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

SINOPSIS

LALALALALALLA Sinopsisul proiectului are rol de introducere, conținând atât o descriere pe scurt a problemei abordate cât și o enumerare sumară a rezultatelor și a concluziilor. Se recomandă ca sinopsisul să fie redactat într-un limbaj accesibil unei persoane nefamiliarizate cu domeniul, dar în același timp destul de specific pentru a oferi rapid o vedere de ansamblu asupra proiectului prezentat. Sinopsisul proiectului va fi redactat atât în română cât și în engleză. Ca dimensiunea recomandată aceasta secțiune va avea maxim 200 de cuvinte pentru fiecare variantă. Împreună, ambele variante se vor încadra într-o singură pagină.

ABSTRACT

The abstract has an introductory role and should engulf both a brief description of the issue at hand, as well as an overview of the obtained results and conclusions. The abstract should be formulated such that even somebody that is unfamiliar with the projects' domain can grasp the objectives of the thesis while, at the same time, retaining a specificity level offering a bird's eye view of the project. The projects' abstract will be elaborated in both Romanian and English. The recommended size for this section is limited to 200 words for each version. Together, both versions will fit in one page.

1 SOLUTIE PROPUSA

Solutia mea presupune implementarea algoritmului ORB-SLAM2. Acesta are 2 scopuri fundamentale:

- sa estimeze pentru fiecare cadru in parte matricea de pozitie si orientare a camerei, reconstruind astfel traseul parcurs in timpul functionarii algoritmului
- sa creeze o harta locala a mediului inconjurator pentru a memora zonele prin care a mai trecut si pentru a imbunatatii estimarea traiectoriei

Matricea de pozitie si orientare a camerei (pose matrix) are dimensiuni 4x4 si are formatul prezentat mai jos, unde R reprezinta matricea de rotatie 3x3, iar t este vectorul coloana de dimensiune 3, reprezentand translatia fata de punctul de origine $(0, 0, 0)$. Aceasta mai este denumita si matricea de conversie din sistemul de coordonate global (world space) in sistemul de coordonate al camerei (camera space) si este notata in implementarea mea ca T_{cw}. Inversa acestei matrice notata T_{wc} realizeaza operatia de conversie dintre cele 2 sisteme de coordonate in sens opus.

$$T_{cw} = \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} \tag{1}$$

Algoritmul primeste ca date de intrare: sursa de la care va obtine imaginile de tip RGB pe care va trebui sa le prelucreze, acestea pot sa provina atat de la un video, set de date consacrat sau chiar in timp real direct de la camera, parametrii de distorsiune a imaginii si matricea parametrilor interni ai camerei, avand dimensiunea 3x3 si notata in mod traditional cu K. Aceasta contine 4 constante importante: distanta focala a camerei pe axa x si pe y f_x, f_y si c_x, c_y reprezentand coordonatele centrului imaginii. Aceasta matrice trebuie modificata de fiecare data cand este schimbata camera cu care se realizeaza filmarea, sau cand se fac operatii de modificare a dimensiunii imaginilor fata de modul in care ar fi acestea extrase natural. Matricea are urmatoarea forma:

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

Algoritmul va returna un fisier text in care se vor afla estimarile matricilor de pozitie impreuna cu timestamp-ul asociat pentru fiecare cadru in parte in ordine cronologica. Rezultatul poate fi comparat cu fisiere care contin valorile reale si care respecta acelasi format pentru a verifica corectitudinea algoritmului. Diagrama UML prezinta interactiunea dintre principale componente descrise din punct de vedere functional, dar si flow-ul natural al algoritmului. In continuare voi detalia logica fiecarei componente din punct de vedere al algoritmilor folositi, ale valorilor de intrare si de iesire ale acestora.

1.1 Achizitia datelor

Scopul acestei componente este sa citeasca imaginea de tip RGB de la camera, sa extraga matricea de adancime asociata cadrului curent, de exemplu: prin intermediul unei camere stereo, a unei camere de tip RGBD sau cu ajutorul unei retele neurale si sa creeze o estimare initiala pentru pozitia curenta a camerei pe baza masuratorilor anterioare. In viitor, o alta functie a acestei componente ar putea fi extragerea datelor de la instrumente de masura precum giroscop sau accelerometru, pentru a obtine informatii suplimentare cu privire la orientarea si distanta efectuata de catre camera care ar putea imbunatatii considerabil estimarea initiala a pozitiei.

1.2 Extragere Feature-uri folosind ORB

Ca date de intrare aceasta componenta primeste doar imaginea de tip RGB, si extrage aproximativ 1000 de puncte cheie, denumite in engleza keypoints, care vor fi ulterior folosite pentru a gasi asocieri intre cadrele consecutive. Daca algoritmul de extragere functioneaza corect iar traiectoria camerei este una stabila, fara schimbari bruste ale directiei de deplasare, feature-uri similare ar trebui sa fie observate in ambele imagini. Asocierile dintre ele, ne pot da informatii despre modul in care s-a deplasat camera intre cele 2 cadre. Folosesc termenul de feature-uri si keypoints ca sinonime deoarece se refera la acelasi concept. Un keypoint reprezinta o zona

circulara de interes din imagine de dimensiuni reduse, dar cel mai adesea se lucreaza cu centrul acestei zone, fiind definit ca un punct de coordonate (x, y). In implementarea mea, diametrul unui keypoint are 20 de pixeli. In cazul algoritmului ORB, aceste zone se afla la frontiera obiectelor care alcatuiesc imaginea. De exemplu, daca poza surprinde un televizor, algoritmul ORB va gasi multe puncte cheie la marginea ecranului si foarte putine pe suprafata ecranului deoarece nu exista variatii in intensitatea luminoasa. Pe de alta parte, o camera complet mobilata ar fi o zona puternic texturata, iar un astfel de algoritm s-ar descurca sa gaseasca suficiente feature-uri. Fiecare dintre aceste keypoint-uri, are asociat un vector de valori numit descriptor, acesta este folosit pentru a compara gradul de similaritate intre 2 keypoint-uri gasite in imagini diferite. Algoritmul pe care il folosesc se numeste Oriented Fast and Rotated Brief (ORB). Acesta a fost creat in anul 2011 ca alternativa pentru alti algoritmi de extragere de feature-uri precum SIFT si SURFMotivul pentru care acesta a ajuns atat de popular se datoreaza mai multor factori:

- Este mult mai rapid decat SIFT si SURF fiind mult mai potrivit pentru sisteme realtime si pentru dispozitive embedded.
- la momentul realizarii lucrarii stiintifice despre ORB-SLAM2 atat SIFT cat si SURF se aflau sub protectia drepturilor de autor, ORB nu avea vreo astfel de restrictie
- ORB se descurca foarte bine a regasi aceleasi feature-uri in imagine indiferent de modul in care aceasta este redimensionata. Acest lucru este deosebit de util in situatiile in care camera se deplaseaza in fata sau in spate, fiindca va vedea in continuare aceleasi feature-uri pe tot parcursul miscarii, permitandu-ne sa deducem directia de deplasare doar analizand mai multe cadre consecutive

1.3 Harta punctelor din spatiu

Unul dintre scopurile fundamentale ale algoritmului de ORB-SLAM2, pe langa cel de estimare al traseului camerei este cel de creare a hartii locale a mediului inconjurator. Problema este ca, in comparatie cu versiuni mai avansate ale acestui algoritm, special modificate pentru o reconstructie cat mai fidela a mediului, algoritmul nostru trebuie sa functioneze pentru un sistem embedded care nu are capacitate de procesare suficient de mare, fiind nevoit astfel sa simuleze mediul printr-un nor de puncte cu o densitate redusa (sparse). Cele 2 sarcini sunt

dependente una de cealalta, fiecare element din norul de puncte actioneaza ca o referinta, o caracteristica a mediului care ar trebui sa fie observata de fiecare data cand punctul se afla in frustum-ul camerei. De exemplu: presupunem ca avem o imagine in care este observata in totalitate o masa in interiorul unei incaperi. ORB va identifica aproape instantaneu feature-urile (colturile mesei) si teoretic, indiferent de modul in care ne-am roti in jurul mesei, aceleasi feature-uri ar trebui sa fie observate de fiecare data, mai mult de atat, considerand ca mediul este static, acestea sunt mereu asociate cu acelasi punct din spatiu, devenind astfel o referinta pe baza careia putem estima modul in care s-ar deplasa camera. In literatura de specialitate aceste puncte din spatiu sunt denumite MapPoint-uri iar functionalitatea corecta a algoritmului depinde strict de modul in care aceste MapPoint-uri sunt observate cadru cu cadru. Un astfel de punct in spatiu este creat dintr-un keypoint, dar nu vom avea nevoie de toate punctele din regiunea respectiva si vom considera ca centrul este punctul cel mai semnificativ, avand coordonate x si y, si distanta fata de camera fiind estimata ca fiind d. Mai mult ne vom folosi de matricea transformarii din coordonatele camerei in coordonatele globale si de parametrii interni ai camerei f_x, f_y distanta focala, si c_x, c_y coordonatele centrului imaginii. Vectorul coloana cu 3 dimensiuni reprezinta pozitia in spatiu a feature-ului gasit in cadrul curent pe care il analizam, astfel am creat primul MapPoint. Ca alternativa, pentru a nu lucra cu matrici de dimensiuni 4x4 putem folosi R_{wc} reprezentand matricea de rotatie si t_{wc} vectorul de translatie.

