1290

UNIVERSIDADE Ð COIMBRA

Francisco Carvalho Tavares

AUTOMATED PARAMETER TUNING FOR EVALUATION OF SLAM METHODS
RUST BASED APPROACH

Automated Parameter Tuning for Evaluation of SLAM Methods

Rust based approach

Supervisor:

Prof. David Portugal

Co-Supervisor:

Eng. Mário Cristovão

Jury:

Prof. Jury1

Prof. Jury2

Prof. Jury3

Dissertation submitted in partial fulfillment for the degree of Master of Science in Electrical and Computer Engineering.

5th of November, 2024

Acknowledgments

Morbi luctus, wisi viverra faucibus pretium, nibh est placerat odio, nec commodo wisi enim eget quam. Quisque libero justo, consectetuer a, feugiat vitae, porttitor eu, libero. Suspendisse sed mauris vitae elit sollicitudin malesuada. Maecenas ultricies eros sit amet ante. Ut venenatis velit. Maecenas sed mi eget dui varius euismod. Phasellus aliquet volutpat odio. Vestibulum ante ipsum primis in faucibus orci luctus et ultrices posuere cubilia Curae; Pellentesque sit amet pede ac sem eleifend consectetuer. Nullam elementum, urna vel imperdiet sodales, elit ipsum pharetra ligula, ac pretium ante justo a nulla. Curabitur tristique arcu eu metus. Vestibulum lectus. Proin mauris. Proin eu nunc eu urna hendrerit faucibus. Aliquam auctor, pede consequat laoreet varius, eros tellus scelerisque quam, pellentesque hendrerit ipsum dolor sed augue. Nulla nec lacus.

Resumo

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

Abstract

Quisque ullamcorper placerat ipsum. Cras nibh. Morbi vel justo vitae lacus tincidunt ultrices. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. In hac habitasse platea dictumst. Integer tempus convallis augue. Etiam facilisis. Nunc elementum fermentum wisi. Aenean placerat. Ut imperdiet, enim sed gravida sollicitudin, felis odio placerat quam, ac pulvinar elit purus eget enim. Nunc vitae tortor. Proin tempus nibh sit amet nisl. Vivamus quis tortor vitae risus porta vehicula.

"He who only wishes and hopes does not interfere actively with the course of events and with the shaping of his own destiny."

<div align="right">

Ludwig Von Mises

</div>

x

Contents

List of Acronyms

APE   Absolute Pose Error

ATE   Absolute Trajectory Error

BO   Bayesian Optimization

DEA   Dissertation's Example Acronym

DoG   Derivative of Gaussian

EA   Evolutionary Algorithm

EI   Expected Improvement

HB   Hyperband

HPO   Hyperparameter Optimization

HPT   Hyperparameter Tuning

ICP   Iterative Closest Point

IMU   Inertial Measurement Unit

LIDAR   Light Detection and Ranging

LoG   Laplacian of Gaussian

NDT   Normal Distributions Transform

PI   Probability of Improvement

PSO   Particle Swarm Optimization

RANSAC   Random Sample Consensus

RMSE   Root Mean Square Error

RPE	Relative Pose Error
RUSTLE	Reliable User-friendly and Straightforward Tool for Localization Experiments
SA	Simulated Annealing
SH	Successive Halving
SMBO	Sequential Model-Based Optimization
SLAM	Simultaneous Localization and Mapping
ULCB	Upper/Lower Confidence Bound
VO	Visual Odometry

List of Figures

List of Tables

1 Introduction

1.1 Context and Motivation

Simultaneous Localization and Mapping (SLAM) is one of the most studied topics in robotics [9, 10]. Its purpose is to simultaneously estimate the robot's position and orientation(pose) and map the robot's environment. SLAM methods often integrate data from devices like Light Detection and Ranging (LIDAR), cameras, or ultrasonic sensors with algorithms like Kalman filters and particle filters, or more advanced aproaches, such as Visual and LIDAR SLAM. These methods are crucial for applications in robotics, such as autonomous navigation.

The accuracy of a SLAM method is usually evaluated using the Absolute Pose Error (APE) and the Relative Pose Error (RPE). The APE refers to the absolute difference between the estimated pose and the ground truth, while the RPE refers to the difference between consecutive estimated poses and the respective ground truth consecutive poses. SLAM's efficacy is dependent on various factors, such as sensor quality, available computational resources, loop closure and algorithm type. The algorithm type is particularly important because different scenarios demand different types of SLAM methods. On the other hand, different SLAM methods have different number of hyperparameters, impacting the size of the parameter space and the difficulty of arriving at the optimal hyperparameter configuration for a given scenario.

One of the biggest hurdles in SLAM research is determining the best hyperparameter configuration for a given dataset. A tool like RUSTLE streamlines the process, allowing different SLAM methods to be run asynchronously, while giving performance reports, allowing for the manual tuning of SLAM's hyperparameters.

There is, however, a missing and crutial component from frameworks like RUSTLE: in literature, SLAM methods are usually not optimally tuned, or the comparisons are between methods that are almost optimally tuned and methods that are tuned just enough to give sufficiently

satisfactory results, and so any comparisons cannot be considered fair and conclusive. Applying hyperparameter optimization algorithms and automating the tuning process would not only relieve the user of the laborious process of manually searching the parameter space for an optimal configuration, but also allow for a more fair comparison between SLAM methods.

1.2   Main goals

The main goals of this thesis are:

- Develop an automatic Hyperparameter Tuning (HPT) framework within RUSTLE to better optimize and compare the performance of multiple SLAM solutions.

- Evaluate the efficiency of the developed optimization algorithms and compare them to each other.

- Summarize the developed work and identify lessons learned and potential future improvements.

- Evaluate the impact(importance) different hyperparameters have on several SLAM methods.

1.3   Document overview

2 Background and Related Work

This chapter goes over the topic of SLAM, its subtypes and metrics, as well as the topic of hyperparameter optimization and its specific relevance in the context of SLAM optimization. Finally, related relevant work is discussed, so as to provide a context and a starting point for the work developed in this thesis.

2.1 SLAM

SLAM is a fundamental problem in robotics and computer vision, wherein a robotic system moves around in an unknown environment, and builds a map of said environment while simultaneously determining its own position within said map [11].

SLAM systems use sensors such as LIDAR's, cameras, IMU's and GPS to collect data about the environment, which then is processed by the backend itself [11], which is implementation dependent.

Generally speaking, a SLAM method involves the following steps(not necessarily in this order):
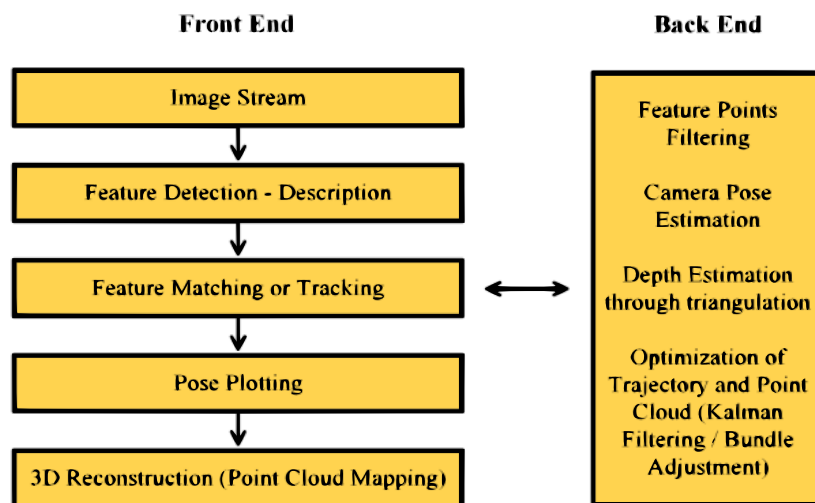

Figure 2.1: Common steps of a generic Visual SLAM system [1].

Firstly, sensor data is collected from the different sensors the system has, such as cameras, LIDARs, Inertial Measurement Unit (IMU)'s, etc. Given that most systems have sensors that collect data at different frequencies, it is critical for the data to be time-aligned. All data is timestamped relative to a common clock(system clock). This synchronization step ensures that features, poses and inertial readings from multiple sensors correspond to the same physical moment[12].

Next, features are detected. In the case of Visual SLAM, this tipically mean detecting corners, which are often present in more than one frame, allowing for later matching of the same 3D points and for the estimation of the pose of the robotic system. Popular choices for corner detectors are the Harris detectors, which detects points with large intensity changes across two perpencicular directions[13], and blob detectors(guassian or laplacian based), which detect regions(not points) where the Laplacian of Gaussian (LoG) or Derivative of Gaussian (DoG) response is extreme in scale and space[14, 15].

Feature matching is simply the process of searching for pairs of feature descriptors across different image frames that likely correspond to the same 3D scene point [16]. The simplest and most common baseline matching algorithms simply compare every descriptor in image A with every descriptor in image B and compute a distance metric(euclidian, hamming, etc), which "describes" the similarity between two descriptors. The smaller the distance, the more similar the descriptors are, and therefore, the chance the descriptors correspond to the same 3D point is higher as well [17].

