată atât pentru a descrie poziția și orientarea în spațiu cât și pentru a schimba din sistemul de coordonate global în cel al camerei. În sistemul de coordonate global, un punct se află la exact aceeași poziție indiferent de camera care îl privește. În sistemul de coordonate al camerei, poziția unui MapPoint o să difere pentru fiecare cadru cheie în parte. În etapa a doua MapPoint-ul este în sistemul de referință al camerei, coordonatele fiind reprezentate prin vectorul coloana X_{camera}. Vom considera a 3-a valoare a acestui vector Z_c. Aceasta reprezintă distanța dintre planul camerei și punctul pe care il analizăm. Z_c ne spune dacă un MapPoint poate fi observat în imagine. Dacă valoarea Z_c este mai mică sau egală cu 0, înseamnă că punctul se proiectează în spatele camerei, făcându-l invalid. Altfel, vom realiza conversia în coordonatele omogene ale imaginii cu ajutorul următoarei formule, u fiind asociat axei x și v fiind asociat axei y. Dacă valorile u și v au valori mai mari ca 0 și mai mici decât dimensiunea imaginii, vectorul coloană $[u, v, 1]$ este rezultatul căutat.

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \mathbf{K} \cdot \begin{bmatrix} \frac{X_c}{Z_c} \\ \frac{Y_c}{Z_c} \\ 1 \end{bmatrix}, \quad \mathbf{K} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{6}$$

4.5.2 Motion Only Bundle Adjustment

Algoritmul folosit în această etapă se numește Motion Only Bundle Adjustment[20]. Acesta modifică doar matricea poziției curente a camerei. Coordonatele punctelor din spațiu rămân nealterate. Algoritmul este unul iterativ, minimizând o functie de cost. Forma generală pentru funcția de eroare este suma erorilor de proiecție pentru toate perechile (feature, MapPoint).

$$\mathbf{R}_{cw}, \mathbf{t_{cw}} = \min_{\mathbf{R}_{cw}, t_{cw}} \sum_{i=1}^{N} \rho \left(\| \mathbf{x}_i - K \cdot (\mathbf{R}_{cw} \cdot \mathbf{X}_i + \mathbf{t_{cw}}) \|^2 \right) \tag{7}$$

În această formulă, x_i reprezintă poziția unui feature în sistemul de coordonatele al imaginii iar X_i reprezintă coordonatele globale ale MapPoint-ului pentru care calculăm eroarea de proiecție. Simbolul $\rho(\cdot)$ reprezintă funcția Huber[21] pentru scalarea valorilor de eroare. Dacă o asociere între un feature și un MapPoint este incorectă, diferența dintre centrul feature-ului și proiecția MapPoint-ului este mai mare decât un prag prestabilit. Această eroare, lăsată nemodificată, ar destabiliza algoritmul făcând ineficient procesul de optimizare. Parteea bună este că o astfel de problemă este ușor de observat. Dacă modificarea matricei de poziție duce la variații enorme a orientării sau a translației între 2 cadre consecutive, atunci cel mai probabil asocierile între feature-uri și MapPoint-uri aveau valori eronate. Termenul de $outlier$ este folosit pentru a descrie o pereche incorectă. Funcția de pierdere Huber reduce valoarea acestor outlier-ere permițându-le în același timp să facă parte din algoritmul de optimizare. În acest fel, Motion Only Bundle Adjustment devine mai robust și capabil ajungă la o valoare optimă în mai puține iterații. Mai jos este prezentată formula matematică a funcției Huber Loss, unde δ reprezintă un număr real pozitiv, toleranța erorii de proiectie.

$$\rho(s) = \begin{cases} \frac{1}{2}s^2 & \text{if } |s| \leq \delta \\ \delta(|s| - \frac{1}{2}\delta) & \text{if } |s| > \delta \end{cases} \tag{8}$$

In urma executiei algoritmului obtinem matricea de pozitie optimizata, mai mult de atat, stim care dintre perechile (feature, MapPoint) au avut statutul de outlier si le putem elimina pentru a nu influenta in mod negativ functionalitatea algoritmului.

4.6 Crearea unui cadru cheie

Aceasta componenta primeste ca date de intrare absolut toate informatiile procesate de pana acum pentru cadrul curent: imaginea de tip rgb, matricea de adancime, punctele cheie, descriptorii, asocierile (feature, MapPoint) si matricea estimarii pozitiei. Toate acestea impreuna vor alcatui un cadru cheie care va fi salvat in memorie. Salvarea pozitiilor cadrelor anterioare ne poate ajuta in 2 feluri. Putem folosi doar 2 cadre anterioare pentru a estima pozitia celui care urmeaza bazandu-ne pe legea inertiei. Consider ca o data inceputa deplasarea camerei intr-o anumita directie, este foarte probabil ca aceea miscare sa fie mentinuta si la urmatorul cadru. Fie T_{cw} matricea de pozitie pentru cadrul la care vrem sa estimam deplasarea, iar T_{cw1}, T_{cw2} matricile de pozitie a celor 2 cadre imediat predecesoare. Formula de estimare a pozitiei curente este:

$$\mathbf{T_{cw}} = \mathbf{T_{cw1}} \cdot \left(\mathbf{T_{cw2}}^{-1} \cdot \mathbf{T_{cw1}}\right) \tag{9}$$

Al doilea motiv pentru care avem nevoie de cadre cheie este recreearea mediului si a traseului parcus. Incercam sa salvam numarul minim de cadre necesare pentru a reproduce harta de MapPoint-uri a mediului inconjurator. Un cadru cheie nou (KeyFrame) aduce cu sine MapPoint-uri noi, extrase din feature-urile gasite in imaginea respectiva. Functionarea corecta a urmarii cadru cu cadru, este determinata de numarul de MapPoint-uri gasite in imaginea curenta in comparatie cu un cadru de referinta. In momentul in care numarul de puncte cheie gasite in imaginea curenta scade sub un anumit prag, stim ca este necesar un nou cadru cheie care: sa stabilizeze urmarirea, sa introduca noi MapPoint-uri, si sa ajute la optimizarea intregii harti a mediului.

4.6.1 Optimizare harta locala

Harta locala este alcatuita din KeyFrame-uri si MapPoint-uri. Pentru a optimiza harta trebuie sa adaugam noi puncte de tip MapPoint si noi KeyFrame-uri in ea. Pentru a valida conexiunile care deja exista. Modul in care sunt create si sterse punctele urmeaza o abordare numita survival of the fittest. La fiecare nou KeyFrame adaugat, sunt create in aproximativ 100 de noi MapPoint-uri si inserate in harta. Acestea vor fi supuse unui test care sa evalueze cat de

usor sunt observate feature-urile pe care le reprezinta. Din punct de vedere matematic, un MapPoint este observat de un KeyFrame daca proiectia acestuia in imagine este un vector valid in sistemul de coordonate al KeyFrame-ului respectiv. Cu cat mai multe Keyframe-uri observa acelasi MapPoint, cu atat mai stabil este punctul respectiv din spatiu. Intr-un caz ideal, ar trebui ca orice MapPoint creat sa fie stabil. De cele mai multe ori nu se intampla acest lucru din cauza erorilor de feature matching. Astfel, doar cele mai evidente feature-uri raman salvate in harta pana la finalul algoritmului. Scopul acestei componente este adaugarea KeyFrame-urilor noi, eliminarea celor redundante si testarea stabilitatii tuturor MapPoint-urilor create. Pentru a salva Keyframe-ul curent in harta urmatoarele operatii trebuie urmate:

1. Folosind harta de adancime, sunt selectate cele mai apropiate N feature-uri care nu au un MapPoint asociat si au valoarea adancimii mai mare ca 0. Coordonatele acestor feature-uri sunt proiectate in spatiu pentru a obtine noi MapPoint-uri.
2. KeyFrame-ul curent este comparat cu alte cadre cheie, pentru a vedea cu cine imparte cele mai multe puncte comune. Keyframe-urile sunt stocate in harta intr-o structura de tip graf neorientat unde nodurile sunt cadrele cheie iar arcele sunt numarul de MapPoint-uri comune dintre ele.
3. sunt eliminate punctele cheie redundante sau care au fost observate in prea putine cadre pentru a fi luate in considerare.
4. se executa un algoritm numit Local Bundle Adjustment in care KeyFrame-urile care au cele mai multe puncte comune cu cadrul curent analizat vor avea matricea de pozitie optimizata si coordonatele globale ale MapPoint-urilor asociate acestora.

Local Bundle Adjustment este similar cu Motion Only Bundle Adjustment. In continuare vorbim de un algoritm iterativ care incearca sa minimizeze o functie de cost, folosind metoda scaderii gradientului. Diferenta este ca optimizarea se aplica pe mai mult de un cadru cheie: atat matricea poztiei si punctele din spatiu asociate (MapPoint-urile) vor fi optimizate. Avem urmatoarele etape:

1. Se creeaza lista de cadre mobile. Plecand de la cadrul curent, se vor selecta toti vecinii de gradul 1 si 2 din graful neorientat stocat in harta. Aceste Keyframe-uri sunt considerate $mobile$ deoarece matricea lor de pozitie se va modifica.
2. Se creeaza lista de MapPoint-uri ale caror coordonate vor fi optimizate. Fiecare Keyframe din multimea cadrelor mobile observa un numar de puncte in spatiu, toate aceste

puncte vor fi folosite de catre algoritmul de optimizare.

3. Se creeaza lista de cadre fixe pentru care matricea de pozitie nu se va modifica. Pentru fiecare punct din lista de MapPoint-uri ce vor fi optimizate se va itera prin lista de KeyFrame-uri care observa acel MapPoint. Daca un KeyFrame apartine multimii de cadre mobile va fi ignorat, iar daca nu, va fi adaugat in lista de cadre fixe. Acestea sunt incluse in algoritm pentru a garanta ca modificarea coordonatelor unui MapPoint nu va strica asociera (feature, MapPoint) in cadrele care nu vor avea matricea de pozitie modificata.

