POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

Development and validation of a parametric optimization tool for internal combustion engines based on Gasdyn

TESI DI LAUREA MAGISTRALE IN
MECHANICAL ENGINEERING
INGEGNERIA MECCANICA

Author: **Carlos Enrique Ramírez Ovalle**

Student ID: 11067897
Advisor: Gianluca D'Errico
Co-advisor: Andrea Marinoni
Academic Year: 2025-26

ACKNOWLEDGEMENTS

First, I want to thank Professor D'Errico and Andrea Marinoni to help me throughout the development of the project by listening to me and making constructive suggestions that made the project reach its current state. I would like to thank you for your patience and commitment which made me felt supported at every stage.

I want to thank my girlfriend, Laura, who has always been present, listening to me, cheering for me and providing me with the support I needed during the hard weeks. I have to say any of this would have been possible without you, so this achievement is as mine as it is yours. I hope you recognize yourself across these pages, it was you I was trying to explain everything to. Do not doubt there is love in every word.

I also thank my family for their constant support and sacrifice without which I would never have had the opportunity to come to Milano.

I would also like to thank God for giving me the strength whenever I felt I was falling.

And I also thank you readers for checking the work I believe is worth looking at.

ABSTRACT

Traditional internal combustion engine calibration extensively relies on experimental test benches and expensive Design of Experiments (DoE) matrices, which are heavily constrained by time, cost, and increasingly stringent environmental regulations. While 1D fluid dynamics simulation tools like Gasdyn offer an accurate, physics-preserving alternative to empirical setups, they often require external standalone scripting architectures to run optimization loops. This thesis presents the development, structural implementation, and multi-variable validation of an embedded parametric optimization tool integrated directly within the Gasdyn 1D simulation software suite.

The software architecture is built on a multi-hierarchical modular framework using Python. It offers generality by using a parametric approach to optimization problems that can be implemented to almost every variable inside Gasdyn and modularity by implementing new functionalities through auxiliary python classes. The tool implements two primary optimization schemes: a highly modified, discrete Mesh Adaptive Direct Search (MADS) algorithm that uses dynamic polling direction sets based on historical vectors of success, and a commercial genetic algorithm (SciPy's Differential Evolution). Both are adapted to respect distinct physical parameter accuracies via general rounding methods. Additionally, an exterior penalty function method was incorporated to effectively enforce complex mechanical and environmental pollutant emissions constraints. Computational efficiency is prioritized through an automated dictionary-based cache system and user-defined stagnation limits.

Validation campaigns were executed using a rapid-simulation single-cylinder engine model at 6000 rpm and an experimentally verified V12 Lamborghini engine model across operating spectra from 1000 to 8000 rpm. The framework successfully optimized continuous variables (Variable Valve Timing and spark advance strategies) and handled binary variables (opening and closing of duct length modifying elements). After the application of full calibration trajectory (variable valve timing, spark timing and intake duct length) the algorithm achieved an average increase of 5.6% and 6.5% for volumetric efficiency and torque respectively. The tool was successful at showing its capacity of being used for general optimization problems including highly complex ones that mix discrete and continuous variables and offers an initial robust framework for further developments that generalize even more its use.

Key words: Internal Combustion Engines, 1D Models, Mesh Adaptive Direct Search, Differential Evolution, Engine Design Optimization, Black-Box Optimization, Gasdyn.

SOMMARIO

La calibrazione dei motori a combustione interna si basa tradizionalmente in modo estensivo su banchi prova sperimentali e costose matrici di “Design of Experiments” (DoE), fortemente limitati da tempi, costi e normative ambientali sempre più stringenti. Sebbene gli strumenti di simulazione fluidodinamica 1D come Gasdyn offrano un'alternativa accurata e fisica ai setup empirici, essi richiedono spesso architetture di scripting esterne e standalone per eseguire i cicli di ottimizzazione. Questa tesi presenta lo sviluppo, l'implementazione strutturale e la validazione multivariabile di uno strumento di ottimizzazione parametrica integrato direttamente all'interno della suite software di simulazione 1D Gasdyn.

L'architettura software è sviluppata su un framework modulare multi-gerarchico in Python. Essa offre generalità grazie all'utilizzo di un approccio parametrico ai problemi di ottimizzazione, applicabile a quasi tutte le variabili interne a Gasdyn, e modularità attraverso l'implementazione di nuove funzionalità tramite classi Python ausiliarie. Lo strumento implementa due schemi di ottimizzazione principali: un algoritmo “Mesh Adaptive Direct Search” (MADS) discreto e altamente modificato, che utilizza vettori direzionali di polling dinamici basati sullo storico dei punti di successo, e un algoritmo genetico commerciale (Differential Evolution di SciPy). Entrambi sono adattati per rispettare le specifiche tolleranze e risoluzioni fisiche dei parametri tramite metodi generali di arrotondamento. Inoltre, è stato incorporato un metodo con funzione di penale esterna per vincolare efficacemente i complessi limiti meccanici e di emissione degli inquinanti ambientali. L'efficienza computazionale è prioritaria grazie a un sistema di cache automatizzato basato su dizionari e a limiti di stagnazione definiti dall'utente.

Le campagne di validazione sono state eseguite utilizzando un modello di motore monocilindrico a simulazione rapida a 6000 giri/min e un modello di motore V12 Lamborghini verificato sperimentalmente su uno spettro operativo da 1000 a 8000 giri/min. Il framework ha ottimizzato con successo variabili continue (strategie di “Variable Valve Timing” e anticipo d'accensione) e ha gestito variabili binarie (apertura e chiusura di elementi di modifica della lunghezza dei condotti). In seguito all'applicazione dell'intera traiettoria di calibrazione (fasi valvole, anticipo d'accensione e lunghezza dei condotti di aspirazione), l'algoritmo ha ottenuto un aumento medio del 5.6% per il rendimento volumetrico e del 6.5% per la coppia motrice. Lo strumento ha dimostrato con successo la sua capacità di essere impiegato per problemi di ottimizzazione generale, compresi quelli altamente complessi che combinano variabili discrete e continue, offrendo un solido framework iniziale per futuri sviluppi volti a generalizzarne ulteriormente l'utilizzo.

