Optimizing Robotic Movements for Efficient Sensor Calibration: A Metaheuristic Approach

Fares Al-Tantawi Utkarsh Aiddasani Balen Seeton

Anish Mahto Vincent Tang Andrew Seo

***Abstract* - In our research, we explore an advanced method to optimize the movements of a robotic arm for sensor-fusion calibration, which is crucial for improving the efficiency of automated systems. The project involves using a robotic arm to position sensor rigs in specific spatial coordinates. This process is vital for the calibration algorithm that calculates essential sensor calibration values, including camera field of view, focal length, and IMU accuracy. A key focus of our study is determining the most effective sensor rig positions to maximize calibration precision.**

We applied three distinct metaheuristic optimization strategies: genetic mutation with Variable Neighborhood Search (VNS), parent selection with VNS, and annealing search. The effectiveness of these methods was evaluated by their impact on minimizing the re-projection error and total run time in calibration trials. Both values should ideally be as low as possible, since re-projection error represents accuracy of the calibration, and total run time represents the productive efficiency of calibration. Our findings indicate that not all the optimization methods were effective, with the genetic algorithm yielding the most useful results. This study not only showcases the practical application of metaheuristic algorithms in complex, real-world robotic tasks but also provides valuable insights for professionals in robotics, automation, and sensor technology, emphasizing the potential of genetic style algorithms in enhancing automated sensor calibration processes.

Keywords — GNS, VNS, GA

I. INTRODUCTION

Our research focuses on developing and testing an optimization algorithm tailored to robotic arm calibration. This optimization is pivotal as it directly impacts the accuracy of the robot's movements and the effectiveness of sensor data integration. The real-world applicability of our algorithm is validated through experimental testing, a necessity due to the inability to accurately estimate the cost function of the calibration process without physical trials.

We present a comprehensive study that explores various metaheuristic and heuristic optimization techniques in the context of a robotic arm's calibration process. The experimental setup involves a robotic arm performing a series of movements based on coordinates supplied by an optimization algorithm. This process is coupled with data collection from onboard cameras, feeding into a calibration algorithm that calculates extrinsic calibration parameters. The ultimate goal is to minimize re-projection error, a measure of calibration accuracy, while also considering the efficiency of the calibration process.

The paper is structured as follows: Firstly, we explain the methodology, detailing the experimental approach chosen for this study. Next, we discuss the algorithm testing method, elaborating on the robotic setup and the calibration process. Following this, we delve into the simulations conducted using Python, outlining the implementation of three metaheuristic algorithms: genetic mutation with Variable Neighborhood Search (VNS), basic local search with Generalized Neighborhood Search (GNS), and simulated annealing. Each method's foundation and practical implementation are explored. The paper concludes with experimental results, highlighting the challenges and insights gained from real-world testing.

Our research introduces an automated calibration system using a robotic arm to strategically position sensor rigs. We focus on optimizing the arm's movements to enhance calibration accuracy. To this end, we explore three metaheuristic optimization techniques: genetic mutation with Variable Neighborhood Search (VNS), Variable Neighborhood Search with parent selection, basic local search with Generalized Neighborhood Search (GNS), and annealing search. These methods are evaluated based on their effectiveness in reducing re-projection error during calibration. Our findings not only advance the application of optimization techniques in robotic calibration but also provide insights into the most effective strategies for automated sensor-fusion calibration, with a particular emphasis on the superior performance of the genetic algorithm approach.

II. METHODOLOGY

A. Algorithm Testing Method

The methodology for this study main consists of experimental testing with a Realsense D435i camera on a Kinova Gen3 robotic arm because it is not possible to estimate the cost function without physical running a trial in the real world. More importantly, this limitation introduces a significant challenge to the development and testing of this optimization algorithm. In the real world, each trial would take around two minutes. Thus, it is not feasible to run more than a couple hundred trials to validate an algorithm.

