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 bune, algoritmul fiind capabil să funcționeze pentru maxim 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.

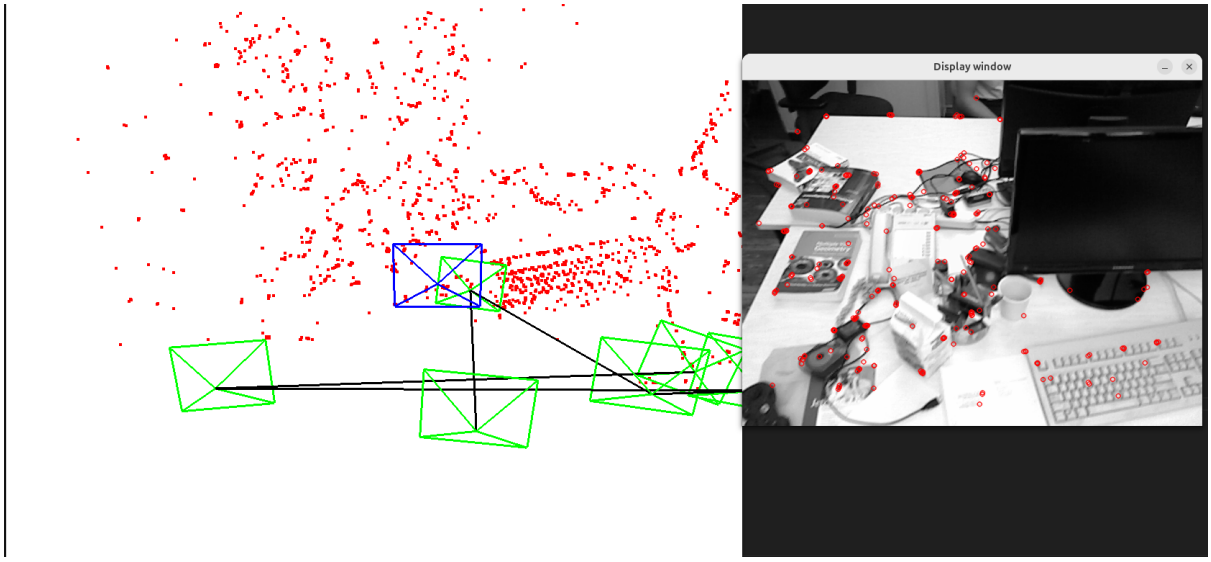

Figura 1: Interfața grafică, stânga reprezentarea hărții, dreapta extragere feature-uri cu ORB

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 etalon, cu multe posibilități de extindere. 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 punct de vedere al algoritmilor folosiți și al valorilor de intrare și de ieșire asociate acestora.

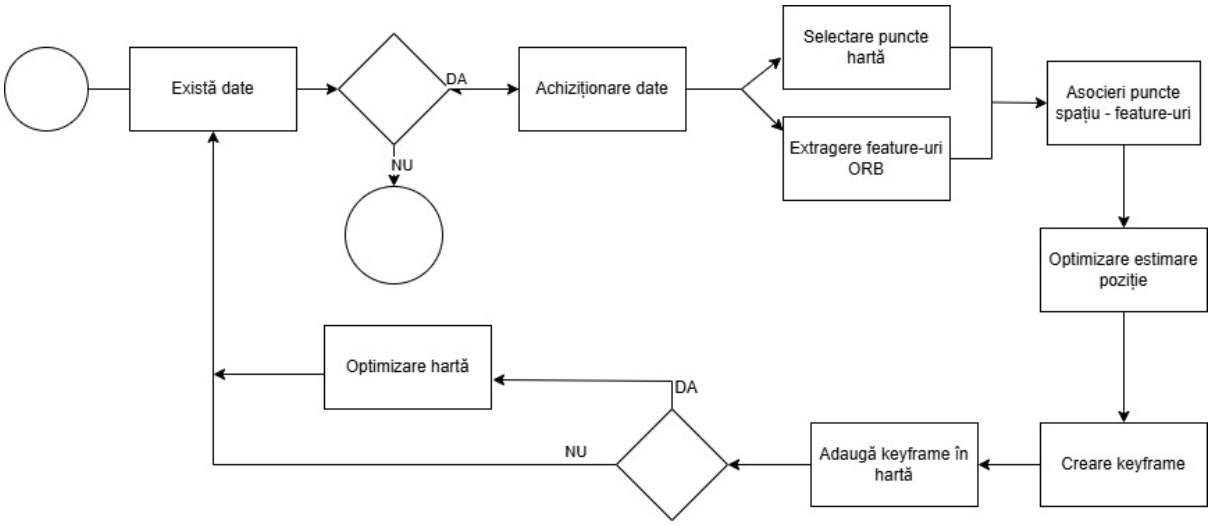

Figura 2: Diagrama UML de activități a întregului sistem

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 funcționalitate a acestei componente ar putea fi extragerea datelor de la alți senzori precum un giroscop sau un 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 primește 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 ăentru a 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 pixelii de pe suprafață. 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 cu mai multe coordonate în imagine. În schimb, o cameră complet mobilată ar fi o zonă puternic texturată, iar un algoritm de detecție 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 între 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 restricție.
- 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 biților despre care se știe că 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 atât 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 varianță. Astfel, au fost alese primele 256 dintre acestea pentru a alcătui descriptorul.

4.3 Harta punctelor din spațiu

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 că 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 găsească 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 asociați și calculând distanța 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. Asocierile sunt făcute cu ajutorul descriptorilor. Se calculează distanța Hamming, iar dacă valoarea obținută 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. Dacă N este numărul de feature-uri din primul set, M numărul de feature-uri din al doilea set și D, dimensiunea descriptorului, în cazul nostru 32, atunci complexitatea algoritmului devine $O(N * M * D)$. Destul de costisitor de folosit pentru un sistem în 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. Odată ce 2 keypoint-uri au fost considerate ca făcând referință la același punct din spațiu, cadrului curent îi este asociat un nou MapPoint.

4.5 Optimizarea Estimării Poziției Inițiale

Această componentă primește ca date de intrare estimarea poziției curente a camerei T_{cw} și o asociere bijectivă între 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 poziție T_{cw} atât matricea de rotație R_{cw}, cât și vectorul coloană de translație 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 coloană X_{camera}. Vom considera a 3-a valoare a acestui vector Z_c. Aceasta reprezintă distanța dintre planul camerei și punctul pe care îl 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. Partea 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 proiecție.

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

În urma execuției algoritmului obținem matricea de pozitie optimizată. Mai mult decât atât,

știm care dintre perechile (feature, MapPoint) au avut statutul de outlier și le putem elimina pentru a nu influența în mod negativ funcționalitatea algoritmului.

4.6 Crearea unui cadru cheie

Această componentă primește ca date de intrare absolut toate informațiile procesate de până acum pentru cadrul curent: imaginea de tip rgb, matricea de adâncime, punctele cheie, descriptorii, asocierile (feature, MapPoint) și matricea estimării poziției. Toate acestea vor alcătui împreună un cadru cheie care va fi salvat în memorie. Salvarea pozițiilor cadrelor anterioare ne poate ajuta în 2 feluri. Putem folosi 2 cadre anterioare pentru a estima poziția celui care urmează bazându-ne pe legea inerției. O dată începută deplasarea camerei într-o anumită directie, este foarte probabil ca aceea miscare să fie menținută și la următorul cadru. Fie T_{cw} matricea de poziție pentru cadrul la care vrem să estimăm deplasarea, iar T_{cw1}, T_{cw2} matricile de poziție a celor 2 cadre imediat predecesoare. Formula de estimare a poziției 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 și a traseului parcus. Încercam să salvăm numărul minim de cadre necesare pentru a reproduce harta de MapPoint-uri a mediului înconjurător. Un cadru cheie nou (KeyFrame) aduce cu sine MapPoint-uri noi, extrase din feature-urile găsite în imaginea respectivă. Funcționarea corectă a urmăririi cadru cu cadru, este determinată de numărul de MapPoint-uri găsite în imaginea curentă în comparatie cu un cadru de referintă. În momentul în care numărul de puncte cheie găsite în cadrul curent scade sub un anumit prag, știm că este necesară salvarea acestuia și creearea unor noi MapPoint-uri pentru a stabiliza urmărirea.

4.6.1 Optimizare hartă locală

Harta locală este alcătuită din KeyFrame-uri și MapPoint-uri. Pentru a optimiza harta trebuie să adăugam noi puncte de tip MapPoint și noi KeyFrame-uri în ea pentru a valida conexiunile care deja exista. Modul în care sunt create și șterse punctele urmează o abordare numită

survival of the fittest. La fiecare nou KeyFrame adăugat, sunt create aproximativ 100 de noi MapPoint-uri și sunt inserate în hartă. Acestea vor fi supuse unui test care o să evalueze cât de ușor sunt observate feature-urile pe care le reprezintă. Din punct de vedere matematic, un MapPoint este observat de un KeyFrame dacă proiecția acestuia în imagine este un vector valid în sistemul de coordonate al KeyFrame-ului respectiv. Cu cât mai multe Keyframe-uri observă același MapPoint, cu atât mai stabil este punctul respectiv din spațiu. Într-un caz ideal, ar trebui ca orice MapPoint creat să fie stabil. De cele mai multe ori nu se întâmplă acest lucru din cauza erorilor de feature matching. Astfel, doar cele mai evidente feature-uri rămân salvate în hartă până la finalul algoritmului. Scopul acestei componente este adăugarea KeyFrame-urilor noi, eliminarea celor redundante și testarea stabilității tuturor MapPoint-urilor create. Pentru a salva Keyframe-ul curent în hartă, următoarele operații trebuie urmate:

1. Folosind harta de adâncime, sunt selectate cele mai apropiate N feature-uri care nu au un MapPoint asociat și au valoarea adâncimii mai mare ca 0. Coordonatele acestor feature-uri sunt proiectate în spaîiu pentru a obîine noi MapPoint-uri.
2. KeyFrame-ul curent este comparat cu alte cadre cheie, pentru a vedea cu cine împarte cele mai multe puncte comune. Keyframe-urile sunt stocate în hartă într-o structură de tip graf neorientat unde nodurile sunt cadrele cheie, iar arcele sunt numărul de MapPoint-uri comune dintre ele.
3. Sunt eliminate punctele cheie redundante sau care au fost observate în prea puține cadre pentru a fi luate în considerare.
4. Se execută un algoritm numit Local Bundle Adjustment în care KeyFrame-urile care au cele mai multe puncte comune cu cadrul curent analizat vor avea matricea de poziție optimizată și coordonatele globale ale MapPoint-urilor asociate acestora.