Parole chiave: Motori a Combustione Interna, Modelli 1D, Mesh Adaptive Direct Search, Evoluzione Differenziale, Ottimizzazione per Design dei Motori, Ottimizzazione Black-Box, Gasdyn.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

The design of fluid machines has historically relied on simplified models and experimental prototyping that allows to validate the sizing procedures that are achieved using these simple models. Mean line codes in turbomachinery, for example, are based on fluid dynamic considerations along with thermodynamical equations accounting for compressibility. More complex analytical modelling is added to account for the presence of variations in radial direction such as stream tubes approach. Internal combustion engines are another group of machines that had been designed using correlations allowing the designer to approximate the final dimensions, for example the duct length. Intake and exhaust duct lengths can be analytically sized considering the dynamic characteristics from the waves that exist inside these systems. Even if analytical models are not totally accurate, they effectively give first guesses about how the design procedure should go on.

Before the advent of high and cheap computational power, all the conclusions collected from the models and the expertise from engineers had to be tested in special measuring rigs. These experiments always have some special mismatches with the actual operating scene but are still the only way to verify if the guesses made match the reality of the operation of the system. Rigs are of many types, and they can go to highly complex ones as mountings that allow engineers to test the whole machine to wind tunnels where scale models can be tested. This works as a feedback loop where the experiments are used to reinforce the models and reduce the errors. One relevant example of this way of working is the famous loss correlations for turbomachinery. These relations are derived from physical principles, but their coefficients can only be derived and certified experimentally, and this is exactly what engineers have been doing across history in many design areas.

Another way to use experimental results is by testing specific components to enhance the modelling from them. Poppet valves are a famous case for internal combustion engines. These valves have been measured in special rigs to verify how their flow coefficient varies as the lift of the valve is increased. As commented in [1] poppet valves present a highly unstable behavior that depends on the pressure variations upstream, generated by the already commented pressure oscillations in the intake and exhaust ducts and on the way in which the flow area is modified as the crank angle is modified. However, it has been observed that testing in steady conditions, that can be achieved in a rig, helps to understand the problem and enhance the modelling that engineers have of this highly relevant system. Notice that this is still a model of the system and that the experimental rig does not actually match the real operating conditions. This leads to the fact that further testing was needed with the whole machine to guarantee that the developed models were working. In the same reference they state that the models correctly matched the experimental results.

One large challenge regarding this collected information is the sharing of it. This information is normally developed in-house and used only by the engineers from the companies that have developed the models, generating a technological disadvantage between different design groups. This is particularly observable in the world of turbomachinery where companies save their loss correlations jealously to keep their advantage. This is not to say that this is wrong, it is part of normal competition between companies but states the point about a way to generate this information

without the need for the high time and money costs that are normally introduced when experimental measurements are done.

As commented in the past few paragraphs, experiments can be used for modelling the machines during the design procedure but also to verify their performance and validate models. These experiments are highly relevant but complex. For example, the test rig explained in [2] was specially specialized since it was a one-cylinder engine that had its head made with quartz. Quartz guaranteed a visually accessible combustion chamber that enabled the verification of the way in which the flame front was formed. Additionally, quartz has piezoelectric characteristics so it could also be used to measure the pressure vs crank angle curve for this specific cylinder. This is commented to emphasize the complexity and technical rigor that is used when preparing these verification experiments. This introduces money and time challenges that make them basically prohibitive to fast-paced industrial design groups. Therefore, engineers need tools that allow for fast verification with enough accuracy to avoid the expensive repetition of these experiments.

The designing of machines through the modelling and verification tools have been largely altered by the arrival of cheap computational power. This has led to the use of simulation tools that enable the use of more complex analytical models that intend to describe the whole flow dynamics inside a fluid machine. In turbomachinery it is obvious that the flow field is extremely distorted due to the presence of complex and intricate geometries, cross flows and different flow regimes. Internal combustion engines are not different. Constantly they experience the effects of pressure oscillations in intake and exhaust systems, sonic choked flows in the valves, in cylinder motion to enhance turbulence inside the mixture, chemical diffusion when fuel is sprayed in port fuel injection systems and in direct injection systems and in many other conditions where understanding the flow is imperative. The detailed understanding of the flow is achieved when all the parts of it are determined thermodynamically by understanding their velocity, temperature, pressure, density and entropy. This can be done by using computational fluid dynamics (CFD) tools.

CFD tools have as main objective the solution of the thermodynamic states inside a flow. This is done by modelling the flow using the Navier-Stokes equations. This set of relations is formed by three partial differential equations that describe vectorially the flow considering mass conservation, energy conservation and the application of Newton's second law to the specific conditions of a differential element of fluid. They are not solvable by themselves; they need two additional ingredients. First, an equation of state that describes the relationship between basic thermodynamic quantities for a fluid and second, boundary conditions that allow the mathematical solvers to correctly solve these equations. The solution is based on finite element modelling where the flow is subdivided into individual elements of flow and all the properties are measured in the middle of each element, then the partial differential equations can be manipulated to become a set of solvable equations, and the thermodynamic states are found at each of these nodes. Then the information is extrapolated to other regions by using underlying functions that model the flows, just like shape functions in dynamic finite element analysis.

