UNIVERSIDAD DE MÁLAGA

E.T.S. INGENIERÍA INFORMÁTICA
UNIVERSIDAD DE MÁLAGA

GRADO EN INGENIERÍA INFORMÁTICA

SLAM VISUAL 3D usando Optimización de Grafo de Poses
3D Visual SLAM using Pose Graph Optimization

Realizado por
Alejandro Naranjo Núñez

Tutorizado por
Javier González Jiménez
José Raúl Ruiz Sarmiento

Departamento
Ingeniería de Sistemas y Automática
UNIVERSIDAD DE MÁLAGA

MÁLAGA, JUNIO DE 2022

Abstract

In the next few years, mobile robots will become more and more common in our daily life, landing in areas like education, health care, or assisting humans in everyday tasks. To accomplish these tasks autonomously robots must be able to recognize its working environment (i.e. to build a map) and be able to localize them-self within it. These two problems need to be solved simultaneously, a key robotic paradigm called SLAM (Simultaneous localization and mapping), which is a huge topic of research nowadays. This bachelor's thesis develops a SLAM system using *Pose-graph optimization*, which achieves a solution of the sequence of robot poses (position and orientation) in the plane (2D). Such system is denominated *Full SLAM*, since the whole trajectory of the robot is estimated. Observations from the environment are provided by a pair of *calibrated stereo cameras* on board the robot from which 3D landmarks coordinates are gathered. Thus, the whole process is termed *Visual SLAM*. An additional goal of this project is to implement the Visual SLAM pipeline in such a way that it can be used for illustrative and educational purposes in Computer Vision and Robotics related courses, with the intent of showing how these trendy fields can work together. For that, computer vision techniques are applied to the stereo images captured by the mobile robot in order to extract *keypoints* from the images, establish reliable matches between them, and triangulate the correspondence pairs to obtain 3D landmarks of the space. In addition to the implementation of a Visual SLAM system with the aforementioned features, this project also provides a virtual model of the environment, making use of *3D Unity library*, where mobile robots equipped with RGB cameras can be simulated in order to generate datasets to feed the system.

Keywords: Mobile robots, Visual SLAM, Computer vision, Graph optimization, Stereo cameras.

Resumen

En los próximos años, los robots móviles serán más y más comunes en nuestra vida cotidiana, llegando a áreas como educación, salud o asistir a humanos en tareas del día a día. Para efectuar estas tareas de forma autónoma los robots deben ser capaces de reconocer su entorno de trabajo (es decir, construir un mapa) y ser capaces de localizarse dentro de ellos. Estos dos problemas deben resolverse simultáneamente, un paradigma clave en robótica llamado SLAM (Simultaneous localization and mapping), el cual es un importante área de investigación hoy en día. Este trabajo de fin de grado desarrolla un sistema SLAM usando *optimización de grafo de poses*, que logra una solución de la secuencia de poses (posición y orientación) del robot en el plano (2D). Este tipo de sistemas se denominan *Full SLAM*, ya que se estiman la trayectoria completa del robot. Observaciones del entorno son dadas por un par de *cámaras estéreo calibradas* montadas en el robot desde donde se obtienen las coordenadas 3D de los landmarks. Por esto, el proceso se denomina *SLAM Visual*. Un objetivo adicional de este proyecto es implementar el sistema de SLAM Visual de manera que pueda ser usada de forma ilustrativa y educacional en clases relacionadas con la Visión por Computador y la Robótica, con la intención de mostrar cómo estos campos que están de moda pueden trabajar juntos. Para ello, técnicas de visión por computador son aplicadas a las imágenes estéreo obtenidas por el robot móvil para extraer *keypoints* de las imágenes, establecer correspondencias fiables entre ellos y triangular los pares correspondientes para obtener landmarks 3D en el espacio. Además de la implementación del sistema de SLAM Visual con las características previamente mencionadas, este proyecto también proporciona un modelo virtual del entorno, haciendo uso de la librería *Unity 3D*, donde robots móviles equipados con cámaras RGB pueden ser simulados para generar datasets para ser usados por el sistema.

Palabras clave: Robots móviles, SLAM Visual, Visión por computador, Optimización de grafos, Cámaras estéreo.

Contents

1 Introduction

Mobile robots are becoming more and more common in our daily life, being the different Roomba's models [17] a clear example of that. Recently, there have been many developments towards making them able to have a greater grade of interaction with humans, like Astro [1] from Amazon (see Fig.1), REEM [26] from PAL Robotics or Camello [25] from Outsaw Digital. This kind of mobile robots work in an environment that is initially unknown and, to be able to accomplish their tasks in an efficient way, they must be able to recognize their working environment and be able to localize themselves within them. The combination of these problems is known as Simultaneous Localization And Mapping (SLAM) [33], and it is the problem that we are going to address in this project.

Figure 1: Amazon's Astro interacting with a user.

1.1 Motivation

A mobile robot is a robot capable of moving around its environment, which is usually unknown to it. Mobile robots need to be autonomous, meaning that they must be capable of navigating without the need of guidance by humans. To achieve this, it needs to create an internal representation of said working environment, which is called a map, and to know its pose (position and orientation) within it. Those problems are usually solved online using a technique called SLAM (Simultaneous localization and mapping). There are two main approaches when designing a SLAM system:

- Online SLAM: Only estimates the last pose of the robot, an example would be EKF (Extended Kalman Filter) [23].
- Full SLAM: Estimates the full path of the robot, an example would be Graph SLAM [40].

Full SLAM has a better accuracy for the estimations as it estimates the full path and makes it consistent, which is the approach that we follow in this project. Specifically pose graph SLAM which is a simplification of Graph SLAM where only the robot poses are optimized and not the landmarks, as will be further described in this report.

With this in view, the main motivation behind this project is to build a Visual SLAM system that results illustrative and educative about its constituent parts. Visual SLAM systems rely on information obtained by cameras mounted on the robot. The images from cameras are later processed in order to obtain keypoints (distinctive patches of the image), which are matched with keypoints of other images in order to find correspondences. Those correspondences, when projected back to space in a global, common reference frame (usually called the *world frame*), become landmarks with an associated position, forming in this way the aforementioned map. With those landmarks and odometry information (wheel information on how the robot has moved) the Visual SLAM system estimates both robot and landmarks location.

Visual SLAM systems are complex solutions with numerous steps, so they are usually hard to understand as a whole. For doing so, the developed code will be part of a document built with the Jupyter Notebook technology, which permits us to merge text, equations, figures, videos, etc. with executable code cells, resulting in a powerful educational tool [28]. This resource is planned to be used in Computer Vision and Robotics related subjects at the University of Málaga.

1.2 Objetives

The main goal of this project is to develop a Visual SLAM system which receives information from a pair of cameras mounted on a robot, computes the 3D position of the observed keypoints and uses that information, combined with the odometry, to create a graph of poses which is later optimized to make it more accurate. This way we are building a representation of the environment and localizing within it at the same time.

Another objective of this project is to design a virtual environment where mobile robots equipped with stereo cameras can be instantiated. This way, the virtual environment serves as a sandbox where simulations can be done while recovering information relative to robot poses and stereo images, that is, capturing datasets to support the development of Visual SLAM systems. This will be implemented in the Unity 3D ecosystem [37].

The result of this project will be integrated into illustrative and educational Jupyter Notebooks profitable in Computer Vision and Robotics related courses. The Computer Vision part corresponds to the so-called *frontend*, which is what obtains the information using keypoint detection, matches these keypoints with those detected in the other frame of the stereo pair, and applies triangulation to obtain 3D coordinates of what the robot is seeing. The coordinates of these keypoints are later used in the *backend*, which corresponds to the Robotics part, to build the pose graph using the information given by the frontend and additional information retrieved from the robot like the odometry estimation from the motion. The backend also optimizes the graph using a Gauss-Newton optimization [12].

1.3 Methodology

During the development of this project the following software engineering methodology was used:

- The implementation followed an iterative development method, having multiple intermediary versions of the program to test different parts.

- To save the different versions of the program we used Git and a remote repository in Bitbucket.

- To test the correct output of the program we generated a dataset in a virtual environment created with *Unity*, exploiting the ground truth of the robot paths to see how well the system approximated the robot's poses in the environment.
- For development the chosen programming language was python, the editor was Vim and for debugging pudb [2] was used.

1.4 Structure

The document is structured as follows:

- **Chapter 1: Introduction.** Describes the context of this project, provides the motivation behind it and the objectives to accomplish.
- **Chapter 2: Background.** Contains information that is necessary to understand the task of visual SLAM and how it is addressed in the literature.
- **Chapter 3: Used technologies.** Describes the libraries, frameworks and technologies used in this project.
- **Chapter 4: Implementation.** Explains how the Visual SLAM system works, what the main problems we faced are, how we tried to solve them and how we use Unity to generate the dataset.
- **Chapter 5: Results.** Shows the outcomes of different benchmarks used to determine how accurate and performant the system is.
- **Chapter 6: Conclusions and future work.** Contains a discussion of the work done in this project and possible future lines of work.

2 Background

SLAM (Simultaneous localization and mapping) is a problem that is composed of two very important problems in mobile robotics: i) localization, which is the ability to determine the position and orientation of the robot within its frame of reference, and ii) mapping, which is the ability to create a representation of the environment. Due to their correlation they are usually solved at the same time which leads to the SLAM problem.

To be able to build a map and localize itself within it the mobile robot must be able to perceive its environment. To do this many different sensors can be used, for example, lasers, sonars or cameras. Cameras are sensors that are cheap and efficient, because they provide information using few resources, SLAM systems that use cameras to obtain information are called Visual SLAM systems. Several SLAM systems exist that use cameras, as far as we know, the first Visual SLAM system was A. Davison's Mono-SLAM in the beginning of the century [6]. Mono-SLAM uses keypoints to represent landmarks in the map, uses frame-to-frame matching to obtain their 3D coordinates and updates the state vector (which is composed of the last robot pose plus each feature 3D position) using an Extended Kalman Filter (EKF) [23]. On the other hand the first stereo-based visual SLAM system was proposed in 2002 by Set et al[32] which used SIFT [22] features to detect landmarks and the stereo pair to get the 3D coordinates of the keypoints in absolute scale.

In this project we use a pair of stereo cameras, since using only one camera and monocular vision wouldn't be able to get the absolute scale of the scene, which means that stereo-based systems are more accurate.

To implement a SLAM system the robot also needs to obtain odometry information, which is an estimation of its movement. For this it can use techniques such as Visual odometry, which is the process of estimating the motion of a robot using only the associated camera images, or wheel odometry, reading how much the wheels rotate using sensors. Odometry, technically, is a measurement rather than a control action, but usually treated as control to simplify the modeling.

In the context of this project Pose graph SLAM is used (see Fig.2), which is a simplication of the full SLAM technique based on Graph SLAM [40] that only optimizes robot poses and not landmarks. Pose graph SLAM consists of building a graph which is made of nodes that represent the different robot positions, that have edges between them which are the odometry commands that serve as a constraint, and landmark observations (red stars) that also serve as a constraint when they are observed from multiple poses.