Lucrarea stiintifica care sta la baza ORB-SLAM2, implementeaza deja functia de cost pe care algoritmul de Local Bundle Adjustment o foloseste. Pentru a intelege mai usor formula matematica, aceasta trebuie privita de la dreapta la stanga. E_{kj} reprezinta eroarea de proiectie a unui MapPoint pe feature-ul asociat. Indicele k apartine KeyFrame-ului, j reprezinta ordinul perechii (feature, MapPoint) pentru care calculam eroarea. Simbolul $\rho(\cdot)$ este asociat functiei Huber, folosita pentru a ameliora efectele perechilor de tip outlier. X_k este o notatie pentru multimea tuturor asocierilor (feature, MapPoint) pentru un Keyframe k. Suma erorilor tuturor perechilor este calculata pentru toate cadrele fixe si mobile. Parametrii care vor fi optimizati sunt: coordonatele MapPoint-urilor selectate de catre algoritm X_i cat si matricile de pozitie pentru cadrele mobile K_l. Algoritmul optimizeaza valorile pana cand ajunge la o valoare de minim sau pentru un numar de iteratii.

$$\{\mathbf{X}_i, \mathbf{R}_l, \mathbf{t}_l \mid i \in \mathcal{P}_L, l \in \mathcal{K}_L\} = \arg\min_{\mathbf{X}_i, \mathbf{R}_l, \mathbf{t}_l} \sum_{k \in \mathcal{K}_L \cup \mathcal{K}_F} \sum_{j \in \mathcal{X}_k} \rho(E_{kj}) \tag{10}$$

$$E_{kj} = \|\mathbf{x}_j - K \cdot (\mathbf{R}_k \mathbf{X}_j + \mathbf{t}_k)\| \tag{11}$$

4.6.2 Reteaua Neurala FastDepth

Retele Neurale Artificiale sunt o tehnica des intalnita in Machine Learning pentru a rezolva sarcini complexe pentru care nu exista solutii algoritmice clar definite sau implementarea acestora este mult prea costisitoare. Conform [22] si [23] retele neurale definesc o functie nonliniara care gaseste o corespondenta intre un set multivariat de date de intrare x si un set multivariat de date de iesire y, modificand un set de parametrii ϕ, $f(x, \phi) = y$. Aceasta este alcatuita dintr-un numar enorm de elemente de procesare care contin parametrii functiei

ϕ, conectate intre ele intr-o structura de tip graf si dispuse pe straturi. Cele mai importante fiind: stratul de intrare si de iesire, unde se stabileste forma generala pe care trebuie o sa respecte datele care vor parcurge reteaua si modul in care va arata rezultatul obtinut. Celelalte nivele sunt denumite straturi ascunse. Aceastea fac prelucrarea informatiei primite de la straturile anterioare si o transmit mai departe. Spunem ca o retea neurala invata din datele primite, daca isi modifica parametrii ϕ astfel incat sa reprezinte cu mai multa acuratete corespondenta intre datele de intrare x si cele de iesire y. FastDepth[17] este o arhitectura de retea neurala folosita pentru estima adancimii in imagini. Aceasta primeste o imagine de tip RGB al interiorului unei incaperi si returneaza o matrice cu valori in intervalul $0m - 10m$ estimand pentru fiecare pixel in parte distanta de la planul de proiectie al imaginii pana la punctul din spatiu surprins de fotografie. Scopul nostru este antrenarea unei retele neurale care sa produca o matrice de adancime cu valori cat mai apropiate de distanta reala la care se afla obiectele fata de camera. ORB-SLAM2 foloseste o camera tip RGBD / Stereo care descrie cu foarte mare acuratete distanta pana intr-un anumit punct din spatiu, dar creeaza o matrice rara de valori, suprafetele lucioase sau cele car nu au putut fi clar observate vor avea adancimea 0 pentru a arata ca distanta nu a putut fi corect estimata in pixelii respectivi. Un motiv pentru care retelele neurale sunt o alternativa buna este ca ele vor avea o estimare pentru fiecare pixel din imagine. Reteaua FastDepth pare sa realizeze o pseudosegmentare a zonelor din imagine, identifica conturul obiectelor si atribuie valori ale distantei asemanatoare pentru pixelii ce apartin aceleasi entitati. Arhitecturile de mari dimensiuni nu pot fi folosite in timp real fara a utiliza un GPU, dar nu este cazul si pentru arhitectura FastDepth care poate procesa aproximativ 100 de cadre pe secunda. De asemenea consuma o cantitate redusa de memorie, parametrii retelei pot fi stocati intr-un fisier ONNX ce ocupa mai putin de 8 MB, fiind usor de integrat intr-un dispozitiv embedded.

Un posibil dezavantaj al acestei arhitecturi este limitarea de 10m, fiind nepotrivit de folosit afara, dar ideal pentru un spatiu inchis de mici dimensiuni. Un alt dezavantaj este ca valorile aproximate vor avea o acuratete mai slaba decat cele obtinute de camerele Stereo/RGBD.

Exista mai multe filozii cand vine vorba de modul in care ar trebui sa arate arhitectura retelelor neurale si operatiile pe care ar trebui sa le realizeze fiecare strat. Feed forward neural network a fost printre primele arhitecturi definite. Elementele de procesare sunt dispuse pe straturi, si fiecare strat primeste input-ul de la stratul precedent si transmitea output-ul la stratul imediat urmator. Informatia circula liniar, de la intrarea in retea pana la finalul acesteia. Abordarea

s-a dovedit eficienta in situatiile in care era nevoie de retele neurale de mici dimensiuni, cu un numar redus de straturi si parametrii. In momentul in care crestea complexitatea, abordarea de feed forward neural network devenea greu de antrenat si dadea rezultate mai slabe[22]. Pentru a rezolva aceasta problema au aparut arhitecturile de tip residual network. Acestea au aplicatii in procesarea imaginilor[24], unde datele de intrare au dimensiuni mari si este nevoie de multe nivele pentru a extrage suficiente informatii. Principiul de functionare este utilizarea unor straturi reziduale denumite si skip connections, in care rezultatul unui strat este salvat si transmis ca data de intrare la un alt nivel decat la cel imediat urmator. Abordarea aceasta pastreaza din informatiile initiale ale datelor de intrare in straturile viitoare stabilizand antrenarea. O alta arhitectura des intalnita este cea de encoder-decoder folosita in numeroase aplicatii practice in ceea ce priveste imaginile: sarcini de colorare a imaginilor gri[25], reconstructie a imaginilor care contin parti lipsa si de generare de imagini: un exemplu fiind Variational Auto Encoder[26].

Pe langa tipurile de arhitecturi propuse, exista mai multe categorii de straturi in retele neurale. Primele folosite erau cele fully connected unde fiecare element de procesare era conectat cu toate celelalte elemente de procesare din stratul urmator. Matematic, operatia poate fi vazuta ca o inmultire de matrici, o operatie costisitoare, iar utilizarea exclusiva a straturilor complet conectate creea o retea neurala incapabila sa reprezinte functii nonliniare, scazand capacitatea de generalizare. De cele mai multe ori straturile liniare sunt folosite impreuna cu functii de activare nonliniare precum ReLU sau Sigmoid dar in continuare ramane problema numarului mare de parametrii care trebuie antrenati. Din aceasta cauza au fost create straturile convolutionale care folosesc mai putini parametrii si au aplicabilitate in procesarea imaginilor. Principiul teoretic pe care se bazeaza este ca pixelii alaturati in imagine au aceeasi semnificatie, reprezentand acelasi feature. Operatia de convolutie trebuie realizata pe o zona a imaginii iar modificarea parametrilor afecteaza output-ul generat de mai multi pixeli. Se stabileste un kernel, o matrice de mici dimensiuni, in FastDepth folosindu-se kernel-uri de (3, 3), acestea vor stoca parametrii w_{mn} pe care reteaua neurala ii va antrena pentru stratul convolutional. In formula h_{ij} reprezinta intensitatea pixelului dupa calculul operatiei de convolutie, iar x_{ij} este valoarea intensitatii pixelului de pe coloana i si linia j. Litera a reprezinta functia de activare folosita iar β este o valoare numerica denumita bias. Acesta poate fi modificat in

timpul antrenarii si creste capacitatea de generalizare a functiei de convolutie[23].

$$h_{ij} = a\left[\beta + \sum_{m=1}^{3}\sum_{n=1}^{3}\omega_{mn}x_{i+m-2,j+n-2}\right] \tag{12}$$

Stratul de convolutie este in continuare prea costisitor pentru a crea o arhitectura in timp real de mari dimensiuni. Presupunem ca avem un vector de intrare pentru un strat de convolutie cu dimensiunile $[d_{in}, h, w]$ unde d_{in} este numarul de canale, h inaltimea si w latimea vectorului. Kernelul folosit are dimensiunile $[k, k, d_{in}, d_{out}]$, unde d_{out} este numarul de canale rezultate in urma convolutiei. In total se vor executa $h \cdot w \cdot d_{in} \cdot d_{out} \cdot k \cdot k$ operatii. Pentru a rezolva aceasta problema a fost creat un strat numit Depthwise Separable Convolutions[27], obtinut prin compunerea a 2 straturi de convolutie, unul numit depthwise convolution, iar celalalt pointwise convolution. Aceasta abordare creste viteza de procesare si imbunatateste acuratetea in sarcini de clasificare pentru seturi de date precum ImageNet ILSVRC2012. In cazul depthwise convolution, fiecare canal al datelor de intrare este procesat de un singur kernel al stratului de convolutie. Pointwise convolution uneste printr-o combinatie liniara rezultatul procesarii fiecarui canal. Complexitatea temporala obtinuta astfel este de: $h \cdot w \cdot d_{in} \cdot (k^2 + d_{out})$[17].

FastDepth foloseste tehnica de skip connections, straturile finale primind ca date de intrare valorile calculate de straturile aflate la inceput, si urmeaza o arhitectura encoder-decoder. Encoder-ul transforma datele de intrare intr-o forma mai compacta asemeni unei operatii de arhivare. Aceasta este realizata folosind o alta retea neurala numita Mobile_Net[28] si ulterior Mobile_Netv2[29] care reduce numarul de parametrii si creste viteza de procesare fara a impacta acuratetea. Partea de decoder este alcatuit din 5 straturi de tip depthwise convolution fiecare urmate de o interpolare liniara care dubleaza dimensiunea rezultatului, ultimul strat fiind un pointwise convolution care uneste canalele obtinute si returneaza matricea de adancime. Pentru Mobile_Netv2 exista parametrii preantrenati in libraria Pytorch[30] pe setul de date ImageNet fiind un motiv in plus de a folosi aceasta arhitectura in dezvoltarea FastDepth. Mobile_Netv2 foloseste atat depthwise convolution cat si pointwise convolution intr-un strat numit Inverted Residual, acesta fiind alcatuit din urmatoarele componente unde t este factorul de multiplicare al numarului de canale, s este parametrul de stride, determina daca se micsoreaza numarul de canale, h, w, d sunt dimensiunile matricei de intrare:

Input	Operator	Output
$h \times w \times k$	1x1 conv2d , ReLU6	$h \times w \times (tk)$
$h \times w \times tk$	3x3 dwise s=s, ReLU6	$\frac{h}{s} \times \frac{w}{s} \times (tk)$
$\frac{h}{s} \times \frac{w}{s} \times tk$	linear 1x1 conv2d	$\frac{h}{s} \times \frac{w}{s} \times k'$

Figura 3: Straturile unui bloc de tip Inverted Residual, preluat din[29]

5 DETALII DE IMPLEMENTARE

5.1 Limbaje de programare si librarii folosite

Implementarea este realizata in C++17. Pentru management-ul librariilor si al codului folosesc CMake 3.28.3. Acesta imi permite sa grupez in foldere codul scris de mine si face operatia de linking automat cu binarele pachetelor folosite. Librariile principale sunt OpenCV[31] 4.9.0, Ceres[32] 2.2.0, Eigen 3.4.0, DBoW2[33] si ultima versiune de Sophus pana la data de ianuarie 2025. In comparatie cu alte librarii care inca mai trec prin diverse update-uri, Sophus a intrat intr-o etapa de mentenanta, dezvoltarea efectiva a acestuia fiind finalizata din iunie 2024. O prima problema pe care am intalnit-o a fost gasirea unei versiuni compatibile de C++ cu toate aceste pachete. Am incercat mai multe variante printre care C++11, C++14, C++17 si C++20. Preferinta mea ar fi fost sa folosesc o versiune cat mai noua cu putinta, dar care sa poata fi compatibila cu toate librariile mentionate. C++11 si C++14 nu erau compatibile cu Ceres, versiunea minima pentru aceasta librarie era C++17. C++20 si C++23 nu era compatibil cu Sophus si cu Eigen, iar ambele librarii sunt fundamentale deoarece implementeaza metode puternic optimizate de a lucra cu matrici iar API-ul lor era mai simplu decat cel din OpenCV. Singura optiune ramasa a fost C++17 care era incompatibila cu DBoW2. Libraria folosea o versiune mai veche a functiei throw pentru erori. In momentul in care am eliminat aceasta directiva, am putut recompila codul ca librarie. Bibliotecile utilizate sunt urmatoarele: OpenCV este o librarie de computer vision. Contine implementari ale algoritmilor de extragere de trasaturi precum FAST, ORB, SIFT, SURF, API-uri pentru procesare video: citirea unui video cadru cu cadru, procesarea de imagini: aplicarea de filtre, transformarea in grayscale, eliminarea distorsiunii cauzata de camera. Foarte importante sunt structurile ce abstractizeaza matricile si parametrii prin care se indentifica trasaturile: cv::Mat si cv::KeyPoint. Structura KeyPoint este fundamentala pentru implementarea algoritmului deoarece stocheaza numeroase informatii despre zona pe care o reprezinta: orientarea acesteia, coordonatele centrului si nivelul la care a fost observat, parametrii de care am avut nevoie in fiecare etapa de procesare a cadrelor. Pe langa aceste lucruri, OpenCV are un modul dedicat pentru citirea parametrilor

retelelor neurale din fisierele care urmeaza un format de tip ONNX, fiind o alternativa potrivita daca vreau sa utilizez un model doar pentru sarcini de inferenta.