Local Bundle Adjustment este similar cu Motion Only Bundle Adjustment. În continuare, vorbim de un algoritm iterativ care încearca să minimizeze o funcție de cost, folosind metoda scăderii gradientului. Diferența este că optimizarea se aplică pe mai mult de un cadru cheie: atât matricea pozției, cât și punctele din spațiu asociate (MapPoint-urile) vor fi optimizate. Avem urmatoarele etape:

1. Se creează lista de cadre mobile. Plecând de la cadrul curent, se vor selecta toți vecinii de gradul 1 și 2 din graful neorientat stocat în hartă. Aceste Keyframe-uri sunt considerate $mobile$, deoarece matricea lor de poziție se va modifica.

2. Se creează lista de MapPoint-uri ale căror coordonate vor fi optimizate. Fiecare Keyframe din mulțimea cadrelor mobile observă un număr de puncte în spațiu și toate aceste puncte vor fi folosite de către algoritmul de optimizare.
3. Se creează lista de cadre fixe pentru care matricea de poziție 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 observă acel MapPoint. Dacă un KeyFrame aparține mulțimii de cadre mobile, acesta va fi ignorat, iar dacă nu, va fi adăugat în lista de cadre fixe. Acestea sunt incluse în algoritm pentru a garanta că modificarea coordonatelor unui MapPoint nu va strica asociera (feature, MapPoint) în cadrele care nu vor avea matricea de poziție modificată.

Lucrarea științifică care stă la baza ORB-SLAM2, implementează deja funcția de cost pe care algoritmul de Local Bundle Adjustment o folosește. Pentru a înțelege mai ușor formula matematică, aceasta trebuie privită de la dreapta la stânga. E_{kj} reprezintă eroarea de proiecție a unui MapPoint pe feature-ul asociat. Indicele k aparține KeyFrame-ului, j reprezintă ordinul perechii (feature, MapPoint) pentru care calculăm eroarea. Simbolul $\rho(\cdot)$ este asociat funcției Huber, folosită pentru a ameliora efectele perechilor de tip outlier. X_k este o notație pentru mulțimea tuturor asocierilor (feature, MapPoint) pentru un Keyframe k. Suma erorilor tuturor perechilor este calculată pentru toate cadrele fixe și mobile. Parametrii care vor fi optimizați sunt: coordonatele MapPoint-urilor selectate de către algoritm X_i și matricile de poziție pentru cadrele mobile K_l. Algoritmul optimizează valorile până când ajunge la o valoare de minim sau pentru un număr de iterații.

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

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

4.6.2 Rețeaua Neurală FastDepth

Rețelele Neurale Artificiale sunt o tehnică des întâlnită în Machine Learning pentru a rezolva sarcini complexe pentru care nu există soluții algoritmice clar definite sau implementarea acestora este mult prea costisitoare. Conform[22] și[23], rețelele neurale definesc o funcție nonliniară care găsește o corespondență între un set multivariat de date de intrare x și un set

multivariat de date de ieșire y, modificând un set de parametrii ϕ, $f(x, \phi) = y$. Aceasta este alcătuită dintr-un număr enorm de elemente de procesare care conțin parametrii funcției ϕ, conectate între ele într-o structură de tip graf și dispuse pe straturi. Cele mai importante fiind: stratul de intrare și de ieșire, unde se stabilește forma generală pe care trebuie să o respecte datele care vor parcurge rețeaua și modul în care va arăta rezultatul obținut. Celelalte nivele sunt denumite straturi ascunse. Aceastea fac prelucrarea informației primite de la straturile anterioare și o transmit mai departe. Spunem că o rețea neurală învață din datele primite, dacă își modifică parametrii ϕ astfel încât să reprezinte cu mai multă acuratețe corespondența între datele de intrare x și cele de ieșire y. FastDepth[17] este o arhitectură de rețea neurală folosită pentru a estima adâncimi în imagini. Aceasta primește o imagine de tip RGB a interiorului unei încăperi și returnează o matrice cu valori în intervalul $0m - 10m$, estimând pentru fiecare pixel distanța de la planul de proiecție al imaginii până la punctul din spațiu surprins de fotografie. Scopul nostru este antrenarea unei rețele neurale care să producă o matrice de adâncime cu valori cât mai apropiate de distanța reală la care se află obiectele față de cameră. ORB-SLAM2 folosește o cameră tip RGBD / Stereo care descrie cu foarte mare acuratețe distanța până într-un anumit punct din spațiu, dar creează o matrice rară de valori. Suprafețele lucioase sau cele car nu au putut fi clar observate vor avea adâncimea 0 pentru a arăta că distanța nu a putut fi corect estimată în pixelii respectivi. Un motiv pentru care rețelele neurale sunt o alternativă bună este că ele vor avea o estimare a distanței pentru fiecare pixel din imagine. Rețeaua FastDepth pare să realizeze o pseudosegmentare a zonelor din imagine, identifică conturul obiectelor și atribuie valori ale distanței asemănătoare pentru pixelii ce aparțin aceleași entități. Arhitecturile de mari dimensiuni nu pot fi folosite în timp real fără a utiliza un GPU, dar nu este cazul și pentru arhitectura FastDepth care poate procesa aproximativ 100 de cadre pe secundă. De asemenea, consumă o cantitate redusă de memorie, astfel, parametrii rețelei pot fi stocați într-un fișier ONNX ce ocupă mai puțin de 8 MB, fiind ușor de integrat într-un dispozitiv embedded.

Un posibil dezavantaj al acestei arhitecturi este limitarea distanței maxime estimate la 10 metri, fiind nepotrivit de folosit afară, dar ideal pentru un spațiu închis de mici dimensiuni. Un alt dezavantaj este faptul că valorile aproximate vor avea o acuratețe mai slabă decât cele obținute de camerele Stereo/RGBD.

Există mai multe filozofii când vine vorba de modul în care ar trebui să arate arhitectura rețelelor neurale și operațiile pe care ar trebui să le realizeze fiecare strat. Feed forward neural

network a fost printre primele arhitecturi definite. Elementele de procesare sunt dispuse pe straturi, și fiecare strat primește input-ul de la stratul precedent și transmite output-ul la stratul imediat următor. Informația circulă liniar, de la intrarea în rețea până la finalul acesteia. Abordarea s-a dovedit eficientă în situațiile în care era nevoie de rețele neurale de mici dimensiuni, cu un număr redus de straturi și parametri. În momentul în care creștea complexitatea, abordarea de feed forward neural network devenea greu de antrenat și dădea rezultate mai slabe[22]. Pentru a rezolva această problemă au apărut arhitecturile de tip residual network. Acestea au aplicații în procesarea imaginilor[24], unde datele de intrare au dimensiuni mari și este nevoie de multe nivele pentru a extrage suficiente informații. Principiul de funcționare este utilizarea unor straturi reziduale denumite și skip connections, în care rezultatul unui strat este salvat și transmis ca dată de intrare la un alt nivel decât cel imediat următor. Abordarea aceasta pastrează din informațiile inițiale ale datelor de intrare, în straturile viitoare, stabilizând antrenarea. O altă arhitectură des întâlnită este cea de encoder-decoder, folosită în numeroase aplicații practice în ceea ce privește imaginile: sarcini de colorare a imaginilor alb-negru[25], reconstrucție a imaginilor care conțin părți lipsă și generarea de imagini folosind Variational Auto Encoder[26].

Pe lângă tipurile de arhitecturi propuse, există mai multe categorii de straturi în rețele neurale. Primele folosite erau cele fully connected unde fiecare element de procesare era conectat cu toate celelalte elemente de procesare din stratul următor. Matematic, operația poate fi văzută ca o înmulțire de matrici, o operație costisitoare, iar utilizarea exclusivă a straturilor complet conectate creează o rețea neurală incapabilă să reprezinte funcții nonliniare, scăzând capacitatea de generalizare. De cele mai multe ori, straturile liniare sunt folosite împreună cu funcții de activare nonliniare precum ReLU sau Sigmoid dar în continuare rămâne problema numărului mare de parametri care trebuie antrenați. Din această cauză au fost create straturile convoluționale care folosesc mai puțini parametri și au aplicabilitate în procesarea imaginilor. Principiul teoretic pe care se bazează este reprezentat de faptul că pixelii alăturați în imagine au aceeași semnificație, fiind același feature. Operația de convoluție trebuie realizată pe o zonă a imaginii, iar modificarea parametrilor afectează output-ul generat de mai mulți pixeli. Se stabilește un kernel, o matrice de mici dimensiuni, în FastDepth folosindu-se kernel-uri de (3, 3). Acestea vor stoca parametrii w_{mn} pe care rețeaua neurală îi va antrena pentru stratul convoluțional. În formulă, h_{ij} reprezintă intensitatea pixelului după calculul operației de convoluție, iar x_{ij} este valoarea intensității pixelului de pe coloana i și linia j.

Litera a, reprezintă funcția de activare folosită, iar β este o valoare numerică denumită bias. Acesta poate fi modificat în timpul antrenării și crește capacitatea de generalizare a funcției de convoluție[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 convoluție este în continuare prea costisitor pentru a crea o arhitectura în timp real de mari dimensiuni. Presupunem că avem un vector de intrare pentru un strat de convoluție cu dimensiunile $[d_{in}, h, w]$, unde d_{in} este numărul de canale, h înălțimea și w lățimea vectorului. Kernelul folosit are dimensiunile $[k, k, d_{in}, d_{out}]$, unde d_{out} este numărul de canale rezultate în urma convoluției. În total se vor executa $h \cdot w \cdot d_{in} \cdot d_{out} \cdot k \cdot k$ operații. Pentru a rezolva această problemă a fost creat un strat numit Depthwise Separable Convolutions[27], obținut prin compunerea a 2 straturi de convoluție, unul numit depthwise convolution, iar celălalt pointwise convolution. Această abordare crește viteza de procesare și îmbunătățește acuratețea în sarcini de clasificare pentru seturi de date precum ImageNet ILSVRC2012. În cazul depthwise convolution, fiecare canal al datelor de intrare este procesat de un singur kernel al stratului de convoluție. Pointwise convolution unește printr-o combinație liniară rezultatul procesării fiecărui canal. Complexitatea temporală obținută astfel este de: $h \cdot w \cdot d_{in} \cdot (k^2 + d_{out})$[17].