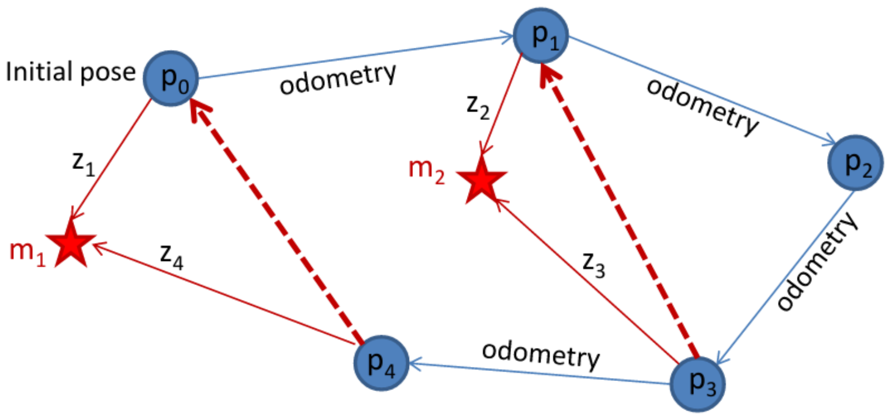


Figure 2: Pose graph [10].

Loops in the graph create inconsistencies as the robot movement is not perfect, these inconsistencies create errors and we can use these errors to optimize the poses, i.e. minimize the overall square error of the relative poses.

But this approach has a few problems that we need to solve, the first one is detecting and matching features. We need a good algorithm to detect and describe the features but it also needs to be fast as this is a real time problem. We also need to obtain the odometry, either by calculating it using visual odometry or by using the wheel odometry.

Another problem is that Visual SLAM systems are very sensitive to incorrect data, so bad matches are a big problem. To reduce the amount of bad matches many approaches can be used, one of them is using RANSAC [35] in cases when we have a model to fit.

SLAM systems are very complex and due to this they are usually divided into two parts: frontend and backend. The frontend's job is to build the graph, that is, obtain all the information needed and transform it in a way the backend can use it. On the other hand, The backend has the responsability to optimize the graph using techniques like Gauss-Newton optimization [12].

2.1 Components of a Visual SLAM system

2.1.1 Motion estimation

Odometry or motion estimation consists in using data from sensors to estimate how the position has changed over time [4]. It's mainly used by robots with legs or wheels to estimate their position relative to a starting point. The ability to estimate motion is imperative to SLAM systems as it gives us a first estimation of the path. As we have mentioned earlier in this section, odometry is technically a measurement insted of a control action but is considered a control action to simply the modeling.

There are three main ways to estimate motion:

- **Wheel odometry:** Is based on sensors that provide data for the rotation of the robot's wheels.
- **Visual odometry:** Consists of determining the pose of the robot by analyzing the images it captures [31].
- **Dead reckoning:** Is the process of estimating the current position of a moving object using a previous known position then adding estimates of speed, direction and elapsed time [42].

All these approaches suffer from cumulative errors and loop closure techniques are a necessity to keep these errors from getting too big. For this reason, robust systems need to implement constraints from odometry and loop closure to have an accurate system.

2.1.2 Map generation

To be able to solve the mapping problem, the robot needs to be able to generate a representation of the working environment, which is commonly referred as *map*. As previously commented, to accomplish this the robot must be able to perceive its environment, for which different sensors can be used (cameras, laser scanners, sonars, etc.). Different types of sensors lead to different SLAM algorithms which make their own premises. Systems based on cameras and systems based on lasers are similar as they provide detail for many points or landmarks within an area, and it's that information what is used to generate a map.

There a different types of maps but the most important to SLAM systems are:

- **Landmarks-based:** Landmarks are represented by their position in the world frame plus a descriptor to distinguish them between each other. It's used by systems based on cameras or laser scanners.

- **Occupancy grid:** Consists of a representation of an evenly spaced field of binary random variables, each variable corresponds to the presence of an obstacle at that location. It's mainly used in systems with noisy and uncertain sensors like sonars and lasers.

In the context of a visual SLAM system, to obtain the landmarks a computer vision algorithm is needed. Usually SIFT [22] or SURF [3] are chosen for this as they are invariant to lighting, scale and rotation. They consist of a detector that finds keypoints, and a descriptor that is later used to determine how similar two keypoints are for matching purposes. With these algorithms visual SLAM systems solve multiple problems like obtaining the 3D coordinates of a point to be able to place it on a landmark map, or detect that the robot is in a position that it has already visited as it is observing something that it has seen before.

ORB [30] was created as an alternative to SIFT and SURF, and sees plenty of use as the computation cost is usually lower, the matching performance is similar and unlike ORB, SIFT and SURF are both patented software.

2.1.3 Loop closure

Loop closure is the ability to recognize that the robot has returned to a previously visited location. This is an important part of a SLAM system as when the robot moves it accumulates errors and might lead to significant errors in the motion estimation. These errors cause incompatibilities, as a single region of the map might have multiple representations.

In recent years, there have appeared approaches based on building a database of the images obtained online by the robot, calculating the closest one previously seen and, if they are close enough, creating a loop closure. Many of these algorithms are based on comparing the images as numerical vectors in the bag of words space. Bag of words approaches are often competent and fast image matchers, but they are not fail-proof. Because of this a verification step is needed later to check that the matching images are geometrically consistent as introducing erroneous loop closure can result in disastrous errors. One example of this approach is DBoW [9] developed by Gálvez-López and Tardós in 2012, where they use FAST+BRIEF features to improve the computation time it takes to extract features from the images as this is the step that takes the most amount of time.

Loop closure is a way to make systems more robust and provide better estimations but, surprisingly, the work of Prokhorov et al [27] showed that using ORB-SLAM2 [24], a world reference project in visual SLAM with a loop closure module based on a bag of words apporach, and using common open source datasets for RGB-D cameras like TUM RGB-D [36] with the loop closure module activated and deactivated, loop closure didn't significantly influence the robustness of the system.

2.1.4 Pose estimation

Pose estimation is the last component of a SLAM system. This component refines the approximation obtained from motion estimation using the information obtained from the sensors. As we have mentioned before, the amount of poses the system has to optimize is defined by the type of SLAM system we have: Online SLAM techniques as EFK [23] only estimate the last pose of the robot and every pose of the landmarks; Full SLAM techniques like Graph SLAM [40] estimates the full path of the robot plus every landmark in the map, because of this, Full SLAM solutions are very computationally expensive (both in memory usage and execution time), but Full SLAM solutions usually result on more accurate pose estimations.

Landmarks reobserved by the robot create inconsistencies because when they are reobserved, they are estimated to be in different positions than the previous observations. This is due to the fact that neither the robot movement nor the sensors are perfectly accurate.

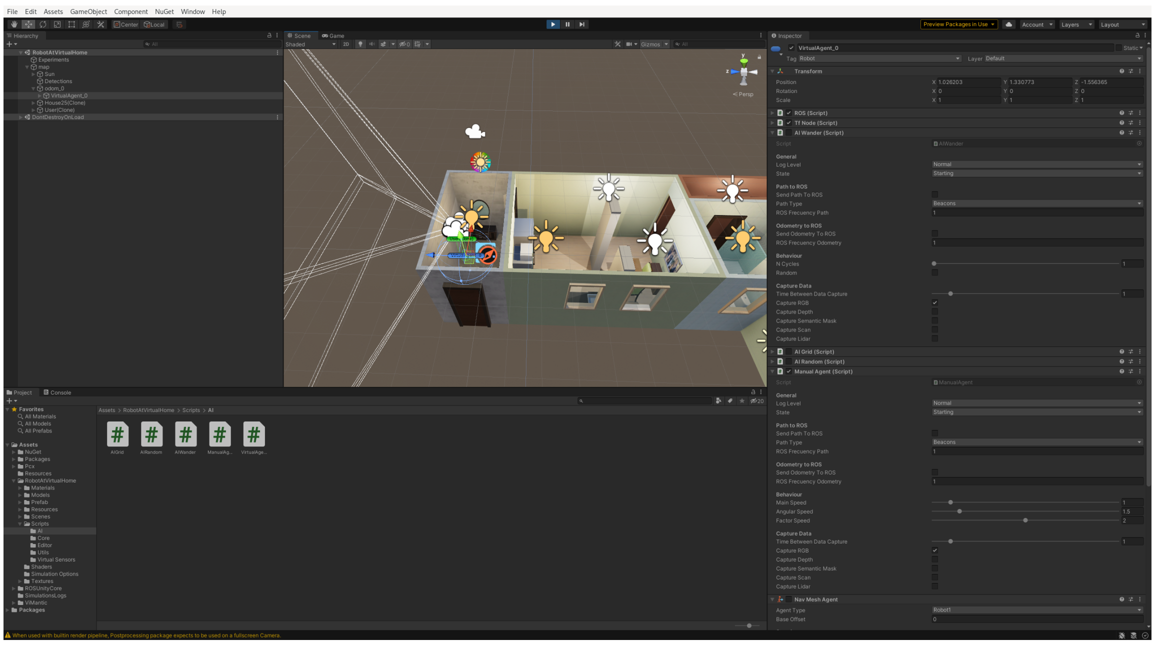
Using these inconsistencies and the motion estimation as restrictions, SLAM systems are able to move around the different poses to minimize the error made in the observations and the movement trying to achieve global consistency of the state vector (robot pose, or robot pose plus landmarks positions, depending on the SLAM method).

3 Used Technologies

3.1 Unity

Unity (see Fig.3) is a game engine that has support for desktop, mobile, console and virtual reality platforms [38].

The engine can be used to create 3D and 2D games, and also real world simulations. The engine has been used in industries outside of video games like cinema, automobile, engineering and the United States Armed Forces [37].



VirtualAgent_0
Tag Robot   Layer Default

Transform
Position X 1.026203 Y 1.330773 Z -1.556365
Rotation X 0 Y 0 Z 0
Scale X 1 Y 1 Z 1

ROS (Script) ✓
Tf Node (Script) ✓
AI Wander (Script)
Script AIWander

General
Log Level Normal
State Starting

Path to ROS
Send Path To ROS ☐
Path Type Beacons
ROS Frecuency Path 1

Odometry to ROS
Send Odometry To ROS ☐
ROS Frecuency Odometry 1

Behaviour
N Cycles 1
Random ☐

Capture Data
Time Between Data Capture 1
Capture RGB ✓
Capture Depth ☐
Capture Semantic Mask ☐
Capture Scan ☐
Capture Lidar ☐

AI Grid (Script)
AI Random (Script)
Manual Agent (Script) ✓
Script ManualAgent

General
Log Level Normal
State Starting

Path to ROS
Send Path To ROS ☐
Path Type Beacons
ROS Frecuency Path 1

Odometry to ROS
Send Odometry To ROS ☐
ROS Frecuency Odometry 1

Behaviour
Main Speed 1
Angular Speed 1.5
Factor Speed 2

Capture Data
Time Between Data Capture 1
Capture RGB ✓
Capture Depth ☐
Capture Semantic Mask ☐
Capture Scan ☐
Capture Lidar ☐

Nav Mesh Agent
Agent Type Robot1
Base Offset 0

When used with builtin render pipeline, Postprocessing package expects to be used on a fullscreen Camera.