Ca librarie de optimizare am avut de ales intre Ceres si g2o[34]. In implementarea oficiala g2o era cel folosit. Motivul principal fiind ca permite abstractizarea parametrilor care trebuie optimizati si a relatiilor dintre acestia sub forma unui graf neorientat. API-ul de g2o permite activarea si dezactivarea anumitor noduri, pentru a face implementarea mai robusta impotriva perechilor de tip outlier, si pentru a putea reintroduce noduri eliminate temporar in graful de optimizare. Ceres din pacate nu permite acest lucru. O data create conditiile initiale acestea pot fi dezactivate si nu mai este permisa reutilizarea lor in aceeasi problema de optimizare. La finalizarea algoritmului memoria folosita de catre noduri este eliberata. Cu toate acestea, Ceres are un API usor de utilizat si are o viteza comparativa cu cel din g2o.

Folosesc libraria Eigen deoarece este mai simplu API-ul de calcul cu matrici decat cel din OpenCV. Pentru a accesa elementele unei matrici in OpenCV se foloseste o referinta la vectorul de date facand accesarea elementelor mult mai nesigura iar verificarea indicelui este facuta la runtime. In cazul matricilor din Eigen, accesarea elementelor si operatiile cu matrici sunt verificate la compile time, prevenind astfel erorile inainte de a rula programul.

Sophus este o librarie care imi permite sa lucrez cu algebra de tip Lie. In loc de a vedea estimarile pozitiei ca pe niste matrici de 4×4, le pot vedea ca pe un vector alcatuit din 7 elemente. Primii 4 parametrii alcatuiesc un quaternion, aceasta fiind o exprimare vectoriala a unei matrici 3×3 de rotatie, iar ultimii 3 parametrii reprezinta un vector de translatie. Biblioteca implementeaza operatii care imi permit sa lucrez cu acesti vectori, care fac parte dintr-un grup numit $se(3)$ si garanteaza ca rezultatul obtinut este scalat corespunzator pentru a face parte in continuare din aceeasi categorie.

DBoW2 este o metoda de tip bag of words pentru compararea imaginilor intre ele. Este utilizat pentru operatii precum feature matching intre imagini consecutive, relocalizari si recunoastera zonelor prin care a trecut pentru a inchide buclele create de mai multe cadre cheie salvate in harta. Acesta este alcatuit dintr-o structura de tip arbore. Fiecare nivel este obtinut din realizarea unui algoritm de clusterizare a descriptorilor de tip ORB ca de exemplu kmeans++, separarea tuturor descriptorilor in functie de centroizi si reluarea aceleasi operatii in fiecare dintre clusterele nou create. Nodurile de tip frunza sunt alcatuite dintr-un singur descriptor. Construirea arborelui se realizeaza intr-o etapa offline. In cazul librariei DBoW2, setul de date folosit a fost Bovisa 2008-09-01. Au fost alese 10K imagini iar pentru fiecare cadru in parte

extrasi 1000 descriptori ORB. Acestia au fost folositi pentru a crea un arbore de adancimea 6 iar numarul de clustere pe care le creeaza fiecare iteratie a algoritmului kmeans++ este de 10. Pe ultimul strat exista un milion de frunze, si tot aceeasi lungime o va avea si vectorul de feature-uri care va reprezenta o imagine. Fiecare descriptor va primi o valoare numerica numita greutate, invers proportionala cu frecventa pe care o are acesta. Cu cat este mai rar un anumit descriptor, cu atat este mai util pentru a diferenta o imagine de multe altele. Scopul principal al librariei este sa primeasca ca data de intrare descriptorii ORB ai unei imagini si sa calculeze vectorul sau bag-of-words. Vectorul bag-of-words este alcatuit in principal din valori de 0. Fiecare descriptor al imaginii parcurge arborele de la radacina spre frunze, parcurgerea realizandu-se prin calcularea distantei Hamming dintre descriptor si toate nodurile de pe un anumit nivel, si alegerea nodului cu distanta Hamming minima. Nodul frunza la care va ajunge va avea asociat un index, in cazul de fata cu valori de la 0 la un milion. La acelasi index va fi modificata valoarea din vectorul bag-of-words in valoarea greutatii descriptorului stocat in arbore. Apelul de biblioteca returneaza de asemenea un vector de feature-uri in care fiecare element este o pereche de forma $(int, vector\_descriptori)$, primul element este indexul clusterului de la nivelul 4 al arborelui DBOW2, iar cel de-al doilea element reprezinta un vector de descriptori din imaginea curenta care se potrivesc in acelasi cluster. Doua imagini pot fi comparate intre ele prin intermediul acestui vector de feature-uri, lucru care va fi detaliat in descriera clasei OrbMatcher. In implementarea ORB-SLAM2, nu este practica creearea unui vector de tip bag of words cu un milion de elemente, mai ales ca majoritatea valorilor sunt 0, asa ca o reducere a dimensionalitatii vectorului ar creste viteza de calcul a sistemului. Din aceasta cauza, compararea descriptorilor se realizeaza doar pana la nivelul 4 in arbore, vectorul bow avand doar 1000 de elemente, iar cel de feature-uri avand acelasi numar de elemente cu numarul de descriptori.

5.2 Mediu de lucru si principalele clase

Structura de fisiere este una simpla, in folderul radacina se regaseste fisierul de CmakeLists.txt care va fi interpretat de utilitarul cmake pentru a genera automat Makefile-ul. Acest Makefile va contine regulile de build si de clean pentru proiect. In fisierul main.cpp vor fi initializate componentele si se va putea selecta pe care dintre cele 2 seturi de date se va aplica algoritmul. Aceste seturi de date contin de fapt cadrele dintr-un video facut cu o camera RGBD Micro-

soft Kinetic impreuna cu matricile de adancime si pozitiile acestora in spatiu pentru fiecare cadru in parte. Aceste seturi de date sunt suficient de complexe pentru a permite evaluarea functionarii algoritmului de ORB-SLAM2. Tot in main.cpp, se va realiza citirea fisierului ORBvoc.txt, acesta contine datele pe care le va folosi clasa ORBVocabulary pentru a initializa arborele folosit de libraria DBOW2.
Tot in folderul radacina se regaseste si fisierul fast_depth.onnx, in care este stocata arhitectura si parametrii retelei neurale FastDepth pentru estimarea adancimii. In folderul de include se afla antetele claselor pe care le voi implementa si in folderul de src se regaseste codul de C++ ce implementeaza logica programului. Am observat ca separarea codului in acest fel este o practica des intalnita in proiectele de mari dimensiuni si garanteaza flexibilitate in includirea dependintelor intre fisiere. Algoritmul ORB-SLAM2 este unul complex, depinzand de o multitudine de parametrii care pot influenta acuratetea. Cei mai importanti sunt cei corelati cu camera. In fisierul config.yaml se regaseste matricea K, parametrii de distorsiune ai imaginii si alte constante pe care le-am considerat ca fiind niste hiperparametrii ai algoritmului. Acestia vor trebui modificati in functie de mediul in care va rula ORB-SLAM2 pentru a garanta functionarea corecta.

Clasa TumDatasetReader este responsabila de achizitia de date si de scrierea in fisier a traiectoriei pe care o estimeaza algoritmul cadru cu cadru. Achizitia de date presupune citirea din memorie a matricei RGB, convertirea acesteia in grayscale pentru o procesare mai rapida de catre algoritmul ORB si de obtinerea hartii de adancime pentru cadrul respectiv. Acest lucru poate fi realizat in 2 feluri: matricea de distante este citita din setul de date TUM RGBD si a fost inregistrata cu o camera RGBD tip Microsoft Kinetic, sau se foloseste reteaua neurala FastDepth care estimeaza in timp real distanta pentru fiecare pixel din cadrul curent. Imaginea RGB si harta de adancime vor fi transmise ca parametrii clasei Tracker. TumDatasetReader stocheaza estimarile pozitiilor camerei pentru fiecare cadru in parte. Cadrele cheie, cele salvate in clasa Map, vor avea matricea de pozitie stocata nealterat in memorie, ele sunt deja relative fata de primul cadru citit. Pentru celelalte cadre, matricea de pozitie salvata in clasa TumDatasetReader este relativa la un cadru cheie, de preferat ultimul cadru cheie creat pana la citirea imaginii curente. Motivul pentru care se realizeaza salvarea pozitiilor in acest fel, este ca doar cadrele din clasa Map sunt salvate in memorie si pot fi optimizate de catre algoritmul Bundle Adjustment asa ca doar acestea ar trebui sa aiba valoarea lor salvata explicit.

Clasa MapPoint este fundamentala pentru buna functionare a algoritmului ORB-SLAM2. Aceasta este formata cu ajutorul unui KeyPoint si al unui KeyFrame asociat acestuia. In etapa anterioara, am prezentat modul in care se face proiectia coordonatelor unui punct cheie in spatiu, acestea devenind coordonatele globale ale MapPoint-ului pe care il creem. Punctului din spatiu i se asociaza de asemenea descriptorul acelui keypoint care l-a creat, pentru compararea ulterioara cu alte KeyPoint-uri din alte imagini. Un MapPoint are nevoie de un vector de orientare, acesta ajuta in verificarea proprietatii unui MapPoint de a fi sau nu vizibil dintr-un KeyFrame. Pentru a calcula acest vector de orientare prima data se determina coordonatele globale ale centrului camerei pentru cadru cheie care a creat acel KeyPoint, acest lucru se realizeaza in felul urmator:

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \end{bmatrix} = -R_{wc}^t * t_{wc}, \quad T_{wc} = \begin{bmatrix} R_{wc} & t_{wc} \\ 0 & 1 \end{bmatrix} \tag{13}$$

Normalizarea diferentei intre coordonatele globale ale centrului camerei si ale MapPoint-ului creeaza vectorul de orientare. Acesta poate fi modificat, daca se constata ca mai multe cadre observa acelasi punct. In situatia respectiva, vectorul de orientare final va fi media aritmetica a celorlalti vectori de orientare individuali.