Pose plotting is the process of estimating and visualizing the pose of a moving camera or robot system within a global coordinate frame over time. Each pose represents a snapshot of where the sensor was located and how it was oriented when an image or measurement was captured. By connecting these poses, one obtains a trajectory that shows the motion path through the environment. Accurate pose plots are essential for evaluating localization performance, identifying drift, and debugging SLAM algorithms. Pose plotting relies on solving geometric constraints between observed image features across frames, often using techniques such as Perspective-n-Point (PnP), bundle adjustment, or pose-graph optimization[18].

Lastly, 3D reconstruction focuses on building a geometric model of the environment using the estimated poses and visual data. Once camera poses are known, corresponding image features can be triangulated into 3D points to form a map, initially a sparse point cloud and, with additional processing, a dense point cloud[19]. This reconstructed structure represents the physical

world captured by the SLAM system. High quality 3D reconstruction depends directly on reliable pose estimates, i.e, errors in trajectory estimation propagate into geometric distortions in the map [19, 20].

2.1.1 Types of SLAM solutions

If one were to categorize SLAM solutions using the sensor types as a criteria, three broad categories would emerge: Visual SLAM(VSLAM), LIDAR SLAM and multi sensor SLAM.

Visual SLAM

Visual SLAM uses cameras to understand an environment by detecting and tracking features over time [21]. Common subtypes of visual SLAM are(according to the type of cameras they use): Monocular SLAM, which uses only one camera, Stereo SLAM, which uses 2 cameras, separated by a known baseline, and RGB-D SLAM, which uses cameras that in addition to visual and color information, also provide pixel depth data(distance from camera to an object).
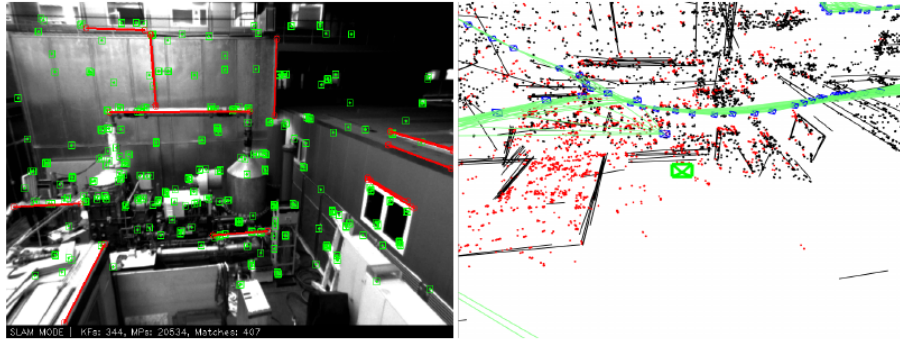
Figure 2.2: Visual SLAM [2].

In terms of hyperparameters, lets consider a few of the most important ones and their effect on the performance of the visual SLAM system.

When performing feature detection and matching, a few important parameters are: **feature matching threshold**, **feature detection sensitivity** and the **scale factor**(in the case of pyramid-based detectors). The **feature matching threshold** determines how similar two feature descriptors must be to be considered a valid correspondence [22]. It directly affects how the system associates points between frames or keyframes. A high threshold ensures that only highly similar descriptors are matched [22], which increases precision but risks losing correspondences when illumination or viewpoint changes. A low threshold increases the number of matches

but allows more outliers, which can corrupt motion estimation [22]. The **feature detection sensitivity** defines how easily the detector identifies keypoints in an image, usually based on a response strength or cornerness threshold. A higher sensitivity results in many detected features, improving robustness in textured environments, but it adds computational cost and potential noise. A lower sensitivity limits detections to the strongest points, making tracking faster but potentially unstable in low-texture areas. Lastly, the **scale factor**(only applicable for pyramid-based detectors) determine how much each image pyramid level is downsampled relative to the previous level [23]. A smaller scale factor adds more levels, improving the ability of the system to handle zooming and depth, but at the cost the aditional computational cost [23]. A higher scale factor reduces the robustness to scale variations but speeds up feature extraction [23]. In short, this parameter controls a trade-off between real time performance and scale invariant tracking.

On the topic of pose estimation, Random Sample Consensus (RANSAC) is a particularly important algorithm. Specifically, the **maximum number of iterations** and the **minimum number of data points** to fit the model are important parameters. The **maximum number of iterations** specifies how many hypotheses RANSAC should run when estimating the camera pose. A higher value increases the probability of finding a correct model, even with many outliers, but adds computational cost [24], while a lower value is too permissive when accepting points into the model, and can lead to unreliable pose estimates [24]. Its a trade-off between robustness to outliers and real time performance. The **minimum number of data points**(or minimum inlier count) controls how many features correpondences are required to accept a pose estimation as valid. A low value makes the system acccept unstable or incorrect poses, resulting in drift [24], while a high value might make tracking very hard, when very few feature matches are available [24].

Finally, on the topic of loop closure, the **loop detection threshold** is one of the most important parameters, which determines how similar two keyframes must be(using a bag of words similarity score, for instance) [25]. A high threshold only triggers loop closure on highly confident matches. This prevents some false positives but might miss some oportunities to correct drift [25]. A low threshold increases loop closure sensitivity, but can incorrectly detect loops that might corrupt the map [25].

It should be noted that these parameters are only a fraction of the total number of parameters at play in visual SLAM solutions, and are only briefly mentioned here. Also, other, more *exotic*,

less used algorithms might be used throughout the execution of the algorithm, which requires a different set of hyperparameters that those presented previously.

LIDAR SLAM

LIDAR SLAM uses, as the name suggests, LIDAR sensors, which unlike Visual SLAM, can measure distances directly using laser pulses. It is overall a more robust method for low light and featureless environments [26].
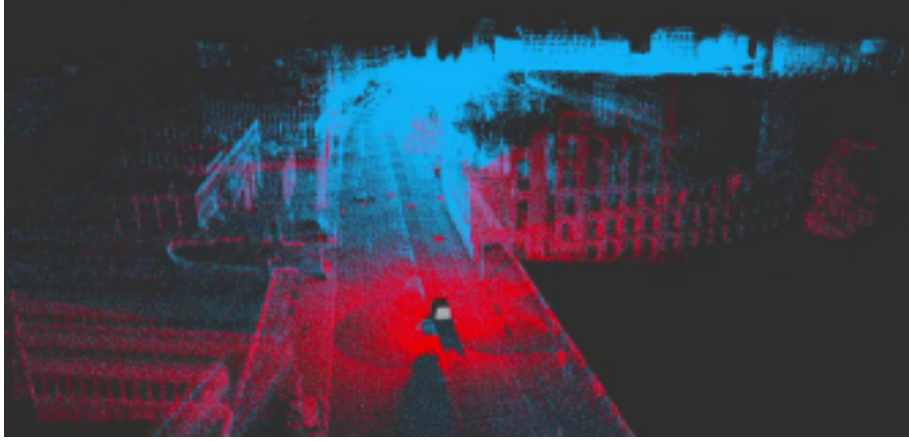

Figure 2.3: LiDAR SLAM point cloud example[3].

In terms of hyperparameters, a few of the most important hyperparameters often present in LIDAR based SLAM methods are: **voxel size**, **map resolution**, **maximum correspondence distance** and **keyframe insertion threshold**. The **voxel size** determines how much the LIDAR point cloud is downsampled before registration. A smaller value helps to preserve detail and accuracy, but sharply increases computation time and noise sensitivity [27]. Larger voxels speed up the system and are more robust to noise, but might increase drift in the process [27]. The **map resolution** affects the granularity of the internal representation(voxel grid). A high resolution captures more fine-grained features but consumes more memory and processing time [28], while a coarse map runs faster but loses detail, affecting localization accuracy[28]. The **maximum correspondence distance** determines how far two points can be from each other to be considered a match during scan registration. If too small, the algorithm(Iterative Closest Point (ICP), Normal Distributions Transform (NDT)) may fail to produce enough correspondences, leading to unstable pose estimations**wang2023comparative**. On the other extreme, if the value set is too high, then incorrect matches will introduce more drift and distort the produced map [29]. The **keyframe insertion threshold**(based on on distance, for example) controls how often new keyframes are created. Low thresholds yield many keyframes, increasing

accuracy but slowing optimization [30], while high thresholds risk under sampling, introducing drift and lowering loop closure detection [30].