CFD has been shown to be a robust modelling tool and even if it continues to carry some of the problems previously mentioned from the experiments, it can run accurate verifications that can be easily modified to check new configurations. The problems previously mentioned are related to how expensive testing can be, specially when talking about time, and the fact that you still need some

experimental validation to guarantee the results are accurate. This last problem has not yet been lifted in any design approach. The computational power that is needed to solve these problems can be high, making the simulations extremely time-consuming and the idea of using it as an engine model basically intractable. Therefore, it can only be used as final verification through a single simulation or to closely study the engine. Normally, they focus on this second aspect by modelling specific components, such as the combustion chamber or the poppet valves to verify models and qualitative behavior since simulating a complete engine is still a huge computational challenge. For example, CFD can be accurately used to verify how in-cylinder motion of the charge is developed depending on the modification to the head of the piston. This helps to model these phenomena without the need of introducing complex measuring systems into a piston to measure the motion of the charge. Furthermore, you can use it to test new geometries of the combustion chamber to optimize this characteristic. It has also been used to study the scavenging of two stroke engines. In general, most of its recent use is in highly localized transient phenomena and offers detailed insights into these complex phenomena, but its computational cost makes it prohibitive when trying to use it to understand the general behavior of the engine.

This is where 1D models enter the scene. 1D models are still based on the Navier Stokes equation as described earlier but make a fundamental simplification. The flow is not studied anymore by using the complete set of equations since only a single direction of the flow is studied, equations are not vectorial anymore. Instead of generating an enormous three-dimensional grid they simply take the flow direction and portion the flow along it into finite elements. Using the same principle as the one presented earlier, taking a single representative node to calculate the properties all along the element, these models can predict the general performance of the engine with way fewer nodes. This offers a model that can be readily solved to generate new data. The agreement of these models with internal combustion engines was discussed by Heywood in [3] and it was confirmed that when fitted with experimental data they are capable of accurately predicting the general performance of the engine.

This completely modifies the design procedure as it was described earlier since a simple model exists now. This model can be first generated through the same procedure as was done originally, taking models and empirical knowledge and fitting them into a prototype. This prototype can be verified by using the 1D model and checking if the model is functional and getting the expected performance. Then 3D CFD validation can be done to verify if the model is approximating the data correctly or to enhance it in such a way that it does. With the verified model near final geometries are achieved and the engine can be constructed and tested. During the first construction of the engine, it must be rigorously used for experimentation, and the collected data is fed into the 1D model to enhance it and achieve better correlation. In this way future iterations of the engine can be first verified with the 1D model and then tested experimentally. Notice that the trial-and-error loop based on experimentation that was described at the beginning of this section totally disappears, the engineers are not in need of doing expensive experiments to carry out improvements, with a single validated model a whole family of engines can be developed and modified depending on the application.

Keeping in mind the previously commented idea the next step is to use the model to maximize performance at given operating conditions. Imagine that a client is asking for a completely new application for the same engine. Fitting it without the previously developed models would need extensive testing but now there is a fast computational model that can be used to achieve the

desired performance. This is where the importance of optimization tools arises but is not the only application scene that there is.

Optimizing an internal combustion engine is not an easy task because it is a nonlinear optimization problem due to the large number of parameters that are closely related to each other. It also manages different variable types. In some cases, it may even introduce discrete variables, for example when using open and closed valves. This means that it can be said accurately that the optimization problem that is faced when optimizing these machines is highly complex under the light of what was commented by Belotti et. al. in [4]. An optimization tool for this application must be robust enough to face the challenges of a multiple integer non-linear problem (MINLP) but also guarantee the physics of the problem keeping in mind that the optimization variables must comply with specific accuracies and should have physical meaning. This introduces a set of constraints that will further increase the complexity of the problem. Figure 1 presents the general structure for an optimization tool for this problem by using a 1D model for the engine.

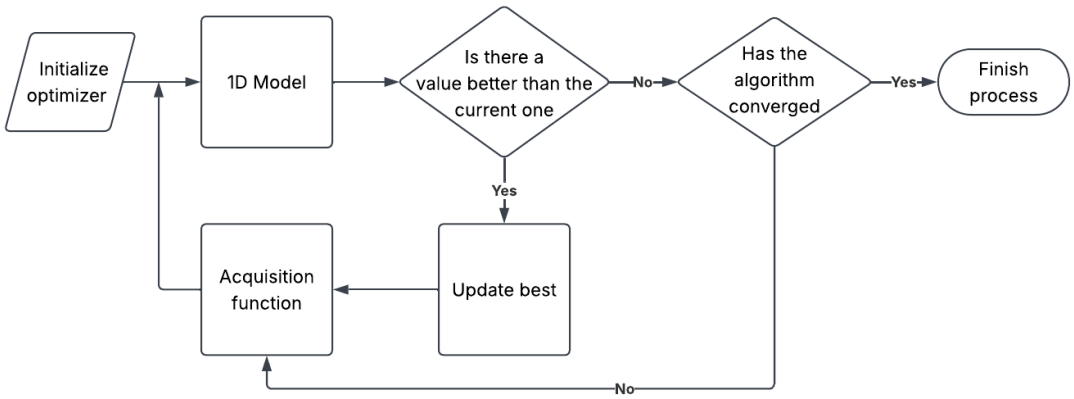

Figure 1. Optimization loop for a tool based on a 1D model for the engine

It can be observed in figure 1 that the optimization loop relies on the 1D model. This is a major difference between normal optimization systems and the one presented in this figure, there is no true mathematical modelling of the objective function, it is an experimentally based method. This suggests the use of experimentally based optimization algorithms such as Bayesian optimization, mesh adaptive direct search (MADS) algorithms or other heuristics such as evolutive algorithms. Any of these mathematical tools can be used but they must guarantee that the restrictions of this individual problem are complied with. The main part of any of these algorithms is the acquisition function, which will decide which new points to test to identify better points considering a certain objective function. One approach is to consider the constraints through the objective function which must consider this and the main physical variable that wants to be optimized. Constraints may include environmental ones while the physical variables to be optimized include the relevant performance metrics for an internal combustion engine.