To run an algorithm experimentally, the algorithm would call a wrapper function for robot control that takes a list of coordinates. The robotic arm’s internal control system would compute the trajectories between the points and move to every coordinate supplied sequentially. A data collection node runs in parallel while the robot moves to collect images from the two cameras mounted on-board. After all coordinates have been reached, the data recording stops. An open-source calibration algorithm package, Kalibr, reads the recorded data to perform some image recognition and vector calculus to determine the extrinsic calibration parameters about the cameras, such as focal length and lens offset. [3] When the parameters are determined, the calibration algorithm reapplies the parameters to the images to determine thew new offsets on the extrinsic data of the cameras. Finally, these new offsets can be translated into re-projection error, which indicate how far the determined calibration target coordinates are from their perceived location. [3] This measure of re-projection error can

©2023 IEEE

be used as a summarizing indicator for the effectiveness of a calibration trial. A lower re-projection error represents better calibration.

If re-projection error was the only parameter of concern, this would be a very simple problem, because the robotic arm could travel through the entire feasible landscape and collect as much data as possible. However, that would not suffice a second objective of this study – efficiency. If the robot moves an unnecessary amount just for very small gains in re-projection error, the opportunity cost of shorter time is high. Thus, time is also used an input factor to determine the overall score of a calibration trial.

The objective function is set-up as:

$$C = a(e_{re-project}) + b(\Delta t)$$

a and b are weighing factors that adjusts the magnitude of influence on total score by the re-projection error and total run time. Since re-projection error in pixels is generally on the magnitude of 10^{-3} and total time is measured in seconds, factor a must be very large to bring the influence from both variables to a reasonable amount. [4]

III. Simulations

In our study, we evaluated the effectiveness of three metaheuristic optimization techniques—genetic mutation with Variable Neighborhood Search (VNS), basic local search with Generalized Neighborhood Search (GNS), and annealing search—through a series of simulations developed in Python. Python was chosen for its robustness and extensive support for scientific computing, making it well-suited for complex algorithmic implementations.

The simulation component of our study was implemented in Python, employing a detailed and structured approach to optimize the movement of a robotic arm for sensor-fusion calibration. The code encapsulates a two-part structure: the RobotControl class, which simulates the robotic arm's movements and calibration process, and the Optimizer class, which applies metaheuristic optimization techniques.

Each simulation run began with an initial guess of four waypoints, each comprising six coordinates (x, y, z, pitch, roll, and yaw). This multi-dimensional setup allowed for a thorough exploration of the robot arm's potential movements and orientations. The RobotControl class's move_and_calibrate function simulated the arm's movement and the resulting calibration, returning values for reprojection error and total time, which collectively formed the basis for the objective function used in optimization.

The first optimization method programmed was local search. This is an algorithm that first generates a population of potential solutions per iteration. Then iterating through all these potential solutions, each one was evaluated based on the objective function – a weighted sum of reprojection error and total time. Then comparing the score to the current best, decisions are made on what the new best solution is. Key methods include neighbourhood generation and optimization iterations.

The Optimizer class implemented the core of selected algorithm or search method. This class managed a population of potential solutions (sets of waypoints), each evaluated based on the objective function – a weighted sum of reprojection error and total time.

The optimization process iteratively generated new candidates, improved them, and updated the best solution based on the lowest score from the objective function. This process continued until a convergence threshold was met or the maximum number of trials was reached.

IV. Implementation

A. *Simulated Annealing*

The next optimization algorithm implemented is simulated annealing, in the SimulatedAnnealing python class. The class is constructed from the following hyperparameters; the initial temperature (corresponds to the maximum number of iterations), the cost function, and the starting search position (a list of 4 coordinates, each with the 6 dimensions as mentioned above). [2]

On a call to SimulatedAnnealing.search(), the algorithm begins executing iterations and keeping track of the best solution found thus far at the end of every iteration. On each iteration, we perform the standard simulated annealing procedure; evaluate the cost of a neighbor, and accept it if either the local neighbor is strictly better than the current position or accept it with probability $e^{-\Delta \mathrm{cost}/\alpha \mathrm{T}}$, if it's worse, where T is the current temperature and α is a hyperparameter. We somewhat arbitrarily chose α=0.05 and decrement T by 1 every iteration (recall it is initialized by the provided initial temperature parameter). [2] Due to the complexities and time constraints of running our algorithms for many iterations on the physical robot, we were unable to experiment with different temperature decay functions (ex. exponential decay instead of just subtraction) or tune α based on experimental results.