Multi sensor SLAM

Multi sensor SLAM is simply a category for the SLAM solutions which make use of several different types of sensors, taking advantage of the strengths of each one, making the final map and trajectories more robust than if each sensor was used on a independent SLAM system.
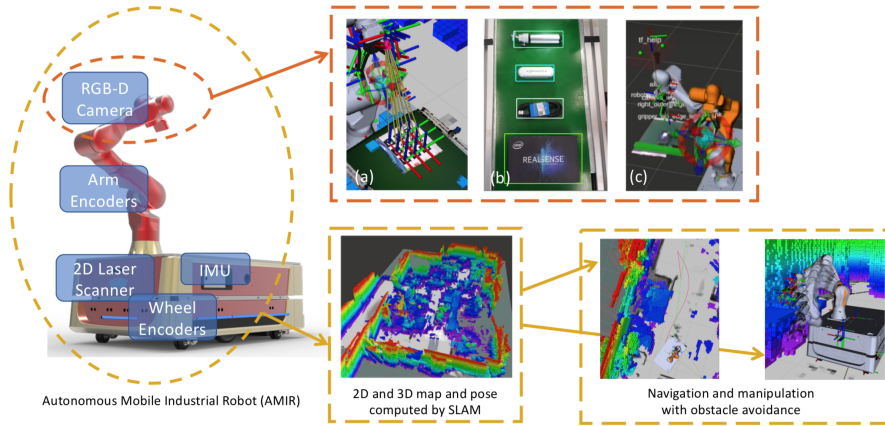

Figure 2.4: Multi sensor SLAM system for an industrial robot[4].

A few common multi sensor SLAM solutions include: visual inertial SLAM [31], Lidar inertial SLAM [32] and visual-lidar-inertial SLAM [33]. **Visual inertial SLAM** combines the good spatial information of a camera with the higher temporal resolution of the IMU to make up for the other sensor's weaknesses. When the IMU accumulates too much drift, the cameras help correct it, and the IMU helps when vision is low. Its main disadvantage is the high sensitivity to visual degradation, such as fog and dark areas. **Lidar-Inertial SLAM**'s main advantage is the high robustness in low light or outdoor environments and the resilience to visual noise, present in nightly, foggy and even dusty environments. Two main limitations are the more expensive/heavier setups and the limitation in texture representation(pure geometry only). In **Visual-Lidar-Inertial SLAM**, rich visual features, accurate range data and motion tracking are all present. It however requires a complex calibration and synchronization process, being more expensive and requiring more computational power than simpler multi sensor approaches to SLAM.

In terms of hyperparameters, multi sensor SLAM's paramaters are largely an amalgamation of their individual sensor constituents.

2.1.2 Evaluation of SLAM

Assessing SLAM performance requires quantitative and qualitative metrics that evaluate how accurate, robust and efficient the estimated map and trajectory are. A few evaluation criteria include the Absolute Trajectory Error (ATE), RPE, resource usage(memory and CPU time consumption), etc.

The **ATE** measures the global deviation between estimated and ground truth trajectories. At each point in the trajectory, the difference between the estimated pose and the ground truth pose is computed. For a more general measure of the ATE, one might also calculate the Root Mean Square Error (RMSE), as follows:

$$ATE_{RMSE} = \sqrt{\frac{1}{N} \sum_{i=1}^{N} \|S(p_i^{est}) - p_i^{gt}\|^2} \tag{2.1}$$

The **RPE** measures the local consistency of the trajectory by evaluating the difference in relative motion between estimated and ground truth poses over a fixed time interval. As with the ATE, the formula for the RMSE of the RPE is as follows:

$$RPE_{RMSE} = \sqrt{\frac{1}{N} \sum_{i=1}^{N} \|(T_i^{est})^{-1} T_{i+\Delta t}^{est} - (T_i^{gt})^{-1} T_{i+\Delta t}^{gt}\|^2} \tag{2.2}$$

It is important to take into account the system resource utilization, both the memory and CPU time consumption, when benchmarking and comparing different SLAM solutions, due to the trade-offs between accuracy/noise robustness and memory utilization/time consumption. Depending on the application, it might not be worth it to use a more computationally intensive (but more accurate) SLAM solution.

In addition to the metrics mentioned previously, one might want to optimize for other metrics not included on this list. In that case, it might be useful to use a fitness function where normalized weights are attributed to each metric, according to the importance each metric has to the user.

Finally, when evaluating and comparing different SLAM solutions, it is important to establish a fair playing field for all the algorithms to be compared. One of the ways to do that is to

use publicly available standardized datasets, such as the KITTI Dataset [34]. As for evaluation frameworks, one of the most widely used is EVO [35], which provides error metrics and visualization tools, so as to better compare the performance of different SLAM solutions.

2.2 Hyperparameter optimization techniques

Hyperparameter optimization techniques can be broadly categorized into search-based, model-based and population-based approaches. Each approach uses different strategies to search the parameter space and obtain an approximation of the optimal solution, such as randomly sampling configurations (random search), mimicking physical processes to get a faster convergence (Simulated Annealing) or even by pre-selecting a parameter space and dropping half of the worst performing configurations at each pass (successive halving). Some of these approaches require a budget to be defined, meaning a time limit, or a maximum number of configurations to be tested.

2.2.1 Search based approaches

One popular type of approach to the problem of Hyperparameter Optimization (HPO), which will be used as a baseline on this thesis' work, is a search based approach, such as Grid Search and Random Search. These approaches are popular due to their implementation simplicity and paralelization possibilities.

Grid Search

Grid Search is a basic solution for HPO. It simply consists of an exhaustive search and evaluation of all possible hyperparameter combinations within a predefined parameter space [36]. In spite of the development of more sophisticated and specific algorithms in recent decades, Grid Search remains popular due to its simple implementation and trivial paralelization [36]. The main drawback is its computational cost, due to the curse of dimensionality being a serious problem in models with large numbers of hyperparameters [36]. This might be summarized by the following equation:

$$N_c = \prod_{n=1}^{k} N_{P_i} \tag{2.3}$$

where N_c is the total number of configurations and N_{P_i} is the number of possible values for hyperparameter i of the model.

By looking at equation 2.3, it becomes clear this algorithm is not very scalable, due to the rapid increase in the number of configurations, which makes this the main hurdle in search based approaches. One way of getting around it is paralelizing the execution of various configurations across several CPU cores and threads. Another way is to pre-select the parameters to optimize, and/or discarding hyperparameters which have little effect on the model's performance. However, this latter strategy has its downsides. If the initial analysis of the relevant parameters is off, one could throw away parameters that actually matter in some regions of the space [37]. Additionally, parameters that individually have small effects on the output may have greater effects when combined [38].
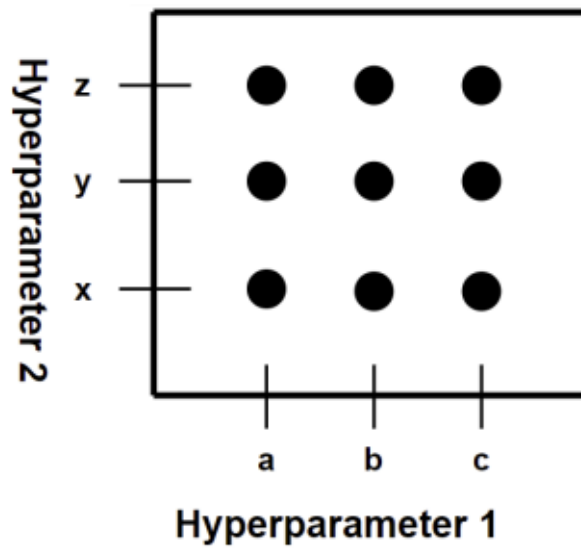
Figure 2.5: Graphical illustration of parameter space exploration in grid search [5].

Random Search

Random Search is a variation of Grid Search. It randomly samples configurations in the aforementioned parameter space [39]. Both Grid Search and Random Search are very similar implementation wise, with one major difference: Random Search requires a budget be specified, whether it is time, number of configurations, etc [37]. The main advantage Random Search has over Grid Search is the faster convergence over a local or global optima [37], although this advantage gets slimmer the larger the parameter space.
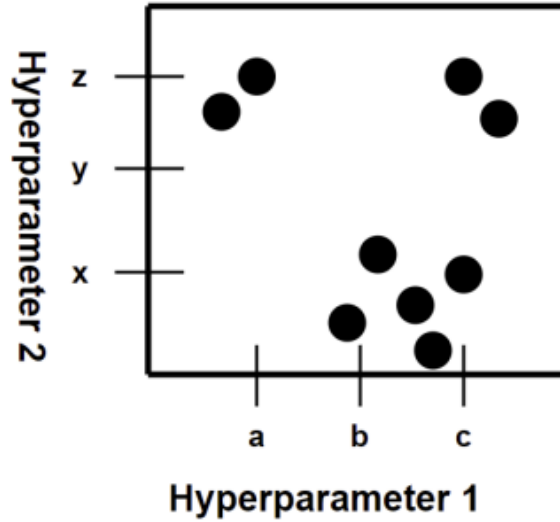
Figure 2.6: Graphical illustration of parameter space exploration in random search [5].