The correct implementation of this tool leads to maximizing the usage of such a comprehensive approximation as the 1D model for the engine and closes the whole design loop. To summarize what has been discussed up until here, the importance of simulation tools in the context of internal combustion engine design is not little. These tools have reduced the cost and time needed for the

development of internal combustion engines by keeping the use of experiments only for validation and verification of models. 3D models are really accurate and are worth using where local phenomena with a lot of transients need to be studied but their computational cost makes them prohibitive to study the general dynamics of an engine. For this task it is better to use a 1D model which has been verified to accurately match the general behavior of the engine after experimental validation. This 1D model is fast and can be used to rapidly modify things in the engine to change its behavior. This last characteristic can be coupled with an optimization tool as described in figure 1 to achieve rapid modifications to get desired performance for any application.

In general, this optimization tool can be used for two tasks, to increase the performance from an engine by changing its configuration or by tuning a pre-existing configuration by selecting the correct parameters to fit a certain desired performance. This second problem is one of the most important steps in the design of an internal combustion engine. These complex machines have many parameters that have a high level of synergy that must be considered to achieve the best possible performance. Relevant performance metrics include volumetric efficiency, brake torque and brake specific fuel consumption. Some of the parameters available for tuning are the valve timings, the spark timing and the exhaust gas recirculation strategy. Additionally, intake and exhaust system dimensions can be modified during the design phase to achieve specific performance fitted for special operating points. During recent years the problem has started to become more complex, the introduction of variable length systems for the intake of the engines or the possibility of modifying the lifting profiles are some of the most recent innovations that are making the problem more complex.

Yu et. al. commented on modern optimization methods to carry out this optimization process [5]. They also highlighted the original method to achieve calibration. Engines can be manually calibrated in experimental rigs where the range of possibilities is discretized forming a grid. Each node in the grid is measured to verify the performance metrics. The point with the maximum performance is selected as the calibrated point. This method is known as *manual test bench-based dyno calibration* and is only one of the different used methods. However, all of them rely on extensive experimental measurements.

The main problem with these methods is that the growing number of parameters and most importantly, the regulations being applied on engines, are making it highly expensive both time and moneywise to execute these procedures. An additional problem arises when thinking about the fact engines suffer from aging and their performance derates with the usage. To address it, engineers have also developed on-board calibration methods which can modify the operation parameters of the engine to ensure optimal performance even in these modified conditions. These systems collect data of the engine during its performance to effectively achieve the optimization.

The authors in [5] discussed the current trend towards not relying completely on measurements from the engine but also complementing them with the use of models based on discrete experiments. These experiments are done during a first phase called the design of experiments (DoE). Engineers use this data to build surrogate models that can be rapidly optimizable since they do not rely on the highly time-consuming experimental measurements that are needed usually. Patnaik et. al. [6] highlighted how machine learning methods can be used to utilize the data from the DoE to generate models that can accurately predict performance of the engine. Artificial neural

networks, for example, offer reliable prediction if trained correctly and can be rapidly evaluated thanks to their simple architecture. The fact that the model can be rapidly evaluated can be used to develop first guesses that can be used afterwards to smooth the manual calibration methods.

Still, this is not the only way to model the engine. As commented earlier, the 1D simulation model is also an accurate tool. Boccardo et. al. [7] used a model of this type that uses an artificial neural network to accurately define pollution metrics from the engine. This enlarges the fidelity of 1D models which can be complemented with robust systems to minimize their prediction error. These models are not as fast as the ones that can be developed with neural networks, but they are more accurate. Furthermore, they guarantee the physics of the problem without additional modifications. Therefore, it is feasible to follow an optimization process based completely on the results from these models and the fact that individual components can be better modelled using additional tools, as Boccardo et. al. discussed, makes them really robust.

Murugravel [8] included a complete description of the fundamentals for the Gasdyn solver developed by the Department of Energy Engineering at Politecnico di Milano. This code is a 1D simulation tool that was extensively exploited by the previously mentioned author for the tuning of internal combustion engines. The tool developed by Murugravel was the base point for the current project. This tool was completely developed on python and used the Gasdyn solver. It used a one-cylinder motorcycle engine for validation. A mesh adaptive direct search algorithm was employed to use the engine model for the calibration of the performance parameters by solving different optimization problems. The first part of the present work included the close study and correction of the *engine_optimizer* code developed in [8]. During this part the concepts and possibilities to enhance the code were observed. A new parametric tool implemented inside the Gasdyn software was compared with the enhanced version of *engine_optimizer* and an improvement plan for the new parametric tool was sketched. This plan was intended to enhance the capabilities of the new tool to equate the functionality level of both tools. The MATLAB based *OptimICE* code presented by Stringhetti in his thesis [9] was also used as input for the plan since it represented the other optimization tool that had been developed based on Gasdyn. This last tool was and still is the most complete tool of the three.

The final product of the current thesis is one functional interface inside the Gasdyn software that can solve multi-input constrained optimization problems and can do most of the tasks that the other two tools can do. The optimizer was demonstrated to have the ability to manage both continuous and binary variables. The most important difference with the previous optimization tools using the Gasdyn solver is that it is embedded inside the complete simulation suite. The previous versions worked as auxiliary scripts that could be used to execute optimization tasks. The present work focuses on this tool and its validation. Section 2 focuses on the code from a mathematical perspective and elaborates on the theory behind optimization and details the mathematics of the mesh adaptive direct search code that was implemented for this project along with some descriptions of a commercial genetic algorithm, SciPy's differential evolution, that was also implemented to match the optimizing capabilities of the *OptimICE* code and to demonstrate the use of commercial optimization software with the new tool. Sections 3 and 4 explain in detail the structure of the code and elaborate on its parametric, hierarchical and modular nature. These sections will also explain what the place of this new addon inside the whole Gasdyn environment is. Section 5 will focus on the final validation from the tool. Six experiments are discussed to deepen

the original contributions made in the work, discuss the applicability of the tool and compare the two optimization algorithms implemented to justify this redundance. Finally, section 6 concludes the text and discusses some future lines of work.