FastDepth folosește tehnica de skip connections, straturile finale primind ca date de intrare valorile calculate de straturile aflate la început și urmează o arhitectură encoder-decoder. Encoder-ul transformă datele de intrare într-o formă mai compactă asemenea unei operații de arhivare. Prima parte a rețelei este realizată folosind o altă arhitectură numită Mobile_Net[28] și ulterior Mobile_Netv2[29], care reduce numărul de parametri și crește viteza de procesare fără a impacta acuratețea. Decoder-ul este alcătuit din 5 straturi de tip depthwise convolution, fiecare urmat de o interpolare liniară care dublează dimensiunea ieșirii. Ultimul strat face parte din categoria pointwise convolution. Acesta unește canalele obținute și returnează matricea finală de adâncime. Pentru Mobile_Netv2 există parametri preantrenați în librăria Pytorch[30], pe setul de date ImageNet, fiind un motiv în plus de a folosi această arhitectură. Rețeaua neurală folosește atât depthwise convolution,

5 DETALII DE IMPLEMENTARE

5.1 Limbaje de programare și librării folosite

Implementarea este realizată în C++17. Pentru management-ul pachetelor și al codului folosesc CMake 3.28.3. Librariile principale sunt OpenCV[31] 4.9.0, Ceres[32] 2.2.0, Eigen 3.4.0, DBoW2[33] și ultima versiune de Sophus până la data de ianuarie 2025. În comparație cu alte librării care încă mai au parte de modificări, Sophus a intrat într-o etapă de mentenanța, dezvoltarea efectivă a acestuia fiind finalizată din iunie 2024. O primă problemă pe care am întâlnit-o a fost găsirea unei versiuni compatibile de C++ cu toate aceste pachete. Am încercat mai multe variante printre care C++11, C++14, C++17, C++20 și C++23. Preferința mea ar fi fost să folosesc o versiune cât mai nouă. C++11 si C++14 nu erau compatibile cu Ceres, versiunea minimă pentru care funcționează această librărie este C++17. C++20 și C++23 nu erau compatibile cu Sophus și cu Eigen. Utilizarea acestor librării este deosebit de importantă, deoarece implementează metode puternic optimizate pentru a lucra cu matrici. Singura opțiune rămasă este C++17 care este incompatibilă cu DBoW2. Librăria folosește o versiune mai veche a funcției throw pentru erori. În momentul în care am eliminat această directivă, am putut recompila codul ca librărie. Voi prezenta în continuare motivele pentru care am ales fiecare librărie.

OpenCV este folosit în computer vision. Conține implementări ale algoritmilor de extragere de trăsături precum FAST, ORB, SIFT, SURF, are un API pentru procesare video, numeroase funcții de procesare a imaginilor: aplicarea de filtre, transformarea în alb-negru, eliminarea distorsiunii cauzate de cameră. Foarte utile sunt clasele care abstractizează matricile și parametrii prin care se indentifică trăsăturile: cv::Mat și cv::KeyPoint. Structura KeyPoint este fundamentală pentru implementarea algoritmului, deoarece stochează numeroase informații despre zona pe care o reprezintă: orientarea acesteia, coordonatele centrului și nivelul la care a fost observat. Pe langa aceste lucruri, OpenCV are un modul dedicat pentru citirea parametrilor rețelelor neurale din fișierele care urmează un format de tip ONNX, fiind o alternativă potrivită dacă vreau să utilizez un model doar pentru sarcini de inferență.

Ca librărie de optimizare, am avut de ales între Ceres și g2o[34]. În implementarea oficială g2o era cel folosit. Motivul principal este că permite abstractizarea parametrilor care trebuie optimizați și a relațiilor dintre aceștia sub forma unui graf neorientat. API-ul de g2o poate activa și dezactiva un nod al problemei de optimizare, pentru a face implementarea mai robustă împotriva perechilor de tip outlier și pentru a putea reintroduce noduri eliminate temporar din graful de optimizare în timpul rulării problemei. Ceres din păcate nu permite acest lucru. O dată create condițiile inițiale acestea pot fi doar dezactivate și nu mai este permisă reutilizarea lor în aceeași problemă de optimizare. La finalizarea algoritmului, memoria folosită de către noduri este eliberată. Cu toate acestea, Ceres are un API ușor de utilizat și are o viteza comparativă cu cel din g2o.

Folosesc librăria Eigen, deoarece este mai simplu API-ul de calcul cu matrici decât cel din OpenCV. Pentru a accesa elementele unei matrici în OpenCV se folosește o referință la vectorul de date, iar verificarea corectitudinii accesării unui element din vector este făcută la runtime. În cazul matricilor din Eigen, verificarea operațiilor cu matrici este realizată la compile time, prevenind astfel erorile înainte de a rula programul.

Sophus este o librărie care îmi permite să lucrez cu algebra de tip Lie. În loc de a vedea estimările poziției ca pe niște matrici de 4×4, le pot converti într-un vector alcătuit din 7 elemente. Primii 4 parametrii reprezintă un quaternion, aceasta fiind o exprimare vectorială a unei matrici 3×3 de rotație, iar ultimii 3 parametrii alcătuiesc un vector de translatie. Biblioteca implementează operații care îmi permit să lucrez cu acești vectori, ei făcând parte dintr-un grup numit $se(3)$ și garantează că rezultatul obținut este scalat corespunzător pentru a face parte în continuare din aceeași categorie.

DBoW2 este o metodă de tip bag of words pentru compararea imaginilor între ele. Este utilizat pentru operații precum feature matching între imagini consecutive, relocalizări și loop closure. Librăria folosește o structură de tip arbore. Fiecare nivel este obținut din realizarea unui algoritm de clusterizare a descriptorilor de tip ORB, folosind algoritmi precum kmeans++. Această împărțire este realizată în mod recursiv, până când adâncimea arborelui atinge nivelul dorit. Nodurile de tip frunză sunt alcătuite dintr-un singur descriptor. Construirea arborelui este realizată într-o etapă offline de antrenare. În cazul librăriei DBoW2, setul de date folosit a fost Bovisa 2008-09-01. Au fost alese 10K imagini, iar pentru fiecare cadru în parte au fost extrași 1000 de descriptori ORB. Aceștia au fost folosiți pentru a crea un arbore de adâncime 6, iar numărul de clustere pe care le creează fiecare iterație a algoritmului kmeans++ este

de 10, acesta fiind factorul de branching. Pe ultimul strat, există un milion de frunze și tot aceeași lungime o va avea și vectorul de feature-uri care va reprezenta o imagine. Fiecare descriptor va primi o valoare numerică numită greutate, invers proporțională cu frecvența cu care a fost întâlnit descriptorul în setul de date de antrenare. Cu cât este mai rar un anumit descriptor, cu atât este mai util pentru a diferenția o imagine. Scopul principal al librăriei este să primească ca dată de intrare descriptorii ORB ai unei imagini și să calculeze vectorul său bag-of-words. Vectorul bag-of-words este alcătuit în principal din valori de 0. Fiecare descriptor al imaginii parcurge arborele de la radacină spre frunze. Compararea cu un nod, se realizează prin calcularea distanței Hamming dintre descriptor și centroidul clusterului nodului respectiv. Parcurgerea se realizează prin compararea descriptorului cu toate nodurile de pe un anumit nivel și alegerea nodului cu distanța Hamming minimă. Nodul frunză la care va ajunge va avea asociat un index, în cazul de față cu valori de la 0 la un milion. La același index o să fie modificată valoarea din vectorul bag-of-words în valoarea greutății descriptorului stocat în arbore. Apelul de bibliotecă returnează, de asemenea, un vector de feature-uri în 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 reprezintă un vector de descriptori din imaginea curentă care se potrivesc în același cluster. Două imagini pot fi comparate între ele prin intermediul acestui vector de feature-uri, lucru care va fi detaliat în descriera clasei OrbMatcher. În implementarea ORB-SLAM2, nu este practică creearea unui vector de tip bag of words cu un milion de elemente, mai ales că majoritatea valorilor sunt 0. Astfel, reducerea dimensiunii vectorului ar crește viteza de calcul a sistemului. Din această cauză, compararea descriptorilor se realizează doar până la nivelul 4 în arbore. Vectorul bow va avea dimensiunea 1000, iar cel de feature-uri va avea numărul de elemente egal cu cel de descriptori.

5.2 Mediu de lucru și principalele clase

Structura de fișiere este una simplă, în folderul rădăcină se regăsește fișierul de CmakeLists.txt care va fi interpretat de utilitarul cmake pentru a genera automat Makefile-ul. Acest Makefile va conține regulile de build și de clean pentru proiect. În fișierul main.cpp vor fi inițializate componentele și se va putea selecta pe care dintre cele 2 seturi de date se va aplica algoritmul. Aceste seturi de date conțin de fapt cadrele dintr-un video făcut cu o cameră RGBD

Microsoft Kinetic, împreună cu matricile de adâncime și pozițiile acestora. Aceste seturi de date sunt suficient de complexe pentru a permite evaluarea algoritmului ORB-SLAM2. Tot în main.cpp, se va realiza citirea fișierului ORBvoc.txt, acesta conține datele pe care le va folosi clasa ORBVocabulary pentru a inițializa arborele folosit de librăria DBOW2.
Tot în folderul rădăcină se regăsește și fișierul fast_depth.onnx, în care sunt stocate arhitectura și parametrii rețelei neurale FastDepth. În folderul de include, se află antetele claselor pe care le voi implementa și în folderul de src se regăsește codul de C++ pentru logica programului. Am observat că separarea codului în acest fel este o practică des întâlnită în proiectele de mari dimensiuni și garantează flexibilitate în includirea dependințelor între fișiere. Algoritmul ORB-SLAM2 este unul complex, depinzând de o multitudine de parametrii care pot influența acuratețea. Cei mai importanti sunt cei corelați cu camera. În fișierul config.yaml se regăsește matricea K, parametrii de distorsiune ai camerei și alte constante pe care le-am considerat ca fiind niște hiperparametrii ai algoritmului. Aceștia vor trebui modificați în funcție de mediul în care va rula ORB-SLAM2 pentru a garanta funcționarea corectă.