Clasa Feature, aceasta componenta nu exista in implementarea oficiala a ORB-SLAM, dar am considerat ca utilizarea acesteia ar simplifica codul. Extinde clasa KeyPoint fara a o mosteni explicit, are asociata distanta, extrasa din matricea de adancime, descriptorul si o valoare de tip boolean care arata daca punctul este monocular sau stereo. Aceasta clasificare se obtine prin compararea adancimii cu o valoare foarte apropiata de 0. In cazul implementarii mele, daca distanta este mai mica decat $1e-2$, consider ca valoarea estimata de camera RGBD ori de reteaua neurala nu este corecta si ca punctul respectiv este monocular, altfel consider ca este stereo. Pentru fiecare KeyPoint extras, se va crea o instanta a clasei Feature care va fi stocata direct in KeyFrame. Fiecare Feature are setat pe null la initilializare o referinta la un obiect de tip MapPoint. Pentru a garanta functionarea in real time a algoritmilor, asocierea (Feature, MapPoint) trebuie sa poata fi accesata in $O(1)$. Vectorul de elemente Feature, impreuna cu o structura de tip dictionar unde cheia va fi MapPoint si valoarea de tip Feature, vor face acest lucru posibil. Singura problema este ca cele 2 structuri incearca sa reprezinte

aceleasi corelatii, in cazul vectorului am indicele unui Feature drept cheie si incerc sa accesez MapPoint-ul asociat, iar in cazul dictionarului, am referinta unui MapPoint si incerc sa obtin adresa unui Feature. Ambele structuri trebuie sa contina aceleasi perechi, altfel comportamentul algoritmului devine nedefinit.

Clasa KeyFrame, contine mai multe elemente legate de cadrul curent. Pentru a mentine functionarea sistemului in timp real, trebuie sa stocam in memorie rezultatele calculelor noastre. In aceasta clasa se vor regasi matricea de adancime, vectorul de instante ale clasei Feature, vectorul de trasaturi calculat de metoda bag-of-words implementata in DBOW2 si cadrul initial, convertit in format grayscale ce va fi folosit ulterior pentru afisarea in timp real a performantelor algoritmului. In interiorul constructorului acestei clase sunt mai multe operatii realizate, majoritatea necesare pentru a creste eficienta accesarii datelor. De exemplu: vectorul de tip Feature in medie contine 1000 de elemente care nu sunt sortate. In situatia in care proiectam un MapPoint in plan ar trebui sa comparam coordonatele proiectiei cu pozitia fiecarui Feature in parte pentru a stabili care este cel mai apropiat. O modalitate de a rezolva acest lucru este segmentarea suprafetei in K zone, in cazul meu am ales $K = 100$, fiecare reprezentand o portiune din imaginea initiala, avand asociate referintele Feature-urilor care se gasesc pe suprafata respectiva. In acest fel, in functie de zona in care este proiectat un MapPoint, vom stii ce Feature are o posibilitate mare de a corespunde, reducand astfel numarul de comparatii. Considerand ca toate valorile de tip Feature sunt dispuse in mod egal pe suprafata imagini atunci complexitatea devine, $O(N/K)$ unde N reprezinta numarul de Feature-uri iar K reprezinta numarul de zone in care a fost impartita imaginea. Constructorul este responsabil de initializarea structurilor de tip Feature, partitionarea lor in functie de coordonatele in imagine, si de memorarea estimarii curente a pozitie camerei si a centrului camerei in coordonate globale. Tot in aceasta clasa se regaseste structura de tip dictionar (MapPoint, Feature), care va fi adaptata pe tot parcursul algoritmului. Alta metoda importanta este: $get\_vector\_keypoints\_after\_reprojection$. Aceasta primeste ca date de intrare coordonatele proiectiei unui MapPoint, valoarea ferestrei de proiectie, si octava minima si maxima. Octavele reprezinta nivelul la care a fost observat un Keypoint in imagine si o estimare grosiera a distantei dintre camera si punctul din spatiu observat. Acesta poate sa aiba valori inte 0 si 7 inclusiv si ne spune de cate ori s-a facut resize la imagine pentru a surprinde o anumita trasatura. De exemplu: daca un Keypoint are valoarea octavei 0, inseamna ca algoritmul a

detectat-o in imaginea nemodificata. Daca ar fi 1, atunci dimensiunea imagini a fost redusa o singura data cu 0.8 din valoarea initiala si asa mai departe. Feature-ul care are asociat un MapPoint trebuie sa aiba valori ale octavei apropiate intre ele. Daca aceasta situatie nu s-ar respecta, ar introduce erori de estimare a distantei, ne asteptam ca Feature-uri corespondente, sa fie aproximate la aceeasi distanta. Altfel ar putea inseamna ca cele 2 puncte din spatiu sunt diferite. Daca mediul are o structura simetrica, de exemplu: o sala de clasa cu bancile aliniate una in fata celeilalte, algoritmul ar putea observa 2 colturi ce apartin de 2 mese diferite, daca nu ar avea aceasta separare pe baza octavei, urmatorul cadru care observa aceleasi mese ar putea sa asocieze eronat punctele intre ele, afectand estimarea pozitiei. Fereastra de proiectie reprezinta cat de departe poate sa fie Feature-ul de coordonatele punctului de proiectie ale unui MapPoint pentru a fi considerata corecta o asociere intre cele 2 elemente. In functie de dimensiunea ferestrei aceasta poate intersecta 1, 2 sau 4 subsectiuni din cele 100 in care este impartita imaginea. Problema cea mai mare pe care am avut-o cu clasele KeyFrame si MapPoint era dependenta circulara. MapPoint-urile aveau nevoie de un KeyFrame si un Feature pentru a fi create si trebuia sa mentina o lista a KeyFrame-urilor care observa MapPoint-ul respectiv. In cazul KeyFrame-ului, acesta trebuie sa pastreze referinte asupra tuturor MapPoint-urilor pe care le observa. Pentru a rezolva aceasta problema am folosit o clasa aditionala care face operatii cu cele 2 structuri si am folosit forward declaration.

Clasa Map implementeaza harta pe care o foloseste algoritmul ORB-SLAM2. Aceasta este responsabila de stocarea corecta a KeyFrame-urilor, a MapPoint-urilor si rezolva problema dependentei circulare a celor 2 clase. Aici am implementate metodele de adaugare/stergere a unui MapPoint dintr-un KeyFrame. De asemenea, clasa MapPoint contine referinte la toate KeyFrame-urile care o observa. Aceste referinte sunt adaugate / sterse de catre 2 metode care se regasesc aici. Clasa Map creaza o structura de tip graf ponderat neorientat, in care nodurile sunt reprezentate de KeyFrame-uri. Arcele arata daca exista mai mult de 15 puncte comune intre 2 KeyFrame-uri iar ponderea lor este determinata de numarul de MapPoint-uri comune. Clasa Map realizeaza operatii pe graful de KeyFrame-uri, adauga/sterge noduri si face interogari pentru a afla vecinii directi sau cei pe nivel 2. Am ales sa implementez aceasta structura folosind $std :: unordered\_map$. Drept cheie va avea KeyFrame-ul curent iar valoarea returnata de structura de tip dictionar va fi un alt $std :: unordered\_map$, ce va contine toate celelalte KeyFrame-uri cu care este direct conectata dar si ponderea conexiunii.

In acest fel accesarea vecinilor de ordinul 1 va fi o operatie ce se poate realiza in timp constant. Functia $track\_local\_map$ este folosita de catre clasa Tracking. Aceasta primeste ca date de intrare cadrul curent si ultimul cadru cheie salvat. Nu returneaza nimic, doar incearca sa gaseasca cate un MapPoint pentru Feature-urile care inca nu au fost corelate cu un punct din spatiu. Aceasta operatie este costisitoare si functioneza in felul urmator:

1. sunt cautate toate KeyFrame-urile vecine de gradul 1 si 2 cu ultimul KeyFrame adaugat
2. din aceste KeyFrame-uri sunt extrase toate MapPoint-urile observate de catre ele
3. MapPoint-urile sunt proiectate si sunt cautate potriviri pentru Feature-urile care inca nu au MapPoint-uri asociate.

Pentru a nu fi necesar sa calculam de fiecare data KeyFrame-urile vecine si harta locala de MapPoint-uri, le stochez ca variabile in interiorul clasei Map. Acestea vor fi modificate in momentul in care un KeyFrame este adaugat in harta. Intr-un caz ideal, ar trebui ca pentru fiecare Feature adaugat sa se gaseasca un MapPoint, dar acest lucru rareori se intampla. In situatia in care s-au gasit mai putin de 30 de puncte din spatiu care s-au proiectat corect in imagine, se considera ca a aparut o eroare de urmarire si algoritmul incepe o etapa de relocalizare.

Clasa OrbMatcher este responsabila de realizarea urmarii feature-urilor asemanatoare intre cadre consecutive. Inainte de a incepe prezentarea metodelor implementate, voi descrie pipeline-ul de procesare al unui punct din spatiu pentru a fi considerat observabil de catre camera. Avem o instanta a obiectului MapPoint mp, daca una dintre operatiile prezentate esuaeaza, punctul respectiv este ignorat de catre KeyFrame-ul curent.

1. Se proiecteaza coordonatele globale ale mp in planul imaginii folosind matricea de estimare a pozitiei T_{cw} si matricea parametrilor camerei K. Se verifica daca coordonatele proiectiei sunt valide pentru imagine.
2. Se calculeaza distanta d de la centrul camerei la mp. In functie de valoarea octavei stocata in acest MapPoint, se pot estima o limita minima si maxima pentru d. Daca valoarea obtinuta nu se incadreaza in acest interval se considera ca punctul este invalid.
3. Cu ajutorul geometriei analitice se obtine ecuatia dreptei care uneste mp si centrulul camerei. Aceasta dreapta si vectorul de directie al MapPoint-ului, trebuie sa creeze un unghi cu o valoare mai mare de 60 de grade pentru a fi considerat mp observabil.

Daca aceste 3 verificari au fost realizate cu succes se considera ca punctul poate fi observat de catre camera. Exista 2 functii responsabile de asocierile intre cadrele curente, scopul acestora este ca gaseasca corespondente intre Feature-urile din cadrul curent si MapPoint-urile din spatiu. Pentru a se gasi perechea Feature f si MapPoint mp, trebuie ca mp sa se proiecteze in vecinatatea f iar descriptorii asociati atat Feature-ului cat si al MapPoint-ului sa aiba distanta Hamming sub un prag, setat in aceasta implementare la 50. O metoda ar fi compararea tuturor Feature-urilor din spatiu, cu totalitatea MapPoint-urilor observate de cadrul curent. Dar aceasta metoda ar fi ineficienta. O alta abordare ar fi separarea Feature-urilor in clustere in functie de distanta Hamming a descriptorilor, abordare stabila dar lenta si preferabil de utilizat cand nu ne putem baza pe estimarea matricei de pozitie a cadrului anterior. Iar cealalta abordare o reprezinta clusterizarea in functie de coordonatele in imagine ale Feature-ului.
Functie $match\_frame\_reference\_frame$ implementeaza prima metoda. Aceasta primeste ca parametru 2 vectori de feature-uri calculati de libraria DBoW2, unul asociat cadrului curent, pentru care estimam matricea de pozitie si unul asociat cadrului anterior, pentru care cunoastem deja matricea de pozitie si asocierile de tip (Feature, MapPoint). Elementele acestor vectori sunt de tip $(int, vector\_descriptori)$. Daca 2 astfel de perechi au prima valoare egala intre ele, inseamna ca cei 2 vectori de descriptori fac parte din acelasi cluster, conform arborelui din libraria DBOW2. Fiecare descriptor are asociata o instanta a clasei Feature. In cadrul anterior, instanta poate avea sau nu, un MapPoint corespondent. Daca exista acel MapPoint se poate proiecta in imagine. Descriptorul intern al MapPoint-ului este comparat cu ceilalti descriptori din acelasi cluster din cadrul curent si se aplica testul de proportionalitate Lowe pentru a garanta ca descriptorul cu distanta Hamming minima este cel mai bun. Functia $match\_consecutive\_frames$ este mai simpla si implementeaza a doua metoda. MapPoint-ul din spatiu este proiectat in imagine si toate Feature-urile dintr-o zona circulara de raza de variabila sunt considerati posibili candidati pentru a crea o asociere (Feature, MapPoint). Se calculeaza distanta Hamming intre descriptorul MapPoint-ului si cel al Feature-ului. Iar descriptorul cu distanta minima si mai mica decat un prag setat la 100 este considerat ca fiind cel mai potrivit. Feature-ul asociat acelui descriptor, va pastra o referinta a MapPoint-ului.