2.2.2 Model based approaches

Model based approaches tackle the optimization problem in a different way, by building a surrogate model that describes the relationship between hyperparameter configurations and algorithm performance. The inner workings of the algorithm to be optimized are unspecified and it is therefore treated as a black box [40]. These kind of HPO techniques are better suited to more complex optimization problems and clarify the relationship between algorithm performance and hyperparameter settings, which might prove to be a good option to pre select the most important hyperparameters to optimize when there are dozens or hundreds of parameters to optimize.

Bayesian Optimization

Bayesian Optimization (BO) is a probabilistic model-based approach that optimizes black box functions that are expensive to evaluate [41]. It is particularly useful when the objetive function lacks an analytic expression and its evaluations are very expensive, which is the case for SLAM methods.

Algorithm 1 Bayesian Optimization

1: **for** $t = 1, 2, \ldots$ **do**

2:      Find x_t by optimizing the acquisition function over the GP:

$$x_t = \arg\max_x u(x \mid \mathcal{D}_{1:t-1})$$

3:      Sample the objective function: $y_t = f(x_t) + \epsilon_t$

4:      Augment the data $\mathcal{D}_{1:t} = \{\mathcal{D}_{1:t-1}, (x_t, y_t)\}$ and update the GP.

5: **end for**

Figure 2.7: The Bayesian Optimization algorithm [6].

Bayesian Optimization has two main components:

- A Surrogate model, which is a probabilistic, fast proximate model that stands in for, and is much easier to evaluate than, an unknown(black box) or expensive function [42]. Instead of repeatedly evaluating the true function, which might take a long time, the surrogate model(usually a Guassian Process [39]) learns from previous evaluations, and predicts both the expected mean value of the true function(in this case, the RMSE of the RPE of a SLAM solution) and the uncertainty at each point. The uncertainty is what informs and guides the algorithm on what point to sample next, by balancing promising regions with unexplored ones(higher uncertainty).

- An Aquisition function, which is the rule BO uses to choose the next objective function point to evaluate [42]. It uses the surrogate model's predicitons(mean and uncertainty) and combines them into a single score, which is then used to evaluate the best candidate point. Common aquisition functions include Expected Improvement (EI), which chooses points that are expected to be better than the current best result, Probability of Improvement (PI), which focuses on the **chance** of beating the current best result, and Upper/Lower Confidence Bound (ULCB), which explicitly trades-off mean and uncertainty using a tunable parameter [43].

BO uses the following iterative process to optimize the objetive function(see figure 3.4):

1. Select the next point to be sampled with the aquisition function.

2. Evaluate the true function $f(x_1, x_2...x_n)$ at the selected point.

13

3. Update the surrogate model with the new information using gaussian process regression, which is the process of adding additional information of the sampled points to the **prior**.

4. Repeat steps 1, 2 and 3 until some stopping criteria is met(exhausted budget, achieved convergence, etc).

One major drawback of Bayesian Optimization is the impossibility of paralelization when compared to other baseline techniques, due to the fact the the surrogate model uses new points to update its parameters, meaning the learning process needs to finish before a new one can be launched [39].

2.2.3 Population based Approaches

These types of approaches are defined by a population of candidate solutions (sets of hyper-parameters) that are iteratively updated to optimize an objective function [44], such as the APE or the RPE in the case of SLAM optimization.

Simulated Annealing

Simulated Annealing (SA) is an optimization technique that mimicks the physical process of heating a metal and then cooling it slowly [45]. Analogously, the algorithm freely explores solutions in the beginning, even ones that seem worse at face value, so as to maximize exploration, and then, as the temperature decreases, according to a predefined cooling schedule, it focuses on refining a solution [46, 45].

The method starts by assigning an initial value to each hyperparameter, one that is high enough to allow compreensive search over the parameter space [46], due to the high variance of the parameter values in early stages of the algorithm. Then, it makes small changes to each parameter at a time and evaluates this new configuration, called a neighbor [45]. The aceptance of the neighbor as being the better solution depends on a probabilistic distribution [45], which in itself depends on the temperature and the difference between the current solution's and the neighbor solution's evaluation, like so:

$$P = e^{-\frac{\Delta E}{T}}$$

where T is the current temperature and ΔE is the difference between the cost of the new solution (the neighbor's) and the cost of the current solution, that is, $\Delta E = f'(x) - f(x)$.

Then, the algorithm updates the temperature, using the following expression: $T' = \alpha T$, where α is the cooling factor, which is usually given a value in the interval $[0.8, 0.99]$.

The algorithm repeats the previous steps until the temperatures reaches a minimum value or a stopping condition is triggered, such as a maximum number of iterations [46]. Once execution stops, the best solution is returned as an approximation to the actual optimal solution. Figure 2.8 shows a graphic representation of SA's behavior.
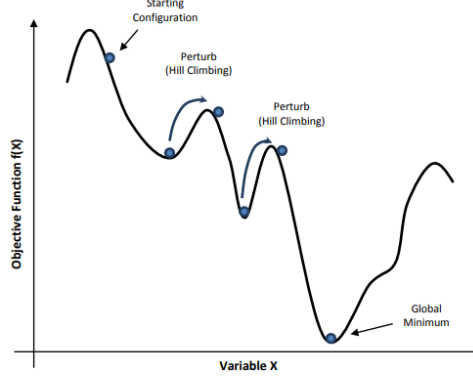
Figure 2.8: A graphic representation of the simulated annealing algorithm.

Successive Halving

Successive Halving (SH) is an optimization method that efficiently allocates resources to the most promising configurations while reducing investment into less promising ones [47]. Similar to SA, this technique requires a budget to be specified (execution time, number of iterations, etc.).

At first, n candidates are generated and evaluated, and the constrained resources are assigned equally to all configurations. Then, **the worst half of all configurations is discarded** and this process is repeated until only 1 configuration remains [48]. The hyperparameter configurations used in this method can be generated in a variety of different ways. One can define a parameter space similar to the one used in search-based approaches and use either Grid Search or Random Search to generate the desired number of configurations. There is also the possibility of manually selecting both the hyperparameters to optimize and their values.

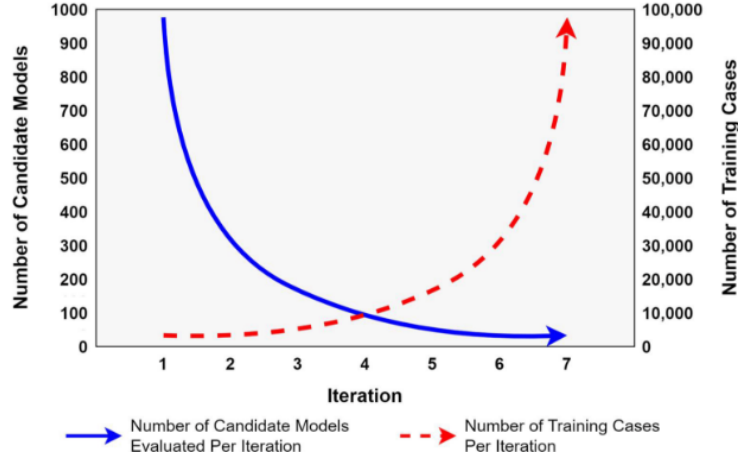Figure 2.9: A graphic representation of the Successive Halving algorithm [7].

Hyperband

Hyperband is a more efficient technique that builds on the Successive Halving algorithm [42]. By dynamically allocating computational resources, Hyperband quickly identifies promising configurations and discards ill-performing ones early on, saving both time and CPU time.

Hyperband works as follows:

1. Define a total Budget B and the proportion of configurations to evaluate at each rung, η. Also, a maximum budget R for a single rung is required [49].

2. Generate a large number of configurations [49].

3. Allocate a small initial budget to all configurations and perform the successive halving algorithm with one modification: halt its execution when only the top $1 / \eta$ configurations remain, instead of halting when only the best configuration remains. Then, increase the budget for the remaining configurations [49].

4. Repeat step 3 until maximum budget R is reached or all configurations are exhausted, meaning only the best one remains [49].

By increasing the budget allocated to a decreasing number of configurations at each execution of the Successive Halving algorithm, Hyperband transitions from an exploration-focused method to an exploitation-focused one.
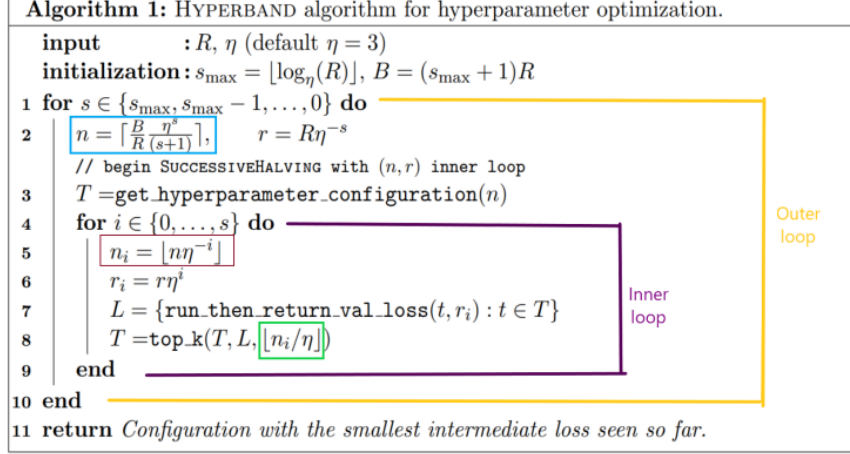
Figure 2.10: HB algorithm described in a more systematic way [8].