Clasa TumDatasetReader este responsabilă de achiziția de date și de scrierea în fișier a traiectoriei pe care o estimează algoritmul. Achiziția de date presupune citirea din memorie a matricei RGB, convertirea acesteia în format alb-negru pentru o procesare mai rapidă de către algoritmul ORB și obținerea hărții de adâncime pentru cadrul respectiv. Acest lucru poate fi realizat în 2 feluri: matricea de distanțe este citită din setul de date TUM RGBD și este înregistrată cu o cameră RGBD tip Microsoft Kinetic, sau se folosește rețeaua neurală FastDepth, care estimează în timp real distanța pentru fiecare pixel din cadrul curent. Imaginea RGB și harta de adâncime o să fie transmise ca parametri clasei Tracker. TumDatasetReader stochează estimările pozițiilor camerei pentru fiecare cadru în parte. Dacă a fost salvat în clasa Map cadrul curent, va avea matricea de poziție salvată, nealterată în clasa TumDatasetReader. Pentru celelalte cadre, matricea de poziție salvată o să fie relativă la un cadru cheie, de preferat ultimul cadru cheie creat până la citirea imaginii curente. Motivul pentru care se realizează stocarea pozițiilor în acest fel, este faptul că doar cadrele din clasa Map sunt salvate în memorie și pot fi optimizate de către algoritmul Bundle Adjustment, astfel, doar acestea ar trebui să aibă valoarea lor salvată explicit.

Clasa MapPoint este fundamentală pentru buna funcționare a algoritmului ORB-SLAM2. Aceasta este formată cu ajutorul unui KeyPoint și al unui KeyFrame asociat acestuia. În etapa anterioară, am prezentat modul în care se face proiecția coordonatelor unui punct cheie în spațiu, obținându-se coordonatele sale globale. Acestuia i se asociază descriptorul keypoint-ului care l-a creat, pentru compararea ulterioară cu alte KeyPoint-uri din alte imagini. Un MapPoint are nevoie de un vector de orientare pentru a putea verifica dacă un MapPoint este vizibil dintr-un KeyFrame. Pentru a calcula acest vector de orientare, prima dată se determină coordonatele globale ale centrului camerei pentru cadrul cheie care a creat acel KeyPoint. Acest lucru se realizează în 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 diferenței între coordonatele globale ale centrului camerei și ale MapPoint-ului creează vectorul de orientare. Acesta poate suferi ulterior modificări dacă se constată că mai multe cadre observă același punct. În situația respectivă, vectorul de orientare final va fi media aritmetică a celorlalți vectori de orientare individuali.

Clasa Feature nu există în implementarea oficială a ORB-SLAM, dar am considerat că utilizarea acesteia ar simplifica codul. Aceasta extinde clasa KeyPoint fără a o moșteni explicit, are asociată distanța extrasă din matricea de adâncime, descriptorul și o valoare de tip boolean care arată dacă punctul este monocular sau stereo. Această clasificare se obține prin compararea adâncimii cu o valoare foarte apropiată de 0. În cazul implementării mele, dacă distanța este mai mică decât $1e-2$, consider că valoarea estimată de camera RGBD ori de rețeaua neurală nu este corectă și că punctul respectiv este monocular, altfel, consider că este stereo. Pentru fiecare KeyPoint extras, se va crea o instanță a clasei Feature care va fi stocată direct în KeyFrame. Fiecare Feature are setat pe null la inițializare o referință la un obiect de tip MapPoint. Pentru a garanta funcționarea în timp real a algoritmilor, asocierea (Feature, MapPoint) trebuie să poată fi accesată în $O(1)$. Vectorul de elemente Feature, împreună cu o structură de tip dicționar unde cheia va fi un MapPoint și valoarea un Feature, vor face acest lucru posibil. Singura problemă este că cele 2 structuri încearcă să reprezinte aceleași corelații. În cazul vectorului am indicele unui Feature drept cheie și încerc să accesez MapPoint-ul asociat, iar în cazul dicționarului, am referința unui MapPoint și încerc să obțin adresa unui Feature. Ambele structuri trebuie să conțină aceleași perechi, altfel comporta-

mentul algoritmului devine nedefinit.

Clasa KeyFrame, conține mai multe elemente legate de cadrul curent. Pentru a menține funcționarea sistemului în timp real, trebuie să stocăm în memorie rezultatele calculelor noastre. În această clasă se vor regăsi matricea de adâncime, vectorul de instanțe ale clasei Feature, vectorul de trăsături calculat de metoda bag-of-words și cadrul inițial în format alb-negru. În interiorul constructorului acestei clase sunt mai multe operații realizate, majoritatea fiind necesare pentru a crește eficiența accesării datelor. De exemplu, vectorul de tip Feature în medie conține 1000 de elemente care nu sunt sortate. În situația în care proiectăm un MapPoint în plan, ar trebui să comparăm coordonatele proiecției cu poziția fiecărui Feature în parte pentru a stabili care este cel mai apropiat. O modalitate de a rezolva acest lucru este segmentarea suprafeței în K zone, în cazul meu am ales $K = 100$, fiecare reprezentând o porțiune din imaginea inițială, având asociate referințele Feature-urilor care se găsesc pe suprafața respectivă. În acest fel, în funcție de zona în care este proiectat un MapPoint, vom știi ce Feature are o posibilitate mare de a corespunde, reducând astfel numărul de comparații.

Considerând că toate valorile de tip Feature sunt dispuse în mod egal pe suprafața imagini, atunci complexitatea devine $O(N/K)$, unde N reprezintă numărul de Feature-uri, iar K numărul de zone în care a fost împărțită imaginea. Constructorul este responsabil de inițializarea structurilor de tip Feature, partiționarea lor în funcție de coordonatele în imagine, și de memorarea estimării curente a poziție camerei și a centrului camerei în coordonate globale. Tot în această clasă se regăsește structura de tip dicționar (MapPoint, Feature), care va fi modificată pe tot parcursul algoritmului. Altă metodă importantă este: $get\_vector\_keypoints\_after\_reprojection$. Aceasta primește ca date de intrare coordonatele proiecției unui MapPoint, valoarea ferestrei de proiecție, și octava minimă și maximă. Octavele reprezintă nivelul la care a fost observat un Keypoint în imagine și creează estimare grosieră a distanței dintre cameră și punctul din spațiu observat. Acesta poate să aibă valori de la 0 la 7 inclusiv și ne spune de câte ori s-a făcut resize la imagine pentru a surprinde o anumită trăsătură. De exemplu, dacă un Keypoint are valoarea octavei 0, înseamnă că algoritmul a detectat-o în imaginea nemodificată. Dacă ar fi 1, atunci dimensiunea imagini a fost redusă o singură dată cu 0.8 din valoarea inițială și așa mai departe. Feature-ul care are asociat un MapPoint, trebuie să aibă valori ale octavei apropiate între ele. Dacă această

condiție nu s-ar respecta, ar introduce erori de estimare a distanței. Ne așteptăm ca Feature-uri corespondente, să fie aproximate la aceeași distanță. Altfel, ar putea înseamna că cele 2 puncte din spațiu sunt diferite. Dacă mediul are o structură simetrică, de exemplu o sală de clasă cu băncile aliniate una în fața celeilalte, algoritmul ar putea observa 2 colțuri ce aparțin de 2 mese diferite. Dacă nu ar avea această separare pe baza octavei, următorul cadru care observă aceleași mese ar putea să asocieze eronat punctele între ele, afectând estimarea poziției. Fereastra de proiecție reprezintă cât de departe poate să fie Feature-ul de coordonatele punctului de proiecție ale unui MapPoint pentru a fi considerată corectă o asociere între cele 2 elemente. În funcție de dimensiunea ferestrei aceasta poate intersecta 1, 2 sau 4 subsecțiuni din cele 100 în care este împărțită imaginea. Problema cea mai mare pe care am avut-o cu clasele KeyFrame și MapPoint era dependența circulară. MapPoint-urile aveau nevoie de un KeyFrame și un Feature pentru a fi create și trebuia menținută o lista a KeyFrame-urilor care observă MapPoint-ul respectiv. În cazul KeyFrame-ului, acesta trebuie să păstreze referințe despre toate punctele de tip MapPoint pe care le observă. Pentru a rezolva această problemă am folosit o clasă adițională care face operații cu cele 2 structuri și am folosit forward declaration.

Clasa Map implementează harta pe care o folosește algoritmul ORB-SLAM2. Aceasta este responsabilă de stocarea corectă a KeyFrame-urilor, a MapPoint-urilor și rezolvă problema dependenței circulare a celor 2 clase. Aici am implementate metodele de adăugare/ștergere a unui MapPoint dintr-un KeyFrame. De asemenea, clasa MapPoint conține referințe la toate KeyFrame-urile care o observă. Aceste referințe sunt adăugate/șterse de către 2 metode care se regăsesc aici. Clasa Map creează o structură de tip graf ponderat neorientat, în care nodurile sunt reprezentate de KeyFrame-uri. Arcele arată dacă există mai mult de 15 puncte comune între 2 KeyFrame-uri, iar ponderea lor este determinată de numărul de MapPoint-uri comune. Clasa Map realizează operații pe graful de KeyFrame-uri, adaugă/șterge noduri și face interogări pentru a afla vecinii direcți sau pe cei pe nivel 2. Am ales să implementez această structura folosind $std :: unordered\_map$. Cheia va fi KeyFrame-ul curent, iar valoarea returnată va fi un alt $std :: unordered\_map$. Acesta va conține toate celelalte KeyFrame-uri cu care este direct conectată cheia, dar și ponderea conexiunii. În acest fel accesarea vecinilor de ordinul 1 va fi o operație ce se poate realiza în timp constant. Funcția $track\_local\_map$ este folosită de către clasa Tracking. Aceasta primește ca date de intrare

cadrul curent și ultimul cadru cheie salvat. Nu returnează nimic, doar încearcă să găsească asocieri (Feature, MapPoint) pentru Feature-urile care încă nu au fost corelate cu un punct din spațiu. Această operație este costisitoare și funcționează în felul următor:

1. sunt căutate toate KeyFrame-urile vecine de gradul 1 și 2 cu ultimul KeyFrame adăugat
2. din aceste KeyFrame-uri sunt extrase toate MapPoint-urile observate de către cadrele cheie
3. MapPoint-urile sunt proiectate în cadrul curent și sunt căutate potriviri pentru Feature-urile care încă nu au MapPoint-uri asociate.