$$\begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = T_{wc} * \begin{bmatrix} \frac{x-c_x}{f_x} * d \\ \frac{y-c_y}{f_y} * d \\ d \\ 1 \end{bmatrix}, \quad \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = R_{wc} * \begin{bmatrix} \frac{x-c_x}{f_x} * d \\ \frac{y-c_y}{f_y} * d \\ d \end{bmatrix} + t_{wc} \tag{3}$$

In literatura de specialitate MapPoint-urile sunt considerate ca fiind niste ancore (landmark) pozitionate dinamic de catre algoritm, acestea sunt asociate cu un anumit cadru cheie si ne vor ajuta in optimizarea matricei de pozitie dar si pentru sarcina de relocalizare si de memorare a zonelor cunoscute.

1.4 Asociere puncte din spatiu cu feature-uri ORB

Ca date de intrare avem feature-urile si descriptorii extrasi din imagine, matricea de adancime si harta de MapPoint-uri. Scopul acestei componente este sa gaseasca cat mai multe asocieri de 1:1 intre feature-uri si MapPoint-uri. Intr-un caz ideal fiecare feature gasit ar trebui sa aiba asociat un MapPoint, dar in realitate nu se poate intampla acest lucru din 2 motive: imperfectiuni ale algoritmului ORB de detectie ale feature-urilor: acesta nu garanteaza ca acelasi feature va fi gasit de fiecare data pentru cadre consecutive si faptul ca modelul isi schimba orientarea, facand ca MapPoint-urile aflate la limita campului vizual al camerei sa nu mai poata fi observate. Un MapPoint este un feature al unui cadru anterior, proiectat in spatiu. In final, aceasta componenta realizeaza tot o comparare de feature-uri intre cadrul curent, si multiple cadre anterioare. Aceasta operatie de comparare se realizeaza prin intermediul distantei Hamming dintre descriptori, cu cat valoarea obtinuta este mai mica, cu atat cele 2 feature-uri sunt mai asemanatoare. Exista mai multe tipuri de algoritmi folositi pentru feature matching, dar cel folosit in implementarea curenta este Brute Force Feature Matching optimizat. Acest algoritm primeste ca date de intrare 2 seturi de feature-uri si incearca sa gaseasca asocieri intre ele. Asocierele sunt facute cu ajutorul descriptorilor, se calculeaza distanta Hamming iar daca valoarea obtinuta este minima, perechea respectiva de feature-uri se considera ca a fost corect asociata, pentru ORB-SLAM2 acest lucru reprezinta ca am gasit exact acelasi punct din spatiu, in 2 imagini diferite. Daca N este numarul de feature-uri din primul set, M numarul de feature-uri din al doilea set si D fiind dimensiunea descriptorului, in cazul nostru fiind 32, complexitatea algoritmului devine $O(N * M * D)$. Facand-ul un algoritm destul de costisitor de folosit pentru un sistem in timp real, mai mult de atat, este predispus la erori, compararea feature-urilor nu tine cont de locatia acestora in imagine, obtinandu-se astfel asocieri care matematic par corecte, dar ele nu au sens din punct de vedere logic. Pentru a rezolva aceasta problema si a reduce complexitatea temporala se stabileste o fereastra patrata de lungime prestabilita in jurul punctului de proiectie unde se pot cauta feature-uri. In final se obtin asocierile intre feature-uri, sunt cautate MapPoint-urile corespunzatoare feature-urilor din cadrele anterioare si altfel se obtin asocierele (feature, MapPoint) de care are nevoie algoritmul.

1.5 Optimizare Estimare Pozitie Initiala

Aceasta componenta primeste ca data de intrare estimarea pozitiei curente a camerei T_{cw}, si o asociere bijectiva intre feature-urile gasite in imagine si punctele care exista la momentul respectiv in spatiu. Ca date de iesire vom avea doar matricea pozitiei curente a camerei optimizata. Daca asocierile intre feature-uri si MapPoint-uri sunt perfecte, ar trebui ca proiectia punctului din spatiu pe imagine sa se suprapuna pe centrul keypoint-ului. Rareori se petrece acest lucru in practica, iar distanta dintre proiectia unui MapPoint si coordonotale centrului feature-ului reprezinta eroarea de asociere. Pentru a minimiza aceasta eroare, exista 2 optimizari care se pot face: prima este modificarea valorilor matricei de pozitiei, iar cea de-a doua este modificarea coordonatelor din spatiu ale MapPoint-ului. Inainte de a prezenta algoritmul de optimizare folosit, voi arata modul in care se proiecteaza un MapPoint in plan.

1.5.1 Proiectarea MapPoint in planul imaginii

Aceasta operatie de proiectie poate fi vazuta ca aplicarea unui functii $\pi(\cdot)$ ce primeste ca date de intrare coordonatele globale ale punctului, iar ca rezultat va returna coordonatele omogene in planul imaginii. Aceasta transformare se petrece in 2 etape:

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

In prima etapa putem folosi coordonatele omogene, pentru a face conversia in mod direct. Alternativ, putem extrage din matricea de pozitie T_{cw} atat matricea de rotatie R_{cw} cat si vectorul coloana de translatie t_{cw}.

$$\mathbf{X}_{camera} = \mathbf{T}_{cw} \cdot \begin{bmatrix} \mathbf{X}_w \\ 1 \end{bmatrix}, \quad \mathbf{T}_{cw} = \begin{bmatrix} \mathbf{R_{cw}} & \mathbf{t_{cw}} \\ \mathbf{0}^T & 1 \end{bmatrix}, \quad \mathbf{X}_{camera} = \mathbf{R_{cw}} \cdot \mathbf{X_w} + \mathbf{t_{cw}} \qquad (4)$$

Matricea T_{cw} este utilizata atat pentru a descrie pozitia si orientarea in spatiu cat si pentru a schimba din sistemul de coordonate global in cel al camerei. In sistemul de coordonate global, un punct se afla la exact aceeasi valoare indiferent de pozitia camerei care il priveste, in sistemul de coordonate al camerei, pozitia unui MapPoint o sa difere de fiecare data. In

etapa a doua MapPoint-ul respectiv este in sistemul de referinta al camerei, coordonatele fiind reprezentate prin vectorul coloana X_{camera}. Vom considera a 3-a valoare a acestui vector Z_c. Aceasta reprezinta distanta dintre planul camerei si punctul pe care il analizam. Z_c ne spune daca punctul respectiv poate fi observat in imagine. Daca valoarea Z_c este mai mica sau egala cu 0, inseamna ca punctul se proiecteaza in spatele camerei, facandu-l invalid. In situatia in care Z_c este mai mare decat 0, vom realiza conversia in coordonatele omogene ale imaginii cu ajutorul urmatoarei formule, u fiind asociat axei x si v fiind asociat axei y. Daca valorile u si˘ au valori mai mari ca 0, si mai mici decat dimensiunea imaginii. Vectorul coloana $[u, v, 1]$ este rezultatul cautat.