2.2.4   HPT in SLAM

When applying HPT algorithms to fine tune a SLAM method's parameters, there are a few things to keep in mind.

First, not all parameters are tunable. Better yet, some parameters have fixed or default values and are treated as constants during the tuning process, such as a camera's intrinsic parameters in VSLAM, or an IMU's noise and Bias models in Visual-Inertial SLAM. This is an important aspect to keep in mind, especially when applying search based approaches, as it may provide a way to significantly reduce the number of configurations to be executed.

As it pertains to the effectiveness of tuning strategies, search based methods are usually considered baseline methods in research, and not very effective when dealing with very high dimensional parameter spaces, due to the high stochasticity of grid and random search. In this regard, simulated annealing achieves better results, incorporating local refinement and controlled exploration. But in order to obtain near optimal results, and faster, other approaches are needed. Successive Halving and Hyperband work well within constrained environments(hardware, time etc), and Bayesian Optimization surpasses the previous 2 strategies in sample efficiency(how many full evaluations of the objective function are necessary to find a good configuration). Bayesian Optimization also works very well when the objective function is expensive to evaluate, or when the parameter space is low dimensional, which might be the case in a SLAM setting, provided the researcher carefully hand-picks the parameters to optimize.

17

2.3 Related Work

This subsection goes over published research in optimizing hyperparameters in SLAM systems. It presents a brief description of each relevant article and the used SLAM method and optimizing strategy, as well the results from the experiments that were made. It then goes over the current literature research gaps and presents the main contributions of this thesis.

On search-based approaches, there has been some research on the topic. Particularly, Putra et al. have hand-selected and manually searched(brute force) the parameter space of a g-mapping SLAM solution [50]. In this specific case, four parameters were tuned: **linear update step**, **angular update step**, **quantity of particle**, and **resampling threshold**. The particle number directly determines the number of possible robot position hypotheses the algorithm tracks. The linear update step is the minimum distance(in meters) the robot must move before updating the map. Similarly, the angular update step is the minimum angle the robot must rotate before triggering a map update. Finally, the resampling threshold controls the frequency at which particles are resampled. Through several executions of the algorithm, it was possible to reduce the navigation time of a robot from 32 seconds to 25 seconds.

Another study applied grid search to optimize 5 parameters in a feature based monocular visual odometry setting [51]. In all 10 sequences, measured ATE significantly decreased, showing the main advantage of this simple tuning approach. While the optimal configuration wasn't the same in all sequences, proving the optimal values change when scenery varies, in order to find a balanced setup, the authors chose parameter values that consistently produced reduced ATE values across all sequences. It is also worth noting that while search based approaches are more suited for a very reduced number of parameters, it is best to use grid search than manual search (brute force). Additionally, both papers presented previously show that, with good knowledge of the system, one can significantly reduce the parameter space down to the very few and impactful hyperparameters and a significantly better configuration than a default one when using grid and random search.

As far as model based approaches go, BO is the most used algorithm in research. One paper used a variant of it, Sequential Model-Based Optimization (SMBO) to optimize the hyperparameters of a LiDAR-based Odometry algorithm, without knowledge of the inner workings of the system, e.g the system is treated as a black box [52]. Although data augmentation was used to prevent overfitting(by adding random noise to the point clouds, reversing the data order, and

applying a random transformation to the point cloud), it was still possible to observe a noticeable decrease in the odometry drift error (in most tested sequences). Another article, although outside the scope of this dissertation(in this dissertation, camera parameters are considered constants) used BO to fine tune the extrinsic parameters of a camera in a Visual Inertial SLAM system [53].

With regards to population-based approaches, there isn't much research in SLAM itself. However, a study by Kostusiak and Skrzypczyński [44] used a Particle Swarm Optimization (PSO) algorithm and an Evolutionary Algorithm (EA) to tune the parameters of an RGB-D Visual Odometry system to improve motion estimation accuracy. Using the TUM RGB-D dataset for training and the PUT Kinect dataset for testing, the authors applied the two previously mentioned methods to adjust only five influential hyperparameters related to feature detection (AKAZE feature detector threshold) and outlier rejection (two RANSAC distance and inlier/outlier thresholds) [44]. The optimization, guided by the values of the ATE and RPE, showed that tuning these few parameters, particularly the AKAZE feature threshold and RANSAC distance limits, significantly reduced trajectory errors, with ATE dropping from about 1.6m to 0.29m [44]. While PSO achieved the best accuracy out of the 2 tuning strategies, the EA achieved similar results in one-fifth the time [44]. Alternatively, the authors also manually tuned the parameters, with significantly worse results than PSO [44].

Article(s)	SLAM category	Optimization method	Results
Z. Zheng (2020) [51]	Feature-based Visual Odometry(not full SLAM)	Grid Search	Across 8 optimized sequences, average ATE decreased, on average, 64.68%
A. Putra and P. Prajitno (2019) [50]	G-mapping SLAM(particle filter)	Brute Force	Navigation time of a predefined path was reduced from 32 to 25 seconds
K. Koide et al (2021) [52]	Lidar Odometry(not full SLAM)	based on Bayesian Optimization	**With data augmentation**, in both tested environments, both components (translational and rotational) of the drift error decreased by at least 9.8%.
A. Kostusiak and P. Skrzypczyński (2019) [44]	RGB-D Visual Odometry	Particle Swarm Optimization	Both ATE and RPE decreased by well over 50%, and optimized parameters generalize well to other sequences
A. Kostusiak and P. Skrzypczyński (2019) [44]	RGB-D Visual Odometry	Evolutionary Algorithm	With EA, the tuning duration is about 5x shorter than PSO,

Table 2.1: Summary of HPT methods used in literature.

2.3.1 Literature gaps

A few notable research gaps exist in the field of SLAM HPT. Most notably:

- Lack of diversity in optimizing strategies. As presented earlier, most of the research use either a search based approach or a model-based approach (similar to BO), as well as only using one optimization method for their particular SLAM solution. For SLAM systems where there is a high degree of knowledge of the inner working of the sensors and how different parameters affect the performance of the solution, it is possible to manually set and stick with default values for some of the hyperparameters, and can then apply a simple baseline(search-based) approach to the tuning of the remaining hyperparameters. As for the latter, it is used most when there is no extensive knowledge of the system at hand. Therefore, BO treats it as a black box and optimizes it. Apart from [44], there isn't much research on population-based approaches to optimizing SLAM, whether variations of particle swarm algorithms, evolutionary algorithms, or other meta heuristics.

- Lack of a framework to automatically optimize SLAM methods, allowing for an even playing field in SLAM research. Athough a framework like **SLAMBench2** [54] provides a platform with a controlled dataset and with standardized metrics, it lacks an automatic tuning feature. Similarly, **RUSTLE**, the tool which will be built and improved upon during the course of this dissertation, already allows for a somewhat organized approach to testing and optimizing complete SLAM solutions, but it lacks any optimization feature, meaning the only way to currently approach HPT in RUSTLE is by brute force optimizing an algorithm.

2.3.2 Statement of contributions

Given that academic research often focuses on a single optimization method for a particular SLAM system for an even more specific use case, the main contribution of this dissertation is new software features(within RUSTLE) for the optimization of multiple SLAM solutions in an organized and systematic manner.

3 Methodology

This chapter will explain the methodology and requirements for the SLAM tuning module, developed for RUSTLE. For each implemented tuning algorithm, a high level overview of its architecture is provided. Test settings, including test algorithms and dataset choices, are also justified.