Figure 3: Unity example.

Unity can be used along Blender, 3ds Max or Maya among others to create 3D models and import them to the scene. In 2D environments, Unity permits importing sprites and a 2D world renderer. In 3D environments, Unity permits specifying mipmaps, texture compression, and resolution settings for every platform that the game engine supports, and also supports parallax mapping, dynamic shadows using shadow maps, bump mapping, screen space ambient occlusion (SSAO), full-screen post-processing effects, render-to-texture and reflection mapping. [39]

One of the most important things about Unity is how big the community is. This gives us access to plenty of documentation, communities and free or paid assets from the store which can be used to speed up development.

In this project it is used with the project Robot@VirtualHome[8] to create a simulation of a robot moving within a house. We use this simulation to generate the dataset to test the project.

3.2 OpenCV

OpenCV (Open Source Computer Vision Library) is an open source computer vision and machine learning software library. Its main purpose is to offer a base infrastructure for computer vision applications and to increase the usage of machine perception in commercial products. [16]

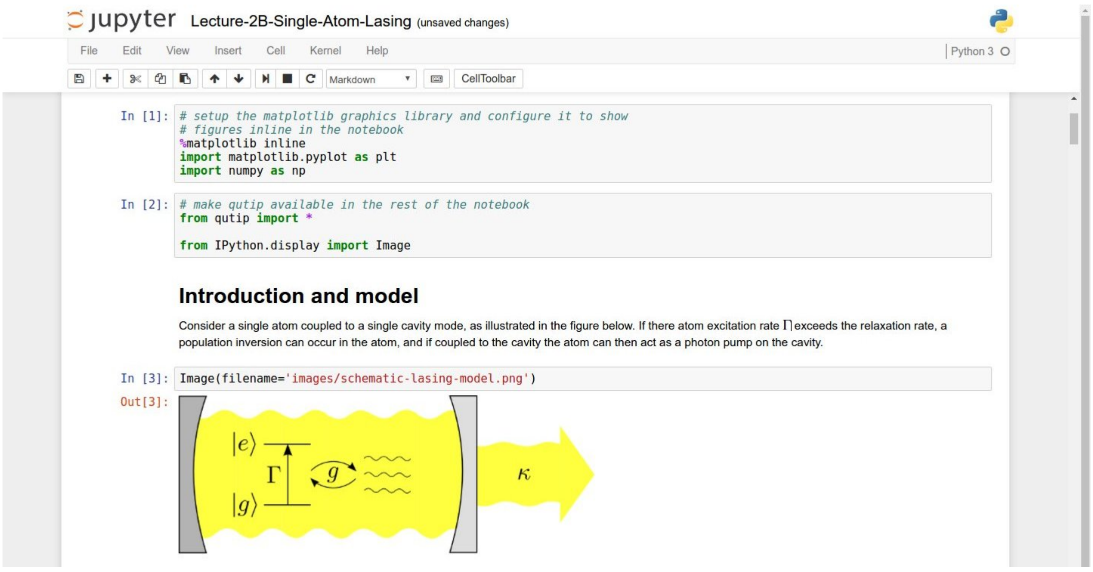
OpenCV launched in 1999, it began as an Intel Research initiative to advance computer vision-based commercial applications. [18] The project was formed by optimization experts from Intel Russia and Intel's Performance Library Team. The main goals of the project were [19]:

- Provide open source and optimized code for computer vision infrastructure.
- Create a common infrastructure for engineers to build on, so that the code could be more readable and portable.
- Push ahead commercial vision applications by making the code available for free with a license that allowed the code to be proprietary.

In this project we are using many of the tools provided by the OpenCV library such as the ORB [30] detector and descriptor, the brute force matcher and the functionality to calculate the fundamental matrix of a camera to use RANSAC.

3.3 Jupyter Notebook / Python

Jupyter Notebook [28] (see Fig.4) is a web-based interactive environment that can be used to create notebook documents. The term "notebook" can be used to refer to multiple different ideas, mainly the Jupyter web application, the python Jupyter web server or the document format Jupyter, depending on the context. A Jupyter notebook document is a JSON document following a versioned schema that contains a sorted list of input/output cells which can contain code, text (using Markdown), mathematical expressions, graphics and rich text, which usually ends with the extension ".ipynb". Jupyter Notebook is an open source project which creates a platform that lets you use different programming languages. It lets you create a notebook which can execute code and write text alongside LaTeX ecuations, video and everything that a browser can display.



```
In [1]: # setup the matplotlib graphics library and configure it to show
        # figures inline in the notebook
        %matplotlib inline
        import matplotlib.pyplot as plt
        import numpy as np

In [2]: # make qutip available in the rest of the notebook
        from qutip import *

        from IPython.display import Image
```

Introduction and model

Consider a single atom coupled to a single cavity mode, as illustrated in the figure below. If there atom excitation rate Γ exceeds the relaxation rate, a population inversion can occur in the atom, and if coupled to the cavity the atom can then act as a photon pump on the cavity.

```
In [3]: Image(filename='images/schematic-lasing-model.png')
Out[3]:
```

Figure 4: Jupyter Notebook example.

A notebook can be converted to multiple output formats such as HTML, slides, LaTeX or PDF among others.

A Jupyter Notebook has two main components:

- **Kernel:** It's responsible of executing the code in the notebook. By default it executes python code but other kernels for different programming languages can be installed.
- **Dashboard:** It displays the notebooks and manages the different kernels.

The result of this project will be used in a Jupyter Notebook for Computer vision and Robotics related courses as a way to see how both fields can work together.

4 Implementation

4.1 System overview

The developed system for 3D Visual SLAM (see Fig.5) consists of: i) a frontend that is responsible for collecting all the information necessary and creating a graph (see Fig.6), and ii) a backend which is responsible for optimizing the graph.

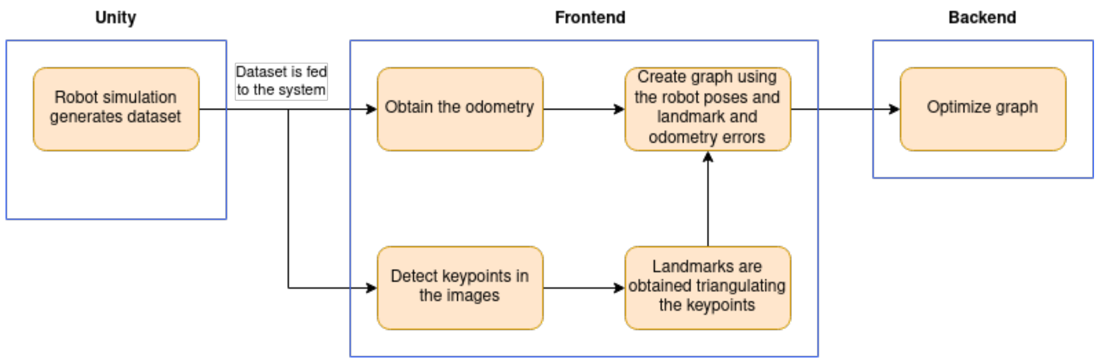

Unity
Frontend
Backend
Robot simulation generates dataset
Dataset is fed to the system
Obtain the odometry
Create graph using the robot poses and landmark and odometry errors
Optimize graph
Detect keypoints in the images
Landmarks are obtained triangulating the keypoints

Figure 5: Flowchart of the Visual SLAM system.

For supporting the system development, we simulate a mobile robot in a real world environment called Robot@VirtualHome [8] using the game engine Unity (see Sec. 4.2). This robot moves freely on the floor using its wheels, so the movement is planar which means that we only need to estimate its x, y and θ which correspond to the position in the plane (x, y) and the rotation (θ). From now on, the combination of position and rotation will be called a pose. In real world environments, the robot movement is not perfect as things like wheel slippage can happen, so we need an estimation of the movement of the robot, the odometry, which in this project is provided by the Unity environment.

To estimate the pose of the robot we also need to use information obtained using a calibrated stereo camera system. This system gives us 3D points of the features detected in the images coming from the cameras, and these 3D points are what are called *landmarks*. The frame of reference of the robot is a plane so, to simplify, the landmarks are assumed to be on the floor.

The system pipeline is as follows:

1. For each step the robot moves:

   1.1 Using the movement command, we add the odometry (which exhibits errors) to the graph (see Sec.4.3.1).

   1.2 The robot uses the stereo pair to detect keypoints and triangulate their positions in 3 dimensions. These detections also present inaccuracies (see Sec.4.3.2 to Sec.4.3.4).

   1.3 Each point is composed with the robot pose to obtain its position in the world reference frame.

   1.4 As each point has a descriptor, we use those descriptors to match the current observation to previous observations, creating loops in the graph (Sec.4.3.5 and Sec.4.3.6).

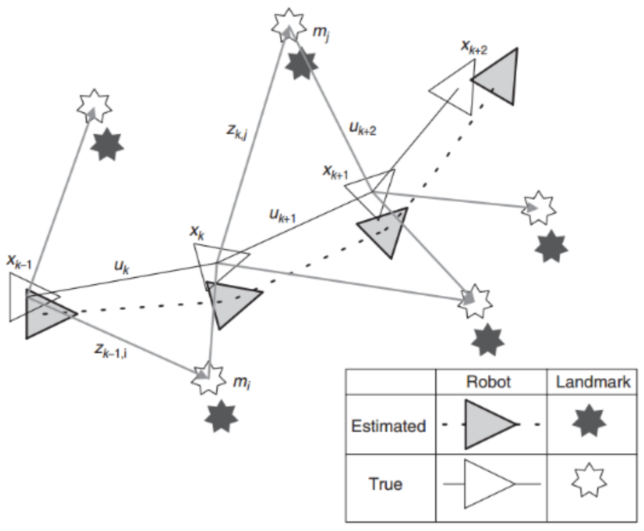


Figure 6: Pose graph [7]. Where the triangles represent the robot (X_k), stars represent observed landmarks(m_j) and the edges between them represent odometry constraints (U_k) and observations of landmarks by the robot ($Z_{k,j}$) in each instant of time k. Filled figures represent the estimated poses, where we think they are, and blank figures represent the true pose, where they actually are in the real world.

2. After the movement is done, the optimization step begins (see Sec.4.4):

   2.1 Calculate the inverse of the error's covariance matrix.

   2.2 Iterate until the result converges or the limit of iterations is reached. In each iteration:

      2.2.1 Calculate the jacobians of the error vector.

      2.2.2 Calculate delta (the change which is going to be applied to the estimated poses) using the inverse of the covariance as weight.

      2.2.3 Subtract delta to the estimated poses.

4.2 The virtual environment

To generate the dataset of stereo images and robot poses used during the system development, we modified an existing virtual environment called Robot@VirtualHome [8] (see Fig.7).