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \mathbf{K} \cdot \begin{bmatrix} \frac{X_c}{Z_c} \\ \frac{Y_c}{Z_c} \\ 1 \end{bmatrix}, \quad \mathbf{K} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{5}$$

1.5.2 Motion Only Bundle Adjustment

Algoritmul folosit in aceasta etapa se numeste Motion Only Bundle Adjustment. Acesta modifica doar matricea pozitiei curente a camerei. Coordonatele punctelor din spatiu sunt considerate ca fiind constante. Algoritmul este unul iterativ, minimizand o functie de cost. Forma generala a functiei de cost este suma erorilor de proiectie pentru toate perechile (feature, MapPoint). Iar formula generala este aceasta.

$$\mathbf{R}_{cw}, \mathbf{t_{cw}} = \min_{\mathbf{R}_{cw}, t_{cw}} \sum_{i=1}^{N} \rho \left(\|\mathbf{x}_i - K \cdot (\mathbf{R}_{cw} \cdot \mathbf{X}_i + \mathbf{t_{cw}})\|^2 \right) \tag{6}$$

In aceasta formula, x_i reprezinta coordonatele omogene feature-ului in coordonatele imagininii, iar X_i reprezinta coordonatele globale ale MapPoint-ului pentru care calculam eroarea de proiectie. Simbolul $\rho(\cdot)$ reprezinta functia Huber pentru scalarea valorilor de eroare. Daca o asociere intre un feature si un MapPoint nu este potrivita, diferenta dintre centrul feature-ului si proiectia MapPoint-ului este mai mare decat un prag prestabilit. Aceasta diferenta, lasata nemodificata, ar destabiliza algoritmul. Iar o astfel de problema este usor de observat, daca modificarea matricei de pozitie duce la variatii enorme a orientarii sau a translatiei intre 2 cadre consecutive, atunci cel mai probabil asocierile intre feature-uri si MapPoint-uri aveau valori eronate, termenul de $outlier$ fiind folosit in aceasta situatie. Functia Huber Loss reduce

valoarea acestor outlier-ere permitandu-le in acelasi timp sa faca parte din algoritmul de optimizare. In acest fel algoritmul devine mai robust si capabil ajunga la o valoare optima in mai putine iteratii. Mai jos este prezentata formula matematica a functiei Huber Loss, unde δ reprezinta un numar real pozitiv, toleranta a erorii de proiectie.

$$\rho(s) = \begin{cases} \frac{1}{2}s^2 & \text{if } |s| \leq \delta \\ \delta(|s| - \frac{1}{2}\delta) & \text{if } |s| > \delta \end{cases} \tag{7}$$

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

1.6 Crearea unui cadru cheie

Aceasta componenta primeste ca date de intrare absolut toate informatiile procesate de pana acum pentru cadrul curent: imaginea de tip rgb, matricea de adancime, punctele cheie, descriptorii, asocierile (feature, MapPoint) si estimarea matricii pozitiei curente a camerei. Toate aceste data impreuna vor alcatui un cadru cheie care va fi salvat in memorie, pentru termen scurt sau lung. Exista 2 motive principale pentru care dorim sa facem aceasta operatie In primul rand ne ajuta sa estimam pozitia urmatorului cadru bazandu-ne pe legea inertiei. Consider ca o data inceputa deplasarea camerei intr-o anumita directie, este foarte probabil ca aceea miscare sa fie mentinuta si la urmatorul cadru. Fie T_{cw} matricea de pozitie pentru cadrul la care vrem sa estimam deplasarea, iar T_{cw1}, T_{cw2} matricile de pozitie a celor 2 cadre imediat predecesoare. Formula de estimare a pozitiei curente este:

$$\mathbf{T_{cw}} = \mathbf{T_{cw1}} \cdot \left(\mathbf{T_{cw2}}^{-1} \cdot \mathbf{T_{cw1}}\right) \tag{8}$$

Al doilea motiv pentru care avem nevoie de cadre cheie este recreearea mediului. Incercam sa salvam numarul minim de cadre necesare pentru a reproduce harta de MapPoint-uri a mediului inconjurator. Un cadru cheie nou (KeyFrame) aduce cu sine MapPoint-uri noi, extrase din feature-urile gasite in imaginea respectiva. Functionarea corecta a urmarii cadru cu cadru, este determinata de numarul de MapPoint-uri gasite in imaginea curenta in comparatie cu un cadru

de referinta. In momentul in care numarul de puncte cheie gasite in imaginea curenta scade sub un anumit prag, stim ca este necesar un nou cadru cheie care: sa stabilizeze urmarirea, sa introduca noi puncte cheie, si sa ajute la optimizarea intregii harti a mediului.

1.6.1 Optimizare harta locala

Harta locala este alcatuita din KeyFrame-uri si MapPoint-uri, avand intre ele o relatie de many-to-many. Se considera ca un KeyFrame si un MapPoint sunt conectate intre ele daca un MapPoint dat poate sa fie observat dintr-un KeyFrame. Matematic se traduce ca proiectia punctului respectiv pe imaginea stocata in KeyFrame poate fi asociat cu un feature valid. Avem acelasi exemplu cu masa dintr-o incapere. Mai multe cadre consecutive observa acelasi colt al piesei de mobilier. Acel feature are asociat un MapPoint, ceeea ce inseamna ca MapPoint-ul respectiv este observat din mai multe KeyFrame-uri. Cu cat mai multe Keyframe-uri observa acelasi MapPoint, cu atat mai stabil este punctul respectiv din spatiu. Intr-un caz ideal, ar trebui ca orice MapPoint creat sa fie stabil. De cele mai multe ori nu se intampla acest lucru din cauza erorilor de feature matching. Astfel, doar cele mai evidente feature-uri raman salvate in harta pana la finalul algoritmului. Pentru a salva Keyframe-ul curent in harta trebuie sa se petreaca in ordine urmatoarele operatii:

1. Exista feature-uri in cadrul curent care nu au fost asociate cu un MapPoint. Folosindune de harta de adancime si de coordonatele fiecarui punct cheie din imagine selectam cele mai apropiate n astfel de puncte si le proiectam in spatiu pentru a obtine noi MapPoint-uri.
2. KeyFrame-ul curent este comparat cu alte cadre cheie, pentru a vedea cu cine imparte cele mai multe puncte comune. Keyframe-urile sunt stocate in harta intr-o structura de tip graf neorientat unde nodurile sunt cadrele cheie iar arcele sunt numarul de MapPoint-uri comune dintre ele.
3. sunt eliminate punctele cheie redundante sau care au fost observate in prea putine cadre pentru a fi luate in considerare.
4. se executa un algoritm numit Bundle Adjustment pentru a optimiza atat pozitiile punctelor din spatiu dar si matricea de pozitie a cadrelor cheie

Bundle Adjustment este similar cu Motion Only Bundle Adjustment. In continuare vorbim de

un algoritm iterativ care incearca sa minimizeze o functie de cost, folosind metoda scaderii gradientului. Diferenta in aceasta situatie este ca Bundle Adjustment se aplica pe mai mult de un cadru si atat matricea poztiei si punctele din spatiu (MapPoint-urile) vor fi optimizate. Avem urmatoarele etape:

1. Se creeaza lista de cadre mobile. Plecand de la cadrul curent, se vor selecta toti vecinii de gradul 1 si 2 din graful neorientat stocat in harta. Aceste Keyframe-uri sunt considerate ca fiind $mobile$ deoarece matricea lor de pozitie se va modifica.
2. Se creeaza lista de MapPoint-uri care vor fi optimizate. Fiecare Keyframe din multimea cadrelor mobile observa un numar de puncte in spatiu, toate aceste puncte vor fi folosite de catre algoritmul de optimizare.
3. Se creeaza lista de cadre fixe, pentru acestea, matricea de pozitie nu se va modifica. Avand lista de MapPoint-uri ce vor fi optimizate, pentru fiecare punct din spatiu vom itera prin lista de KeyFrame-uri care observa acel MapPoint. Daca un KeyFrame apartine multimii de cadre mobile in vom ignora, iar daca nu, vom considera ca face parte din lista de cadre fixe. Acestea sunt incluse in algoritm pentru a garanta ca modificarea coordonatelor MapPoint-ului nu va strica asociera (feature, MapPoint) in cadrele care nu vor avea matricea de pozitie modificata.