Clasa MotionOnlyBA implementeaza in Ceres algoritmul Motion Only Bundle Adjustment, primeste ca date de intrare KeyFrame-ul curent si returneaza matricea de pozitie optimizata. Libraria lucreaza cu o notiune din C++ numita functori. Acestea sunt clase/structuri pentru

care s-a facut overload la operatorul $()$. Clasa BundleError se afla din aceeasi categorie si implementeaza functia de eroare obtinuta din proiectarea unui MapPoint si asocierea acestuia cu un Feature. Pentru a crea problema de optimizare, clasa ceres::Problem trebuie sa stie care parametrii trebuie optimizati si functia de eroare pe care trebuie sa o minimizeze. In cazul acestui algoritm, singurul lucru care va fi modificat este matricea de pozitie a KeyFrame-ului pe care o voi converti in forma $se(3)$, transformand-o intr-un vector de 7 elemente. Iar pentru functia de eroare, nu voi scrie explicit ca este suma erorilor de proiectie. In schimb, voi initializa pentru fiecare asociere de tip (Feature, MapPoint) cate un element al clasei BundleError. Algoritmul de optimizare implementat de libraria Ceres, va incerca in mod independent sa reduca valoarea erorii pentru fiecare pereche in parte, modificand pe rand vectorul pozitiei. Exista un motiv pentru care schimb modul in care este exprimata pozitia camerei, matricea de pozitie contine 2 componente: matricea de rotatie R si un vector de translatie t. Pentru t nu exista restrictii de modificare atata timp cat aceasta nu aduce modificari mari intre pozitiile a doua cadre consecutive, orice mod in care ar varia parametrii acestui vector, in continuare semnificatia lui de vector de translatie ramane nealterata, in aceasta situatie putem spune ca parametrii sunt alterati de catre libraria Ceres folosind $EuclidianManifold$, mici modificari bazate pe calcularea derivatelor partiale ale acestora din functia de eroare definita in BundleError, asemanator modului in care sunt modificati parametrii in retele neurale. Pentru matricea de rotatie R nu se mai poate aplica aceeasi logica. Aceasta trebuie sa faca parte din structura de tip grup numit $SO(3)$, adica sa respecte egalitatea $R * R^t = R^t * R = I$ si trebuie sa reprezinte o rotatie reala pe cele 3 axe. Alterarea aleatorie a parametrilor ar duce la o matrice invalida. Din aceasta cauza, modificarea rotatiei trebuie facuta cu un anumit unghi iar acest lucru se poate realiza printr-o inmultire de 2 matrici de rotatie valide. Din pacate nu exista implementare in forma matriceala pentru schimbarea unghiului de rotatie, dar este pentru Quaternioni. Din aceasta cauza fac conversia din matrice de pozitie in vector din categoria $se(3)$, iar pentru primii 4 parametrii asociati rotatiei, optimizarea lor se realizeaza folosind $QuaternionManifold$. Aceasta abordare rezolva problema instabilitatii numerice si garanteaza ca rezultatul operatiei de optimizare este un element valid in $se(3)$, ce poate fi ulterior convertit in forma matriceala. In functie de categoria din care face parte Feature-ul, acesta este considerat monocular sau stereo. Functia de eroare implementata de clasa BundleError este identica pentru ambele, cu exceptia ca pentru punctele stereo, este verificata si distanta la care se afla punctul fata de valoarea la care a fost estimata de camera RGBD.

Pentru a preveni instabilitatea cauzata de punctele de tip outlier, functia Huber descrisa in capitolul anterior este folosita in calcularea finala a erorii de proiectie. In implementara oficiala realizata de g2o, agloritmul de optimizare este rulat de 4 ori, si dupa fiecare executie sunt eliminate punctele de tip outlier. Experimental, am observat ca etapa de optimizare cadru cu cadru este cea mai costisitoare operatie pe care o realizeaza algoritmul de ORB-SLAM2, executia acesteia de 4 ori, nu creste semnificativ acuratetea si reduce viteza de prelucrarea la aproximativ 5 cadre pe secunda, facandu-l nepotrivit pentru un sistem in timp real. Am observat ca obtin rezultate foarte bune, ruland o singura data Motion Only Bundle Adjustment, urmat apoi de o etapa de eliminare a corelatiilor (Feature, MapPoint) de tip outlier. Daca mai putin de 3 asocieri raman, se considera ca algoritmul a acumulat prea multe erori in urmarirea cadru cu cadru si trece intr-o stare de relocalizare.

Clasa Tracker realizeaza urmarirea traiectoriei cadru cu cadru. Aceasta integreaza fiecare dintre componentele definite anterior, si este responsabila de captarea cadrului curent, transformarea acestuia in KeyFrame si luarea deciziei daca va fi salvat in Map pentru a completa harta mediului inconjurator. Pasii urmatori se executa pentru fiecare cadru in parte:

1. Se creeaza KeyFrame-ul curent.
2. Se estimeaza matricea de pozitie pe baza legii de miscare.
3. Se realizeaza asocierea intre Feature-urile (puncte 2D) din cadru curent si MapPoint-urile observate de cadru anterior (puncte 3D)
4. Pe baza asocierilor respective realizate anterior, se optimiza matricea de pozitie a KeyFrame-ului curent, sunt eliminate asocierile de tip outlier
5. Este proiectata harta locala pe cadrul curent, si se gasesc noi asocieri (Feature, MapPoint), se executa din nou aceeasi operatie de optimizare Motion Only Bundle Adjustment
6. Este evaluat KeyFrame-ul curent, se verifica daca trebuie salvat in clasa Map.

Cadrul curent si matricea de adancime sunt citite de TumDatasetReader. In imaginea RGB se foloseste ORB pentru a extrage un vector de KeyPoint-uri si un vector de descriptori. Acestea sunt folosite pentru a initializa un obiect de tip KeyFrame. Pentru algoritmul ORB se foloseste o versiune modificata implementata in clasa ORBextractor si este conceputa sa extraga aproximativ 1000 de puncte cheie, acestea fiind distribuite cat mai egal pe suprafata imaginii. Daca un numar foarte mare de keypoint-uri s-ar obtine din aceeasi zona, acuratetea

estimarii ar avea de suferit, pixelii din zonele aflate mai aproape de camera se misca cu o viteza mai mare decat cei aflati in departare, daca am considera doar punctele dintr-o anumita in zona in realizarea estimarii, am obtinute variatii in miscare prea bruste / lente depenzind de locul unde s-au gasit majoritatea punctelor. Parametrii setati pentru algoritmul ORB sunt urmatorii: 1000 de feature-uri, factorul de scalare al imaginii este 1.2, exista maxim 8 nivele, si algoritmul FAST care face extragerea initiala de KeyPoint-uri sa ia in considerare zona respectiva daca diferenta de intensitate intre pixeli este de la 20 in sus. Daca in schimb, zona este slab texturata atunci poate sa seteze aceasta diferenta la 7, pentru a garanta ca vor fi gasite feature-uri chiar si in cele mai dezavantajoase zone din imagine. De-a lungul duratei de viata a algoritmului, clasa Tracker pastreaza 4 referinte de tip KeyFrame: cadrul curent care este analizat, 2 cadre imediat anterioare care vor fi folosite la estimarea pozitiei si ultimul cadru referinta care a fost creat. Cadrul referinta este ultimul KeyFrame adaugat in Map si indica aproximativ in ce zona se afla camera si care MapPoint-uri ar trebui sa fie vizibile. Cadrele mai vechi care nu au fost salvate in Map au fost sterse pentru a reduce cantitatea de memorie folosita. Pentru ultimul KeyFrame creat urmeaza etapa de estimare a pozitiei curente, aceasta se face pe baza legii de miscare, iar valorile matricii vor fi calculate folosindu-ne de cele 2 cadre salvate in Tracker. Important aici de observat ca pentru primul cadru citit, pozitia acestuia este matricea identitate 4*4, acest lucru sugerand ca dispozitivul care inregistreaza mediul considera ca primul KeyFrame este chiar originea sistemului de coordonate, iar toate matricile de pozitie viitoare sunt de fapt transformari relative fata de origine. Primul KeyFrame va fi salvat intotdeauna in clasa Map si este utilizat pentru a initializa primele puncte de tip MapPoint: pentru toate Feature-urile de tip stereo din imagine, se vor creea puncte in spatiu. Din cauza acestui mod de initializare, ORB-SLAM2 este sensibil pana la aparitia urmatorului cadru cheie, estimarile facute de acesta in prima etapa fiind predispuse la erori. Uneori algoritmul isi pierde orientarea cu totul, fiind necesara o etapa de relocalizarea, sau de reluare a executiei acestuia. ORB-SLAM3, implementeaza o metoda mult mai robusta de initializare, generand mai multe harti locale in situatia in care urmarirea cadru cu cadru esueaza si le uneste intre ele in momentul in care recunoaste o zona pe care a vizitat-o deja. Dupa ce a fost create KeyFrame-ul si a fost facuta estimarea initiala a pozitiei, clasa OrbMatcher este folosita pentru a gasit corelatii intre Feature-uri si MapPoint-uri. Alegerea metodei care indeplineste acest lucru fiind determinata de numarul de KeyFrame-uri create de la ultima relocalizare sau de la adaugarea unui nou cadru cheie in Map. Daca nu se vor gasi minim

15 asocieri, se va considera ca algoritmul si-a pierdut orientarea, altfel, asocierile respective vor fi utlizate de catre MotionOnlyBA pentru a realiza optimizarea pozitiei. Perechile de tip outlier vor fi eliminate si noua pozitie a KeyFrame-ului va fi returnata. Daca vor ramane mai putin de 3 asocieri se va considera, din nou, ca algoritmul si-a pierdut orientarea. In final, se foloseste clasa Map pentru a proiecta toate punctele din harta locala pe cadrul curent iar asocierile gasite vor trece din nou printr-un proces de optimizare. Daca nu se gasesc minim 50 de perechi (Feature, MapPoint) inseamna ca urmarirea cadrului curent a esuat. Altfel se trece la etapa urmatoare si se va decide daca vom stoca in Map KeyFrame-ul curent. Acest lucru se va intampla daca urmatoarele conditii vor avea loc simultan.