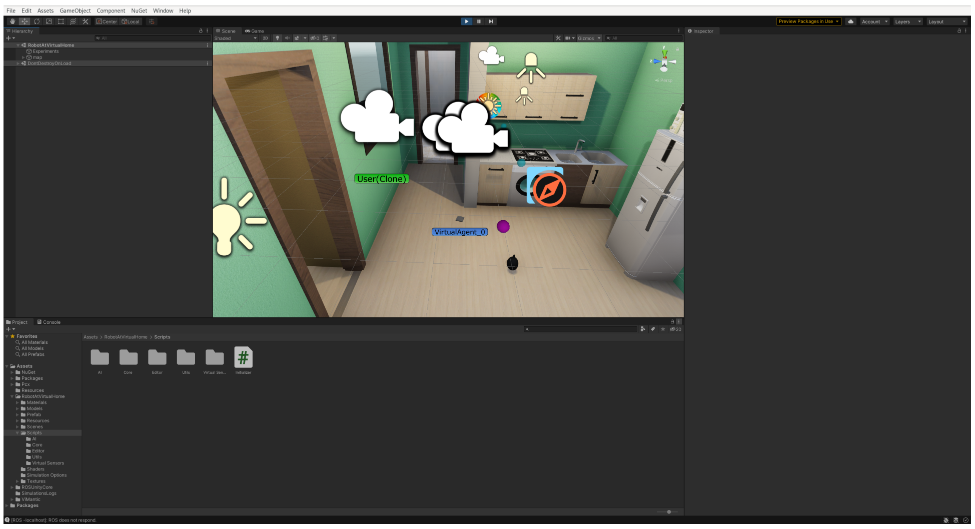


Figure 7: A screenshot of the unity editor during a simulation using the Robot@VirtualHome environment.

This environment offers up to 30 virtual houses, these houses have been designed based on real world resources (plans, images, point clouds, etc.) from houses hosted in Idealista.com. To transverse the environment Robot@VirtualHome offers multiple types of virtual agents:

- **Wanderer:** The robot visits every room in the house, the order can be specific or random.

- **Grid:** Multiple nodes are generated in the house where the robot will be placed to perform a 360º turn capturing data.
- **Manual:** The robot can be controlled using a keyboard, namely **w, a, s, d** to move and **q, e** to turn the camera horizontally.

To be able to capture data, the robot must be equipped with a sensor. Robot@VirtualHome includes two different sensors:

- **Smart camera:** This camera can capture intensity and depth images.
- **Laser scanner:** A virtual scanner that, unlike real world lasers, can measure to infinity. This can be limited to make them more realistic, the field of view and resolution can also be configured.

Robot@VirtualHome also provides already built robots, they are in the *Prefab* folder. These robots come with one *Smart Camera* by default but, as we want to generate a stereo dataset, we need to add another to the virtual agent prefab to form a stereo pair. This stereo pair has been modeled after some industry common products to provide a more realistic setup. The main parameter is the baseline (the distance between the cameras) that will be used in section 4.3.4. Our choice for the baseline is 20cm, close to the 17.5cm of the Zed 2 [20] for example.

As the system is not prepared for robots with multiple cameras, the script of the virutal agents also has to be modified to capture images with both cameras. The result is a moving robot which navigates around a house and captures stereo images and its position when getting said images (see Fig.8).

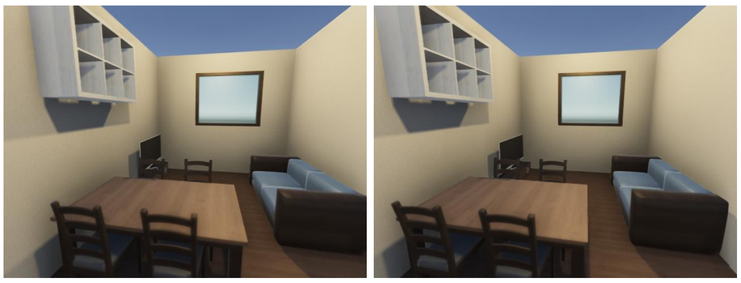


Figure 8: A pair of images taken by the stereo pair at the same instant of time during a simulation of the Robot@VirtualHome environment.

4.3 Frontend

4.3.1 Odometry

The frontend builds the graph, to do so, each time the robot moves, the frontend computes the movement the robot has done to create an odometry restriction. It includes a node in the graph which represents the current robot pose and an edge (which is the restriction) between that node and the previous one representing the movement command given to the robot (see Fig.9).

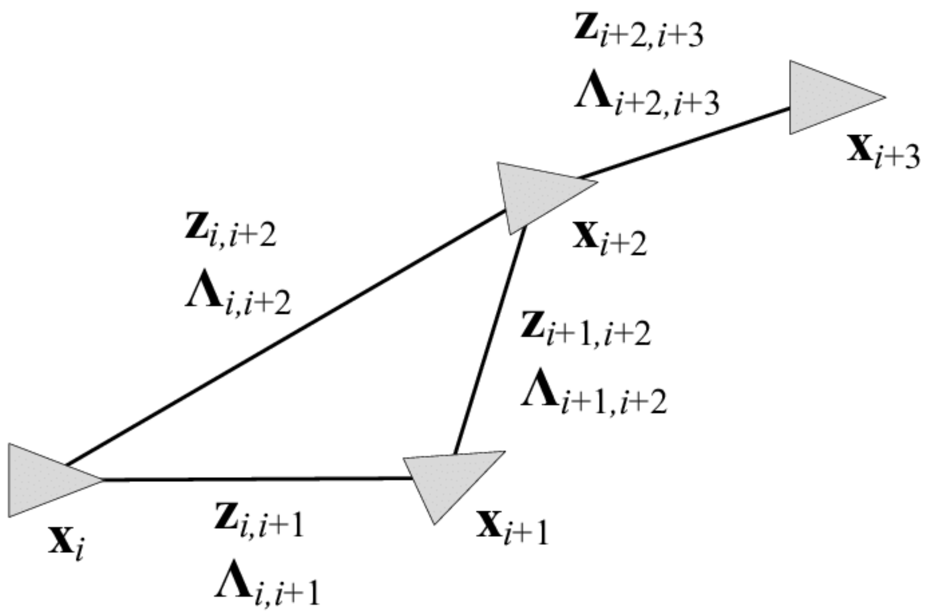


Figure 9: Pose graph with only robot poses and no landmarks [21], the triangles represent the different robot poses and the edges represent the odometry restrictions.

The movement in the real world is not perfect, things like wheel slippage can happen making the movement not perfect. To take this into consideration we have to model the uncertainty of the robot pose. The pose is represented as a random variable (X_t) that follows a normal distribution.

$$X_t \sim \mathcal{N}(\bar{x_t}, \Sigma x_t)$$

Since the data we are working on comes from a simulation, the movement is perfect so the true robot pose and the odometry pose are exactly the same (and the project would finish!). To consider a more realistic scenario, some noise is added to the movement command, making the true pose the same as in the simulated environment in Unity and the odometry pose different from the true pose. This movement command (u_t or Δx_t) also follows a normal distribution where the covariance of the movement (Σ_{u_t}) is the noise we have added.

$$u_t \sim \mathcal{N}(\bar{u}, \Sigma u_t)$$

As the robot moves the uncertainty keeps growing, so we have to calculate the new covariance every time the robot moves to have it available for the backend. To calculate this we have to take into account the current covariance ($\Sigma_{x_{t-1}}$) and the modeled covariance of the movement (Σ_{u_t}). This way, the covariance associated with the robot position at time instant t is retrieved by:

$$\Sigma_{x_t} \approx \frac{\partial g}{\partial x_{t-1}} \Sigma_{x_{t-1}} \frac{\partial g}{\partial x_{t-1}}^T + \frac{\partial g}{\partial \Sigma_{\Delta x_t}} \Sigma_{\Delta x_t} \frac{\partial g}{\partial \Sigma_{\Delta x_t}}^T$$

where the jacobians are:

$$\frac{\partial g}{\partial x_{t-1}} = \begin{bmatrix} 1 & 0 & -\Delta x_t \cos\theta_{t-1} - \Delta y_t \sin\theta_{t-1} \\ 0 & 1 & \Delta x_t \cos\theta_{t-1} - \Delta y_t \sin\theta_{t-1} \\ 0 & 0 & 1 \end{bmatrix}$$

$$\frac{\partial g}{\partial \Sigma_{\Delta x_t}} = \begin{bmatrix} \cos\theta_{t-1} & -\sin\theta_{t-1} & 0 \\ \sin\theta_{t-1} & \cos\theta_{t-1} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

and the function g is the composition of poses (also represented as the symbol $\oplus$) and is defined as:

$$p_1 \oplus p_{12} = \begin{bmatrix} x_1 + x_{12}\cos\theta_1 - y_{12}\sin\theta_1 \\ y_1 + x_{12}\sin\theta_1 + y_{12}\cos\theta_1 \\ \theta_1 + \theta_{12} \end{bmatrix}$$

4.3.2 Detecting keypoints

If we only had odometry restrictions the graph would be in a consistent state, to fix this we need to add a new type of restriction that is generated by observing the same landmark from multiple poses. This restriction comes from the fact that when we observe a landmark for the first time we place it in the map with the pose composition $m_i = p_i \oplus z_i$ which takes into account the current pose of the robot p_i (see Sec.4.3.6).

To get these observations z_i, the robot captures images with the stereo pair and detects keypoints (see Fig.10) using the **ORB** [30] detector from the OpenCV [16] library.

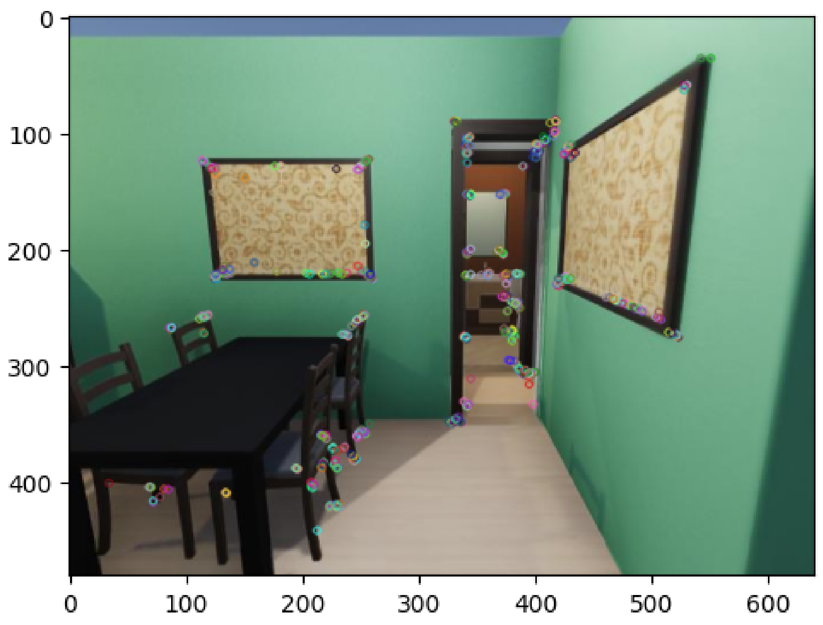


Figure 10: Image from the "ManualAgent2" dataset with keypoints detected by the ORB algorithm drawn.

The ORB detector was chosen after benchmarking several detector and descriptor combinations in a multitude of different environments. It turned to be the best one (the one finding the most keypoints) while being orders of magnitude faster (see Sec.5.1).