Pentru a nu fi necesar să calculam de fiecare dată KeyFrame-urile vecine și harta locală de MapPoint-uri, le stochez ca variabile în interiorul clasei Map. Acestea vor fi modificate în momentul în care un KeyFrame este adăugat în hartă. Într-un caz ideal, ar trebui ca pentru fiecare Feature adăugat să se găsească un MapPoint, dar acest lucru rareori se întâmplă. În situația în care s-au găsit mai putin de 30 de puncte din spațiu care s-au proiectat corect în imagine, se consideră că a aparut o eroare de urmărire și algoritmul începe o etapă de relocalizare.

Clasa OrbMatcher este responsabilă de realizarea urmăririi feature-urilor asemănătoare între cadre consecutive. Înainte de a începe prezentarea metodelor implementate, voi descrie pipeline-ul de procesare al unui punct din spațiu pentru a fi considerat observabil de către cameră. Avem o instanță a obiectului MapPoint mp, dacă una dintre operațiile prezentate nu funcționează, punctul respectiv este ignorat de către KeyFrame-ul curent.

1. Se proiectează coordonatele globale ale mp în planul imaginii folosind matricea de estimare a poziției T_{cw} și matricea parametrilor camerei K. Se verifică dacă coordonatele proiecției sunt valide pentru imagine.
2. Se calculează distanța d de la centrul camerei la mp. În funcție de valoarea octavei stocată în acest MapPoint, se poate estima o limită minimă și maximă pentru d. Dacă valoarea obținută nu se încadrează în acest interval se consideră ca punctul este invalid.
3. Cu ajutorul geometriei analitice se obține ecuația dreptei care unește mp și centrulul camerei. Această dreaptă și vectorul de direcție al MapPoint-ului trebuie să creeze un unghi cu o valoare mai mare de 60 de grade pentru a fi considerat mp observabil.

Dacă aceste 3 verificări au fost realizate cu succes, se consideră că punctul poate fi observat

de către cameră. Există 2 funcții responsabile de asocierile între imagini. Scopul acestora este să găsească corespondențe între Feature-urile din cadrul curent și MapPoint-urile din spațiu. Pentru a se găsi perechea Feature f și MapPoint mp, trebuie mp să se proiecteze în vecinătatea lui f, iar descriptorii asociați atât Feature-ului, cât și MapPoint-ului, să aibă distanța Hamming sub un prag prestabilit. O metodă ar fi compararea tuturor Feature-urilor din spațiu, cu totalitatea MapPoint-urilor observate de cadrul curent. Dar această metodă ar fi ineficientă și ar duce la estimări eronate. Separarea Feature-urilor în clustere în funcție de distanța Hamming a descriptorilor este o abordare stabilă dar lentă și preferabil de utilizat când nu ne putem baza pe estimarea matricei de poziție a cadrului curent. Cealaltă abordare este clusterizarea în funcție de coordonatele din imagine ale Feature-ului.
Funcția $match\_frame\_reference\_frame$ implementează prima metodă. Aceasta primește ca parametrii 2 vectori de feature-uri calculați de librăria DBoW2: unul asociat cadrului curent, pentru care estimăm matricea de poziție și celălalt asociat cadrului anterior și perechilor de tip (Feature, MapPoint). Elementele acestor vectori sunt de tip $(int, vector\_descriptori)$. Dacă 2 astfel de perechi au prima valoare egală, înseamnă că cei 2 vectori de descriptori fac parte din același cluster, conform arborelui din librăria DBOW2. Fiecare descriptor are asociată o instanță a clasei Feature. În cadrul anterior, instanța poate avea sau nu, un MapPoint corespondent. Dacă există, acel MapPoint se poate proiecta în imagine. Descriptorul intern al MapPoint-ului este comparat cu ceilalți descriptori care fac parte din clusterul cadrului curent. Se aplică testul de proporționalitate Lowe, pentru a garanta că descriptorul cu distanța Hamming minimă este cel mai bun. Funcția $match\_consecutive\_frames$ este mai simplă și implementează a doua metodă. MapPoint-ul este proiectat în imagine și toate Feature-urile aflate în apropiere devin posibili candidați pentru a crea o asociere (Feature, MapPoint). Se calculează distanța Hamming între descriptorul MapPoint-ului și cel al Feature-ului. Trăsătura din imagine cu distanța minimă și mai mică decât un prag setat la 100, este considerată ca fiind cea mai potrivită. Feature-ul asociat acelui descriptor, vă păstra o referință a MapPoint-ului.

Clasa MotionOnlyBA implementează în Ceres algoritmul Motion Only Bundle Adjustment, primește ca date de intrare KeyFrame-ul curent și returnează matricea de poziție optimizată. Librăria lucrează cu o noțiune din C++ numită functori. Acestea sunt clase/structuri pentru care s-a făcut overload la operatorul $()$. Clasa BundleError se află în aceeași categorie și

implementează funcția de eroare obținută din proiectarea unui MapPoint și asocierea acestuia cu un Feature. Pentru a crea problema de optimizare, clasa ceres::Problem trebuie să știe care parametrii trebuie optimizați și funcția de eroare pe care trebuie să o minimizeze. În cazul acestui algoritm, singurul lucru care va fi modificat este matricea de poziție a KeyFrame-ului pe care o voi converti în forma $se(3)$, transformând-o într-un vector de 7 elemente. Pentru funcția de eroare, nu voi scrie explicit că este suma erorilor de proiecție. În schimb, voi inițializa pentru fiecare asociere de tip (Feature, MapPoint) câte un element al clasei BundleError. Algoritmul de optimizare implementat de librăria Ceres, va încerca în mod independent să reducă valoarea erorii pentru fiecare pereche în parte, modificând vectorul de poziție. Schimbarea modului în care este exprimată poziția camerei nu este aleatorie. Matricea de poziție conține 2 componente: matricea de rotație R și un vector de translație t. Pentru t nu există restricții de modificare atâta timp cât rezultatul final nu creează modificări mari între pozițiile a două cadre consecutive. Oricum ar varia parametrii acestui vector, în continuare semnificația lui de vector de translație rămâne neschimbată. În această situație putem spune că parametrii sunt alterați de către librăria Ceres folosind $EuclidianManifold$, mici modificări bazate pe calcularea derivatelor parțiale ale variabilelor din funcția de eroare definită în BundleError, asemănător modului în care sunt modificați parametrii în rețele neurale. Pentru matricea de rotație R nu se mai poate aplica aceeași logică. Aceasta trebuie să facă parte din structura de tip grup numit $SO(3)$, adică să respecte egalitatea $R * R^t = R^t * R = I$ și trebuie să reprezinte o rotație reală pe cele 3 axe. Alterarea aleatorie a parametrilor ar duce la o matrice invalidă. Din aceasta cauză, modificarea rotației trebuie făcută cu un anumit unghi, iar acest lucru se poate realiza printr-o înmulțire de 2 matrici de rotație valide. Din păcate, nu există implementare în formă matriceala pentru schimbarea unghiului de rotație, dar este pentru Quaternioni. Din această cauză realizez conversia din matrice de poziție în vector din categoria $se(3)$, iar pentru primi 4 parametrii asociați rotației, optimizarea lor se realizează folosind $QuaternionManifold$. Această abordare rezolvă problema instabilității numerice și garantează că rezultatul operației de optimizare este un element valid în $se(3)$, ce poate fi ulterior convertit în forma matriceală. În funcție de categoria din care face parte Feature-ul, acesta este considerat monocular sau stereo. Funcția de eroare implementată de clasa BundleError este identică pentru ambele, cu excepția faptului că pentru punctele stereo, este verificată și distanța la care se află punctul față de valoarea la care a fost estimat de camera RGBD. Pentru a preveni instabilitatea cauzată de punctele de tip outlier, funcția

Huber descrisă în capitolul anterior este folosită în calcularea finală a erorii de proiecție. În implementarea oficială realizată în g2o, agloritmul de optimizare este rulat de 4 ori și după fiecare execuție sunt eliminate punctele de tip outlier. Experimental, am observat că etapa de optimizare cadru cu cadru este cea mai costisitoare operație pe care o realizează algoritmul de ORB-SLAM2. Execuția acesteia de 4 ori nu crește semnificativ acuratețea și reduce viteza de prelucrarea la aproximativ 5 cadre pe secundă, făcându-l nepotrivit pentru un sistem în timp real. Am observat că obțin rezultate foarte bune, rulând o singură dată Motion Only Bundle Adjustment, urmat apoi de o etapă de eliminare a corelațiilor (Feature, MapPoint) de tip outlier. Dacă mai puțin de 3 asocieri rămân, se consideră că algoritmul a acumulat prea multe erori în urmărirea cadru cu cadru și trece într-o etapă de relocalizare.

Clasa Tracker realizează urmărirea traiectoriei cadru cu cadru și integrează fiecare dintre componentele definite anterior. Este responsabilă de captarea cadrului curent, transformarea acestuia în KeyFrame și luarea deciziei daca vă fi salvat în Map pentru a completa harta mediului înconjurător. Pașii următori se execută pentru fiecare cadru în parte:

1. Se creează KeyFrame-ul curent.
2. Se estimează matricea de poziție pe baza legii de mișcare.
3. Se realizează asocierea între Feature-urile (puncte 2D) din cadru curent și MapPoint-urile observate de cadru anterior (puncte 3D)
4. Pe baza asocierilor realizate anterior, se optimizează matricea de poziție a KeyFrame-ului curent și sunt eliminate asocierile de tip outlier
5. Este proiectată harta locală pe cadrul curent, și se găsesc noi asocieri (Feature, MapPoint). Se execută, din nou, aceeași operație de optimizare Motion Only Bundle Adjustment.
6. Este evaluat KeyFrame-ul curent, se verifică dacă trebuie salvat în clasa Map.