2. THE OPTIMIZER

The optimizer has two main components, the objective function and the acquisition function. The objective function is the function that is going to be optimized. The acquisition function is a mechanic that allows the algorithm to generate new trial points. The present section is formed by four sections. The first section will cover the objective function which is general for all optimizers. The second section will introduce the MADS algorithm and will describe how it was modified to create a custom acquisition function. Finally, SciPy's differential evolution's acquisition function will be briefly described.

2.1 THE OBJECTIVE FUNCTION

As commented in the introduction, the objective function is general to all optimization algorithms. It is formed by two parts, the objective itself and the constraints. The objective is the main part of the function and includes the engine metric that the user wishes to minimize. Constraints, on the other hand, can cover any simulation output. Constraints are added using the exterior penalty method as described by Rao in [10]. Constraints can be divided into inequality type and equality type. Inequality type constraints are directly added to the objective function as passive members, only active if the boundary is surpassed. While equality constraints are added as quadratic terms, which do not affect by themselves the convexity of the function. This method enables the use of any commercial optimization algorithm, even if it cannot manage constraints directly. Structures for the constraints are presented in table 1.

Table 1. Numerical constraint structure

Lower boundary	Upper boundary	Equality constraint
$W * \max\left(0, \underline{x} - g(x)\right)$	$W * \max(0, \overline{x} - g(x))$	$W\left(X_{obj} - g(x)\right)^2$

W represents a weight, it is an integer bigger than zero and enables the optimizer to know how relaxed the constraint is. If it is low, the optimization algorithm will be able to break the constraint when it is close to the boundary. Really large values make the constraints rigid, and the optimization algorithm will not surpass the boundaries. The function $g(x)$ represents the mapping between a certain trial point x and the values needed to calculate the constraints or the objective. A general form of the objective function is:

$$f(X) = -g_{obj}(X) + \mathrm{W}\left(\sum_{i=1}^{n_{bigger}} \max\left(0, \underline{x}_i - g_{bigger}(X,i)\right) + \sum_{j=1}^{n_{less}} \max\left(0, \overline{x}_j - g_{less}(X,j)\right) + \sum_{k=1}^{n_{equal}} \left(X_{obj_k} - g_{equal}(X,k)\right)^2\right)$$

The lower and upper boundaries needed for the constraints are represented by $\underline{x}$ and $\overline{x}$ respectively while X_{obj} represents the target values that the user wishes to enforce. Notice that this applies for an objective that is going to be maximized. If the opposite case is studied, the first part of the function must be positive. It is also possible to optimize a parameter to reach a target. To do so, the

first term of the objective function is eliminated, and this target is included inside the equality constraints term.

2.2 THE MADS ACQUISITION FUNCTION

The acquisition function serves the purpose of using the gathered information to decide which new points should be tested. This is highly dependent on the type of algorithm that is being used. The developed code uses two optimizers, MADS and SciPy's differential evolution. The implemented MADS algorithm is based on what was exposed on the seminal paper by Audet and Dennis [11]. This reference proposes the main steps of the algorithm. This part of the document will provide a brief description of the algorithm and will highlight the mathematics from the custom implementation. From here after the Audet and Dennis algorithm will be called the original MADS algorithm while the implementation here presented will just be the MADS algorithm.

The original MADS algorithm tries to minimize a function $f(x)$ in a continuous space Ω that is discretized following some mesh size parameters Δ. There are two mesh size parameters Δ_k^m, the mesh size parameter and Δ_k^p, the polling mesh size parameter, for each iteration k. These two parameters are linked by the following inequality $\Delta_k^m \leq \Delta_k^p$ which must be satisfied for all iterations k. The algorithm searches along the mesh with mesh size Δ_k^m in a search step, generating the set of trial points S_k. The logic to generate S_k can vary and the reference proposes that any preferred method can be used. This set is evaluated and a better point is looked for inside it. Then, if no better point is found, it will look for a second set of trial points in a mesh with mesh size Δ_k^p by advancing in a set of predefined n_D directions, that form the matrix D with dimension $n \times n^D$, starting from the polling origin x_k, which is the best point the optimizer has found. This second set is called P_k and is defined in the following way:

$$P_k = \{x_k + \Delta_k^p d : d \in D_k\}$$

To summarize, the first step is called the search step, and the second one is called the poll step. One of the most important features of the algorithm is that the mesh size parameters can be modified dynamically. Normally, the mesh is coarsened when a better point is found and refined when no better point is found but this can be varied as needed. These dynamics are included to accelerate convergence when better points are found and to search locally whenever no better points are found. To do this, minimum and maximum values for the mesh size evolution can be defined along with a value τ which defines the dynamics of the mesh size parameters. The evolution for the mesh size is defined in the following way:

$$\Delta_{k+1} = \tau^{\omega_k}\Delta_k : \omega_k \in \begin{Bmatrix} \{1\} \mid f(x) < f(x_{k-1}) : x \in S_k \cup P_k \\ \{-1\} \mid f(x) \geq f(x_{k-1}) : x \in S_k \cup P_k \end{Bmatrix} \mid \Delta_{min} \leq \Delta_{k+1} \leq \Delta_{max}$$

In the case bounds are not respected the algorithm will fall back to the value at the crossed boundary. This defines the basic mechanisms for the original MADS algorithm. It can be noticed that this description can be organized in an algorithm with four steps: initialization, search step, poll step and updating of mesh size. In the initialization the parameters are defined: $f(x)$, Ω, x_0, $\Delta_0^m, \Delta_0^p, D_0, \tau, \Delta_0, \Delta_{max}$ and Δ_{min}. Then the search and poll steps are carried out and finally the algorithm updates its parameters following the presented logic until convergence is achieved. Notice the search and poll steps form the acquisition function of the algorithm.