ORB is the combination of FAST keypoint detector [29] and BRIEF descriptor [5] with modifications to improve the performance. ORB first uses FAST to detect keypoints, then applies a Harris corner measure [13] to find the top N points as ORB only returns a limited amount of keypoints that can be configured. In this work we use the default parameter in the OpenCV library of 500 keypoints.

4.3.3 Matching the keypoints and applying RANSAC

After we have detected the keypoints seen in the images taken by the stereo pair, we need to match them to know their correspondences in the other image. To do this we use the brute force matcher of OpenCV with the hamming norm as the distance measurement as ORB uses modified BRIEF descriptors [5] which are binary strings. The hamming distance is more efficient to compute than the often used L_2 norm for other descriptors.

To improve the quality of the matches, which is primordial to ensure that the system works correctly, for each keypoint only matches that are bidirectional are considered, that is, for every keypoint in image A that has a descriptor i and every keypoint in image B that has a descriptor j, we only consider the match if j is the best match for descriptor i and viceversa. This is an alternative to the ratio test introduced by D.Lowe in the SIFT paper [22].

When we match two images usually we have matches that are not correct (because the descriptors are similar) (see Fig.11) but the landmarks they are associated to are far away from each other in the world frame, so we can use the distance between the landmarks to filter those matches.

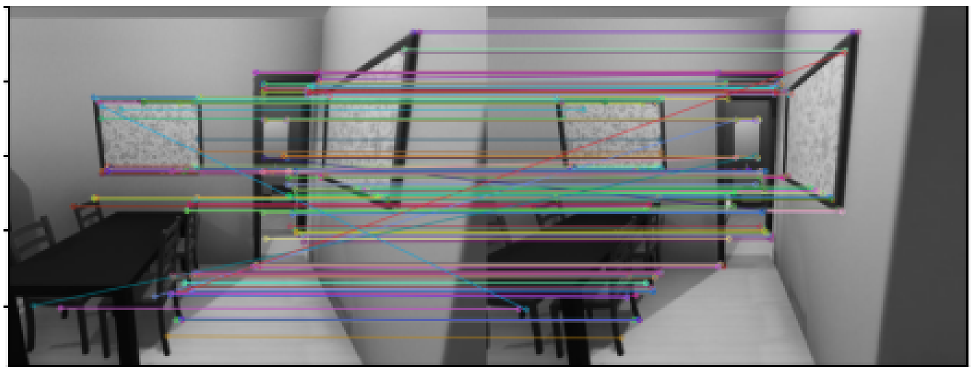


Figure 11: Matches before applying RANSAC.

But even then incorrect matches can still occur and they will negatively impact the accuracy of the estimations. To reduce the amount of bad matches even further (see Fig.12) we use an algorithm called **RANSAC** (RANdom SAmple Consensus) (see Fig.13), which is an iterative method to estimate parameters of a mathematical model from a set of observed data that contains outliers [35].

The algorithm takes a random sample from the data to compute a model. Then calculates the model parameters using samples and scores it by the fraction of inliners within a threshold

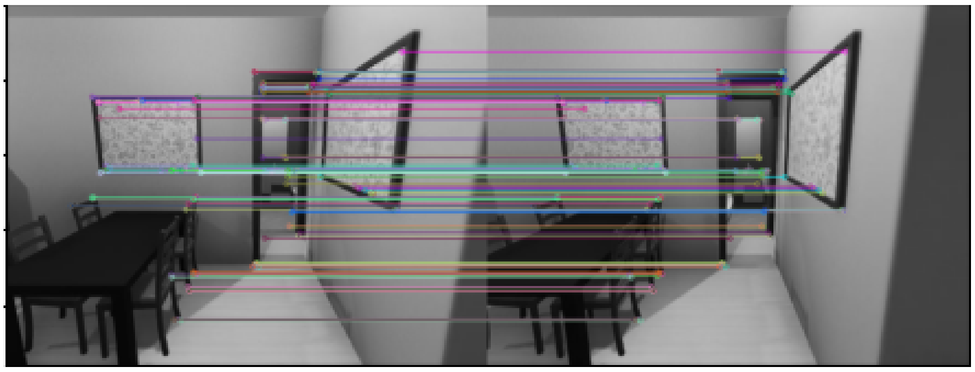


Figure 12: Matches after applying RANSAC.

of the model. This is repeated until a good enough model is found or the maximum number of iterations is reached. The data that do not fit the model (outliers) are discarded.

Algorithm 1 RANSAC

```
1: Select a random subset of the original data. This subset is called "hypothetical inliners".
2: Create a model using the previous set.
3: The rest of the data is tested against the created model. The points that fit the estimated
   model are considered to be part of the consensus set.
4: The model is accepted if enough points are classified as part of the consensus set.
5: If the model isn't good enough, go to 1 if the maximum number of iterations hasn't been
   reached yet.
```

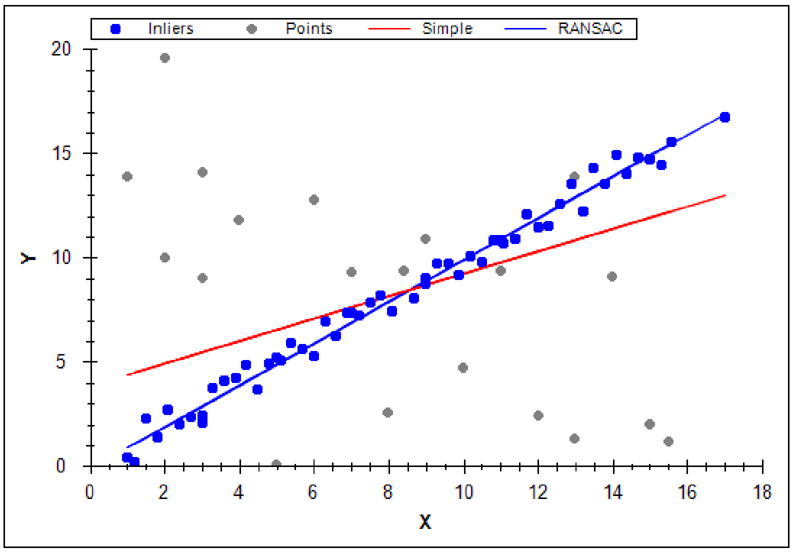


Figure 13: RANSAC example [34].

4.3.4 Triangulation

After we have detected, matched and filtered the information in the images, we need to move that information from the camera frame to the world frame for the robot to be able to use it. To do this we have to do 3D triangulation to transform the pixel coordinates of each match into 3D coordinates.

The stereo camera pair in this work exhibits an ideal configuration (see Fig.14), which is defined as:

- Identical cameras.
- Parallel axes: $Y_l = Y_r = Y$; $y_l = y_r = y$; $Z_l = Z_r = Z$.
- The right camera is along the X axis at a distance b (baseline): $X_l = X_r + b$.

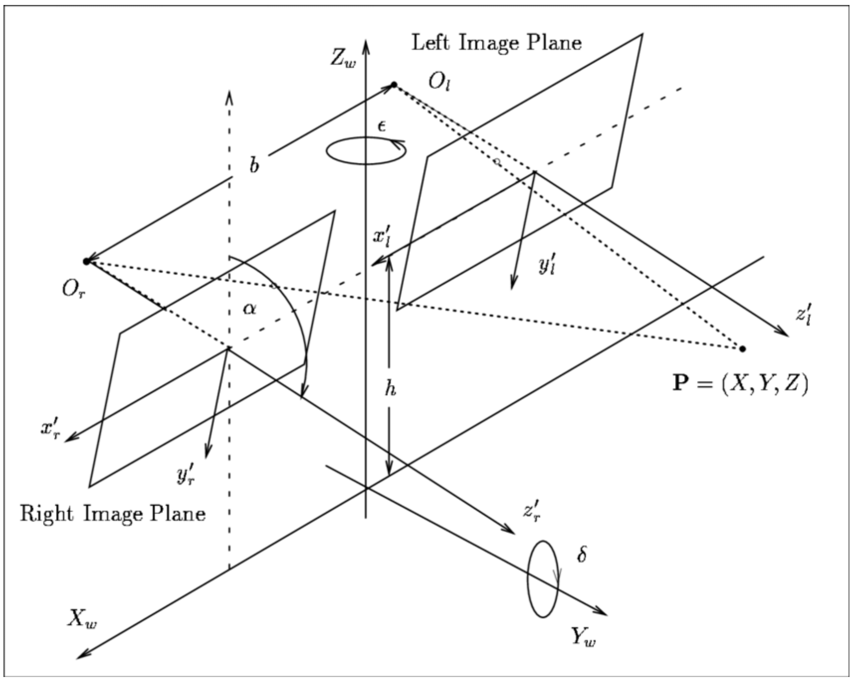


Figure 14: Stereo pair frames [41].

So now we just have to apply the following formula to get the 3D coordinates:

$$X_l = \frac{k_x b}{d_i} \begin{bmatrix} \frac{1}{k_x}(u_l - u_0) & \frac{1}{k_y}(v_l - v_0) & f \end{bmatrix}^T = \frac{b}{d_i} \begin{bmatrix} (u_l - u_0) & (v_l - v_0) & kf \end{bmatrix}^T$$

Where d_i is the disparity in pixels.

After we have the 3D points with $\begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$, we have to compose the coordinates with the robot pose to obtain where what we are seeing is in the world frame. Given that the sensor frame and the robot frame are not the same, we have to change the point to $\begin{bmatrix} Z \\ -X \end{bmatrix}$ to make the axis of the sensor frame be the same as the robot frame (see Fig.15). Also we are applying two simplifications, one is that we are only considering the point to be 2D, i.e. we are putting the point "in the floor" and we are also considering that the origin of the robot frame and the stereo pair one is the same, so there is no distance between both.

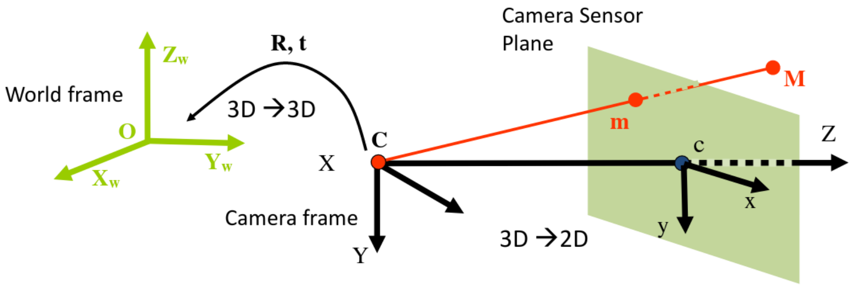


Figure 15: Representation of the world, camera and camera sensor frames and how they correspond to each other [11].

With all of this in mind we just have to apply the pose composition operator but with 2 dimensions, and we get the coordinates of the landmark in the world frame. This results:

$$m_j = p_i \oplus p_{ij} = \begin{bmatrix} x_i + x_{ij} \cos\theta_i - y_{ij} \sin\theta_i \\ y_i + x_{ij} \sin\theta_i + y_{ij} \cos\theta_i \end{bmatrix}$$

Where m_j is the landmark, p_i is the robot pose and p_{ij} is the point of the correspondence.