Lucrarea stiintifica care sta la baza ORB-SLAM2, implementeaza deja functia de cost pe care algoritmul de Bundle Adjustment o foloseste. Pentru a intelege mai usor formula matematica, aceasta trebuie privita de la dreapta la stanga. E_{kj} reprezinta eroarea de proiectie a MapPoint-ului pe feature-ul asociat. Indicele k este asociat KeyFrame-ului, iar j este indicele perechii (feature, MapPoint) pentru care calculam eroarea. Simbolul $\rho(\cdot)$ este asociat functiei Huber, folosita pentru a ameliora efectele perechilor de tip outlier. X_k reprezinta multimea tuturor asocierilor (feature, MapPoint) pentru un Keyframe k. Aceasta suma de erori este calculata pentru fiecare cadru, atat cele fixe cat si cele mobile. Parametrii care trebuiesc optimizati sunt: coordonatele MapPoint-urilor selectate de catre algoritm X_i cat si matricile de pozitie pentru cadrele mobile. Algoritmul returneaza noile valori ale parametrilor care trebuie optimizati.

$$\{\mathbf{X}_i, \mathbf{R}_l, \mathbf{t}_l \mid i \in \mathcal{P}_L, l \in \mathcal{K}_L\} = \arg\min_{\mathbf{X}^i, \mathbf{R}_l, \mathbf{t}_l} \sum_{k \in \mathcal{K}_L \cup \mathcal{K}_F} \sum_{j \in \mathcal{X}_k} \rho(E_{kj}) \tag{9}$$

$$E_{kj} = \|\mathbf{x}_j - K \cdot (\mathbf{R}_k \mathbf{X}_j + \mathbf{t}_k)\| \tag{10}$$

1.6.2 Reteaua Neurala FastDepth

Retele Neurale Artificiale sunt o tehnica des intalnita in Machine Learning pentru a rezolva sarcini complexe pentru care nu exista solutii algoritmice clar definite sau implementarea acestora este mult prea costisitoare. O definitie simplificata este ca retele neurale definesc o functie nonliniara care gaseste o corespondenta intre un set multivariat de date de intrare x si un set multivariat de date de iesire y fara a le modifica $f(x, \phi) = y$. Aceasta este alcatuita dintr-un numar enorm de elemente de procesare care contin parametrii functiei ϕ, conectate intre ele intr-o structura de tip graf si dispuse pe straturi. Cele mai importante fiind: stratul de intrare si de iesire, unde se stabileste forma generala pe care trebuie o sa respecte datele care vor parcurge reteaua si modul in care va arata rezultatul obtinut. Celelalte nivele sunt denumite straturi ascunse. Aceastea fac prelucrarea informatiei primite de la straturile anterioare si o transmit mai departe. Spunem ca o retea neurala invata din datele primite, daca isi modifica parametrii primiti ϕ astfel incat sa reprezinte cu mai multa acuratete corespondenta intre datele de intrare x si cele de iesire y. FastDepth este o arhitectura de retea neurala folosita pentru estima adancimii in imagini. Aceasta primeste o imagine de tip RGB al interiorului unei incaperi si returneaza o matrice cu valori in intervalul $0m - 10m$ estimand pentru fiecare pixel in parte distanta de la planul de proiectie al imaginii pana la punctul din spatiu surprins de fotografie. Scopul nostru este antrenarea unei retele neurale care sa produca o matrice de adancime cu valori cat mai apropiate de distanta reala la care se afla obiectele fata de camera. In ciuda faptului ca algoritmul de tip ORB-SLAM2 functioneaza fara a folosi orice tehnica de Machine Learning. Consider ca adaugarea unor retele neurale care sa indeplineasca anumite sarcini clar definite: cum ar fi extragerea feature-urilor sau pentru estimarea adancimii in imagini ar fi un pas in fata pentru a creste eficienta algoritmului. Revenind la sarcina de estimare a adancimii, ORB-SLAM2 foloseste o camera tip RGBD / Stereo care descrie cu foarte mare acuratete distanta pana intr-un anumit punct din spatiu, dar creeaza o matrice rara de valori, majoritatea avand valori de 0, sugerand incapacitatea de a estima distanta in zonele respective. O retea neurala are capacitatea sa gaseasca o aproximare a distantei pentru absolut fiecare punct din imagine si are un context suficient de bine format pentru a intelege care obiecte sunt mai apropiate si care se afla mai departe. In acest fel, valorile estimate au sens: zone similare din imagini avand valori ale distantei apropiate una de cealalta si atribuie o valoarea valida a adancimii pentru fiecare pixel in parte. Un posibil dezavantaj al acestei

arhitecturi este limitarea de 10m, fiind nepotrivit de folosit afara, dar ideal pentru un spatiu inchis de mici dimensiuni. Cu toate acestea exista 2 probleme pentru care o retea neurala s-ar putea sa nu fie optiunea potrivita: valorile aproximate au o acuratete mai slaba decat cele obtinute de camerele Stereo/RGBD iar viteza acestora de procesare a cadrelor nu este suficient de mare pentru a functiona in timp real. Un motiv pentru care am ales arhitectura FastDepth este ca rezolva in totalitatea problema vitezei procesand aproximativ 130 de cadre pe secunda in plus consuma o cantitate redusa de memorie, totalitatea parametrilor folositi in antrenare ocupand pana in 40 de MB, fiind usor de integrat intr-un dispozitiv embedded. Exista mai multe filozii cand vine vorba de modul in care ar trebui sa arate arhitectura unei astfel de retele neurale si operatiile pe care ar trebui sa le realizeze fiecare strat in parte. Prima astfel de abordare a fost feed forward neural network in care elementele de procesare erau dispuse pe straturi, si fiecare strat primea input-ul de la stratul precedent si transmitea output-ul la stratul imediat urmator. Informatia circula liniar, de la intrarea in retea pana la finalul acesteia. Abordarea s-a dovedit foarte buna pentru situatiile in care aveai nevoie de retele neurale de mici dimensiuni, cu un numar redus de straturi si parametrii. Pentru retele cu zeci de straturi de intrare, abordarea de feed forward netwrork nu era potrivita fiind foarte greu de antrenat corect o astfel de retea desi la nivel de principiu o retea cu dimensiuni mai mari ar trebui sa fie capabila sa generalizeze mai bine. Pentru a rezolva aceasta problema au aparut arhitecturile de tip residual network. Acestea au aplicatii numeroase in clasificarea imaginilor, unde datele de intrare au dimensiuni mari si este nevoie de multe nivele pentru a extrage suficiente informatii pentru a realiza clasificarea. Principiul de functionare este utilizarea unor straturi reziduale denumite si skip connections, in care rezultatul unui strat este salvat si transmis ca data de intrare la un alt nivel mai in fata sau chiar readus in bucla in acelasi strat. Aceasta abordare pastreaza din informatiile initiale ale datelor de intrare in straturile viitoare stabilizand antrenarea. FastDepth foloseste aceasta tehnica, straturile finale primind ca date de intrare valorile calculate de straturi aflate la inceput, tocmai pentru a pastra informatia initiala a imaginii. Pe langa tipurile de arhitecturi propuse, s-au modificat si tipurile de straturi in retele neurale. Primele folosite erau cele fully connected, in care fiecare element de procesare era conectat cu toate celelalte elemente de procesare din stratul urmator. Matematic, operatia poate fi vazuta ca o inmultire de matrici, o operatie costisitoare si mai ales limitata, nu putea fi folosita pentru a reprezinta functii nonliniare, scazand capacitatea de generalizare. De cele mai multe ori straturile liniare erau folosite impreuna cu functii de

activare nonliniare precum ReLU dar in continuare stratul avea prea multi parametrii care trebuiau antrenati. Din aceasta cauza au fost create straturile convolutionale, folosesc mai putini parametrii si sunt folosite in principal pentru procesarea imaginilor. Principiul teoretic pe care se bazeaza este ca pixeli alaturati vor avea acelasi scop in imagine, de exemplu vor reprezentarea aceluiasi feature. In aceasta situatie, operatia de convolutie trebuie realizata pe o zona a imaginii, o filozofie diferita fata de straturile complet conectate, in care valoarea fiecarui neuron este modificata individual. Operatia de convolutie functioneaza in felul urmator, se stabileste un kernel, o matrice de mici dimensiuni, in FastDepth folosindu-se kernel-uri de (3, 3), acestea vor stoca parametrii pe care reteaua neurala ii va antrena pentru stratul convolutional. In formula matematica sunt notati w_{mn}, h_{ij} reprezinta intensitatea pixelului dupa calculul formulei de convolutie, iar x_{ij} este valoarea intensitatii pixelului de pe coloana i si linia j. Ceilalti 2 parametrii a reprezinta functia de activare folosita iar β reprezinta bias-ul care poate fi modificat in procesul de antrenare al retelei neurale si cresc capacitatea de generalizare a functiei de convolutie.

$$h_{ij} = a\left[\beta + \sum_{m=1}^{3}\sum_{n=1}^{3}\omega_{mn}x_{i+m-2,j+n-2}\right] \tag{11}$$

Stratul de convolutie este prea costisitor pentru a fi folosit de mai multe ori pentru a crea o arhitectura in timp real de mari dimensiuni. Presupunem ca avem un vector de intrare pentru un strat de convolutie x cu dimensiunile $[d, h, w]$ unde $[h, w]$ reprezinta inaltimea si latimea vectorului, iar d reprezinta numarul de canale, pentru o imagine de tip RGB, valoarea parametrului d este 3, fiind 3 canale de culoare pentru rosu, verde si albastru. De asemenea presupunem ca avem un kernel de dimensiuni $[k, k, d, d_{out}]$, unde d_{out} este numarul de canale care rezulta in urma operatiei de convolutiei, atunci numarul total de operatii va fi: $h \cdot w \cdot d \cdot d_{out} \cdot k \cdot k$. Solutia pentru aceasta problema este data de utilizarea unei variante derivate a operatiei de convolutie, numita Depthwise Convolution.