Now the MADS algorithm is discussed. The main differences are the fact that Δ_k^m is a fixed value equal to the minimum accuracy for the studied variable and that the search step goes after the poll step and may even not occur depending on the availability of points for evaluation. Since this can be a multidimensional problem, Δ_k^m and Δ_k^p should be generalized as column vector with dimension $n \times 1$ where n is the number of input variables. To avoid this extra complexity, the polling step is redefined in the following way:

$$P_k = \{x_k + \Delta_k^p \alpha d : d \in D_k\} \quad (1)$$

The term α represents a $N \times N$ diagonal matrix containing the accuracies of each parameter on its diagonal. This new formulation means that the minimum value for Δ is simply 1. The optimizer knows all the feasible points at the beginning of the evaluation because all the feasible values in the design space are known. This is true because Ω needs to respect a certain accuracy for the parameters. The search step randomly selects values from the defined design space that replace previously evaluated points that belong to P_k, this is if there are points left to evaluate in the design space. Otherwise, no search step occurs. The final testing set is T_k. The set of points that have not been simulated is called the sampling space, σ, and it changes every iteration. If there are no points in σ, no search step occurs. This step serves two main purposes, it is a way to find other feasible minima, and it allows the algorithm to explore the design space extensively to reach conclusions about it.

One extra step is added before the initialization of the optimization algorithm, the design of experiments (DoE) phase. This is a uniform sampling of the design space Ω and is used to initialize the optimizer. The optimizer uses as initial polling point the minimum found during the DoE phase and initializes the sampling space σ by finding all the feasible points and eliminating from the set the points evaluated during the DoE. In each iteration, the evaluated points will be removed from σ. In iteration 1, it evaluates all the points surrounding the polling origin for the iteration, x_1, using the minimum mesh size. This means that $\Delta_1^p = 1$. The set of polling directions, D_1, for a two-input problem can be defined as:

$$D_{1_{N=2}} = \{(d_1, d_2)^T \neq (0,0)^T : d_1, d_2 \in \{-1, 0, 1\}\}$$

The set of directions to look at the surrounding region in all directions for a general dimension N is called D_1. A better value is looked for and Δ_2^p is calculated as described earlier. If a better value is found, the optimizer will sample points in the same direction as it has advanced. This is done to enhance convergence time. This means that the set of directions for iteration 2, $D_{k=2}$, is redefined as:

$$D_{k=2_{N=2}} = \{(d_1, d_2)^T \neq (0,0)^T : d_1, d_2 \in \begin{Bmatrix} \{0,1,2\} \mid x_{k+1}^n > x_k^n \\ \{0,-1,-2\} \mid x_{k+1}^n < x_k^n \end{Bmatrix}\}$$

The superscript from x tries to depict the fact that each component of d, each d^n, can be spanned from a different set depending on the direction of advancement for that particular dimension. This will form a set of 8 directions at every step k. It repeats this operation until it becomes ineffective, meaning that a possible minimum was reached. At this point the algorithm will go back to exploring the surrounding points which means it will use a set of directions equal to D_1 and the mesh size will be refined as described earlier. To generalize this behavior to N dimensions, the following mathematical expression can be used:

$$D_k = \{(d_1, d_2, \dots, d_N)^T \neq \vec{0}_{N\times 1} : d_1, d_2, \dots, d_N \in \begin{Bmatrix} \{-1, 0, 1\} \mid x_{k+1}^n = x_k^n \\ \{0, 1, 2\} \mid x_{k+1}^n > x_k^n \\ \{0, -1, -2\} \mid x_{k+1}^n < x_k^n \end{Bmatrix} \quad (2)$$

This means that there could be dimensions in which you do not advance and you study the surrounding area or where you are advancing in opposite directions. There is one peculiarity in the mesh size dynamics. When the algorithm restarts the search after identifying a candidate minimum, the mesh size drops down to its minimum value to keep the algorithm from getting too far away from the candidate region. Mathematically this can be written as $D_{k+1} \neq D_k \cup D_k = D_1$. If no better points are found the optimizer finishes and concludes the function was minimized. Remember that the algorithm never goes below the minimum mesh size, so refinement is disabled if the value of mesh size is equal to 1. Other introduced modifications for this step are that if a value added during the search step (a random value) is found to be the new best the mesh size will reach a value of 4 and that for the first three iterations mesh will be coarse even if no better value is found. The reason for the first modification is to make the search faster but controlled in the surroundings of this random value. This will allow the algorithm to explore its surroundings rapidly without getting stuck too fast. The second modification is also made to explore the surroundings of the best point found in the DoE. This is especially important when the functions show a difficult topography and out of luck the algorithm found a best value candidate in its first evaluation. A generalized form of the dynamic updating of Δ_k^p can be written in the following way:

$$\Delta_{k+1}^p = \begin{Bmatrix} \tau^{\omega_k} * \Delta_k : \omega_k \in \{1\} \mid f(x) < f(x_{k-1}) : x \in P_k \\ \Delta_k = 4 \mid f(x) < f(x_{k-1}) : x \notin P_k \\ \tau^{\omega_k} * \Delta_k : \omega_k \in \{1\} \mid f(x) \geq f(x_{k-1}) \cup k < 3 : x \in T_k \\ \tau^{\omega_k} * \Delta_k : \omega_k \in \{-1\} \mid f(x) \geq f(x_{k-1}) : x \in T_k \\ \Delta_k = 1 \mid D_{k+1} \neq D_k \cup D_k = D_1 \end{Bmatrix} \mid \Delta_{k+1}^p \geq 1$$

Notice no maximum mesh size was defined for this new implementation, and the minimum mesh size is equal to one ensuring that target values are always generated with the desired accuracy. It can be observed that the algorithm will generate exactly $3^N - 1$ trial points per iteration following the generalized logic for polling direction definition. The differences between the original algorithm and the one implemented will be further expanded and summarized in section 2.4.