1. au trecut mai mult de 30 de cadre de la ultimul KeyFrame adaugat in Map
2. numarul de MapPoint-uri in cadrul curent este 25% din numarul urmarit de cadrul de referinta
3. cadrul curent are cel putin 70 de Feature-uri de tip stereo, cu distanta dintre centrul camerei si punct este mai mica de 3.2 metri si urmareste cel putin 100 de MapPoint-uri

Clasa LocalMapping este responsabila de optimizarea hartii algoritmului. Aceasta sterge/adauga KeyFrame-uri si MapPoint-uri, iar la fiecare cadru cheie nou, realizeaza operatia de Local Bundle Adjustment. Aceasta metoda optimizeaza matricile de pozitie si toate MapPoint-urile vecinilor directi si cei de categoria a doua pentru KeyFrame-ul abia adaugat. In momentul in care thread-ul de Tracking considera ca un nou cadru cheie trebuie de adaugat in harta, se executa metoda principala $local\_map$, aceasta indeplineste urmatoarele operatii:

1. Creeaza noi MapPoint-uri din primele 100 de Feature-uri de tip stereo, sortate in ordine crescatoare dupa distanta la care se afla acestea de centrul camerei
2. Adauga cadrul curent in graful de KeyFrame-uri stabilind vecinii directi ai acestuia
3. Noile MapPoint-uri create sunt adaugate intr-o lista numita $recently\_added$, pentru a iesi din aceasta lista, punctele trebuie sa treaca un test care dovedeste ca nu sunt rezultatul unui Feature eronat detectat de catre algoritmul ORB, si ca pot fi folosite cu incredere
4. Se executa operatia de $culling$, punctele sunt verificate daca sunt valide iar daca nu, memoria lor este eliberata.
5. Se foloseste operatia de triangulare pentru a crea noi MapPoint-uri din Feature-urile care se potrivesc intre ele si fac parte din cadre cheie diferite.

6. Se detecteaza entitatile de tip MapPoint care reprezinta acelasi punct din spatiu, iar una dintre referinte este stearsa pentru creste coorenta hartii si a creste ponderea conexiunii dintre KeyFrame-urile adiacente
7. Se executa operatia de KeyFrame culling, se verifica daca informatiile pe care le detine un KeyFrame, adica totalitatea valorilor de tip MapPoint pe care le detine, sunt observate si din alte cadre. Daca peste 90% din punctele observate de un anumit cadru sunt vizibile si din alte cadre, KeyFrame-ul analizat este considerat redundant si memoria lui este eliberata. Acest lucru garanteaza ca graful clasei Map, contine doar cadre esentiale pentru reprezentarea norului de puncte.

In etapa a 4-a se executa operatia de $culling$, aceasta elimina punctele care nu sunt de incredere. Singurele puncte care nu vor trece prin aceasta etapa de verificare sunt cele generate de primul KeyFrame, tot primul KeyFrame nu poate fi sters deoarece ar da peste cap sistemul de coordonate local sub care lucreaza ORB-SLAM2. Un punct este considerat de incredere daca din momentul in care a fost creat, el a fost observat in 3 cadre cheie consecutive si daca a fost observat in cel putin 25% din numarul total de cadre care au trecut de la creearea acestuia. Ambele conditii trebuie sa fie respectate simultan in momentul in care se face verificarea punctului respectiv. Politica pe care o urmeaza familia de algoritmi ORB-SLAM este sa genereze multe puncte, fara a impune restrictii, pe care apoi le va supune acestui test de relevanta.

Ultima clasa este cea de MapDrawer pe care o folosesc pentru a afisa norul de MapPoint-uri, cadrul curent analizat si pozitiile cadrelor cheie observate. Folosesc libraria Pangolin si OpenGL pentru desenarea fiecarei structuri, camera urmareste cadrul curent. Interfata grafica scade viteza de procesare a cadrelor dar este o modalitate eficienta de a intelege vizual ce se petrece in algoritm. Implementarea pentru interfata grafica am realizat-o spre final, cand aveam celelalte componente finalizate, lucru care a ingreunat procesul de dezvoltare deoarece lucram cu valori numerice in terminal. Acum daca as reincepe implementarea, interfata grafica ar fi printre primele lucruri pe care le-as realiza. Datorita acestei clase am reusit sa gasesc erori in modul de constructie al grafului ponderat din clasa Map si al modului in care proiectam punctele in spatiu.

5.3 Pipeline antrenare FastDepth

Pentru reteaua Neurala FastDepth pipeline-ul de antrenare a fost scris folosind libraria Pytorch iar pentru operatiile de preprocesare folosesc libraria Albumentations. Setul de date pe care am facut antrenarea se numeste Nyu Depthv2 Dataset[35] si l-am obtinut de pe Kaggle. Rezultatul acestui pipeline trebuie sa fie un fisier de tip ONNX cu valorile parametrilor retelei FastDepth in urma antrenarii pe setul de date. O problema pe care am observat-o la setul de date este ca pentru imaginile de antrenament, adancimile sunt exprimate ca fiind in intervalul $[0, 255]$, pe cand in setul de date de validare, acestea se afla intre $[0, 10000]$ reprezentand valorile in milimetrii ale distantelor. O limitarea acestui set de date este ca nu poate detecta distante mai mari de 10 metri. Dar considerand ca algoritmul trebuie sa functioneze pentru incaperi de mici dimensiuni, consider ca aceasta distanta maxima nu ar trebui sa reprezinte o problema. Pentru antrenare am ales sa urmez lucrarea stiintifica[3] si am setat hiperparametrii:

- Optimizatorul folosit a fost implementarea din Pytorch pentru Stochastic Gradient Descent, torch.SGD, avand un learning rate de $1e{-}3$, o valoare a momentumului de $\beta = 0.9$ si $weight\_decay = 1e - 4$.
- antrenarea s-a realizat pentru 50 de epoci iar durata antrenarii a fost de aproximativ 6 ore jumatate. Laptopul pe care am antrenat este un Asus TUF Gaming A15, avand un procesor AMD Ryzen 7 cu o frecventa de 4.2 GHz si placa video NVIDIA GeForce RTX 2060, cu o memorie de 6GB.
- Imaginile in setul de date au o dimensiune de $(3, 460, 640)$. Pentru a creste viteza de procesare am modificat dimensiunile la $(3, 256, 320)$ si am aplicat o functie de normalizare de tip min_max. Ambele transformari sunt aplicate atat pe setul de date de antrenare cat si pe cel de test.
- un batch de date are dimensiune de 8
- In lucrarea FastDepth functia de pierdere folosita este L1Loss, aceasta fiind suma diferentelor dintre valoarea reala si cea determinata de reteaua neurala in modul. In implementarea mea am ales sa folosesc o functie de pierdere mai robusta conform acestei lucrari stiintifice[36].

Acuratetea a fost verificata prin compararea diferentei relative intre valorile obtinute prin inferenta si cele reale cu un factor $RELATIVE\_ERROR = 0.15$. Aceasta operatie a fost

realizata pentru fiecare pixel in parte, iar acuratetea reprezinta procentul de pixeli cu o valoare care se incadreaza in limita impusa de RELATIVE_ERROR. Pentru a preveni antrenarea pentru intervale lungi fara a obtine rezultate, am avut 2 metode pe care le-am implementat: o strategia de early stopping: in situatia in care valoarea acuratetii nu ar fi crescut pentru 5 epoci antrenarea ar fi fost oprita si o strategie pentru modificarea learning rate-ului in timpul antrenarii. Daca acuratetea nu crestea pentru 3 epoci valoarea parametrului sa fie redusa la 0.3 din valoarea initiala. In practica am observat ca reteaua converge aproape monoton catre o valoare optima. Functia de pierdere primeste ca date de intrare matricea de adancime obtinuta de catre reteaua neurala si matricea cu valori reala din setul de date, denumita groundtruth, si returneaza o valoarea numerica de tip double care exprima cat de departe se afla estimarea noastra de realitate. Ideea antrenarii unei retele neurale este minimizarea acestei valori. Functia de eroare este alcatuita dintr-o combinatie liniara a 3 componente diferite[36]: L1Loss, GradientEdgeLoss si Structural Similarity Loss, formula matematica este:

$$loss = 0.6 \cdot L1Loss + 0.2 \cdot GradientEdgeLoss + StructuralSimilarityLoss \tag{14}$$

Structural Similary Loss se asigura ca media si distributia standard pe care o urmeaza valorile estimate, se apropie de media si distributia standard a matricei groundtruth. In comparatie cu celelalte 2 componente ale functiei de pierdere care sunt aplicate la nivel de pixel, aceasta abstractizeaza rezultatele ca fiind 2 distributii Normale cu parametrii $\mathcal{N}(\mu, \sigma^2)$ care trebuie sa se suprapuna. Principiul de functionare pentru GradientEdgeLoss este ca pixeli din regiuni apropiate trebuie sa aiba cam aceleasi valori de estimare ale distantei si ca diferenta intre pixeli adiacenti pe axele x si y, ar trebui sa fie identica cu cea din imaginea groundtruth. Aceasta poate fi scrisa in felul urmator, unde N reprezinta numarul de pixeli din imagine, iar derivata valorilor pixelilor in raport cu axa de coordonate reprezinta diferenta intre matricea imaginii initiale si aceeasi matrice avand un rand shiftat la dreapta pentru axa X notata $\frac{\partial I}{\partial x}$ si un rand shiftat vertical pentru axa Y notata $\frac{\partial I}{\partial y}$.

$$L_{\text{edges}} = \frac{1}{N} \sum_{i=1}^{N} \left(\left| \frac{\partial I_{\text{pred}}}{\partial x} - \frac{\partial I_{\text{true}}}{\partial x} \right| + \left| \frac{\partial I_{\text{pred}}}{\partial y} - \frac{\partial I_{\text{true}}}{\partial y} \right| \right) \tag{15}$$

6 EVALUARE

6.1 Setul de date TUM RGBD Dataset

Setul de date utilizat pentru a realiza evaluarea se numeste TUM RGBD Dataset[2]. Acesta contine numeroase subseturi, fiecare verificand un aspect diferit al implementarii, ajutand la creearea unui imagini de ansamblu cu privire la robustetea algoritmului in functie de mediu in care se lucreaza si de traiectoria pe care o urmeaza. Cele 2 subseturi pe care le-am considerat potrivite pentru implementarea sunt:

- Subsetul rgbd_dataset_freiburg1_xyz contine cadrele unui video de 35 de secunde, in care traiectoria este in principal alcatuita din translatii, exista foarte putine rotatii fiind ideal pentru a verifica daca estimarea pozitiei in spatiu este corect realizata.
- Subsetul rgbd_dataset_freiburg1_rpy are 27 de secunde si contine foarte putine translatii. Exista in schimb numeroase schimbari bruste de rotatie care reduc acuratetea imaginii captate, testand la maxim capacitatea algoritmului ORB de a extrage feature-uri. Sistemul isi schimba orientarea pe toate cele 3 axe, fiind unul dintre cele mai dificile subseturi de date pe care se poate face antrenarea. Algoritmul ORB-SLAM2 este sensibil la operatiile de rotatie, mai ales atunci cand camera isi schimba orientarea catre o zona necunoscuta. Pentru a crea harta zonei respective sunt generate numeroase KeyFrame-uri si MapPoint-uri, pe care algoritmul trebuie sa le filtreze in clasa de LocalMapping, lucru care creste complexitatea temporala si spatiala si scade acuratetea sistemului.