2 DETALII DE IMPLEMENTARE

Implementarea este realizata in C++17. Pentru management-ul librariilor si al codului folosesc CMake 3.28.3. Acesta imi permite sa grupez in foldere codul scris de mine si face operatia de link-are automat cu binarele librariilor pe care le folosesc. Alte tehnologii pe care le mai folosesc sunt OpenCV 4.9.0, Ceres 2.2.0, Eigen 3.4.0, DBoW2 si ultima versiune de Sophus pana la data de ianuarie 2025. In comparatie cu alte librarii care inca mai trec prin diverse update-uri, Sophus a intrat intr-o etapa de mentenanta, dezvoltarea efectiva a acestuia fiind finalizata din iunie 2024. O prima problema pe care am intalnit-o a fost gasirea unei versiuni compatibile de C++ cu toate aceste pachete. Am incercat mai multe variante printre care C++11, C++14, C++17 si C++20. Preferinta mea ar fi fost sa folosesc o versiune cat mai noua cu putinta, dar care sa poata fi compatibila cu toate librariile mentionate. C++11 si C++14 nu erau compatibile cu Ceres, libraria fiind mult prea complexa pentru a intra si face modificari in codul sursa. C++20 nu era compatibil cu Sophus si cu Eigen, iar ambele librarii sunt fundamentale deoarece implementeaza metode puternic optimizate de a lucra cu matrici iar API-ul lor era mai simplu decat cel OpenCV. Singura optiune ramasa a fost C++17 care era incompatibila cu versiunea de DBoW2, aceasta folosi o versiune mai veche a functiei throw pentru erori. Modificand modul in care functiile returnau erorile, cu ajutorul metodei de biblioteca assert si a codurilor de eroare, am eliminat complet necesitatea vreunei functii de throw si am putut recompila intreg codul pe care l-am putut utiliza ca biblioteca in implementarea ORB-SLAM2. Fiecare din librariile utilizate indeplineste o anumita functionalitate bine definita.

OpenCV este fundamental deoarece contine functii de biblioteca pentru o multitudine de elemente: implementarea algoritmilor de extragere de feature-uri FAST, ORB, SIFT, SURF, procesare video, citirea unui video cadru cu cadru, procesarea de imagini: aplicarea de filtre, transformarea in grayscale, eliminarea distorsiunii cauzata de camera. Dar cele mai importante sunt structurile pentru lucru cu matrici cv::Mat, si cea pentru stocarea unui feature: KeyPoint. Structura KeyPoint este deosebit de utila deoarce stoca numeroase informatii despre zona pe care o reprezinta, orientarea acesteia, coordonatele centrului si nivelul la care a fost observat, parametrii de care am avut nevoie in fiecare dintre componentele algoritmului. Pe langa aceste

lucruri OpenCV are un modul dedicat pentru citirea retelelor neurale scrise din fisierele care urmeaza un format de tip ONNX, fiind o alternativa potrivita daca vreau sa aplic modelul, fara a incerca sa modific in vreun fel parametrii acestuia.

Ca librarie de optimizare am avut de ales intre Ceres si g2o. In implementarea oficiala g2o era cel mai folosit, fiindca filozofia din spate este de a face optimizari pe graf. Se creeaza un graf neorientat cu nodurile care trebuiesc optimizate si sunt conectate intre ele prin intermediul functiei de optimizare care trebuie aplicata. API-ul de g2o permite activarea si dezactivarea anumitor noduri, pentru a face implementarea mai robusta impotriva outlier-elor, pe cand Ceres nu permite acest lucru. O data create conditiile initiale acestea nu pot fi dezactivate pana la finalizarea algoritmului. Cu toate acestea, Ceres are un API mai usor de utilizat si prin rulari repetate am observat ca este cu putin mai rapid decat g2o.

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

Sophus este o librarie care imi permite sa lucrez cu algebra de tip Lie. In loc de a vedea estimarile pozitiei ca pe niste matrici de 4x4, le pot vedea ca pe un vector alcatuit din 7 elemente. Primii 4 parametrii alcatuind un quaternion, aceasta fiind o exprimare vectoriala a unei matrici 3X3 de rotatie, iar ultimii 3 parametrii reprezentand un vector de translatie. Biblioteca implementeaza operatii care imi permit sa lucrez cu acesti vectori, care fac parte dintr-un grup numit $se(3)$ si garanteaza ca rezultatul obtinut este scalat corespunzator pentru a face parte in continuare din aceeasi categorie.

DBoW2 este o metoda de tip bag of words pentru compararea imaginilor intre ele. Acesta foloseste feature-uri de tip FAST si descriptori de tip BIREF asemeni algoritmului ORB.

Structura de fisiere este una simpla, in folderul radacina se regaseste fisierul de CmakeLists.txt care va fi interpretat de utilitarul cmake pentru a genera automat Makefile-ul. Acest Makefile va contine regulile de build si de clean pentru proiectul meu. Am fisierul de main.cpp unde vor fi initializate componentele si se va selecta pe care dintre cele 2 seturi de date se va aplica algoritmul. Tot aici se regaseste si fisierul fast_depth.onnx, in este stocata arhitectura si parametrii antrenati ai retelei neurale FastDepth pentru estimarea adancimii. In plus am fisiele de include unde se afla antetele claselor pe care le voi implementa si fisierul de src

unde se afla codul de C++ si logica programului. Am observat ca separarea codului in acest fel este o practica des intalnita in proiectele de mari dimensiuni si garanteaza flexibilitate in includirea dependintelor intre fisiere. Algoritmul ORB-SLAM2 este unul complex, depinzand de o multitudine de parametrii care pot influenta acuratetea. Cei mai importanti sunt cei corelati cu camera. In fisierul config.yaml se regaseste matricea K, parametrii de distorsiune ai imaginii si alte constante pe care le-am considerat ca fiind niste hiperparametrii ai algoritmului, care vor trebui modificati in functie de mediul in care va rula ORB-SLAM2 pentru a garanta functionarea corecta.

In main.cpp se face citirea fisierului ORBvoc.txt, acesta contine datele pe care le va folosi clasa ORBVocabulary pentru a calcula vectorii de KeyPoint-uri pentru fiecare dintre cadrele cheie. Acesti vectori de KeyPoint-uri vor fi comparati intre ei pentru a determina daca pozele respective provin din acelasi loc pentru a face corelatii intre cadre, relocalizari sau operatii de loop closure. Tot in main.cpp se va face selectia pentru unul din cele 2 seturi de date pe care le va folosi algoritmul in rularea lui. Aceste seturi de date contin de fapt cadrele dintr-un video facut cu o camera RGBD Microsoft Kinetic impreuna cu matricile de adancime si pozitiile acestora in spatiu pentru fiecare cadru in parte. Aceste seturi de date sunt suficient de complexe pentru a-mi permite evaluarea functionarii algoritmului de ORB-SLAM2.

Clasa SLAM initializeaza componentele principale ale algoritmului si monitorizeaza durata fiecarei operatii pentru debugging. Implementarea se bazeaza pe compunerea de clase, fiecare fiind responsabila cu realizarea unei anumite sarcini. Voi prezenta clasele folosite plecand de la cele mai elementare catre cele mai complexe.

Clasa TumDatasetReader este responsabila de partea de achizitii de date, va citi fiecare cadru in parte iar pentru matricea de adancime va avea 2 variante: ori va incarca in memorie matricea de distante din setul de date pentru cadrul respectiv sau o va calcula folosind reteaua neurala FastDepth. Imaginea va fi convertita din RGB in grayscale, pentru o prelucrare mai rapida de catre ORB. Aceste 2 matrici vor fi transmise mai departe catre Tracker.