2.3 THE SCIPY'S DIFFERENTIAL EVOLUTION ACQUISITION FUNCTION

SciPy's differential evolution algorithm was added to present the possibility of using commercial optimization software, to use it as comparison mechanism during the first stages of development of the MADS algorithm where the exactitude of the algorithm was not really known and to offer an alternative for optimization problems where the mesh is not as structured or the function topology presents multiple minimums close to each other. As its name suggests, this is an evolutive algorithm that has an acquisition function which produces candidates based on the mutation of its members [12]. This algorithm also takes a tolerance and finalizes when the dispersion of the testing population reaches a threshold value based on this tolerance. Detailed description of the mathematical behavior of this algorithm is out of the focus of the current document.

2.4 PRACTICAL IMPLEMENTATION

This section covers the practical implementation of the algorithm. The first task to be discussed is the acquisition function. Function $g(x)$ is formed by two parts. The simulation executor, in charge of compiling the input data, executing the simulation and recovering the output files, and the parser function, which extracts information from the output files. The parser function generates a dictionary with the required data. Objectives are extracted from the regimes.csv output file. This file contains some of the most important metrics from the motor such as brake torque, volumetric efficiency and brake specific fuel consumption (BSFC). The values available to build constraints include the values that can be extracted from the regimes.csv output file but also includes the values coming from the cylEmissWetFlow.csv output file. These last values are data from the emissions generated by the engine. The currently used mechanic can be generalized to apply constraints with any of the values in the output files.

Regarding the definition of constraints, one of the most important values to decide if a constraint is flexible or rigid is the weight W. For this case, constraints were thought of as rigid walls that the optimizer should not cross. To this end, the value of W was fixed to be 1e6. This value guaranteed the constraints worked as desired. There was just another value that was fixed which was the value for τ which was set to be two.

The following thing that will be discussed is the practical way in which the acquisition function of the MADS algorithm works. Each trial point is identified by a dictionary that uses as keys the parameter names and assigns the values for each one accordingly. Values are of the float type. Remember from the section dedicated to the acquisition function that, P_k is generated through **equation 1.** Then the search step is done by replacing any of the trial points that had already been evaluated with random points from set σ, thus generating the trial set T_k. To collect the previously evaluated data a cache was implemented. The cache works is a dictionary in which the keys are strings containing the data from the dictionaries identifying each trial point and the values are lists containing the value from the objective function along with the output directory from the simulation. This last directory is a fundamental variable because it allows the optimizer to reach all the data from a particular optimization. Keep in mind that if σ is empty, no random points are included and $T_k = P_k$. This last equality will also hold if all points in P_k are new, and therefore there are no dictionaries related to them in the cache. The implemented algorithm is programmed to execute random evaluations rather than using the cache to maximize exploration.

The amount of exploration after finding a minimum can be controlled through the maximum of the stagnation counter which is the number of iterations the algorithm executes without finding a better value. This is a user input. Stagnation counter is another one of the major practical differences that the implementation includes in the algorithm. Really low values are not recommended because they can lead to “kill” the simulations too fast while big values will make the optimizer more time consuming. The value of three was found to be good, but slightly larger values can enhance exploration and provide more information from the design space.

Simulations are executed in parallel if multiple points are given, enhancing their evaluation time. Keep in mind that SciPy’s differential evolution provides values one by one while the MADS algorithm provides a list.

It is important to clarify how the algorithm chooses a better value. To do this, the algorithm uses the objective function but also a user defined tolerance. The tolerance value is used to define the minimum difference two values should have for them to be considered different. This makes the possibility of better values arising without the algorithm updating the best value. To hint the user on this, the displayed result of the optimizer also includes the best values reached without considering the tolerance that could be found before the end of the optimization process. Lower tolerances can increase the execution time but are more exact. The physical reason for this is that the measuring instrumentation can only collect information with a certain resolution. In an experimental test, values that differ under the resolution of the instrument will not be detected as different.

Figure 2 presents an example for a single dimension optimization. In this example, $\tau = 2$. It can be observed in iteration 7 that this implementation of the algorithm goes back to the minimum mesh size if a directional search step is tried after the code got stuck. As commented earlier, this is particularly important to avoid the algorithm from going away from the region where the existence of a minimum is hypothesized. Two important remarks must be made, the algorithm was stuck in 9 and 7 during two iterations, this situation depicts the use of the stagnation counter. If the stagnation counter was set to be two, the algorithm would not have reached the minimum located at 7. The second remark concerns the importance of the random evaluations and the cache. Iteration 5 shows that the algorithm proposes two values that had already been evaluated in previous iterations. This means these values do not need to be simulated, instead the algorithm will take two random values from the design space to enhance the exploration. In case there are no available points to explore, the cache guarantees the values can be collected and used by the optimizer.