4.3.5 The association problem

At this point we have almost all the pieces of the puzzle, the only thing we are missing is the landmark restrictions that we mentioned earlier in section 4.3.2, which we said that we are going to obtain by seeing the same landmark from two different poses. We calculated all the landmarks in the previous step and now we only need to know which ones are the same but seen from another pose.

To do this we are going to use the descriptors associated to each point we used to generate the landmark and OpenCV to match them. This is something that have been already done before in this project, specifically when we matched keypoints in the stereo pair (see 4.3.3). When we did that we had the problem of having bad matches and we managed to solve that by using RANSAC. However, this time we can't use RANSAC because we don't have a model.

To face this issue we used the distance in a straight line between matched landmarks to filter them (euclidean distance), that is, if two matched landmarks are too far away from each other we don't consider that match. This solves the problem for matches that are very far from each other, but it doesn't take into consideration thing like walls, so we could have a bad match between landmarks that are on opposite sides of a wall. Thankfully this is not a big problem and can be safely ignored in this project.

4.3.6 Generating the error vector

At this point we have everything needed to generate the error vector required by the backend for optimization (see next section). We have two kinds of errors that we call "Odometry" errors and "Landmark" errors:

- **Odometry error**: models the difference between the motion command we sent to the robot and the relative pose between the pose before the movement and after the movement, so this errors starts as 0 and grows when we move the poses around. When we optimize them, this works as a restriction for how far the new estimated poses can be from each other. Odometry errors are calculated as:

$$e^{ij}_{odom} = \ominus p_i \oplus p_j - p_{ij} = p_j \ominus p_i - p_{ij}$$

  where $\ominus$ represents the inverse pose composition.

- **Landmark error**: this error is the difference between two observations of the same landmark, that is, if we observe the same thing twice it should have the same world coordinates, but it doesn't because the robot is not where it should be according to the motion commands we sent. The landmark error is calculated as:

$$e^{ij}_{land} = m^j - m^i = p^j \oplus z^j - p^i \oplus z^i$$

where m is the landmark, p is the robot pose and z is the observation in Cartesian coordinates (coordinates of the correspondence in the robot local frame).

To build the final error vector we just concatenate odomety and landmark errors:

$$e = \left[\dots e_{odom}^{ij} \dots e_{land}^{ij} \dots \right]^T$$

Finally, in addition to the error vector, we also need the covariance associated with those errors, which is used to weight each error individually. The covariance matrix of the observation error e_{land}^{ij} is:

$$\Sigma_{e^{ij}} = \Sigma_{m^i} + \Sigma_{m^j}$$

$$\Sigma_{m^i} = \frac{\partial m^i}{\partial p^i} \Sigma_i (\frac{\partial m^i}{\partial p^i})^T + \frac{\partial m^i}{\partial z^i} Q_i (\frac{\partial m^i}{\partial z^i})^T$$

Where Σ_i is the covariance of the robot pose and Q_i is the covariance of the sensor. Notice that the covariance of the odometry error is just Σ_i as that is the accumulated uncertainty of the movement of the robot after moving to the pose i.

4.4 Backend

The backend performs an estimation of the poses in the full trajectory of the robot. These robot poses are the nodes of the graph while the dependencies between them (recall Sec.4.3.6) are spatial constraints, each with an associated weight given by their covariance, previously calculated by the frontend and represented as edges in the graph.

To compute this estimation we want to obtain the configuration of poses $\hat{p^i}$ that minimizes the overall square error:

$$\{\hat{p^i}\} = argmin_{\{p^i\}} \left[e^T \Sigma_e^{-1} e \right]$$

Where e is the error vector and Σ_e is the covariance of the vector as formulated in the previous section.

To solve this ecuation we propose using a standar Gauss-Newton non-linar least squares optimization algorithm [12]. This approach requires a good initial estimation to start iterating until the algorithm converges. For this initial estimation we use the odometry of the robot (recall Sec.4.3.1) as it's the best approximation we have of the full robot path.

The implemented algorithm for the Gauss-Newton optimization works as follows:

Algorithm 2 Gauss-Newton optimization

```
1: $x \leftarrow$ odometry poses
2: while norm($\delta$) > tolerance and iteration < maximum_iterations do
3:     $\delta \leftarrow (J_e^T \Sigma_e^{-1} J_e)^{-1} J_e^T \Sigma_e^{-1} e$
4:     $x \leftarrow x - \delta$
5: end while
```

Delta represents the changes we have to apply to the poses. The algorithm runs until the maximum number of iterations is reached or until it has converged, i.e. the norm of delta is bigger than a given tolerance value.

In each step we need Σ_e^{-1} which is the inverse of the covariance of the vector error, which is just a matrix where every position is 0 and along the diagonal there is a covariance matrix for each error and it is defined as the inverse of:

$$\Sigma_e = \begin{bmatrix} \Sigma_{x_k} & 0 & \dots & 0 \\ 0 & \Sigma_{x_{k+1}} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \Sigma_{e^{ij}} \end{bmatrix}$$

We also need J_e which is the jacobian of the vector error with respect to the estimated poses and represents how the errors change when the poses change. It is defined as:

$$J_e = \begin{bmatrix} \frac{\partial p_{ij}}{p_i} & \frac{\partial p_{ij}}{p_j} & \dots & 0 \\ 0 & \frac{\partial p_{jk}}{p_j} & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \frac{\partial m_x}{\partial p_y} \end{bmatrix}$$

We have two kinds of jacobians, one for each type of error, and are calculated as follows:

- Odometry error jacobians:

$$\frac{\partial p_{ij}}{p_j} = \begin{bmatrix} \cos\theta_i & \sin\theta_i & 0 \\ -\sin\theta_i & \cos\theta_i & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\frac{\partial p_{ij}}{p_i} = \begin{bmatrix} -\cos\theta_i & -\sin\theta_i & -(x_j - x_i)\sin\theta_i + (y_j - y_i)\cos\theta_i \\ \sin\theta_i & -\cos\theta_i & -(x_j - x_i)\cos\theta_i + (y_j - y_i)\sin\theta_i \\ 0 & 0 & -1 \end{bmatrix}$$

- Landmark error jacobian:

$$\begin{bmatrix} 1 & 0 & -x_j\sin\theta_i - y_j\cos\theta_i \\ 0 & 1 & x_j\cos\theta_i - y_j\sin\theta_i \\ 0 & 0 & 1 \end{bmatrix}$$

5 Results

The main priorities when developing a SLAM system are achieving a high accuracy and a low execution time. On the one hand, to test the accuracy we used the datasets we generated using the Robot@VirtualHome [8] environment (see Fig.16). This environment consists of 30 virtual houses modeled from real houses' resources (plans, images, point clouds, etc.) taken from Idealista.com where a mobile robot and their sensors can be simulated. This way, a robot can move from point to point providing us with the ground truth of its pose, so it is possible to determine how accurate the SLAM system estimations are. On the other hand, to evaluate execution times a computer with an Intel®Core©i5-8400 at 4GHz and 16GB DDR4 RAM at 2400 MHz is used.

Figure 16: Picture of the environment of the dataset "Home05".

As for the system configuration, for every execution in this section we have used a tolerance of 1×10^{-5} and a maximum number of iterations of 100 for the Gauss-Newton optimization (recall Sec.4.4). Every image of the dataset has a resolution 640x480 pixels, and as the cameras are ideal their intrinsic parameters are:

$$\begin{bmatrix} k_x & 0 & u_0 \\ 0 & k_y & v_0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 608.1718 & 0 & 320 \\ 0 & 667.2514 & 240 \\ 0 & 0 & 1 \end{bmatrix}$$

The focal length (f) is 2.28 and the stereo pair has a baseline (b) of 20cm. To simplify the problem we assume that the cameras have no distortion.

5.1 Feature detection

There are many detector and descriptors that can be used to obtain the landmarks of the environment (recall Sec.4.3.2). In this section we are going to analyze a few combinations of them using pairs of stereo images from our datasets (see Fig.17) in order to select the more suitable one to be used in the project.

Different descriptors are invariant to different things like changes of lighting, scale or rotation. Choosing an adequate algorithm is essential for the overall accuracy of the system, as the more information it has from the environment the better the approximations are going to estimate. The combinations that we are going to test are: Harris [13] + NCC (Normalized cross correlation), Harris + ORB (descriptor), ORB [30] and SIFT[22]. The last two are used for both detecting and describing features.

The first metric that we are going to analyze is the number of keypoints detected. Ideally, the more information gets detected in the images, the better, as it is going to let us detect landmarks in the environment that are absolutely necessary for the system to work. The results of this experiment are shown in Tab.1. As we can see both combinations using the Harris detector perform very poorly on all the images. On the other hand ORB is the clear winner being the one that detects the higher number of keypoints with SIFT falling a bit behind in most datasets and having a much worse performance in the home05 dataset.

Next, we are going to analyze the number of matches between keypoints. For that we resort to the hamming distance for the ORB-based descriptors, the norm L2 for the SIFT one

(a) Home02 dataset.

(b) Home05 dataset.

(c) Home23 dataset.

(d) ManualAgent2 dataset.

Figure 17: Left camera images of the different datasets used in this section.

	Harris + NCC	**Harris + ORB**	**ORB**	**SIFT**
home02	20	20	500	348
home05	11	11	313	90
home23	9	9	125	58
ManualAgent2	24	24	500	460
Average	16	16	359.5	239

Table 1: Average number of keypoints detected in the pair of images taken from various datasets using different detector + descriptor combinations.

(also know as the euclidean norm) as it has been shown to produce the best results for this algorithm, and for the NCC descriptor where we have to use small patches of the image and try to match the template in the other image. We also only take the best match for each keypoint except for SIFT where we use the K best matches with a K value of 2. This metric is also extremely important as these matches let's us triangulate the 3D position of the observed keypoints, being what ultimately creates the restriction between the robot poses confirming that we are seeing the same thing from different poses.

	Harris + NCC	**Harris + ORB**	**ORB**	**SIFT**
home02	14	18	323	144
home05	5	7	116	29
home23	9	7	69	26
ManualAgent2	16	18	273	240

Table 2: Number of matches detected in images taken from various datasets using different detector + descriptor combinations.

The Tab.2 reports the output of this test. Again the ORB algorithm is the best performer by a landslide having the highest amount of matches in every dataset.

Now, we are going to analyze the time execution it takes for each algorithm to compute these results. This is relevant since even if an algorithm is the best at detecting and matching features, if it takes too long, it would not be suitable for a SLAM system. First we see the time it takes for each algorithm to detect the keypoints in Tab.3. ORB is simply faster by magnitudes of difference, while the other three combinations are close in time with SIFT being the worst by a slight margin.

	Harris + NCC	**Harris + ORB**	**ORB**	**SIFT**
home02	3.29e-02	3.14e-02	8.95e-05	1.42e-01
home05	6.01e-02	5.98e-02	9.68e-05	2.68e-01
home23	7.44e-02	7.39e-02	2.14e-04	3.94e-01
ManualAgent2	2.69e-02	2.64e-02	6.00e-05	1.27e-01

Table 3: Time taken in seconds to detect keypoints in seconds in images taken from various datasets using different detector + descriptor combinations.