The crux of the simulated annealing comes from selecting an optimal neighbor to search against for every iteration, which we implemented in SimulatedAnnealing.get_neighbor(). Again, due to the nature of our robotic setup and the speed of executing a single search iteration in real time, we decided to limit the algorithm to searching exactly **one** possible neighbor on every iteration. To select a single optimal neighbor to check against per iteration, we search a "close" local neighbor, similar to ILS. In particular, we perturb a single randomly selected coordinate from the current position by disturbing each dimension of that coordinate by a random variable defined by N~(0, (max value for that dimension – min value for that dimension) / 6) - in other words, each dimension for the randomly selected coordinate is shifted by a normal distribution that spans the entire range of that dimension's values (the standard deviation is the range divided by 6 as almost 100% of points in a normal distribution lie within 3 standard deviations to the left or right of the mean). By perturbing dimensions by a normal distribution, we disproportionately favor perturbed positions that are closer to the current coordinate.

After all iterations are completed, SimulatedAnnealing.search() returns the global best found across all the neighbors searched.

B. *Genetic mutations with VNS*

Genetic algorithms mimic natural evolutionary processes, utilizing mechanisms like selection, crossover, and mutation. In our context, each 'individual' in the population represents a set of waypoints for the robotic arm's movement. [1] The fitness of each individual is evaluated based on the objective function that combines re-projection error and total run time.

The genetic mutation component introduces variability into the population. It randomly alters parts of an individual's waypoint data, potentially leading to new and more effective solutions. This process is crucial for maintaining diversity in the population and avoiding early convergence to poor solutions. [1]

Then, VNS is used to complement the genetic algorithm by systematically changing the neighborhood within which the search is conducted. It enhances the algorithm's ability to escape local optima and explore a more diverse solution space. The VNS process in our implementation involves 'shaking' the current solution by making random changes within a defined range and then applying a local search to refine the new solutions. [1]

The Python implementation, illustrated in the attached code, utilizes a two-class structure: RobotControl and Optimizer. The RobotControl class controls robotic arm's movement and calibration, while the Optimizer class is where the genetic mutation and VNS are implemented.

Key methods in the Optimizer class include:

create_random_individual: Generates a new set of waypoints.

crossover: Combines parts of two parent solutions to create a new one, mimicking biological crossover.

select_parents: Randomly selects two individuals from the population to be parents for crossover.

local_search: Refines a candidate solution by iterating through its neighborhood.

mutate: Introduces chaos by giving the child a random mutation within a waypoint.

shake: Randomly alters a solution to create a new starting point for local search.

optimize: The main loop that iteratively evolves the population, evaluates fitness, and applies VNS.

The objective function combines re-projection error and total time, weighted appropriately to balance calibration accuracy and efficiency.

C. *Parent Selection with VNS*

The objective function still combines re-projection error and total time, as discussed in the previous two sections.

Parent selection is a method used to choose potential 'parent' solutions from a population, which are then combined to produce 'offspring'. In our implementation, this process involves choosing sets of waypoints that have shown promise in minimizing re-projection error and optimizing total run time.

The absence of mutation in this approach means that the new generations of solutions are entirely dependent on the characteristics of the selected parents. This method can lead to a more stable convergence pattern, as it avoids the randomness introduced by mutations, but it also risks converging to local optima if the diversity in the parent pool is low. However, in our case, this performed better than the random mutations involved in the previous solution. This is most likely due to the inherit chaos that mutations contain, and in the case of robotic arm movements, is very undesirable.