Clasa MapPoint este fundamentala pentru buna functionare a algoritmului ORB-SLAM2. Aceasta este formata cu ajutorul unui KeyPoint si al unui KeyFrame asociat acestuia. In etapa anterioara, am prezentat modul in care se face proiectia coordonatelor unui punct cheie in spatiu, acestea devenind coordonatele globale ale MapPoint-ului pec are il creem. Punctului din spatiu i se asociaza de asemenea descriptorul keyPoint-ului care l-a creat, pentru compararea ulterioara cu alte KeyPoint-uri din alte imagini. Un MapPoint are nevoie de un

vector de orientare, acesta ajuta in verificarea proprietatii unui MapPoint de a fi sau nu vizibil dintr-un KeyFrame. Pentru a calcula acest vector de orientare prima data se determina coordonatele globale ale centrului camerei pentru cadru cheie care a creat KeyPoint-ul, acest lucru se realizeaza in felul urmator:

$$\begin{bmatrix} X_c \\ Y_c \\ Z_c \end{bmatrix} = -R_{wc}^t * t_{wc}, \quad T_{wc} = \begin{bmatrix} R_{wc} & t_{wc} \\ 0 & 1 \end{bmatrix} \tag{12}$$

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

Clasa Feature, aceasta componenta nu exista in implementarea oficiala a ORB-SLAM, dar am considerat ca utilizarea acesteia ar simplifica codul. Extinde clasa KeyPoint fara a o mosteni explicit, are asociata distanta, extrasa din matricea de adancime, descriptorul si o valoare de tip boolean care setata pe true, inseamna ca punctul este monocular altfel inseamna ca este stereo. Aceasta clasificare se obtine prin compararea adancimii cu o valoare foarte apropiata de 0, de exemplu $1e-1$. Daca distantei este mai mica decat 0.1, consider ca valoarea estimata de camera RGBD ori de reteaua neurala nu este corecta si ca punctul respectiv este monocular, altfel voi considera ca este un punct de tip stereo. Pentru fiecare KeyPoint extras, se va crea un astfel de Feature, aceasta fiind clasa care va fi stocata direct in KeyFrame. Fiecare Feature are setat pe null la initilializare o referinta la un obiect de tip MapPoint. Pentru a garanta functionarea in real time a algoritmilor, asocierea (KeyPoint, MapPoint) trebuie sa poata fi accesata in $O(1)$. Vectorul de elemente de tip Feature, impreuna cu o structura de tip dictionar unde cheia va fi MapPoint si valoarea un Feature, vor face acest lucru posibil. Singura problema este ca cele 2 structuri incearca sa reprezinte aceleasi corelatii, in cazul vectorului am indicele unui Feature drept cheie si incerc sa accesez MapPoint-ul asociat, iar in cazul dictionarului, am referinta unui MapPoint si incerc sa obtin adresa Feature-ului. Ambele structuri trebuie sa contina aceleasi perechi, altfel comportamentul algoritmului devine nedefinit.

Clasa KeyFrame, aceasta contine mai multe elemente legate de cadrul curent. Pentru a men-

tine functionarea sistemului in timp real, trebuie sa stocam in memorie rezultatele calculelor noastre. Din aceasta cauza, in aceasta clasa se vor regasi matricea de adancime, vectorul de Feature-uri, vectorul de feature-uri calculat de metoda bag-of-words implementata in DBOW2 si cadrul initial, convertit in format grayscale ce va fi folosit ulterior pentru afisarea in timp real a performantelor algoritmului. In interiorul constructorului acestei clase sunt mai multe operatii realizate, majoritatea necesare pentru a creste eficienta accesarii datelor. De exemplu: vectorul de Feature-uri in medie contine 1000 de elemente care nu sunt sortate. In situatia in care proiectam un MapPoint in plan ar trebui sa comparam coordonatele proiectiei cu pozitia fiecarui Feature in parte pentru a stabili care este cel mai apropiat. O modalitate de a rezolva acest lucru este segmentarea suprafetei in K zone, in cazul meu am ales $K = 100$, fiecare reprezentand o portiune din imaginea initiala, avand asociate referintele Feature-urilor care se gasesc pe suprafata respectiva. In acest fel, in functie de zona in care este proiectat MapPoint-ul, vom stii ce Feature are o posibilitate mare de a corespunde corect cu MapPoint-ul respectiv, reducand astfel numarul de comparatii. Considerand ca toate valorile de tip Feature sunt dispuse in mod egal pe suprafata imagini atunci complexitatea devine, $O(N/K)$ unde N reprezinta numarul de Feature-uri iar K reprezinta numarul de zone in care a fost impartita imaginea. Constructorul este responsabil de initializarea structurilor de tip Feature, partitionarea lor in functie de coordonatele in imagine, si de memorarea estimarii curente a pozitie camerei si a centrului camerei in coordonate globale. Tot in aceasta clasa se regaseste structura de tip dictionar (MapPoint, Feature), care va fi adaptata pe tot parcursul algoritmului. Alta metoda importanta este: $get\_vector\_keypoints\_after\_reprojection$. Aceasta primeste ca date de intrare coordonatele proiectiei unui MapPoint, valoarea ferestrei de proiectie, si octava minima si maxima. Octavele reprezinta nivelul la care a fost observat un Keypoint in imagine si o estimare grosiera a distantei dintre camera si punctul din spatiu observat. Acesta poate sa aiba valori inte 0 si 7 inclusiv si ne spune de cate ori s-a facut resize la imagine pentru a surprinde o anumita trasatura in imagine. De exemplu: daca un Keypoint are valoarea octavei 0, inseamna ca algoritmul a detectat-o din imaginea nemodificata. Daca ar fi 1, atunci dimensiunea imagini a fost redusa o singura data cu 0.8 din valoarea initiala si asa mai departe. Feature-ul cu care este asociat MapPoint-ul trebuie sa aiba valori ale octavei apropiate intre ele. Daca aceasta situatie nu s-ar respecta, ar introduce erori de estimare a distantei, ne asteptam ca Feature-uri care au corelatie puternica intre ele, sa faca parte din aproximativ acelasi loc, daca estimarea distantei in spatiu ar avea o diferenta prea

mare intre cele 2 ar putea inseamna ca cele 2 puncte din spatiu sunt diferite, dar mediul are o structura simetrica, de exemplu: o sala de clasa cu bancile aliniate una in fata celeilalte, algoritmul ar putea observa 2 colturi ce apartin de 2 mese diferite, daca nu ar avea aceasta separare pe baza octavei, urmatorul cadru care observa aceleasi mese ar putea sa asocieze eronat punctele intre ele, afectand estimarea pozitiei. Fereastra de proiectie reprezinta cat de departe poate sa fie Feature-ul de coordonatele punctului de proiectie pentru a fi considerat corect. In functie de dimensiunea ferestrei aceasta poate intersecta 1, 2 sau 4 subsectiuni din cele 100 in care este impartita imaginea. Problema cea mai mare pe care am avut-o cu clasele KeyFrame si MapPoint era dependenta circulara. MapPoint-urile aveau nevoie de un KeyFrame si un Feature pentru a fi create si trebuia sa mentina o lista a KeyFrame-urilor care observa MapPoint-ul respectiv. In cazul KeyFrame-ului, acesta trebuie sa pastreze referinte asupra tuturor MapPoint-urilor pe care le observa. Pentru a rezolva aceasta problema am folosit o clasa aditionala care face operatii cu cele 2 structuri si am folosit forward declaration.

Clasa Map implementeaza harta pe care o foloseste algoritmul ORB-SLAM2. Aceasta este responsabila de stocarea corecta a KeyFrame-urilor, a MapPoint-urilor si rezolva problema dependentei circulare a celor 2 clase. Aici am implementate metodele de adaugare/stergere a unui MapPoint dintr-un KeyFrame. De asemenea, clasa de MapPoint continea referinte la toate KeyFrame-urile care o observa. Aceste referinte sunt adaugate / sterse de catre 2 metode care se regasesc aici. Clasa Map creaza o structura de tip graf ponderat neorientat, in care nodurile sunt reprezentate de KeyFrame-uri. Arcele arata daca exista mai mult de 15 puncte comune intre 2 KeyFrame-uri iar ponderea lor este determinata de numarul de MapPoint-uri comune. Tot aceasta clasa realizeaza operatii pe acest graf, adauga/sterge noduri si face interogari pentru a afla vecinii directi sau cei pe nivel 2. Am ales sa implementez aceasta structura folosind $std :: unordered\_map$. Drept cheie va avea KeyFrame-ul curent iar valoarea returnata de structura de tip dictionar va fi un alt $std :: unordered\_map$, ce va contine toate celelalte KeyFrame-uri cu care este direct conectata dar si ponderea conexiunii. In acest fel accesarea vecinilor de ordinul 1 va fi o operatie ce se poate realiza in timp constant. Functia $track\_local\_map$ este folosita de catre clasa Tracking. Aceasta primeste ca date de intrare cadrul curent, ultimul cadru cheie salvat. Nu returneaza nimic, doar incearca sa gaseasca cate un MapPoint pentru Feature-urile care inca nu au fost corelate cu un punct din spatiu. Aceasta operatie este costisitoare si functioneza in felul urmator:

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

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

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

1. Se proiecteaza coordonatele globale ale mp in planul imaginii folosind matricea de estimare a pozitiei T_{cw} si matricea parametrilor camerei K. Se verifica daca coordonatele proiectiei sunt valide pentru imagine.
2. Se calculeaza distanta d de la centrul camerei la mp. In functie de valoarea octavei stocata in acest MapPoint, se pot estima o limita minima si maxima pentru d. Daca valoarea obtinuta nu se incadreaza in acest interval se considera ca punctul este invalid.
3. Cu ajutorul geometriei analitice se obtine ecuatia dreptei care uneunghiului dintre aceasta dreapta si vectorul de directie al MapPoint-ului, daca valoarea este mai mica de 0.5. Atunci se considera ca mp nu poate fi observat in cadrul curent.ste punctul din spatiu si coordonatele centrului KeyFrame-ului. Se calculeaza cosinusul unghiului dintre aceasta dreapta si vectorul de directie al MapPoint-ului, daca valoarea este mai mica de 0.5. Atunci se considera ca mp nu poate fi observat in cadrul curent.

