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 feature-uri pentru fiecare dintre cadrele cheie. Acesti vectori de feature-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.

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 Tracker realizeaza urmarirea traiectoriei camerei cadru cu cadru. Tot ceea ce realizeaza este ca pe baza pozitiei a doua cadre imediat anterioare, gaseste noua matrice de pozitie a cadrului curent. Pentru a face acest lucru indeplineste mai multe etape: In primul rand, estimeaza pozitia si orientarea cadrului curent. Algoritmul functioneaza folosind coordonate locale asa ca primul si cel de-al doilea cadru vor avea asociate matricea identitate 4*4, acest lucru sugerand ca dispozitivul care inregistreaza mediul pleaca din originea sistemului de coordonate. Daca prima imagine va fi folosita doar pentru a initializa harta, cea de-a doua va

trece prin intreg pipeline-ul de optimizare al pozitiei, modificandu-si astfel estimarea initiala. De la al treilea cadru pana la finalul algoritmului, estimarea initiala a pozitiei va fi realizata exclusiv pe baza legii de miscare.

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.