Cadrul curent și matricea de adâncime sunt citite de TumDatasetReader. În imaginea RGB se folosește ORB pentru a extrage un vector de KeyPoint-uri și un vector de descriptori. Acestea sunt folosite pentru a inițializa un obiect de tip KeyFrame. Se folosește o versiune modificată pentru algoritmul ORB, implementată în clasa ORBextractor. Aceasta este concepută să extragă aproximativ 1000 de puncte cheie astfel încât să fie distribuite cât mai egal pe suprafața imaginii. Dacă un număr foarte mare de keypoint-uri s-ar obține din aceeași zona, acuratețea

estimării ar avea de suferit, pixelii din zonele aflate mai aproape de cameră se mișcă cu o viteza mai mare decât cei aflați în depărtare. Dacă am considera doar punctele dintr-o anumită zonă în realizarea estimării, am obține variații în mișcare eronate. Parametrii setați pentru algoritmul ORB sunt următorii: 1000 de feature-uri, factorul de scalare al imaginii este 1.2, există maxim 8 nivele și algoritmul FAST care face extragerea inițială de KeyPoint-uri alege o anumită zonă doar dacă diferența de intensitate între pixeli are cel puțin valoarea 20. Când zona aleasă este slab texturată, atunci poate să seteze această diferență la 7, pentru a garanta că vor fi găsite feature-uri în mod uniform, pe suprafața imaginii. Clasa Tracker păstrează 4 referințe de tip KeyFrame: cadrul curent care este analizat, 2 cadre imediat anterioare care vor fi folosite la estimarea poziției și ultimul cadru referință care a fost creat. Cadrul referință este ultimul KeyFrame adăugat în Map și indică la nivel aproximativ, în ce zonă se află camera și care MapPoint-uri ar trebui să fie vizibile. Cadrele mai vechi care nu au fost salvate în Map au fost șterse pentru a reduce cantitatea de memorie folosită. Pentru noul KeyFrame creat, urmează etapa de estimare a poziției curente, aceasta se face pe baza legii de mișcare, iar valorile matricii vor fi calculate folosindu-ne de cele 2 cadre imediat anterioare salvate în Tracker. Important aici de observat este faptul că pentru primul cadru citit, poziția acestuia este matricea identitate 4*4, acest lucru sugerând că dispozitivul care înregistrează mediul consideră că primul KeyFrame este chiar originea sistemului de coordonate, iar toate matricile de poziție viitoare sunt de fapt transformări relative față de origine. Primul KeyFrame va fi salvat întotdeauna în clasa Map și este utilizat pentru a inițializa primele puncte de tip MapPoint. Pentru toate Feature-urile de tip stereo din imagine, se vor creea puncte în spatiu. Din cauza acestui mod de inițializare, ORB-SLAM2 este sensibil până la apariția următorului cadru cheie, estimările făcute de acesta în prima etapă fiind predispuse la erori. Uneori algoritmul își pierde orientarea, fiind necesară o etapă de relocalizarea sau de reluare a execuției acestuia. ORB-SLAM3 implementează o metodă mult mai robustă de inițializare, generând mai multe hărți locale în situația în care urmărirea cadru cu cadru eșuează și le unește între ele în momentul în care recunoaște o zonă pe care a vizitat-o deja. După ce a fost creat KeyFrame-ul și a fost făcută estimarea inițială a poziției, clasa OrbMatcher este folosită pentru a găsi corelații între Feature-uri și MapPoint-uri. Alegerea metodei care îndeplinește această sarcină este determinată de numărul de KeyFrame-uri create de la ultima relocalizare sau de la adăugarea unui nou cadru cheie în Map. Dacă nu se vor găsi minim 15 asocieri de tip (Feature, MapPoint), se va considera că algoritmul și-a pierdut orientarea. Altfel,

etapa de matching a funcționat iar perechile găsite vor fi utlizate de către MotionOnlyBA pentru a realiza optimizarea poziției. Perechile de tip outlier vor fi eliminate și noua poziție a KeyFrame-ului va fi returnată. Dacă vor rămâne mai puțin de 3 asocieri se va considera, din nou, că algoritmul și-a pierdut orientarea. În final, se folosește clasa Map pentru a proiecta toate punctele din harta locală pe cadrul curent, iar asocierile găsite vor trece din nou printr-un proces de optimizare. Dacă nu se găsesc minim 50 de perechi (Feature, MapPoint), înseamnă că urmărirea cadrului curent a eșuat. Altfel se trece la etapa următoare și se va decide dacă noul KeyFrame va fi stocat în Map. Acest lucru se va întâmpla dacă următoarele condiții vor avea loc simultan.

1. au trecut mai mult de 30 de cadre de la ultimul KeyFrame adăugat în Map
2. numărul de MapPoint-uri în cadrul curent este 25% din numărul urmărit de cadrul de referință
3. cadrul curent are cel puțin 70 de Feature-uri de tip stereo, cu distanța estimată de către matricea de adâncime ca fiind mai mică de 3.2 metri și urmărește cel puțin 100 de MapPoint-uri

Clasa LocalMapping este responsabilă de optimizarea hărții algoritmului. Aceasta șterge/adaugă KeyFrame-uri și MapPoint-uri, iar la fiecare cadru cheie nou, realizează operația de Local Bundle Adjustment. Această metodă optimizează matricile de poziție și toate MapPoint-urile vecinilor direcți și cei de categoria a doua pentru KeyFrame-ul abia adăugat. În momentul în care thread-ul de Tracking consideră că un nou cadru cheie trebuie adăugat în hartă, se execută metoda principală $local\_map$. Aceasta îndeplinește următoarele operații:

1. Creează noi MapPoint-uri din primele 100 de Feature-uri de tip stereo, sortate în ordine crescătoare după distanța la care se află acestea de centrul camerei
2. Adaugă cadrul curent în graful de KeyFrame-uri stabilind vecinii direcți ai acestuia
3. Noile MapPoint-uri create sunt adăugate într-o lista numită $recently\_added$. Pentru a ieși din această listă, punctele trebuie să treacă un test care dovedește că nu sunt rezultatul unui Feature eronat detectat de către algoritmul ORB, și că pot fi folosite cu încredere
4. Se execută operația de $culling$, punctele sunt verificate dacă sunt valide, iar dacă nu, memoria lor este eliberată.

5. Se folosește operația de triangulare pentru a crea noi MapPoint-uri din Feature-urile care se potrivesc între ele și fac parte din cadre cheie diferite.
6. Se detectează entitățile de tip MapPoint care reprezintă același punct din spațiu, iar una dintre referințe este ștearsă pentru a garanta că nu există puncte duplicate. Acest pas reduce dimensiunea hărții, îmbunătățește urmărirea cadru cu cadru și întărește conexiunile deja existente între KeyFrame-urile adiacente.
7. Se execută operația de KeyFrame culling, se verifică dacă informațiile pe care le deține un KeyFrame, adică totalitatea valorilor de tip MapPoint asociate lui, sunt observate și din alte cadre. Dacă peste 90% din punctele observate de un anumit cadru sunt vizibile și din alte părți, KeyFrame-ul analizat este considerat redundant și memoria lui este eliberată. Acest lucru garantează că structura de graf a clasei Map, conține doar cadre esențiale pentru reprezentarea norului de puncte.

În a patra etapă, se execută operația de $culling$. Aceasta elimină punctele care nu sunt de încredere. Singurele puncte care nu vor trece prin această etapă de verificare sunt cele generate de primul KeyFrame. Tot primul KeyFrame nu poate fi șters, deoarece ar da peste cap sistemul de coordonate local pe care îl folosește ORB-SLAM2. Un punct este considerat de încredere dacă din momentul în care a fost creat, el a fost proiectat cu succes în 3 cadre cheie consecutive și dacă a fost observat în cel puțin 25% din numărul total de cadre care au trecut de la creearea acestuia. Ambele condiții trebuie să fie respectate simultan în momentul în care se face verificarea punctului respectiv. Politica pe care o urmează familia de algoritmi ORB-SLAM este să genereze multe puncte, fară a impune restricții, pe care apoi le vă supune acestui test de relevanță.

Ultima clasă este cea de MapDrawer pe care o folosesc pentru a afișa norul de MapPoint-uri, cadrul curent analizat și pozițiile cadrelor cheie observate. Folosesc librăria Pangolin și OpenGL pentru desenarea fiecărei structuri, camera urmărește cadrul curent. Interfața grafică scade viteza de procesare a cadrelor, dar este o modalitate eficientă de a înțelege vizual ce se petrece în algoritm. Implementarea pentru interfața grafică am realizat-o spre final, când aveam celelalte componente finalizate, lucru care a îngreunat procesul de dezvoltare, deoarece lucram cu valori numerice în terminal. Acum dacă aș reîncepe implementarea, interfața grafică ar fi printre primele lucruri pe care le-aș realiza. Datorită acestei clase am reușit să găsesc erori

în modul de construcție al grafului ponderat din clasa Map și al modului în care proiectam punctele în spatiu.

5.3 Pipeline antrenare FastDepth

Pentru rețeaua Neurală FastDepth pipeline-ul de antrenare a fost scris folosind librăria Pytorch, iar pentru operațiile de preprocesare folosesc librăria Albumentations. Setul de date pe care am făcut antrenarea se numește Nyu Depthv2 Dataset[35] și l-am obținut de pe Kaggle. Rezultatul acestui pipeline trebuie să fie un fișier de tip ONNX cu valorile parametrilor rețelei FastDepth în urma antrenării. O problemă pe care am observat-o la setul de date este că pentru imaginile de antrenament, adâncimile sunt exprimate ca fiind în intervalul $[0, 255]$, pe când în setul de date de validare, acestea se află între $[0, 10000]$ reprezentând valorile în milimetri ale distanțelor. O limitare a acestui set de date este că nu poate detecta distanțe mai mari de 10 metri. Considerând că algoritmul trebuie să funcționeze pentru încăperi de mici dimensiuni, această limitare nu ar trebui să reprezinte o problemă. Pentru antrenare am ales să urmez lucrarea științifică[3] și am setat hiperparametrii:

- Optimizatorul folosit a fost implementarea din Pytorch pentru Stochastic Gradient Descent, torch.SGD, având un learning rate de $1e{-}3$, o valoare a momentumului de $\beta = 0.9$ și $weight\_decay = 1e - 4$.
- Antrenarea s-a realizat pentru 50 de epoci, iar durata antrenării a fost de aproximativ 6 ore jumătate. Laptopul pe care am antrenat este un Asus TUF Gaming A15, având un procesor AMD Ryzen 7 cu o frecvență de 4.2 GHz și placă video NVIDIA GeForce RTX 2060, cu o memorie de 6GB.@
- Imaginile în setul de date au o dimensiune de $(3, 460, 640)$. Pentru a crește viteza de procesare am modificat dimensiunile la $(3, 256, 320)$ și am aplicat o funcție de normalizare de tip min_max. Ambele transformări sunt aplicate atât pe setul de date de antrenare, cât și pe cel de test.
- Un batch de date are dimensiunea de 8.
- În lucrarea FastDepth, funcția de pierdere folosită este L1Loss, aceasta fiind suma diferențelor dintre valoarea reală și cea determinată de rețeaua neurală în modul. Am ales să folosesc o funcție de pierdere mai robustă conform acestei lucrări științifice[36].