Daca aceste 3 verificari au fost realizate cu succes se considera ca punctul poate fi observat

de catre camera. Exista 2 functii responsabile de asocierile intre cadrele curente. Una este $match\_frame\_reference\_frame$ si cealalta este $match\_consecutive\_frames$. Voi incepe prin a o descrie pe prima, aceasta este folosita pentru a asocia Feature-uri intre primele 3 cadre de fiecare data cand este adaugat un Keyframe nou sau la inceput, cand sistemul tocmai ce a fost initializat. Aici este o situatie in care se folosesc vectorii de trasaturi calculati de libraria DBOW2, acestea gasesc o asociere intre elementele de tip Feature intre cele 2 imagini.

Clasa MotionOnlyBA implementeaza in Ceres algoritmul Motion Only Bundle Adjustment, primeste ca date de intrare KeyFrame-ul curent si returneaza matricea de pozitie optimizata. Libraria lucreaza cu o notiune din C++ numita functori. Acestea sunt clase/structuri pentru care s-a facut overload la operatorul $()$. Clasa BundleError se afla din aceeasi categorie si implementeaza functia de eroare obtinuta din proiectarea unui MapPoint si asocierea acestuia cu un Feature. Pentru a crea problema de optimizare, clasa ceres::Problem trebuie sa stie care parametrii trebuie optimizati si functia de eroare pe care trebuie sa o minimizeze. In cazul acestui algoritm, singurul lucru care va fi modificat este matricea de pozitie a KeyFrame-ului pe care o voi converti in forma $se(3)$, transformand-o intr-un vector de 7 elemente. Iar pentru functia de eroare, nu voi scrie explicit ca este suma erorilor de proiectie, voi initializa pentru fiecare asociere de tip (Feature, MapPoint) cate un element al clasei BundleError. Algoritmul de optimizare implementat de libraria ceres, va incerca in mod independent sa reduca valoarea erorii pentru fiecare pereche in parte, modificand pe rand vectorul pozitiei. Exista un motiv pentru care schimb modul in care este exprimata pozitia camerei, matricea de pozitie contine 2 componente: matricea de rotatie R si un vector de translatie t. Pentru t nu exista restrictii de modificare atata timp cat aceasta nu aduce modificari mari intre pozitiile a doua cadre consecutive, orice mod in care ar varia parametrii acestui vector, in continuare semnificatia lui de vector de translatie ramane nealterata, in aceasta situatie putem spune ca parametrii sunt alterati de catre libraria Ceres folosind $EuclidianManifold$, mici modificari bazate pe calcularea derivatelor partiale ale acestora din functia de eroare definita in BundleError, asemanator modului in care sunt modificati parametrii in retele neurale. Pentru matricea de rotatie R nu se mai poate aplica aceeasi logica. Aceasta trebuie sa faca parte din structura de tip grup numit $SO(3)$, adica sa respecte egalitatea $R * R^t = R^t * R = I$ si trebuie sa reprezinte o rotatie reala pe cele 3 axe. Alterarea aleatorie a parametrilor ar duce la o matrice invalida. Din aceasta cauza, modificarea rotatiei trebuie facuta cu un anumit unghi

iar acest lucru se poate realiza printr-o inmultire de 2 matrici de rotatie valide. Din pacate nu exista implementare in forma matriceala pentru schimbarea unghiului de rotatie, dar este pentru Quaternioni. Din aceasta cauza fac conversia din matrice de pozitie in vector din categoria $se(3)$, iar pentru primii 4 parametrii asociati rotatiei, optimizarea lor se realizeaza folosind $QuaternionManifold$. Aceasta abordare rezolva problema instabilitatii numerice si garanteaza ca rezultatul operatiei de optimizare este un element valid in $se(3)$, ce poate fi ulterior convertit in forma matriceala. In functie de categoria din care face parte Feature-ul, acesta este considerat monocular sau stereo. Functia de eroare implementata de clasa BundleError este identica pentru ambele, cu exceptia ca pentru punctele stereo, este verificata si distanta la care se afla punctul fata de valoarea la care a fost estimata de camera RGBD. Pentru a preveni instabilitatea cauzata de ppunctele de tip outlier, functia Huber descrisa in capitolul anterior este folosita in calcularea finala a erorii de proiectie. In implementara oficiala realizata de g2o, agloritmul de optimizare este rulat de 4 ori, si dupa fiecare executie sunt eliminate punctele de tip outlier. Experimental, am observat ca etapa de optimizare cadru cu cadru este cea mai costisitoare operatie pe care o realizeaza algoritmul de ORB-SLAM2, executia acesteia de 4 ori, nu creste semnificativ acuratea si reduce viteza de prelucrarea la aproximativ 5 cadre pe secunda, facandu-l nepotrivit pentru un sistem in timp real. Am observat ca obtin rezultate foarte bune, ruland o singura data Motion Only Bundle Adjustment, urmat apoi de o etapa de eliminare a corelatiilor (Feature, MapPoint) de tip outlier. Daca mai putin de 3 asocieri raman, se considera ca algoritmul a acumulat prea multe erori in urmarirea cadru cu cadru si trece intr-o stare de relocalizare.

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

1. Se creeaza KeyFrame-ul curent.
2. Se estimeaza matricea de pozitie pe baza legii de miscare.
3. Se realizeaza asocierea intre Feature-urile (puncte 2D) din cadru curent si MapPoint-urile observate de cadru anterior (puncte 3D)
4. Pe baza asocierilor respective realizate anterior, se optimiza matricea de pozitie a KeyFrame-ului curent, sunt eliminate asocierile de tip outlier

5. este proiectata harta locala pe cadrul curent, si se gasesc noi asocieri (Feature, MapPoint), se executa din nou aceeasi operatie de optimizare Motion Only Bundle Adjustment
6. Este evaluat KeyFrame-ul curent, se verifica daca trebuie salvat in clasa Map.

Cadrul curent si matricea de adancime sunt citite de TumDatasetReade. In imaginea RGB se foloseste ORB pentru a extrage un vector de KeyPoint-uri si un vector de descriptori. Acestea sunt folosite pentru a initializa un obiect de tip KeyFrame. Pentru algoritmul ORB se foloseste o versiune modificata implementata in clasa ORBextractor si este conceputa sa extraga aproximativ 1000 de puncte cheie, acestea fiind distribuite cat mai egal pe suprafata imaginii. Daca un numar foarte mare de keypoint-uri s-ar afla in aceeasi zona, acuratetea estimarii ar avea de suferit, pixelii din zonele aflate mai aproape de imagine se misca cu o viteza mai mare decat cei aflati in departare, daca am considera doar punctele dintr-o anumita in zona in realizarea estimarii, am obtinute variatii in miscare prea bruste / lente depenzind de locul unde s-au gasit majoritatea punctelor. Parametrii setati pentru algoritmul ORB sunt urmatorii: 1000 de feature-uri, factorul de scalare al imaginii este 1.2, exista maxim 8 nivele, si algoritmul FAST care face extragerea initiala de KeyPoint-uri sa ia in considerare zona respectiva daca diferenta de intensitate intre pixeli este de la 20 in sus. Daca in schimb, zona este slab texturata atunci poate sa seteze aceasta diferenta la 7, pentru a garanta ca vor fi gasite feature-uri chiar si in cele mai dezavantajoase zone din imagine. De-a lungul duratei de viata a algoritmului, clasa Tracker pastreaza 4 referinte de tip KeyFrame: cadrul curent care este analizat, 2 cadre imediat anterioare care vor fi folosite la estimarea pozitiei si ultimul cadru referinta care a fost creat. Cadrul referinta este ultimul KeyFrame adaugat in Map si indica aproximativ in ce zona se afla camera si care MapPoint-uri ar trebui sa fie vizibile. Cadrele mai vechi care nu au fost salvate in Map au fost sterse pentru a reduce cantitatea de memorie folosita. Pentru ultimul KeyFrame creat urmeaza etapa de estimare a pozitiei curente, aceasta se face pe baza legii de miscare, iar valorile matricii vor fi calculate folosindu-ne de cele 2 cadre salvate in Tracker. Important aici de observat ca pentru primul cadru citit, pozitia acestuia este matricea identitate 4*4, acest lucru sugerand ca dispozitivul care inregistreaza mediul considera ca primul KeyFrame este chiar originea sistemului de coordonate, iar toate matricile de pozitie viitoare sunt de fapt transformari relative fata de origine. Primul KeyFrame va fi salvat intotdeauna in clasa Map si este utilizat pentru a initializa primele puncte de tip MapPoint: pentru toate Feature-urile de tip stereo din imagine, se vor creea puncte in spatiu. Din cauza acestui mod de initializare, ORB-SLAM2 este sensibil pana la aparitia