3.1 RUSTLE
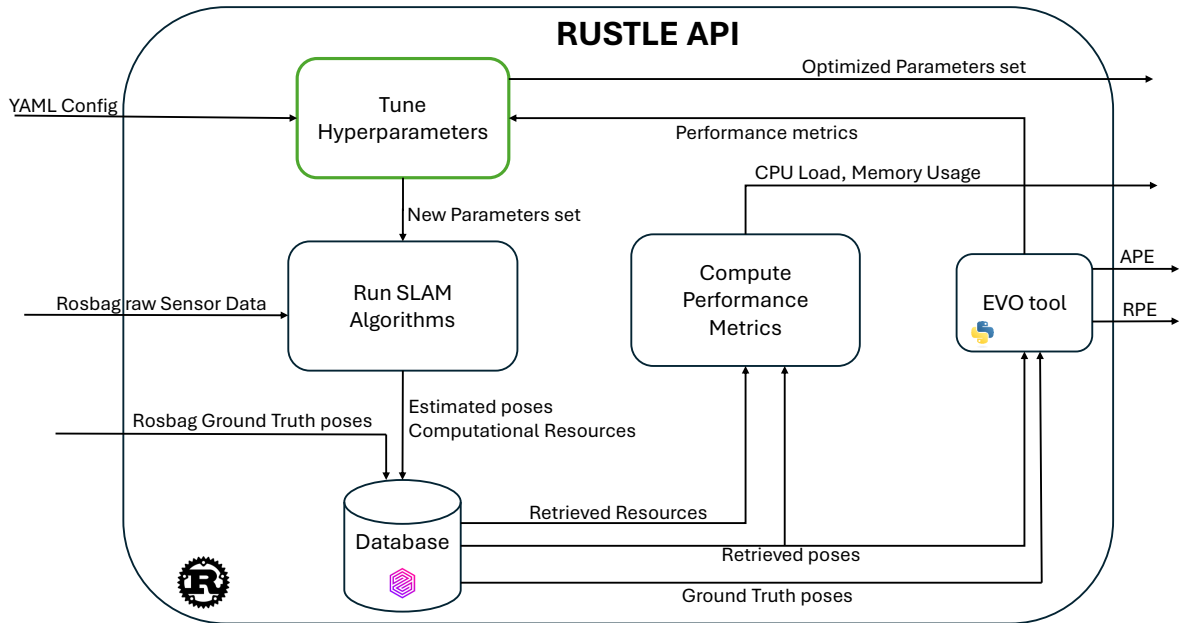

Figure 3.1: RUSTLE diagram

RUSTLE is a command line application, written in Rust, which is designed to simplify the evaluation of SLAM algorithms in mobile robotics. Its main objective is to provide a simple, stream-lined and reproducible way to run and compare SLAM algorithms. It tracks not only trajectory accuracy(APE or RPE), but also runtime, CPU load and memory usage, offering a

more complete assessment of the algorithm's performance.

At its core, RUSTLE takes three main inputs: a dataset in rosbag format, the SLAM algorithm's parameters in yaml format, and the algorithms themselves as docker images. The reason for using Docker is for reliably reproducing the algorithms across many different platforms and environments. During execution, the odometry data is streamed into the database(Surrealdb), which, after conclusion, is used to calculate trajectory errors such as the APE and the RPE, using EVO.

Algorithm executions are run as tests. Each test's configuration is passed as an YAML file, and contains: its name, number of workers, number of iterations, list of algorithms, dataset on which to execute these algorithms and the test type. Additionally, some additional fields might be required, depending on the type of test to be executed.

Tests can be of a few different types, namely:

- Simple - The rosbag file simply plays from begining to end at normal speed

- Speed - Portions of the dataset are run at different speeds

- Cut - Only specified portions of the dataset are run

- Drop - Não faço ideia

Every tuning experiment run on RUSTLE will be run as a simple test, with the caveat that the user can choose the portion of the dataset to run. This is to facilitate development and testing. When used in production, the user should set both the start and duration of the dataset to 0, meaning, the whole dataset will be run.

3.2 Development

Development follows an incremental approach, focusing on small incremental features, well thought and tested. Each big feature, such as random search tuning, is divided into smaller, easier to test and debug sub-features. The reason for this is to make it easier to reason about how to implement the feature, as well as, during development, to make it easier to test and debug each sub-feature.

3.2.1 Feature Implementation

The optimization algorithms that were chosen to be implemented are: Grid Search, Random Search and Simulated Annealing. In section 2, more algorithms were mentioned, specifically Bayesian Optimization, Successive Halving and Hyperband. The reasons for deciding only to implement these three of the six are twofold. Firstly, Grid Search serves as a baseline, exhaustive search method, despite being inefficient for large parameter spaces. Random Search is a natural improvement over Grid Search, finding better configurations faster, particularly under a fixed computational budget [37]. Finally, Simulated Annealing was selected by being a meta-heuristic and a trajectory based method which is capable of escaping local minima. The reason for choosing only three methods out of the six described in section 2 is mostly based on time constraints. Implementing more than 3 methods, particularly Bayesian Optimization and HyperBand, might prove to be too much of a challenge, considering one has to write the code, but also to run each tuning algorithm for multiple runs, for each chosen SLAM solution, all before extracting the results, and discussing them in the present dissertation.

In addition to the optimization algorithms, it will also be necessary to write code for exporting results of a tuning test, for later result extraction, processing and discussion, as well as code for parsing a YAML settings file, with all necessary information to run the optimization experiment. Finally, some python scripts will be developed to plot the results of the tuning experiments, which will be presented in the next chapter of this dissertation. Below is a high-level diagram of the features I intend to implement.
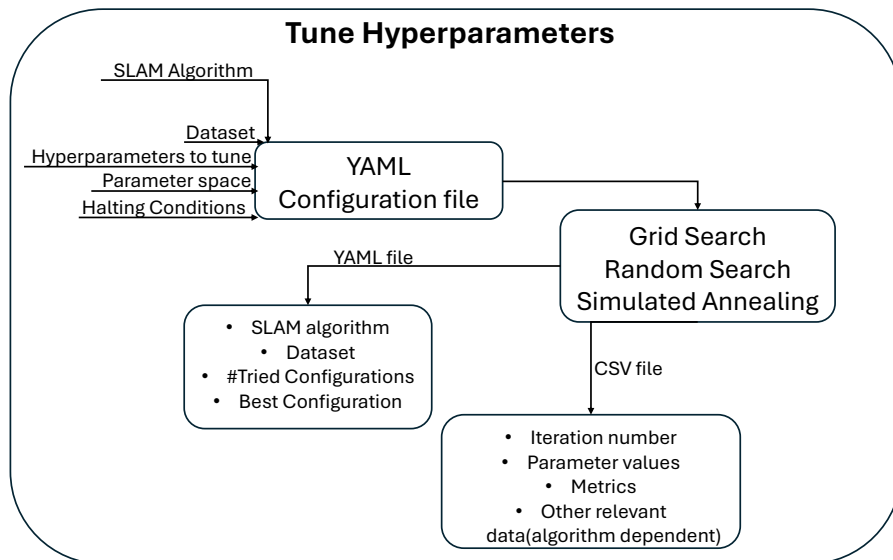

Figure 3.2: low-level diagram of hyperparameter tuning module

Simulated Annealing

The Simulated Annealing algorithm implementation is identical to the one provided by Stefan Kroboth [55].

For floating-point parameters, the used perturbation function samples a normal distribution function and adds the sampled value to the current value of the parameter. Both the mean and the standard deviation of this normal distribution are hyperparameters that can be specified in a YAML configuration file.

$$\Delta_{float} \sim \mathcal{N}(m, d^2) \tag{3.1}$$

For integer parameters, a maximum step size is specified for each parameter, and that step size is multiplied by the current temperature. Then a random integer is generated between the result of that multiplication(see equation 3.2).

$$\Delta_{int} \in [-t_i * max\_step, t_i * max\_step] \tag{3.2}$$

Δ_{int} is then added to the current value of the parameter, and the value is truncated if it is higher that the parameter's upper bound or lower than its lower bound. Both the upper and lower bounds, as well as the maximum step size for each integer parameter, can be customized used a YAML configuration file.

During the execution of the algorithm, it is sometimes necessary to reanneal, that is, to reset the temperature to a previous, higher level, to allow the algorithm to escape(or at least try to) a local minima. With that in mind, 3 parameters are necessary:

- reanneal_fixed - the algorithm resets the temperature no matter what after this many iterations

- reanneal_accepted - the algorithm resets the temperature if no neighbor solution is accepted after this many **consecutive** iterations

- reanneal_best - the algorithm resets the temperature if no solution better than the incumbent is found after this many **consecutive** iterations

26

As for halting conditions, 2 parameters are specified:

- stop_accepted - The algorithm stops its execution if this no neighbor solution is accepted for these many **consecutive** iterations

- stop_best - the algorithm stops its execution if no solution better than the incumbent is found after these many **consecutive** iterations

Grid Search

The Grid Search implementation considers the parameter space as a multi dimensional tensor, where each dimension corresponds to a different parameter, and has a differing number of possible values. In each iteration, to obtain the current values of the parameters

---
Algorithm 2 Grid Search

1: **for** $t = 1, 2, ..., n$ **do**

2:     Obtain the corresponding values of the parameters from the index of the configuration

3:     Evaluate the configuration and extract metrics

4:     If halting conditions are met(time limit or number of configurations), stop the algorithm's execution

5: **end for**

Figure 3.3: Grid Search procedure

The halting conditions can be customized via a YAML configuration file. Similarly, the parameter space can be designed in the following way:

$$parameter\_name : [first\_value, step\_size, number\_of\_values] \tag{3.3}$$

Additionally, the code checks the default configuration and catches most common errors, such as the user passing a String for a parameter that is supposed to be an integer.

Random Search

The Random Search Implementation is similar to Grid Search, except that the index of the configuration to be evaluated is randomly generated, and no configuration is evaluated twice, by way of checking a list of previously evaluated configurations.