Videourile sunt realizate cu ajutorul unei camere RGBD Microsoft Kinetic, avand frecventa de 30 de cadre pe secunda, setul de date contine imaginile de tip RGB, hartile de adancime pentru fiecare cadru in parte, vectorii de pozitie in forma $se(3)$, primii 3 parametrii fiind pozitia in spatiu (tx, ty, tz) iar urmatorii 4 parametrii sunt asociati matricei de rotatie, scrisa sub forma de Quaternion, (qw, qx, qy, qz) si timestamp-urile asociate momentului in care au fost inregistrate fiecare din valorile din setul de date. Cu ajutorul acestor timestammp-uri putem

crea asocieri de tip (imagine RGB, matrice de adancime, pozitie) pe care le putem transmite algoritmul ORB-SLAM2. Pozitia este considerata ca fiind valoarea ideala, groundtruth, si va fi comparata cu rezultatele obtinute. Clasa TumDatasetReader este responsabila de citirea datelor si stocarea matricilor de pozitie obtinute pentru fiecare cadru. Dupa parcurgerea intregului set de date, valorile estimate sunt salvate intr-un fisier de tip text unde vor fi comparate cu cele reale.

6.2 Metrici utilizate

Algoritmul ORB-SLAM2 scrie intr-un fisier estimarile matricilor de pozitie pentru fiecare cadru in parte. Pentru a realiza comparatia cu datele de tip groundtruth din setul de date, folosesc un pachet din python numit evo, acesta este capabil sa creeze un grafic al traiectoriei, permitand astfel o reprezentare vizuala a rezultatelor si o separare a acestora in functie de ceea ce vreau sa evaluez: viteza, translatia sau orientarea. De exemplu, figura de mai jos reprezinta variatia translatie pe fiecare dintre cele 3 axe. Cu albastru este valoarea de tip groundtruth iar cu galben este estimarea realizata de implementarea mea pentru algoritmul ORB-SLAM. Rezultatele sunt obtinute pentru subsetul de date rgbd_dataset_freiburg1_xyz, acesta fiind special conceput pentru a testa corectitudinea estimarii translatiei intre cadre.

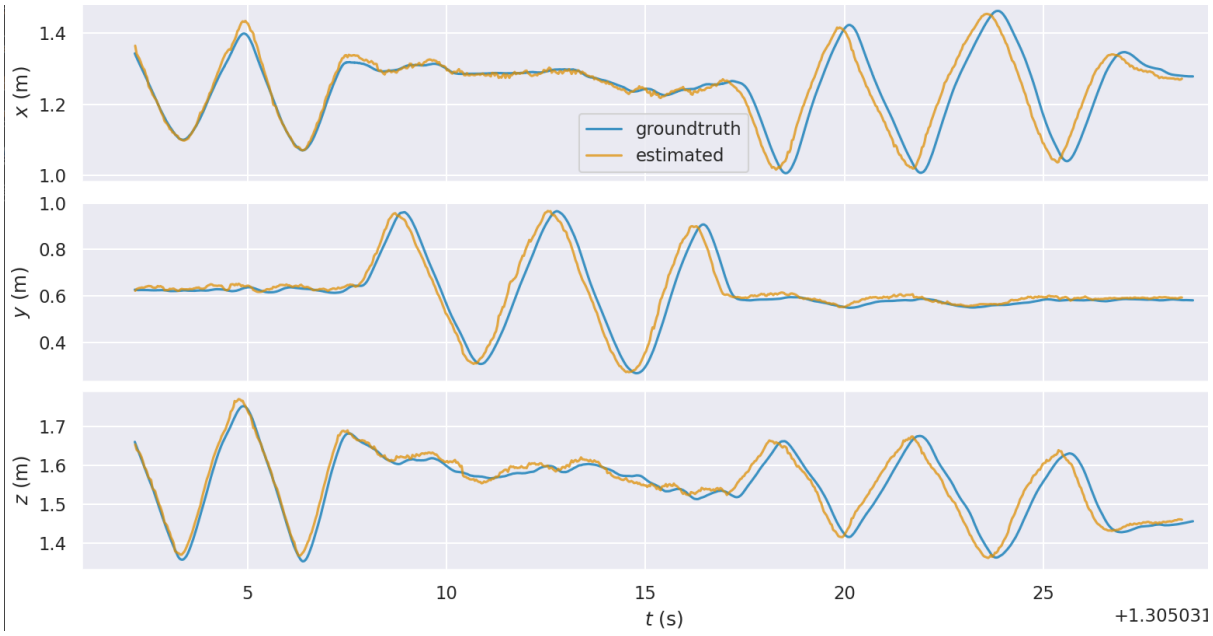

Figura 4: Graficul translatiei pe fiecare din cele 3 axe, groundtruth si estimare ORB-SLAM2

Consider ca o reprezentare grafica in care traiectoria groundtruth se suprapune exact cu ceeea

ce a obtinut estimarea ORB-SLAM2 poate fi considerata in mod neoficial, o metrica pe baza careia sa putem spune daca algoritmul functioneaza corect. Pentru exactitate se poate folosi: APE (Absolute Pose Error), aceasta metrica măsoară distanța euclidiană dintre pozițiile estimate și cele reale, la fiecare moment de timp. In general valorile scorurilor APE obtinute pentru ambele seturi de date sunt pana in 0.05, sugerand ca acuratetea este buna. Alte metrici pe care le folosesc pentru implementarea mea a algoritmului ORB-SLAM2 este numarul de secunde necesar pentru parcurgerea setului de date sau numarul de cadre pe secunda. Numarul de KeyFrame-uri create, in momentul in care sistemul nu mai poate realiza urmarirea cadru cu cadru, acesta insereaza un nou KeyFrame, cu cat sunt mai putine KeyFrame-uri noi adaugate, se poate considera ca traiectoria este usor de interpretat si ca sistemul este stabil. O alta metrica este legata de numarul de relocalizari pe care a trebuit sa le faca algoritmul pentru a parcurge setul de date. Relocalizarea apare in situatia in care urmarirea cadru cu cadru esueaza si este cautat KeyFrame-ul care seamana cel mai bine cu cadrul curent folosind vectorul de feature-uri calculat de metoda bag-of-words. Ideal, numarul necesar de relocalizari ar trebui sa fie 0.

In cazul rularii implementarii mele pe setul de date rgbd_dataset_freiburg1_xyz acesta dureaza in medie 67 de secude, functionand la aproximativ 15 cadre pe secunda, in medie este nevoie intre 5-7 Keyframe-uri noi pentru parcurgerea setului de date. Logica de relocalizare nu este deloc folosita, algoritmul fiind capabil sa realizeze urmarirea cadru cu cadru. Pe setul de date rgbd_dataset_freiburg1_rpy a durat 73 de secunde, videoul avand 27 de secunde, reprezinta aproximativ 10-11 cadre pe secunde. Acest lucru se datoreaza numeroaselor operatii de optimizare a hartii pe care trebuie sa le faca algoritmul deoarece sunt adaugate intre 17-19 KeyFrame-uri pentru a parcurge intreg setul de date, din cauza miscarilor bruste ale camerei care reduc considerabil claritatea imaginilor extrase. In continuare, numarul de relocalizari este 0. Mai jos, am atasat graficul care compara estimarea orientarii pentru fiecare cadru, estimarile fiind descompuse dupa cele 3 dimensiuni ale rotatiei. Graficul realizat de algoritmul ORB-SLAM2 pare sa fie shiftat in timp fata de cel real, dar sa aiba aproximativ aceeasi forma cu cel al valorilor de tip groundtruth. Consider ca problema poate sa porneasca de la modul in care sunt atasate timestamp-urile pentru fiecare cadru in parte, lucru care nu are legatura directa cu modul in care este realizata implementarea, ci cu modul in care setul de date creeaza perechile (imagine RGB, vector pozitie, matrice de adancime). Am incercat utilizarea retelei neurale FastDepth pentru a estima adancimea in loc de a folosi matricea de

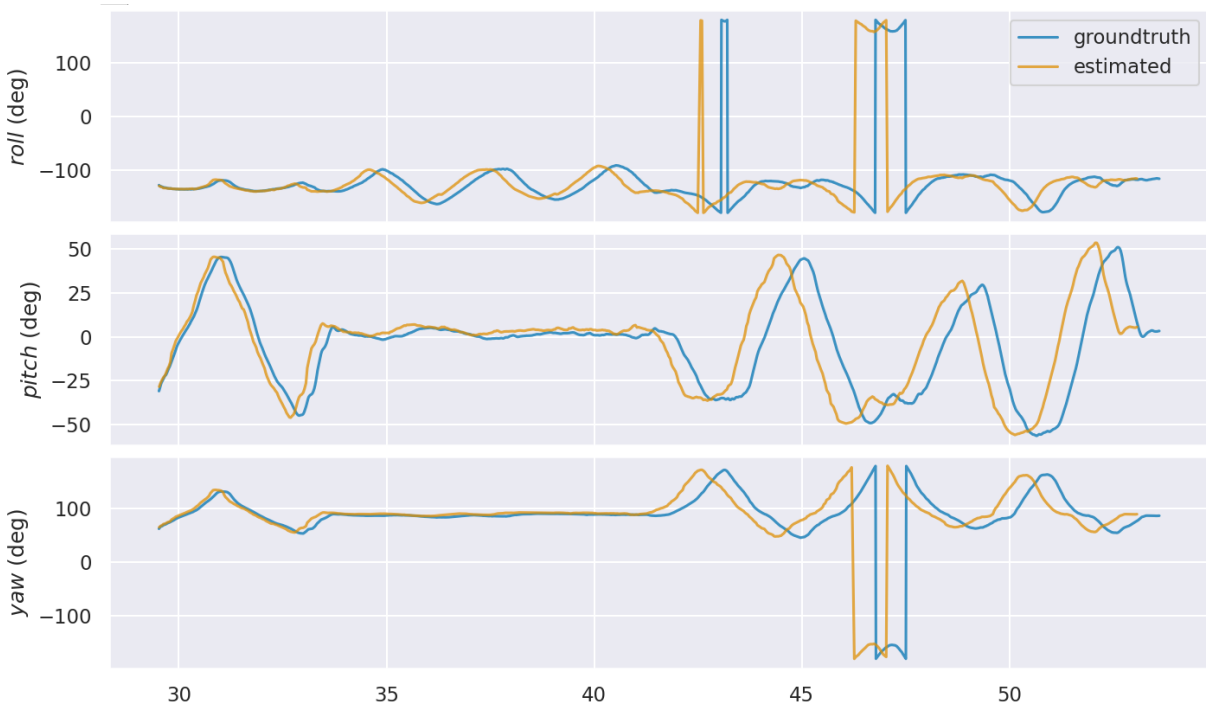

Figura 5: Graficul orientarii pentru setul de date rgbd_dataset_freiburg1_rpy

distante a setului de date TUM RGBD. Pentru a testa arhitectura voi lua doar imaginea si vectorul de pozitie simuland astfel o situatie in care sistemul ar avea doar o camera RGB. Problema cu aceasta implementare este ca reteaua neurala nu este suficient de exacta iar estimarile distantelor intre cadre consecutive in continuare variaza mult. Pentru subsetul de date rgbd_dataset_freiburg1_xyz implementarea intra in etapa de relocalizare in medie dupa primele 50-60 de cadre procesate. Sistemul este mult prea instabil pentru a realiza in mod corect urmarirea cadru cu cadru.