urmatorului cadru cheie, estimarile facute de acesta in prima etapa fiind predispuse la erori. Uneori algoritmul isi pierde orientarea cu totul, fiind necesara o etapa de relocalizarea, sau de reluare a executiei acestuia. ORB-SLAM3, implementeaza o metoda mult mai robusta de initializare, generand mai multe harti locale in situatia in care urmarirea cadru cu cadru esueaza si le uneste intre ele in momentul in care recunoaste o zona pe care a vizitat-o deja. Dupa ce a fost create KeyFrame-ul si a fost facuta estimarea initiala a pozitiei, clasa OrbMatcher este folosita pentru a gasit corelatii intre Feature-uri si MapPoint-uri. Alegerea metodei care indeplineste acest lucru fiind determinata de numarul de KeyFrame-uri create de la ultima relocalizare sau de la adaugarea unui nou cadru cheie in Map. Daca nu se vor gasi minim 15 asocieri, se va considera ca algoritmul si-a pierdut orientarea, altfel, asocierile respective vor fi utlizate de catre MotionOnlyBA pentru a realiza optimizarea pozitiei. Perechile de tip outlier vor fi eliminate si noua pozitie a KeyFrame-ului va fi returnata. Daca vor ramane mai putin de 3 asocieri se va considera, din nou, ca algoritmul si-a pierdut orientarea. In final, se foloseste clasa Map pentru a proiecta toate punctele din harta locala pe cadrul curent iar asocierile gasite vor trece din nou printr-un proces de optimizare. Daca nu se gasesc minim 50 de perechi (Feature, MapPoint) inseamna ca urmarirea cadrului curent a esuat. Altfel se trece la etapa urmatoare si se va decide daca vom stoca in Map KeyFrame-ul curent. Acest lucru se va intampla daca urmatoarele conditii vor avea loc simultan.

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

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

1. Creeaza noi MapPoint-uri din primele 100 de Feature-uri de tip stereo, sortate in ordine crescatoare dupa distanta la care se afla acestea de centrul camerei

2. Adauga cadrul curent in graful de KeyFrame-uri stabilind vecinii directi ai acestuia
3. Noile MapPoint-uri create sunt adaugate intr-o lista numita $recently\_added$, pentru a iesi din aceasta lista, punctele trebuie sa treaca un test care dovedeste ca nu sunt rezultatul unui Feature eronat detectat de catre algoritmul ORB, si ca pot fi folosite cu incredere
4. Se executa operatia de $culling$, punctele sunt verificate daca sunt valide iar daca nu, memoria lor este eliberata.
5. Se foloseste operatia de triangulare pentru a crea noi MapPoint-uri din Feature-urile care se potriveau intre ele si faceau parte din cadre cheie diferite.
6. Se detecteaza entitatile de tip MapPoint care reprezinta acelasi punct din spatiu, iar una dintre referinte este stearsa pentru creste coorenta hartii si a creste ponderea conexiunii dintre KeyFrame-urile adiacente
7. Se executa operatia de KeyFrame culling, se verifica daca informatiile pe care le detine un KeyFrame, adica totalitatea valorilor de tip MapPoint pe care le detine, sunt observate si din alte cadre. Daca peste 90% din punctele observate de un anumit cadru sunt vizibile si din alte cadre, KeyFrame-ul analizat este considerat redundant si memoria lui este eliberata. Acest lucru garanteaza ca graful clasei Map, contine doar cadre esentiale pentru reprezentarea norului de puncte.

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

Ultima clasa este cea de MapDrawer pe care o folosesc pentru a afisa norul de MapPoint-uri, cadrul curent analizat si pozitiile cadrelor cheie observate. Folosesc libraria Pangolin si OpenGL pentru desenarea fiecarei structuri, camera urmareste cadrul curent. Interfata grafica

scade viteza de procesare a cadrelor dar este o modalitate eficienta de a intelege vizual ce se petrece in algoritm. Implementarea pentru interfata grafica am realizat-o spre final, cand aveam celelalte componente finalizate, lucru care a ingreunat procesul de dezvoltare deoarece lucram cu valori numerice in terminal. Acum daca as reincepe implementarea, interfata grafica ar fi printre primele lucruri pe care le-as realiza. Datorita acestei clase am reusit sa gasesc erori in modul de constructieal grafului ponderat din clasa Map si al modului in care proiectam punctele in spatiu.

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

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

ferentelor dintre valoarea reala si cea determinata de reteaua neurala in modul. In implementarea mea am ales sa folosesc o functie de pierdere mai robusta conform acestei lucrari stiintifice (voi cita aici lucrarea)

Acuratetea a fost verificata prin compararea diferentei relative intre valorile obtinute prin inferenta si cele reale cu un factor $RELATIVE\_ERROR = 0.15$. Aceasta operatie a fost realizata pentru fiecare pixel in parte, iar acuratetea reprezinta procentul de pixeli cu o valoare care se incadreaza in limita impusa de RELATIVE_ERROR. Pentru a preveni antrenarea pentru intervale lungi fara a obtine rezultate, am avut 2 metode pe care le-am implementat: o strategia de early stopping: in situatia in care valoarea acuratetii nu ar fi crescut pentru 5 epoci antrenarea ar fi fost oprita si o strategie pentru modificarea learning rate-ului in timpul antrenarii. Daca acuratetea nu crestea pentru 3 epoci valoarea parametrului sa fie redusa la 0.3 din valoarea initiala. In practica am observat ca reteaua converge aproape monoton catre o valoare optima. Functia de pierdere primeste ca date de intrare matricea de adancime obtinuta de catre reteau neurala si matricea cu valori reala din setul de date, denumita groundtruth, si returneaza o valoarea numerica de tip double care exprima cat de departe se afla estimarea noastra de realitate. Ideea antrenarii unei retele neurale este minimizarea acestei valori. Functia de eroare este alcatuita dintr-o combinatie liniara a 3 componente diferite: L1Loss, GradientEdgeLoss si Structural Similarity Loss, formula matematica este:

$$loss = 0.6 \cdot L1Loss + 0.2 \cdot GradientEdgeLoss + StructuralSimilarityLoss \quad (13)$$

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

shiftat vertical pentru axa Y notata $\frac{\partial I}{\partial y}$.

$$L_{\text{edges}} = \frac{1}{N} \sum_{i=1}^{N} \left(\left| \frac{\partial I_{\text{pred}}}{\partial x} - \frac{\partial I_{\text{true}}}{\partial x} \right| + \left| \frac{\partial I_{\text{pred}}}{\partial y} - \frac{\partial I_{\text{true}}}{\partial y} \right| \right) \tag{14}$$

3 EVALUARE

Un motiv pentru care mi-a fost dificil sa testez implementarea pe un video facut de mine a fost faptul ca nu aveam acces la valorile pozitiei in spatiu si a orientarii la fiecare cadru in parte, fiindu-mi imposibil sa verific daca acuratetea implementarii este corespunzatoare. Algoritmul are nevoie de matricea K a camerei, daca valorile ei sunt incorecte algoritmul va creea MapPoint-uri noi pentru aproape fiecare cadru in parte si va pierde capacitatea de a urmari traseul in cateva secunde. Am ales sa folosesc un set de date numit TUM RGBD Dataset, care contine videoclipuri filmate cu o camera RGBD, Microsoft Kinetic, la 30 de cadre pe secunda, imaginile avand rezolutia Acesta include numeroase subseturi de date care testeaza diferite lucruri pe care ar trebui sa le indeplineasca ORB-SLAM2. De exemplu pentru a verifica ca estimarea translatiei cadru cu cadru este realizata corect, folosesc rgbd_dataset_freiburg1_xyz, in care miscarea este facuta in linie dreapta, camera pastrandu-si aproximativ aceeasi orientare pe tot parcursul video-ului: cadrele de tip rgb, matricile de adancime, pozitia lor in spatiu pentru fiecare cadru in parte, parametrii camerei si cei de distorsiune.