Acuratețea a fost verificată prin compararea diferenței relative între valorile obținute prin inferență și cele reale cu un factor $RELATIVE\_ERROR = 0.15$. Această operație a fost realizată pentru fiecare pixel în parte, iar acuratețea reprezintă procentul de pixeli cu o valoare care se încadrează în limita impusă de RELATIVE_ERROR. Pentru a preveni antrenarea pentru intervale lungi fără a obține rezultate, am avut 2 metode pe care le-am implementat: o strategie de early stopping și o strategie pentru modificarea learning rate-ului în timpul antrenării. În situația în care valoarea acurateții nu ar fi crescut pentru 5 epoci, antrenarea ar fi fost oprită, iar în situația în care nu creștea pentru 3 epoci, valoarea ratei de învățare să fie redusă la 0.3 din valoarea inițială. În practică am observat că rețeaua converge aproape monoton către o valoare optimă. Funcția de pierdere primește ca date de intrare matricea de adâncime obținută de către rețeaua neurală și matricea cu valori reale din setul de date, denumită groundtruth, și returnează o valoare numerică de tip double care exprimă cât de departe se află estimarea noastră de realitate. Ideea antrenării unei rețele neurale este minimizarea acestei valori. Funcția de eroare este alcătuită dintr-o combinație liniară a 3 componente diferite[36]: L1Loss, GradientEdgeLoss și Structural Similarity Loss, formula matematică este:

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

Structural Similarity Loss se asigură că media și distribuția standard pe care o urmează valorile estimate, se apropie de media și distribuția standard a matricei groundtruth. În comparație cu celelalte 2 componente ale funcției de pierdere care sunt aplicate la nivel de pixel, aceasta abstractizează rezultatele ca fiind 2 distribuții Normale cu parametrii $\mathcal{N}(\mu, \sigma^2)$ care trebuie să se suprapună.

Principiul de funcționare pentru GradientEdgeLoss este că pixeli din regiuni apropiate trebuie să aibă cam aceleași valori de estimare ale distanței și că diferența între pixeli adiacenți pe axele x și y, ar trebui să fie identică cu cea din imaginea groundtruth. Aceasta poate fi scrisă în felul următor, unde N reprezintă numărul de pixeli din imagine, iar derivata valorilor pixelilor în raport cu axa de coordonate reprezintă diferența între matricea imaginii inițiale și aceeași matrice având un rând shiftat la dreapta pentru axa X notată $\frac{\partial I}{\partial x}$ și un rând shiftat vertical pentru axa Y notată $\frac{\partial I}{\partial y}$.

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

6 EVALUARE

6.1 Setul de date TUM RGBD Dataset

Setul de date utilizat pentru a realiza evaluarea se numește TUM RGBD Dataset[2]. Acesta conține numeroase subseturi, fiecare verificând un aspect diferit al implementării, ajutând la crearea unei imagini de ansamblu cu privire la robustețea algoritmului în funcție de mediul în care se lucrează și de traiectoria pe care o urmează. Cele 2 subseturi pe care le-am considerat potrivite pentru implementare sunt:

- Subsetul rgbd_dataset_freiburg1_xyz conține cadrele unui video de 35 de secunde, în care traiectoria este în principal alcătuită din translații. Există foarte puține rotații, fiind ideal pentru a verifica dacă estimarea poziției în spațiu este corect realizată.
- Subsetul rgbd_dataset_freiburg1_rpy are 27 de secunde și conține foarte puține translații. Există în schimb numeroase schimbări bruște de rotație care reduc acuratețea imaginii captate, testând la maxim capacitatea algoritmului ORB de a extrage feature-uri. Sistemul își schimbă 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 operațiile de rotație, mai ales atunci când camera își schimbă orientarea către o zonă necunoscută. Pentru a crea harta zonei respective sunt generate numeroase KeyFrame-uri și MapPoint-uri, pe care algoritmul trebuie să le filtreze în clasa de LocalMapping, lucru care crește complexitatea temporală și spațială și scade acuratețea sistemului.

Videourile sunt realizate cu ajutorul unei camere RGBD Microsoft Kinect, având frecvența de 30 de cadre pe secundă. Setul de date conține imaginile de tip RGB, hărțile de adâncime pentru fiecare cadru în parte, vectorii de poziție în forma $se(3)$, primii 3 parametri fiind poziția în spațiu (tx, ty, tz), iar următorii 4 parametri sunt asociați matricei de rotație, scrisă sub forma de Quaternion, (qw, qx, qy, qz) și timestamp-urile asociate momentului în care au fost

înregistrate fiecare din valorile din setul de date. Cu ajutorul acestor timestamp-uri putem crea asocieri de tip (imagine RGB, matrice de adâncime, poziție) pe care le putem transmite algoritmului ORB-SLAM2. Poziția este considerată ca fiind valoarea ideală, groundtruth și va fi comparată cu rezultatele obținute. Clasa TumDatasetReader este responsabilă de citirea datelor și stocarea matricilor de poziție obținute pentru fiecare cadru. După parcurgerea întregului set de date, valorile estimate sunt salvate într-un fișier de tip text unde vor fi comparate cu cele reale.

6.2 Metrici utilizate

Algoritmul ORB-SLAM2 scrie într-un fișier estimările matricilor de poziție pentru fiecare cadru în parte. Pentru a realiza comparația cu valorile de tip groundtruth din setul de date, folosesc un pachet din python numit evo. Acesta este capabil să creeze un grafic al traiectoriei, permițând astfel o reprezentare vizuală a rezultatelor pentru viteză, translație și orientare. De exemplu, figura de mai jos reprezintă variația translației pe fiecare dintre cele 3 axe. Cu albastru este valoarea de tip groundtruth, iar cu galben este estimarea realizată de implementarea mea pentru algoritmul ORB-SLAM. Rezultatele sunt obținute pentru subsetul de date rgbd_dataset_freiburg1_xyz, acesta fiind special conceput pentru a testa corectitudinea estimării translației între cadre. Consider că o reprezentare grafică în care traiectoria groundtruth

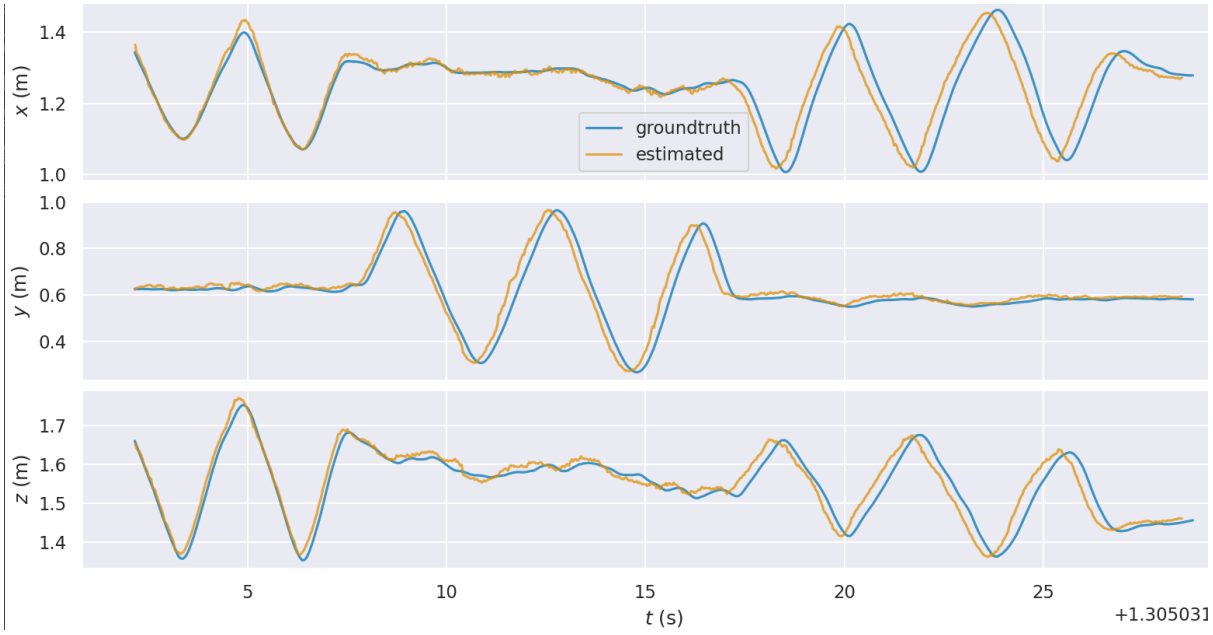

Figura 3: Graficul translației pe fiecare din cele 3 axe, groundtruth și estimare ORB-SLAM2