Algorithm 3 Random Search

1: **for** $t = 1, 2, ..., n$ **do**

2:      Generate a random number between in the interval [0, n], and check if the number is on the previously generated numbers list. If it is, keep generating numbers until it isn't.

3:      Add the number to the previously generated numbers list.

4:      Obtain the parameter's current values and update the SLAM configuration

5:      Evaluate the configuration and extract metrics

6:      **if** (elapsed_time > time_limit) or (evaluated_configs > max_iterations) **then** suspend algorithm execution

7:      **end if**

8: **end for**

Figure 3.4: Random Search procedure

As in Grid Search, in Random Search, both the parameter space and the halting conditions can be specified in a YAML configuration file.

What i will be doing: explain the features i will be developing, where my code inserts itself inside of RUSTLE, which optimization strategies will be implemented and why, what tests i will do to test the code and to extract results.

3.3 Functional requirements

The system must allow users to define all relevant settings for the tuning process using a YAML configuration file. This configuration file will serve as the main interface for specifying the tuning method, SLAM algorithm, and dataset to be used. In addition, it will enable the user to indicate which parameters are to be optimized, with one or more values per parameter. Additionaly, the code must ensure type safety, meaning, for example, that a parameter which is supposed to be a String cannot have a u64 passed as a value. Also, the user must have the option of specifying only the parameters he wants to optimize.

For grid and random search tuning methods, the configuration file must support flexible representations of parameter values. Each tunable parameter can be defined as a single value, representing a fixed setting, or as an array containing multiple candidate values to be tested. For numerical parameters(integers and floats), a third specification pattern must be supported: a

linearly spaced vector. This vector will be described by a three-element array in the form [**start, step, number_of_elements**], which the system will interpret to automatically generate a sequence of values. This feature will simplify the definition of evenly spaced search spaces for numerical parameters and ensures consistency across different tuning runs. It also makes it significantly less cumbersome to define large sequences of values.

Every tuning algorithm has, besides the parameter space, some method specific hyperparameters that control its behavior, such as the budget(grid search) or the cooling factor(SA). Each of these parameters must have a default value, so the user doesn't need to fine tune the tuning algorithm's behavior. As for the budget **B**, a common hyperparameter used in tuning algorithms, it must be either time based, or iterations based(maximum number of iterations to run).

For easier development and integration with the overall code structure of RUSTLE, the tuning of an algorithm's hyperparameters is treated as a test. Each configuration is run as a **simple test**, already defined in the code, changing hyperparameters between iterations. This methodology will allow for greater re-usage of already written code, saving on development time.

When running a tuning test, each configuration and its metrics(results) must be stored for later processing. The tuning method's hyperparameters must be stored as well, mostly for benchmarking purposes.

For benchmarking different tuning algorithms, there must be a command as simple as **tune benchmark -algo xxxxx -dataset xxxxx**, which runs a few preselected tuning algorithms, and displays a few performance metrics.

Whenever possible, configurations should be run concurrently.

The code written should be future proof, allowing for easier future integrations of more tuning methods into the software.

3.4   Testing

3.4.1   Metrics

During the development, as well as during testing stages, only the RPE is begin considered. However, the user can attribute weights to both the APE and RPE using a YAML configuration

file. However, the code supports, with a few smal moddifications, taking into account other metrics(CPU load, memory usage, etc) when computing the fitness value of a configuration.

3.4.2 Algorithms

In this subsection, I explain both the SLAM methods I chose to optimize and the hyperparameters I chose to optimize and why.

One thing that is applicable to both SLAM solutions chosen is that due to the high number of hyperparameters and the memory constraints, it is not possible neither to optimize every hyperparameter nor to give every hyperparameter a very large number of possible values. For that reason, only a select few hyperparameters will be optimized, and the choices of hyperparameters will be fully explained below.

faster-lio

fast-livo2

Write something about faster-lio and fast-livo2 and why they were chosen.

3.4.3 Dataset

During code development, and also during testing and result extraction, I used the BotanicGarden dataset [56], specifically the 1006 03 sequence. This dataset presents a challenging, semi-structured environment with diverse sensor modalities, including a DALSA M1930 stereo camera, a DALSA C1930 stereo camera, a Velodyne VLP16 **LiDAR!** (**LiDAR!**), a Livox Avia **LiDAR!**, and an Xsens Mti-680G IMU. All sensors are hardware-synchronized, ensuring time-aligned measurements suitable for sensor fusion. It also provides a robust ground truth, generated using a static terrestrial laser scanner to acquire high-density, high-resolution point clouds. These point clouds were subsequently aligned to produce ground truth trajectories with an accuracy of approximately 1 cm.

3.4.4 Tuning Strategy

In order to extract good parameter values, the following strategy was devised:

- Run Simulated Annealing. It's likely that more than one run will be required, and the algorithm parameters should be adjusted(reannealing and halting thresholds matter most), as well as some SLAM parameters upper and lower bounds.

- Plot the results, particularly the SLAM parameter values vs fitness function value(in my case, only the RPE is considered). This will help identify "unsafe" and "safe" intervals, meaning, bounds inside which the fitness function value goes up rapidly, and bounds where the fitness function tends to have a low value.

- Using the previously identified "safe" intervals, design a parameter space to further explore.

- Run Grid Search and Random Search. This might help to further reduce the fitness function value, but it also helps to emphasize the advantages in using Random Search over Grid Search.

- Extract the best configuration

4 Results

This chapter goes over 2 examples of tuning 2 distinct SLAM algorithms(faster-lio and fast-livo2), using all three tuning algorithms implemented. Given that the purpose of this dissertation is to optimize SLAM solutions, there is not much sense in comparing the effectiveness of faster-lio and fast-livo2. So, it was decided to optimize each of these algorithms for a different metric, aligning the optimization goal with the algorithm's design goals. faster-lio was optimized for RPE, while fast-livo2 was optimized for APE.

4.1   faster-lio

4.2   fast-livo2

4.3   Possible improvements

Bibliography

[1] S. A. K. TAREEN, "Large scale 3d simultaneous localization and mapping (ls-3d-slam) using monocular vision," 2019.

[2] D. Casado, "Introduction to visual slam: Chapter 1 —introduction to slam," 2021.

[3] F. Andert, O. Böttcher, A. Mushyam, and P. Schmälzle, "Semantic lidar point cloud mapping and cloud-based slam for autonomous driving," pp. 853–860, 2025.

[4] R. C. Luo, S. L. Lee, Y. C. Wen, and C. H. Hsu, "Modular ros based autonomous mobile industrial robot system for automated intelligent manufacturing applications," pp. 1673–1678, 2020.

[5] E. I. M. Castillo, *Hyperparameter Optimization for SLAM: An Approach for Enhancing ORB-SLAM2's Performance.* PhD thesis, University of Alberta, 2022.

[6] E. Brochu, V. M. Cora, and N. De Freitas, "A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning," *arXiv preprint arXiv:1012.2599*, 2010.

[7] D. S. Soper, "Hyperparameter optimization using successive halving with greedy cross validation," *Algorithms*, vol. 16, no. 1, p. 17, 2022.

[8] L. Li, K. Jamieson, G. DeSalvo, A. Rostamizadeh, and A. Talwalkar, "Hyperband: A novel bandit-based approach to hyperparameter optimization," *Journal of Machine Learning Research*, vol. 18, no. 185, pp. 1–52, 2018.

[9] R. Beņkis, E. Grabs, T. Chen, A. Ratkuns, D. Čulkovs, A. Ancāns, and A. Ipatovs, "A survey and practical application of slam algorithms," pp. 1–10, 2024.

[10] P.-Y. Lajoie, B. Ramtoula, F. Wu, and G. Beltrame, "Towards collaborative simultaneous localization and mapping: a survey of the current research landscape," *Field Robotics*, vol. 2, pp. 971–1000, 2022.

[11] H. Taheri and Z. C. Xia, "Slam; definition and evolution," *Engineering Applications of Artificial Intelligence*, vol. 97, p. 104032, 2021.

[12] A. Macario Barros, M. Michel, Y. Moline, G. Corre, and F. Carrel, "A comprehensive survey of visual slam algorithms," *Robotics*, vol. 11, no. 1, p. 24, 2022.

[13] C. Harris, M. Stephens, *et al.*, "A combined corner and edge detector," vol. 15, no. 50, pp. 10–5244, 1988.

[14] T. Lindeberg, "Feature detection with automatic scale selection," *International journal of computer vision*, vol. 30, no. 2, pp. 79–116, 1998.

[15] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International journal of computer vision*, vol. 60, no. 2, pp. 91–110, 2004.

[16] Y. Liao, Y. Di, K. Zhu, H. Zhou, M. Lu, Y. Zhang, Q. Duan, and J. Liu, "Local feature matching from detector-based to detector-free: a survey," *Applied Intelligence*, vol. 54, no. 5, pp. 3954–3989, 2024.