And lastly, the final test we are going to perform is to see the time it takes to compute the matches between the images, which results are reported in Tab.4. This time the only combination that performs noticeably worse than the rest is the Harris + NCC algorithm, while Harris + ORB, ORB and SIFT have similar performances each of them being the fastest depending on the dataset.

	Harris + NCC	**Harris + ORB**	**ORB**	**SIFT**
home02	1.83e-02	2.56e-05	1.66e-05	4.31e-06
home05	2.63e-02	5.71e-06	4.66e-06	5.17e-06
home23	1.19e-02	5.71e-06	2.17e-06	4.23e-06
ManualAgent2	1.78e-02	3.33e-06	3.77e-06	7.13e-06

Table 4: Time taken in seconds to match descriptors in seconds in images taken from various datasets using different detector + descriptor combinations.

After all the benchmarks we have a clear winner, that is the ORB algorithm as it's the best detector that generates the most amount of matches while also taking the least amount of time overall executing.

5.2 Graph optimization performance

The overall time to optimize the graph has a big variance mainly depending on how many iterations are needed to converge. Here we present a table with the real time (see Tab.6) and CPU time (see Tab.7) needed to perform the various steps in the optimization process.

The information showed in these tables have been obtained when working with the different datasets with the parameters shown in Tab.5. It can be noticed that the amount of odometry errors is equal to the amount of poses to optimize as the initial pose doesn't need to be optimized as it has no error and there is a move command between every pose and the immediate previous one.

So overall the backend takes less than a second to perform the optimization step (see Tab.6), as we can see there is no major bottle neck in any specific step as there isn't any that is taking an unacceptable amount of time. The performance of the system is mainly affected by the amount of iterations it takes to converge.

	Iterations to converge	**Poses to optimize**	**Landmark errors**	**Error covariance matrix size**	**Maximum distance between matched landmarks**
home02	12	36	388	884 x 884	0.5
home05	9	26	108	294 x 294	0.5
home23	24	36	271	650 x 650	5
ManualAgent2	6	13	431	901 x 901	5

Table 5: Parameters used for the execution of each dataset.

	Full process	**Calculate inverse covariance matrix**	**Average time calculating jacobians**	**Average time calculating delta**
home02	0.86s	0.016s	0.096s	0.003s
home05	0.19s	0.0018s	0.003s	0.0005s
home23	1.27s	0.008s	0.0015s	0.0074s
ManualAgent2	0.355s	0.017s	0.0079s	0.0045s
Average	0.67s	0.0107s	0.0271s	0.0039s

Table 6: Real time in seconds taken to compute each part of the backend in each dataset.

	Full process	**Calculate inverse covariance matrix**	**Average time calculating jacobians**	**Average time calculating delta**
home02	5.08s	0.084s	0.057s	0.016s
home05	1.122s	0.0099s	0.018s	0.003s
home23	7.54s	0.042s	0.0088s	0.044s
ManualAgent2	2.054s	0.089s	0.046s	0.0242s
Average	3.949s	0.056s	0.032s	0.0218s

Table 7: CPU time in seconds taken to compute each part of the backend in each dataset.

5.3 Pose estimation accuracy

As the main goal of a Visual SLAM system is to estimate the trajectory of the robot, a study on the accuracy of the estimations is necessary. To do this we can use the datasets we generated with the Robot@VirtualHome environment as it contains the ground truth of the robot path and we can compare it to our estimations. The most popular metrics to test the accuracy of SLAM systems are ATE (Absolute Trajectory Error) and RPE (Relative Pose Error).

The Absolute Trajectory Error (ATE) measures the difference between points of the true and the estimated path, this metrics takes into account the global consistency. ATE is generally a good metric to measure the performance of visual SLAM systems. On the other hand, the Relative Pose Error (RPE) computes the error in the relative motion between pairs of poses. RPE is used to measure the drift of a visual odometry system, as our odometry comes from wheel odometry the most interesting metric we can use is ATE.

Usually, to be able to compute the ATE a pre-processing step is needed to associate the ground truth poses to the estimated poses using the timestamps. This is done because the trajectories might be expressed in different reference frames but, as our data comes from a simulation, it is expressed in the same reference frame so this step in unnecessary.

To calculate the ATE, each pose $P_i \in SE(3)$ is assigned to the corresponding ground truth pose $Q_i \in SE(3)$ based on the timestamp values. $SE(3)$ is the group of all rigid transformations in $\mathbb{R}^3$. Then, since both trajectories can be expressed w.r.t. arbitrary coordinate frames, they are aligned using the rigid-body transformation S that can be obtained using the Horn method [27].

The ATE is defined as the root mean square error (RMSE):

$$RMSE = (\frac{1}{n}\sum_{i=1}^{n} ||trans(E_i)||^2)^{\frac{1}{2}}$$

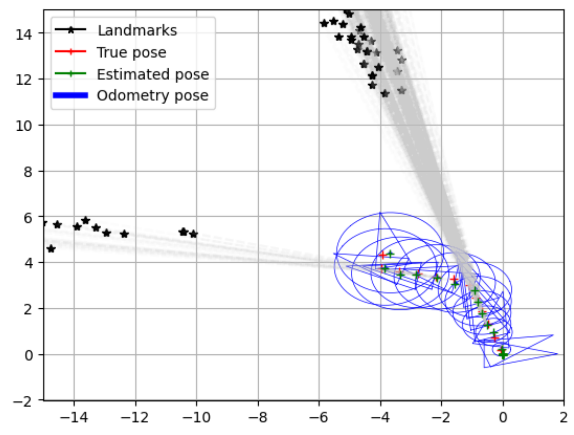
where E_i is the absolute trajectory error matrix defined as:

$$E_i := Q_i^{-1} S P_i$$

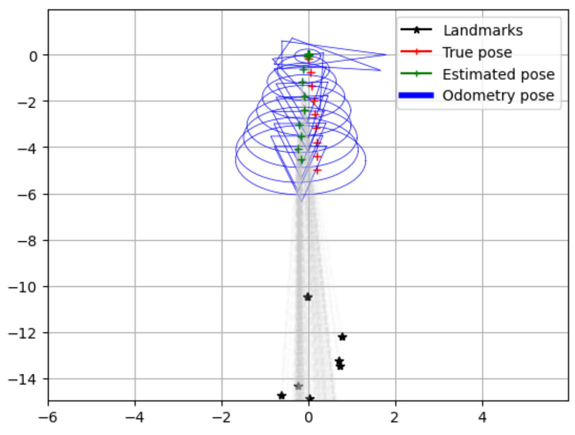
and $trans(M) := t$ returns the translation components t of matrix M.

With this metric we have tested the datasets home02 and home05 with a maximum distance between matched landmarks of 0.5 and the datasets home23 and ManualAgent2 with a maximum distance of 5 (for a visual representation of the trajectories of the robot in each dataset see Fig.18).

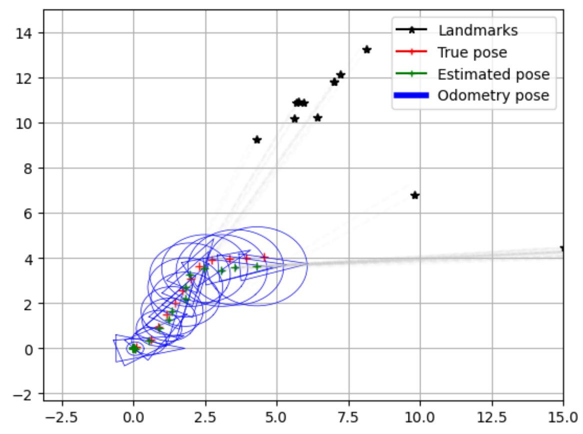
As we can see in the Tab.8, the average error between the estimated poses and the ground truth for every dataset is lower than the ATE of the initial odometry estimation, showing that the optimization step increases the precision of the estimations obtained by the odometry.



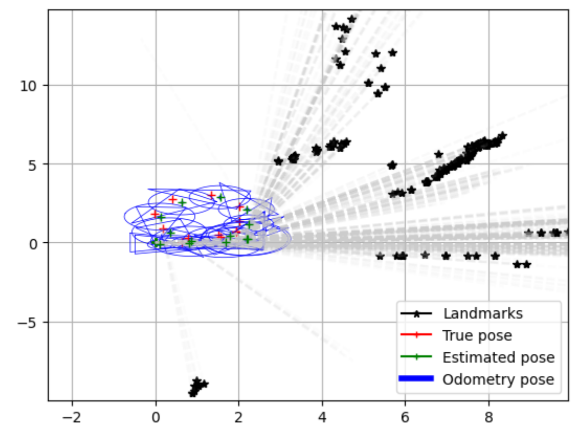
(a) Home02 dataset.



(b) Home05 dataset.



(c) Home23 dataset.



(d) ManualAgent2 dataset.

Figure 18: Paths of the robot while navigating the different datasets used in this section.

	home02	**home05**	**home23**	**ManualAgent2**
ATE Before optimization	0.135	0.192	0.164	0.095
ATE After optimization	0.116	0.182	0.148	0.082

Table 8: ATE (Absolute Trajectory Error) of the approximations of the datasets in meters before and after graph optimization.

6 Conclusions and future work

6.1 Conclusions

This bachelor's thesis proposes a Visual SLAM system based on pose graph optimization with the objective to be used as a pedagogical resource in robotics and computer vision related courses at the University of Málaga.

This document has first described the typical components that form a Visual SLAM system, also offering different state of the art approaches for building them. Then, we have modified the Robot@VirtualHome [8] environment to generate datasets of a robot traversing through multiple real word based houses capturing images with a pair of stereo cameras to feed and test the system. After that, we considered multiple open source implementations for the feature extraction part of the system. The performance of the feature extraction component is extremely important for the overall accuracy of the system, as well as its execution time. Specifically we decided to implement a solution based on the ORB [30] detector and descriptor as it proved to be the one that produced the most keypoints and matches while also not taking an unacceptable amount of time to process the images (see Sec.5.1).

Next, we started implementing the backend using a Gauss-Newton optimization to optimize the poses of the graph and also developed a simple frontend which was the base for the rest of the implementation. The backend was tested using simple circular paths to prove that it worked correctly. After that we implemented the final version of the frontend based on the aforementioned ORB algorithm. It was detected that the system suffers plenty from bad information due to bad matches between descriptors, so two solutions were implemented: RANSAC [35] when we had a model, and a solution based on the distance between landmarks otherwise.

Lastly, we have benchmarked the system to test the accuracy of the estimated poses and the execution time of the system. We weren't able to find any bottle necks in the performance as the time it takes grows mainly with the amount of iterations the algorithm takes to converge, which largely depends on the dataset used, that is, the features in the scene and their distribution.

6.2 Future work

Even though the project has reached its main goal, there's still room for improvement in certain areas like:

- Better reduction of incorrect matches. As commented, the algorithm suffers a lot from incorrect loops in the graph which are caused by incorrect matches. Because of this we have to be very restrictive with our matches, so to accomplish a better reduction of incorrect matches things like Huber loss [15] or M-estimator [14] could be implemented.
- In this project every frame from the stereo pair is being considered, but in a real world application where a camera might record new observations 30 times per second only *keyframes* should be considered to reduce the number of nodes in the graph, as each frame represents another pose to optimize.
- Loop closure. If we go back to the same pose we have previously been to, this algorithm considers them as two different poses, but we could add loop closure there if we were able to recognize them as the same. To do this we could use a Bag of Words approach like DBoW2 [9].