The Python code structure remains similar to the genetic mutation with VNS, utilizing the RobotControl and Optimizer classes. However, the key difference lies in the Optimizer class, especially in the absence of a mutation function.

Key methods in the Optimizer class include:

select_parents: Chooses two individuals from the population, focusing on those with better performance metrics.

crossover: Merges parts of the two parent solutions to create a new solution.

shake: Introduces variability by randomly altering a solution, providing new starting points for local search.

local_search: Iteratively refines a solution within its neighborhood.

optimize: Conducts the main evolutionary loop, focusing on parent selection and the application of VNS.

The objective function remains the same, emphasizing the minimization of re-projection error and total calibration time.

The implementation of parent selection with VNS was rigorously tested through simulations. Compared to the genetic mutation approach, this method demonstrated a more consistent convergence pattern, thanks to the inherit chaos of mutations being absent.

V. Experimental Results

A. *Experimental Setup*

The benchmark of the trials was a set of default waypoints that has been used in the development of the calibration algorithm. Most algorithms started from this as their initial guess.

The experiments were run on a Ubuntu 20.04 machine with a ROS 1 Noetic environment. Some additional code was added to the algorithm scripts to allow interfacing with the robot and cameras.

B. *Experimental Results*

Due to lack of possibility to simulate this optimization problem in any realistic manner, experimental testing is essential. Since every trial involves robot movement and calibration algorithm operation, experimental testing capacity is limited to the magnitude of hundreds of trials. In addition, robot control programs have a small probability of faulting during operation. Thus, not every algorithm was tested to the same number of iterations.

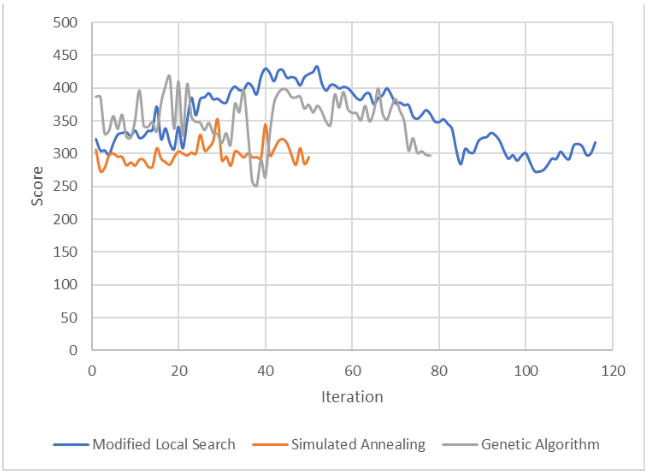

Figure 1: Experimental results of the optimization algorithms

The above graph shows the score of each algorithm at each iteration. None of the algorithms were able to produce a score that was significantly better than the initial score. However, genetic algorithm may have possibly found a local minimum around its 40th iteration. That local minimum is also the best score out of all trials of all algorithms.

Note that comparative framework is based on iterative performance, measured in terms of a 'Score' metric. It is important to note that this metric is presumed to represent the solution quality obtained by each algorithm.

The Modified Local Search (MLS) algorithm, with VNS and parent selection, showed superior performance throughout the experimental trials. A consistent lead in score from the initial to the final iteration was observed (see Fig. 1). Notably, the parent selection mechanism within the MLS algorithm seems to have played a pivotal role in its success. The modified algorithm not only started from a higher baseline score compared to its counterparts but also demonstrated a stable increase and maintenance of solution quality. This could be attributed to an enhanced balance between exploration of the search space and exploitation of promising regions, while inherently having much more stability than the genetic algorithm with mutations.