[17] A. Jakubović and J. Velagić, "Image feature matching and object detection using brute-force matchers," pp. 83–86, 2018.

[18] W. Chen, L. Zhu, Y. Guan, C. R. Kube, and H. Zhang, "Submap-based pose-graph visual slam: A robust visual exploration and localization system," in *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pp. 6851–6856, IEEE, 2018.

[19] Q. Picard, S. Chevobbe, M. Darouich, and J.-Y. Didier, "A survey on real-time 3d scene reconstruction with slam methods in embedded systems," 2023.

[20] Z. Li, F. Ye, and X. Guan, "3d point cloud reconstruction and slam as an input," 2021.

[21] Y. Chen, Y. Zhou, Q. Lv, and K. K. Deveerasetty, "A review of v-slam," in *2018 IEEE International Conference on Information and Automation (ICIA)*, pp. 603–608, IEEE, 2018.

[22] A. Fontan, J. Civera, and M. Milford, "Anyfeature-vslam: Automating the usage of any chosen feature into visual slam," vol. 2, 2024.

[23] T. Lindeberg, ""' scale selection", computer vision: A reference guide,(k. ikeuchi, editor)," 2014.

[24] R. Raguram, J.-M. Frahm, and M. Pollefeys, "A comparative analysis of ransac techniques leading to adaptive real-time random sample consensus," pp. 500–513, 2008.

[25] D. Yan, W. Tuo, W. Wang, and S. Li, "Illumination robust loop closure detection with the constraint of pose," *arXiv preprint arXiv:1912.12367*, 2019.

[26] X. Yue, Y. Zhang, J. Chen, J. Chen, X. Zhou, and M. He, "Lidar-based slam for robotic mapping: state of the art and new frontiers," *Industrial Robot: the international journal of robotics research and application*, vol. 51, no. 2, pp. 196–205, 2024.

[27] D. Zhu, Q. Wang, F. Wang, and X. Gong, "Research on 3d lidar outdoor slam algorithm based on lidar/imu tight coupling," *Scientific Reports*, vol. 15, no. 1, p. 11175, 2025.

[28] J. Jorge, T. Barros, C. Premebida, M. Aleksandrov, D. Goehring, and U. Nunes, "Impact of 3d lidar resolution in graph-based slam approaches: A comparative study," pp. 1–6, 2024.

[29] H. Wang, Y. Yin, and Q. Jing, "Comparative analysis of 3d lidar scan-matching methods for state estimation of autonomous surface vessel," *Journal of Marine Science and Engineering*, vol. 11, no. 4, p. 840, 2023.

[30] N. Stathoulopoulos, V. Sumathy, C. Kanellakis, and G. Nikolakopoulos, "Why sample space matters: Keyframe sampling optimization for lidar-based place recognition," *arXiv preprint arXiv:2410.02643*, 2024.

[31] M. Servières, V. Renaudin, A. Dupuis, and N. Antigny, "Visual and visual-inertial slam: State of the art, classification, and experimental benchmarking," *Journal of Sensors*, vol. 2021, no. 1, p. 2054828, 2021.

[32] Z. Liu, H. Li, C. Yuan, X. Liu, J. Lin, R. Li, C. Zheng, B. Zhou, W. Liu, and F. Zhang, "Voxel-slam: A complete, accurate, and versatile lidar-inertial slam system," 2024.

[33] K. Liu, "A lidar-inertial-visual slam system with loop detection," 2023.

[34] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun, "Vision meets robotics: The kitti dataset," *The international journal of robotics research*, vol. 32, no. 11, pp. 1231–1237, 2013.

[35] M. Grupp, "evo: Python package for the evaluation of odometry and slam.." `https://github.com/MichaelGrupp/evo`, 2017.

[36] L. Yang and A. Shami, "On hyperparameter optimization of machine learning algorithms: Theory and practice," *Neurocomputing*, vol. 415, pp. 295–316, 2020.

[37] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *The journal of machine learning research*, vol. 13, no. 1, pp. 281–305, 2012.

[38] B. Bischl, M. Binder, M. Lang, T. Pielok, J. Richter, S. Coors, J. Thomas, T. Ullmann, M. Becker, A.-L. Boulesteix, *et al.*, "Hyperparameter optimization: Foundations, algorithms, best practices and open challenges. arxiv," *arXiv preprint arXiv:2107.05847*, 2021.

[39] A. Bissuel, "Hyper-parameter optimization algorithms: a short review," *Criteo tech blog, Medium. Disponível em: https://medium. com/criteo-labs/hyper-parameter-optimizationalgorithms-2fe447525903. Acedido em*, 2020.

[40] A. Morales-Hernández, I. Van Nieuwenhuyse, and S. Rojas Gonzalez, "A survey on multi-objective hyperparameter optimization algorithms for machine learning," *Artificial Intelligence Review*, vol. 56, no. 8, pp. 8043–8093, 2023.

[41] J. Mockus, "The application of bayesian methods for seeking the extremum," *Towards global optimization*, vol. 2, p. 117, 1998.

[42] R. Elshawi, M. Maher, and S. Sakr, "Automated machine learning: State-of-the-art and open challenges," *arXiv preprint arXiv:1906.02287*, 2019.

[43] P. I. Frazier, "A tutorial on bayesian optimization," *arXiv preprint arXiv:1807.02811*, 2018.

[44] A. Kostusiak and P. Skrzypczyński, "On the efficiency of population-based optimization in finding best parameters for rgb-d visual odometry," *Journal of Automation Mobile Robotics and Intelligent Systems*, vol. 13, 2019.

[45] R. A. Rutenbar, "Simulated annealing algorithms: An overview," *IEEE Circuits and Devices magazine*, vol. 5, no. 1, pp. 19–26, 1989.

[46] O. Ghasemalizadeh, S. Khaleghian, and S. Taheri, "A review of optimization techniques in artificial networks," *International Journal of Advanced Research*, vol. 4, no. 9, pp. 1668–86, 2016.

[47] D. S. Soper, "Greed is good: Rapid hyperparameter optimization and model selection using greedy k-fold cross validation," *Electronics*, vol. 10, no. 16, p. 1973, 2021.

[48] C. Huang, Y. Li, and X. Yao, "A survey of automatic parameter tuning methods for meta-heuristics," *IEEE transactions on evolutionary computation*, vol. 24, no. 2, pp. 201–216, 2019.

[49] S. Falkner, A. Klein, and F. Hutter, "Bohb: Robust and efficient hyperparameter optimization at scale," in *International conference on machine learning*, pp. 1437–1446, PMLR, 2018.

[50] I. A. Putra and P. Prajitno, "Parameter tuning of g-mapping slam (simultaneous localization and mapping) on mobile robot with laser-range finder 360 sensor," pp. 148–153, 2019.

[51] Z. Zheng, "Feature based monocular visual odometry for autonomous driving and hyperparameter tuning to improve trajectory estimation," vol. 1453, no. 1, p. 012067, 2020.

[52] K. Koide, M. Yokozuka, S. Oishi, and A. Banno, "Automatic hyper-parameter tuning for black-box lidar odometry," pp. 5069–5074, 2021.

[53] Z. Chen, "Visual-inertial slam extrinsic parameter calibration based on bayesian optimization," 2018.

[54] B. Bodin, H. Wagstaff, S. Saeedi, L. Nardi, E. Vespa, J. H. Mayer, A. Nisbet, M. Luján, S. Furber, A. J. Davison, P. H. J. Kelly, and M. O'Boyle, "Slambench2: Multi-objective head-to-head benchmarking for visual slam," 2018.

[55] S. Kroboth, "argmin: Numerical optimization in rust," 2024.

[56] Y. Liu, Y. Fu, M. Qin, Y. Xu, B. Xu, F. Chen, B. Goossens, P. Z. Sun, H. Yu, C. Liu, L. Chen, W. Tao, and H. Zhao, "Botanicgarden: A high-quality dataset for robot navigation in unstructured natural environments," *IEEE Robotics and Automation Letters*, vol. 9, p. 2798–2805, Mar. 2024.

Appendix A

Sample Appendix

Fusce mauris. Vestibulum luctus nibh at lectus. Sed bibendum, nulla a faucibus semper, leo velit ultricies tellus, ac venenatis arcu wisi vel nisl. Vestibulum diam. Aliquam pellentesque, augue quis sagittis posuere, turpis lacus congue quam, in hendrerit risus eros eget felis. Maecenas eget erat in sapien mattis porttitor. Vestibulum porttitor. Nulla facilisi. Sed a turpis eu lacus commodo facilisis. Morbi fringilla, wisi in dignissim interdum, justo lectus sagittis dui, et vehicula libero dui cursus dui. Mauris tempor ligula sed lacus. Duis cursus enim ut augue. Cras ac magna. Cras nulla. Nulla egestas. Curabitur a leo. Quisque egestas wisi eget nunc. Nam feugiat lacus vel est. Curabitur consectetuer.