7 Conclusiones y lineas futuras

7.1 Conclusiones

Este trabajo de fin de grado propone un sistema de SLAM Visual basado en optimización de grafo de poses con el objetivo de ser usado como un recurso pedagógico en clases de la Universidad de Málaga relacionados con la visión por computador y la robótica.

Este documento primero ha descrito los componentes típicos que forman parte de un sistema de SLAM Visual, también ofreciendo diferentes enfoques del estado del arte para desarrollarlos. Después, hemos modificado el entorno Robot@VirtualHome [8] para generar datasets de un robot recorriendo múltiples casas basadas en el mundo real capturando imágenes con un par de cámaras estéreo para probar el sistema. Después de eso, hemos considerado diferentes implementaciones open source para la parte de extracción de features. El rendimiento del componente de extracción de features es extremadamente importante para la precisión general del sistema, además de su tiempo de ejecución. Específicamente hemos decidido implementar una solución basada en el detector y descriptor ORB [30], ya que demostró ser el que producía más keypoints y correspondencias a la vez de no tardar una cantidad demasiado elevada de tiempo para procesar las imágenes (ver Sec.5.1).

Después, empezamos a implementar el backend usando una optimización de Gauss-Newton para optimizar el grafo de poses y también desarrollamos un frontend simple que fue la base para el resto de la implementación. El backend fue testeado usando una trayectoria ciruclar simple para probar que funcionaba correctamente. Después de eso implementamos la versión final del frontend basada en el algoritmo antes mencionado, ORB. Se detectó que el sistema sufría mucho por la información incorrecta debida a malas correspondencias entre descriptores, así que dos soluciones fueron implementadas: RANSAC [35] cuando teníamos un modelo y una solución basada en la distancia entre landmarks cuando no.

Por último, hemos probado el sistema para comprobar la precisión de la estimación de las poses y el tiempo de ejecución del sistema. No fuimos capaces de encontrar ningún cuello de botella en el rendimiento, ya que el tiempo de ejecución crece, principalmente, con el número de iteraciones que el algoritmo necesita para converger, lo cual depende del dataset usado, es decir, de las features de la escena y su distribución.

7.2 Lineas futuras

Aunque el proyecto ha cumplido su objetivo principal aún hay posibilidad de mejora en algunas áreas como:

- Mejor reducción de correspondencias erróneas. Como hemos comentado, el algoritmo sufre mucho debido a los bucles incorrectos en el grafo que son causados por las correspondencias erróneas. Por esto tenemos que ser muy restrictivos con las correspondencias, por lo que para conseguir una reducción de las correspondencias erróneas técnicas como Huber loss[15] o M-estimator [14] pueden ser implementadas.

- En este proyecto todas las imágenes del par estéreo son consideradas, pero en una aplicación del mundo real donde una cámara puede capturar nuevas observaciones 30 veces por segundo solo se deberían considerar los *keyframes* para reducir el número de nodos en el grafo, ya que cada imagen representa una pose más a optimizar.

- Cierre de bucle. Si volvemos a una pose que ya hemos visitado anteriormente este algoritmo las considera como dos poses distintas, pero podríamos añadir un cierre de bucle en ese caso si fuéramos capaces de reconocer que ambas son la misma pose. Para esto podemos usar un enfoque basado en una bolsa de palabras como DBoW2 [9].

8 References

[1] Amazon. Robot astro. `https://www.amazon.com/-/es/Presentamos-Amazon-Astro`. [Online; accessed: 2022-06-11].

[2] Andreas Kloeckner. Pudb. `https://pypi.org/project/pudb/`, 2022. Version 2022.1.1.

[3] Herbert Bay, Andreas Ess, Tinne Tuytelaars, and Luc Van Gool. Speeded-up robust features (surf). *Computer Vision and Image Understanding*, 110(3):346–359, 2008. Similarity Matching in Computer Vision and Multimedia.

[4] Mordechai Ben-Ari and Francesco Mondada. *Robotic Motion and Odometry*, pages 63–93. Springer International Publishing, Cham, 2018.

[5] Michael Calonder, Vincent Lepetit, Christoph Strecha, and Pascal Fua. Brief: Binary robust independent elementary features. In *Eur. Conf. Comput. Vis.*, volume 6314, pages 778–792, 09 2010.

[6] Davison. Real-time simultaneous localisation and mapping with a single camera. In *Proceedings Ninth IEEE International Conference on Computer Vision*, pages 1403–1410 vol.2, 2003.

[7] Marios Xanthidis et al. Slam tutorial slides. eth zurich. `http://www.cs.columbia.edu/~allen/F17/NOTES/slam_pka.pdf`. [Online; accessed 12-June-2022].

[8] D. Fernandez-Chaves, J.R. Ruiz-Sarmiento, Alberto Jaenal, N. Petkov, and J. Gonzalez-Jimenez. Robot@virtualhome, an ecosystem of virtual environment tools for realistic indoor robotic simulation. *Expert Systems with Applications*, 2022 (To appear).

[9] Dorian Gálvez-López and J. D. Tardós. Bags of binary words for fast place recognition in image sequences. *IEEE Transactions on Robotics*, 28(5):1188–1197, October 2012.

[10] J. Gonzalez-Jimenez and J.R. Ruiz-Sarmiento. Apuntes de la asignatura robotica, ingenieria informatica. universidad de malaga, 2021.

[11] J. Gonzalez-Jimenez and J.R. Ruiz-Sarmiento. Apuntes de la asignatura vision por computador, ingenieria informatica. universidad de malaga, 2021.

[12] Serge Gratton, Amos Lawless, and Nancy Nichols. Approximate gauss–newton methods for nonlinear least squares problems. *SIAM Journal on Optimization*, 18:106–132, 01 2007.

[13] Christopher G. Harris and M. J. Stephens. A combined corner and edge detector. In *Alvey Vision Conference*, 1988.

[14] F. Hayashi. *Econometrics*. Amsterdam University Press, 2000.

[15] Peter J. Huber. Robust Estimation of a Location Parameter. *The Annals of Mathematical Statistics*, 35(1):73 – 101, 1964.

[16] Willow Garage Intel Corporation. OpenCV about. `https://opencv.org/about/`, 2022. [Online; accessed: 2022-05-26].

[17] iRobot. Robots aspiradores roomba®. (s. f.). `https://www.irobot.es/roomba`. [Online; accessed: 2022-06-11].

[18] A. Kaehler and G. Bradski. *Learning OpenCV 3: Computer Vision in C++ with the OpenCV Library*, page 25. O'Reilly Media, Inc., 2016.

[19] A. Kaehler and G. Bradski. *Learning OpenCV 3: Computer Vision in C++ with the OpenCV Library*, page 9. O'Reilly Media, Inc., 2016.

[20] Stereo labs. Zed 2 - ai stereo camera | stereo labs. `https://www.stereolabs.com/zed-2/`. [Online; accessed: 2022-05-26].

[21] Donghwa Lee and Hyun Myung. Solution to the slam problem in low dynamic environments using a pose graph and an rgb-d sensor. *Sensors (Basel, Switzerland)*, 14:12467–12496, 07 2014.

[22] D.G. Lowe. Object recognition from local scale-invariant features. In *Proceedings of the Seventh IEEE International Conference on Computer Vision*, volume 2, pages 1150–1157 vol.2, 1999.

[23] L.A. McGee and S. F. Schmidt. *Discovery of the Kalman Filter as a Practical Tool for Aerospace and Industry*. NASA technical memorandum. National Aeronautics and Space Administration, Ames Research Center, 1985.

[24] R. Mur-Artal, J. M. M. Montiel, and Juan D. Tardós. ORB-SLAM: a versatile and accurate monocular SLAM system. *IEEE Transactions on Robotics*, 31(5):1147–1163, 2015.

[25] Otsaw Digital. Camello | delivery robot. `https://otsaw.com/camello/`, 2013. [Online; accessed 15-June-2022].

[26] PAL Robotics. Reem-c. `https://pal-robotics.com/es/robots/reem-c/`, 2013. [Online; accessed 15-June-2022].

[27] David Prokhorov, Dmitry Zhukov, Olga Barinova, Konushin Anton, and Anna Vorontsova. Measuring robustness of visual slam. In *2019 16th International Conference on Machine Vision Applications (MVA)*, pages 1–6, 2019.

[28] Fernando Pérez and Brian Granger. Project jupyter | about us. `https://jupyter.org/about`, 2022. [Online; accessed 15-June-2022].

[29] Edward Rosten and Tom Drummond. Machine learning for high-speed corner detection. In *Comput Conf Comput Vis*, volume 3951, 07 2006.

[30] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. Orb: An efficient alternative to sift or surf. In *2011 International Conference on Computer Vision*, pages 2564–2571, 2011.

[31] Davide Scaramuzza and Friedrich Fraundorfer. Visual odometry [tutorial]. *IEEE Robotics & Automation Magazine*, 18(4):80–92, 2011.

[32] Stephen Se, David G. Lowe, and J. Little. Mobile robot localization and mapping with uncertainty using scale-invariant visual landmarks. *The International Journal of Robotics Research*, 21:735 – 758, 2002.

[33] MATLAB & Simulink. Slam (localización y mapeo simultáneos) – matlab y simulink. (s. f.). https://es.mathworks.com/discovery/slam.html. [Online; accessed: 2022-06-11].

[34] C. Souza. Random sample consensus (ransac) in c# – césar souza. http://crsouza.com/2010/06/02/random-sample-consensus-ransac-in-c/, 2010. [Online; accessed 15-June-2022].

[35] Tilo Strutz. *Data Fitting and Uncertainty (2nd edition).* Springer Vieweg, 01 2016.

[36] Jürgen Sturm, Nikolas Engelhard, Felix Endres, Wolfram Burgard, and Daniel Cremers. A benchmark for the evaluation of rgb-d slam systems. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 573–580, 2012.

[37] Unity Technologies. Government & aerospace. https://unity.com/solutions/government-aerospace. [Online; accessed: 2022-06-10].

[38] Unity Technologies. Unity real-time development platform | 3d, 2d, vr & ar engine. https://unity.com/. [Online; accessed: 2022-06-10].

[39] Unity Technologies. Using directx11 in unity 4. https://web.archive.org/web/20130312140345/http://docs.unity3d.com/Documentation/Manual/DirectX11.html, 2013. [Online; Retrieved February 19, 2013].

[40] Sebastian Thrun and Michael Montemerlo. The graph slam algorithm with applications to large-scale mapping of urban structures. *The International Journal of Robotics Research*, 25(5-6):403–429, 2006.

[41] Joseph Weber, Dieter Koller, Q.-T Luong, and Jitendra Malik. An integrated stereo-based approach to automatic vehicle guidance. *IEEE International Conference on Computer Vision*, 04 1996.

[42] Ning Yu, Xiaohong Zhan, Shengnan Zhao, Yinfeng Wu, and Renjian Feng. A precise dead reckoning algorithm based on bluetooth and multiple sensors. *IEEE Internet of Things Journal*, 5(1):336–351, 2018.