Simulated Annealing (SA) displayed a high degree of variability in its score, particularly during the initial phase of iterations. This behavior aligns with the theoretical underpinnings of SA, where a higher temperature parameter at the start enables a broader search, including the acceptance of suboptimal solutions to escape local optima. As the iterations progressed, a notable stabilization in performance was shown, showing the algorithm's cooling mechanism and its approach to convergence. [5]

The Genetic Algorithm (GA) portrayed an intermediate performance between the MLS and SA algorithms. Despite its robust search capabilities, it did not achieve the same level of solution quality as the MLS. The performance trajectory of the GA suggests a slower rate of convergence, [5] potentially implicating the need for parameter tuning or a review of the crossover and mutation processes within the algorithm, or omission entirely.

A convergence analysis indicates that while all algorithms trend towards a plateau in solution quality, the MLS algorithm achieves a more rapid and stable convergence. This observation underscores the MLS algorithm's efficient search capabilities, which are critical in applications where quick and high-quality solutions are vital. [5]

The experimental results prove the effectiveness of the MLS algorithm. This finding suggests potential for the MLS algorithm in applications demanding high precision in solution quality. However, these results warrant further investigation and testing across various problem sets to affirm the algorithm's usability and reliability.

VI. Conclusion

Through our tests and analysis of our experimental results, we have determined the most efficient optimization method which would enhance the precision of the sensors present on a robotic arm. The main objective was to find the best method to yield the lowest reprojection error weighted against the run time, therefore evaluating both accuracy and speed of our sensor calibration.

The core of this investigation had us iterate through several different optimization strategies, with the three most interesting and successful ones presented in this paper. At first, dummy simulations were run to ensure the algorithm created would work towards an appropriate solution, then were implemented, and ran on the robotic arm, logging the results with every iteration.

The GA with VNS showed a slower convergence, suggesting that further refinement in its parameters may yield stronger results. Simulated annealing showed strong results, however, contained a lot of variability between each iteration, given its nature. However, the MLS algorithm tested was found to perform the best given its time and accuracy.

VII. Future Work

This initial optimization study shows multiple areas for potential improvement. One significant factor noticed during the study is that there may be wasted time during calibration due to the calibration target not being in the camera’s field of view. This results in fewer usable data frames, and thus would significantly speed up the calibration algorithm. However, this is counter-productive, as uncertainty increases with less frames analyzed. Thus, inclusion of the usable video frames ratio should be implemented in the future.

With regards to the metaheuristic optimization algorithms, a custom Variable Neighborhood Search that has the ability to define neighborhoods tailored to each waypoint may be ideal. This could reduce the complexity of the problem without sacrificing the optimization potential. If each coordinate is optimized within their own quadrant neighborhood, each coordinate can step in their own best direction.

With regards to the meta-heuristics optimization algorithm, a custom Variable Neighborhood Search that has the ability to define neighborhoods tailored to each waypoint may be ideal. This could reduce the complexity of the problem without sacrificing the optimization potential. If each coordinate is optimized within their own quadrant neighborhood, each coordinate can step in their own best direction.

References

[1] K. Sun et al., “Hybrid genetic algorithm with variable neighborhood search for flexible job shop scheduling problem in a machining system,” vol. 215, pp. 119359–119359, Apr. 2023, doi: https://doi.org/10.1016/j.eswa.2022.119359.

[2] F. Liang, "Optimization Techniques — Simulated Annealing," Medium, Apr. 21, 2020. https://towardsdatascience.com/optimization-techniques-simulated-annealing-d6a4785a1de7

[3] "IMU-Camera Calibration using Kalibr," Robotics Knowledgebase, May 07, 2019. https://roboticsknowledgebase.com/wiki/sensing/camera-imu-calibration/

[4] "Objective Function: Definition, Types, Formula, and Examples," GeeksforGeeks, Jul. 19, 2023. https://www.geeksforgeeks.org/objective-function/

[5] "Convergence of Algorithms — Scientific Computing with Python," *caam37830.github.io*. https://caam37830.github.io/book/01_analysis/convergence.html#:~:text=Alternatively%2C%20we%20can%20monitor%20the (accessed Dec. 02, 2023).