se suprapune exact cu ceea ce a obținut estimarea ORB-SLAM2 poate fi considerată, în mod neoficial, o metrică pe baza căreia putem spune dacă algoritmul funcționează corect. Pentru exactitate se poate folosi: APE (Absolute Pose Error). Această metrică măsoară distanța euclidiană dintre pozițiile estimate și cele reale, la fiecare moment de timp. În general valorile scorurilor APE obținute pentru ambele seturi de date sunt până în 0.05, sugerând că acuratețea este bună. Alte metrici sunt: numărul de secunde necesar pentru parcurgerea setului de date sau numărul de cadre pe secundă. Adăugarea unui nou cadru cheie este necesară atunci când clasa Tracker nu mai poate urmări traiectoria corect. Putem considera numărul de KeyFrame-uri ca fiind o metrică pentru estimarea stabilitatii sistemului. O altă metrică este legată de numărul de relocalizări pe care a trebuit să le facă algoritmul pentru a parcurge setul de date. Relocalizarea apare în situația în care urmărirea cadru cu cadru eșuează și este căutat KeyFrame-ul care seamănă cel mai bine cu cadrul curent folosind vectorul de feature-uri calculat de metoda bag-of-words. Ideal, numărul necesar de relocalizări ar trebui să fie 0. În cazul rulării implementării mele pe setul de date rgbd_dataset_freiburg1_xyz, acesta durează în medie 67 de secunde, funcționând la aproximativ 15 cadre pe secundă. În medie este nevoie între 5-7 Keyframe-uri noi pentru parcurgerea setului de date. Logica de relocalizare nu este deloc folosită, algoritmul fiind capabil să realizeze urmărirea cadru cu cadru. Pe setul de date rgbd_dataset_freiburg1_rpy a durat 73 de secunde, videoul având 27 de secunde, iar viteza de procesare a fost de aproximativ 10-11 cadre pe secunde. Acest lucru se datorează numeroaselor operații de optimizare a hărții pe care trebuie să le facă algoritmul, deoarece sunt adăugate între 17-19 KeyFrame-uri pentru a parcurge întreg setul de date, din cauza mișcărilor bruste ale camerei care reduc considerabil claritatea imaginilor extrase. În continuare, numărul de relocalizări este 0. Mai jos, am atașat graficul care compară estimarea orientării pentru fiecare cadru, estimările fiind descompuse după cele 3 dimensiuni ale rotației. Graficul realizat de algoritmul ORB-SLAM2 pare să fie shiftat în timp față de cel real, dar să aibă aproximativ aceeași formă cu cel al valorilor groundtruth. Consider că problema poate să pornească de la modul în care sunt atașate timestamp-urile pentru fiecare cadru în parte, lucru care nu are legătură directă cu modul în care este realizată implementarea, ci cu modul în care setul de date creează perechile (imagine RGB, vector poziție, matrice de adâncime). Am încercat utilizarea rețelei neurale FastDepth pentru a estima adâncimea în loc de a folosi matricea de distanțe a setului de date TUM RGBD. Pentru a testa arhitectura o să utilizez doar imaginea și vectorul de poziție, simulând astfel o situație în care sistemul ar avea doar o

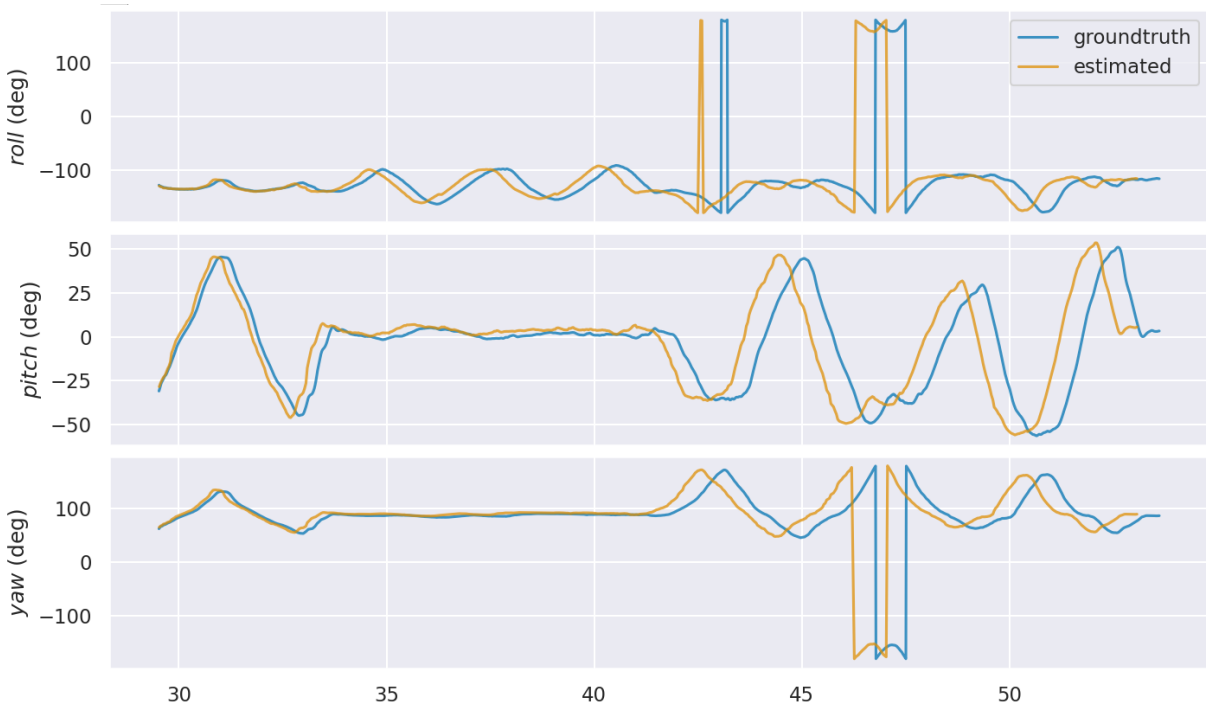

Figura 4: Graficul orientării pentru setul de date rgbd_dataset_freiburg1_rpy

cameră RGB. Problema cu această implementare este că rețeaua neurală nu este suficient de exactă, iar estimările distanțelor între cadre consecutive în continuare variază mult. Pentru subsetul de date rgbd_dataset_freiburg1_xyz, implementarea intră în etapa de relocalizare în medie după primele 50-60 de cadre procesate. Sistemul este mult prea instabil pentru a realiza în mod corect urmărirea cadru cu cadru.

În etapele inițiale ale dezvoltării algoritmului am încercat utilizarea unui video realizat folosind camera telefonului pentru testarea implementării. Au fost 3 probleme pe care le-am întâlnit. Prima problemă este faptul că nu puteam determina parametrii corecți ai camerei telefonului. Aveam nevoie de distanța focală, de coordonatele centrului imaginii și de parametrii de distorsiune. A doua problemă o reprezenta lipsa unei matrici de adâncime pentru fiecare cadru, iar cea de-a treia, era lipsa unui vector de poziție pentru fiecare imagine. Chiar și în situația în care obțineam parametrii camerei telefonului folosind algoritmi implementați în OpenCV, în continuare nu puteam fi sigur dacă estimările realizate de mine sunt cele corecte. Din cauza acestor multe probleme, am ajuns la concluzia că un set de date ar fi o variantă mai potrivită.

7 CONCLUZII

O problemă pe care am avut-o în testarea algoritmului a fost că nu am putut face funcțională implementarea inițială a ORB-SLAM2. Există conflicte între versiunile de biblioteci Eigen și g2o. Versiunea de Eigen folosită la momentul respectiv nu mai există acum în repository-ul oficial de Github.

Librăria Ceres este ușor de folosit pentru problemele de optimizare non-liniare, astfel, consider că cel mai mare plus pe care îl aduce lucrarea mea este faptul că am cea mai completă implementare a algoritmului Bundle Adjustment în această librărie, la care se adaugă o logică de filtrare a punctelor de tip outlier. Aceasta are caz separat de folosire atât pentru punctele monoculare, cât și pentru cele stereo. În plus, am adus optimizări la codul oficial pentru ORB-SLAM2 legate de faptul că implementarea lor nu eliberează absolut deloc memoria pentru MapPoint-urile și KeyFrame-urile considerate invalide. Eu am rezolvat această problemă, extinzând durata pentru care poate rula algoritmul și făcându-l potrivit pentru sistemele embedded cu o memorie redusă. De asemenea, în implementarea oficială nu este folosită nicio structură de tip dicționar. Clasele principale folosite au parametri de stare care își modifică valoarea la fiecare cadru, funcțiile având efecte laterale care generează erori greu de urmărit și corectat. În total am scris 4030 de linii de cod în C++ și am modificat codul pentru numeroase funcții, codebase-ul meu fiind de aproximativ 3 ori mai mic decât cel al implementării oficiale. ORB-SLAM2 îndeplinește 3 funcții: urmărirea cadru cu cadru, corectarea erorilor, și închiderea buclelor. Etapa de închidere a buclelor nu am reușit să o implementez din cauza apropierii termenului de predare, cu toate acestea, algoritmul obține în continuare rezultate bune pentru subseturile de date alese.

În ciuda faptului că utilizarea unei rețele neurale pentru a înlocui o cameră RGBD nu a funcționat așa cum am crezut inițial, algoritmul devine instabil și își pierde complet orientarea după primele 50 de cadre, consider că o rețea neurală precum FastDepth poate fi folosită în estimarea adâncimii pentru punctele care au avut atribuită distanța 0 de către camera tip RGBD. O direcție viitoare ar fi utilizarea unui sistem care combină cele 2 abordări. Separarea straturilor convolutionale în depthwise și pointwise s-a dovedit a fi o tehnică bună pentru a

crește viteza arhitecturii astfel încât să poată fi folosită în timp real.

Un lucru pe care îl regret este că nu am utilizat o interfață grafica încă de la primele etape pentru a detecta erorile în timpul dezvoltării algoritmului. Deși scorul calculat de APE este o metrică bună pentru a vedea cât de bine se potrivesc 2 estimări ale poziției unui cadru, o simplă valoare numerică nu este suficientă pentru a avea o privire generală asupra modului în care funcționează implementarea.

Ca direcții viitoare, aș dori să utilizez arhitecturi de rețele neurale pentru sarcini bine delimitate, cum ar fi extragerea de feature-uri sau găsirea de corelații de tip (Feature, MapPoint). Adăugarea unui modul de object detection și a unui algoritm de planificare de trasee, pentru a îndeplini sarcini simple de găsire a unor obiecte de mici dimensiuni. Până acum algoritmul a folosit doar metode clasice, ORB pentru extragere de feature-uri, Brute Force pentru feature matching, bag-of-words pentru relocalizare. Astfel, acum consider că direcția pe care ar trebui să o urmeze această clasă de algoritmi de tip SLAM este una în care tehnicile de Machine Learning sunt folosite pentru a crește viteza și poate acuratețea operațiilor. Aceasta fiind și direcția încurajată de lucrările ce fac parte din state of the art până 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.