In etapele intiale ale dezvoltarii algoritmului am incercat utilizarea unui video realizat folosind camera telefonului pentru testarea implementarii. Au fost 3 probleme pe care le-am intalnit: nu puteam determina parametrii corecti ai camerei telefonului. Aveam nevoie de distanta focala, de coordonatele centrului imaginii si de parametrii de distorsiune. A doua problema o reprezenta lipsa unei matrici de adancime pentru fiecare cadru iar cea de-a treia, era lipsa un vector de pozitie pentru fiecare imagine. Chiar si in situatia in care obtineam parametrii camerei telefonului folosind algoritmi implementati in OpenCV, in continuare nu puteam fi sigur daca estimarile realizate de mine sunt cele corecte. Din cauza acestor multe probleme, am ajuns la concluzia ca un set de date ar fi o varianta mai potrivita.

7 CONCLUZII

O problema pe care am avut-o in testarea algoritmului a fost ca nu am putut face functionala implementarea initiala a ORB-SLAM2, exista conflicte intre versiunile de biblioteci Eigen si g2o. Versiunea de Eigen folosita la momentul respectiv nu mai exista acum in repo-ul oficial si eu nu am reusit sa o accesez pentru a testa.

Libraria Ceres este usor de folosit pentru problemele de optimizare non-liniare, consider ca cel mai mare plus pe care il aduce lucrarea mea este ca am cea mai completa implementare a algoritmului Bundle Adjustment in aceasta librarie la care se adauga o logica de filtrare a punctelor de tip outlier si are caz separat de folosire atat pentru punctele monoculare cat si pentru cele stereo. In plus am adus optimizari la codul oficial pentru ORB-SLAM2: implementarea lor nu elibereaza absolut deloc memoria pentru MapPoint-urile si KeyFrame-urile considerate invalide, eu am rezolvat aceasta problema, extinzand durata pentru care poate rula algoritmul si facandu-l potrivit pentru sistemele embedded. De asemenea, in implementarea oficiala nu este folosita nicio structura de tip dictionar, clasele principale folosite au parametrii de stare care isi modifica valoarea la fiecare cadru, functiile avand efecte laterale care genereaza erori greu de urmarit si corectat. In total am scris 4030 de linii de cod in C++ si am modificat codul pentru numeroase functii, facandu-l mai usor de inteles si mentinut. ORB-SLAM2 indeplineste 3 functii: urmarirea cadru cu cadru, corectarea erorilor, si inchiderea buclelor. Etapa de inchidere a buclelor nu am reusit sa o implementez din cauza apropierii termenului de predare, cu toate acestea, algoritmul obtine in continuare rezultate bune pentru subseturile de date alese.

In ciuda faptului ca utilizarea unei retele neurale pentru a inlocui o camera RGBD nu a functionat asa cum am crezut initial, algoritmul devine instabil si isi pierde complet orientarea dupa primele 50 de cadre, consider ca o retea neurala precum FastDepth poate fi folosita in estimarea adancimii pentru punctele care au avut atribuita distanta 0 de catre camera tip RGBD. O directie viitoare ar fi utilizarea unui sistem care combina cele 2 abordari. Separarea straturilor convolutionale in depthwise si pointwise s-a dovedit a fi o tehnica buna pentru a creste viteza arhitecturii astfel incat sa poata fi folosita in timp real.

Un lucru pe care il regret este ca nu am utilizat o interfata grafica inca de la primele etape, pentru a detecta erorile in timpul dezvoltarii algoritmului. Desi scorul calculat de APE este o metrica buna pentru a vedea cat de bine se potrivesc 2 estimari ale pozitiei unui cadru, o simpla valoare numerica nu este suficient pentru a avea o privire generala asupra modului in care functioneaza implementarea.

Ca directii viitoare, ma gandeam sa utilizez arhitecturi de retele neurale pentru sarcini bine delimitate, cum ar fi extragerea de feature-uri sau gasirea de corelatii de tip (Feature, MapPoint), adaugarea unui modul de object detection si a unui algoritm de planificare de trasee, pentru a indeplini sarcini simple de gasire a unor obiecte de mici dimensiuni. Pana acum algoritmul a folosit doar metode clasice, ORB pentru extragere de feature-uri, Brute Force pentru matching, bag-of-words pentru relocalizare. Acum, consider ca directia pe care ar trebui sa o urmeze aceasta clasa de algoritmi de tip SLAM este una in care tehnici de Machine Learning sunt folosite pentru a creste viteza si poate acuratetea operatiilor. Aceasta fiind si directia incurajata de lucrarile ce fac parte din state of the art pana la momentul curent.

BIBLIOGRAFIE

[1] Raul Mur-Artal și Juan D. Tardós, "ORB-SLAM2: An Open-Source SLAM System for Monocular, Stereo, and RGB-D Cameras", în *IEEE Transactions on Robotics* 33 (2016), pp. 1255–1262.

[2] Jürgen Sturm et al., "A benchmark for the evaluation of RGB-D SLAM systems", în *2012 IEEE/RSJ international conference on intelligent robots and systems*, IEEE, 2012, pp. 573–580.

[3] Diana Wofk et al., "FastDepth: Fast Monocular Depth Estimation on Embedded Systems", în *2019 International Conference on Robotics and Automation (ICRA)* (2019), pp. 6101–6108.

[4] Hugh Durrant-Whyte și Tim Bailey, "Simultaneous Localization and Mapping: Part I", în *IEEE Robotics & Automation Magazine* 13.2 (2006), pp. 99–110.

[5] Lahav Lipson, Zachary Teed și Jia Deng, "Deep patch visual slam", în *European Conference on Computer Vision*, Springer, 2024, pp. 424–440.

[6] Weifeng Wei et al., "Real-Time Dense Visual SLAM with Neural Factor Representation", în *Electronics* (2024).

[7] Zhiqi Zhao et al., "Light-SLAM: A Robust Deep-Learning Visual SLAM System Based on LightGlue under Challenging Lighting Conditions", în *ArXiv* abs/2407.02382 (2024).

[8] Liming Liu și Jonathan M. Aitken, "HFNet-SLAM: An Accurate and Real-Time Monocular SLAM System with Deep Features", în *Sensors (Basel, Switzerland)* 23 (2023).

[9] Carlos Campos et al., "ORB-SLAM3: An Accurate Open-Source Library for Visual, Visual-Inertial, and Multimap SLAM", în *IEEE Transactions on Robotics* 37.6 (2021), pp. 1874–1890, DOI: 10.1109/TRO.2021.3075644.

[10] Christian Forster, Matia Pizzoli și Davide Scaramuzza, "SVO: Fast semi-direct monocular visual odometry", în *2014 IEEE International Conference on Robotics and Automation (ICRA)* (2014), pp. 15–22.

[11] Zachary Teed, Lahav Lipson și Jia Deng, "Deep Patch Visual Odometry", în *ArXiv* abs/2208.04726 (2022).

[12] Ethan Rublee et al., "ORB: An efficient alternative to SIFT or SURF", în *2011 International Conference on Computer Vision* (2011), pp. 2564–2571.

[13] Tony Lindeberg, "Scale Invariant Feature Transform", în vol. 7, Mai 2012, DOI: 10.4249/scholarpedia.10491.

[14] Herbert Bay, Tinne Tuytelaars și Luc Van Gool, "SURF: Speeded Up Robust Features", în *European Conference on Computer Vision (ECCV)*, Springer, Mai 2006, pp. 404–417, DOI: 10.1007/11744023_32.

[15] Ebrahim Karami, Siva Prasad și Mohamed Shehata, "Image Matching Using SIFT, SURF, BRIEF and ORB: Performance Comparison for Distorted Images", în Nov. 2015, DOI: 10.48550/arXiv.1710.02726.

[16] Richard W. Hamming, "Error Detecting and Error Correcting Codes", în *Bell System Technical Journal* 29.2 (1950), pp. 147–160, DOI: 10.1002/j.1538-7305.1950.tb00463.x.

[17] Edward Rosten și Tom Drummond, "Machine Learning for High-Speed Corner Detection", în vol. 3951, Iul. 2006, ISBN: 978-3-540-33832-1, DOI: 10.1007/11744023_34.

[18] Christopher G. Harris și M. J. Stephens, "A Combined Corner and Edge Detector", în *Alvey Vision Conference*, 1988.

[19] Michael Calonder et al., "BRIEF: Binary Robust Independent Elementary Features", în (2010), ed. de Kostas Daniilidis, Petros Maragos și Nikos Paragios, pp. 778–792.

[20] Sameer Agarwal et al., "Bundle Adjustment in the Large", în *Computer Vision – ECCV 2010*, ed. de Kostas Daniilidis, Petros Maragos și Nikos Paragios, Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 29–42, ISBN: 978-3-642-15552-9.

[21] Peter J. Huber, "Robust Estimation of a Location Parameter", în *Annals of Mathematical Statistics* 35.1 (1964), pp. 73–101, DOI: 10.1214/aoms/1177703732.

[22] Erkam Guresen și Gulgun Kayakutlu, "Definition of artificial neural networks with comparison to other networks", în *Procedia Computer Science* 3 (2011), pp. 426–433, ISSN: 1877-0509, DOI: https://doi.org/10.1016/j.procs.2010.12.071.

[23] Simon J.D. Prince, *Understanding Deep Learning*, The MIT Press, 2023.

[24] Kaiming He et al., *Deep Residual Learning for Image Recognition*, 2015, arXiv: 1512.03385 [cs.CV].

[25] Bikram Shah, Manoj Guragai și Anil Verma, "Image Colorization Using AutoEncoder", în Feb. 2024.

[26] Ashhadul Islam și Samir Brahim Belhaouari, "Fast and Efficient Image Generation Using Variational Autoencoders and K-Nearest Neighbor OveRsampling Approach", în *IEEE Access* 11 (2023), pp. 28416–28426.

[27] Laurent SIfre și Stéphane Mallat, *Rigid-Motion Scattering for Texture Classification*, 2014, arXiv: 1403.1687 [cs.CV].

[28] Andrew G. Howard et al., *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*, 2017, arXiv: 1704.04861 [cs.CV].

[29] Mark Sandler et al., *MobileNetV2: Inverted Residuals and Linear Bottlenecks*, 2019, arXiv: 1801.04381 [cs.CV].

[30] Adam Paszke et al., "Automatic differentiation in PyTorch", în *NIPS-W*, 2017.

[31] Itseez, *Open Source Computer Vision Library*, https://github.com/itseez/opencv, 2015.

[32] Sameer Agarwal, Keir Mierle și The Ceres Solver Team, *Ceres Solver*, versiunea 2.2, Oct. 2023.

[33] Dorian Galvez-López și Juan D. Tardos, "Bags of Binary Words for Fast Place Recognition in Image Sequences", în *IEEE Transactions on Robotics* 28.5 (2012), pp. 1188–1197, DOI: 10.1109/TRO.2012.2197158.

[34] Rainer Kümmerle et al., "G2o: A general framework for graph optimization", în *2011 IEEE International Conference on Robotics and Automation*, 2011, pp. 3607–3613, DOI: 10.1109/ICRA.2011.5979949.

[35] Pushmeet Kohli Nathan Silberman Derek Hoiem și Rob Fergus, "Indoor Segmentation and Support Inference from RGBD Images", în *ECCV*, 2012.

[36] Muhammad Hafeez et al., "Depth Estimation Using Weighted-Loss and Transfer Learning", în *Proceedings of the 19th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications*, SCITEPRESS - Science și Technology Publications, 2024, pp. 780–787, DOI: 10.5220/0012461300003660.