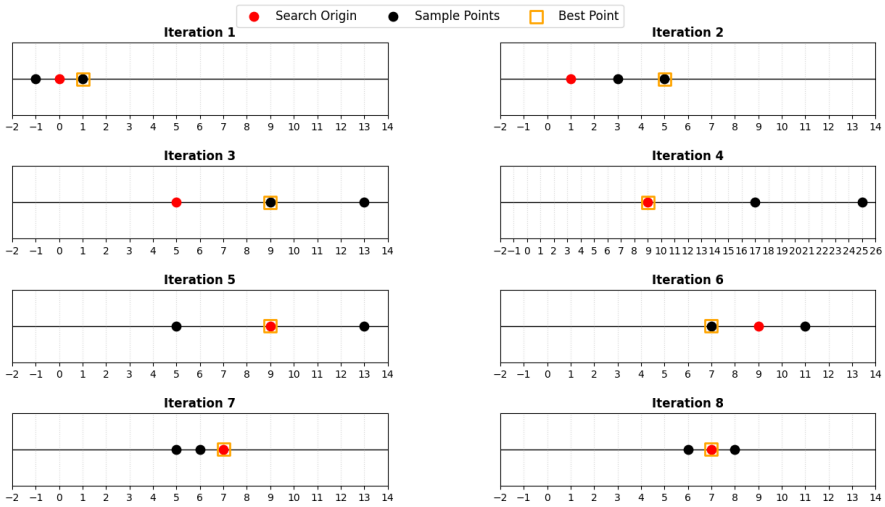

Figure 2. Depiction of a 1 input MADS process.

To summarize, The MADS algorithm here implemented has the following differences:

- Interchanges the order of the poll and search step prioritizing the poll step and making both a single step with its output being the ser of trial points T_k.
- The selection of directions for the polling step is dynamic and considers the direction in which successful advancements have been made.

- The mesh size dynamics only affect Δ^p, since Δ^m is fixed, and are modified to consider the way in which the optimizer has advanced to keep it from evaluating areas far away from the hypothesized minimum and enhance exploration.
- The algorithm is thought to optimize over a design space which is discretized to offer real physical meaning defined by the user defined accuracies.

Regarding SciPy's differential evolution the cache and the stagnation counter were also implemented. The cache is largely used for this type of algorithm because the evolution of the population can lead to proposing previously evaluated points. Consider that for the algorithm to signal convergence the members of the population should be similar to each other. In general, this algorithm depends greatly on the initial population, and it needs higher values of stagnation counters. This is because the algorithm is evaluating the whole population per iteration not only a single maximum value so the stagnation maximum should be relaxed. Notice that as convergence is close, less sample values will need evaluation since they will be already stored in the cache making the algorithm faster at these final iterations. One important part of the implementation that was done is the use of a rounding function. The acquisition function will generate mutations that do not respect the accuracy provided by the user. To avoid this to be a problem, all values are rounded to the accuracy needed before simulations are run. In this way it is guaranteed that even if the values are not respecting the accuracy, they are managed in a way that forces them to. This rounding function was also used to guarantee that the DoE values would respect the accuracy, since the mechanism to generate them did not consider the accuracies beforehand.

3. THE OPTIMIZATION RUNNER

The optimization algorithms presented in the previous section lie in the lower part of a hierarchical program that connects this logic with the user. Understanding the hierarchy is important because it depicts the way in which the user and the Gasdyn software itself can interact with the optimizers. To complete the description of the optimizer the optimization runner must be discussed. This part of the program is responsible for the communication between the optimization logic and the user during the execution of the optimization process and of the control of the optimization algorithm. Regarding the first task, this part of the code parses the output information and writes the output csv files that the user can check to observe how the optimization process is advancing. Regarding the second task, it is responsible for seeding the optimizer with the points from the DoE, manages the compilation of the data for simulations needed to collect numerical values for optimization and manages some of the termination mechanisms, specifically the maximum number of iterations one. This section will cover the most important tasks of the optimization runner beginning with a brief introduction to the queue type on python, fundamental for the understanding of the underlying mechanics of the interaction between the two hierarchies, then a detailed description of the interaction between both hierarchies is exposed and finally an outline of the operation of the runner is provided along with an explanation of how it does some of its most relevant tasks.

3.1 INTRODUCTION TO PYTHON QUEUE

The queue module is a python library providing the user with the ability to generate queues. Their main use is for threaded programming where they allow easy and smooth information exchange between threads [13]. The queues implemented on the code are from the type first in first out (FIFO). FIFO threads deliver the user information in the order it was delivered to the queue. When the user extracts the information from the queue it is eliminated from it. The importance from the method is that the queue will lock the thread where information has been requested until information is placed on it by another thread. This allows the program to run two processes simultaneously but somehow wait for each other to finish cooperating in the best possible way.

To implement it, the main thread and the “auxiliary” one are launched. The “auxiliary” thread is launched from the main one and they interact through the queues to get information from each other. To extract information from the queue, the “.get()” method is used. Every time this method is called the thread where it has been called is blocked until the other thread puts information into the queue. This is done through the “.tell()” method. As soon as this method is executed the blocked thread can retrieve the needed information and keep running. The next section will present the implementation of these mechanics in the real tool.

3.2 THE REAL INTERACTION BETWEEN HIERARCHIES

This mechanism is fundamental for the use of commercial software inside the optimization tool. Commercial software does not have a separable acquisition and objective function, both work together. Regarding the implementation of this commercial software, it must be noted that for it to evaluate its objective function the software needs to run a simulation. This means that the software needs to deliver information about the test point, somehow this point needs to be evaluated, and the information must be retrieved to continue with a new iteration to repeat the cycle. As

commented earlier, the task of the optimization runner is to compile, parse and run the simulations. Therefore, the information about the test point must be delivered to the optimization runner and the information from the simulation must be brought back to the optimizer to carry on with its process. Summarizing the logic, the optimizer must deliver a test point and stop until the result from the simulation is provided.

This is where the queues become relevant. The optimizer is one function in a larger file with other functions enabling communication between the hierarchies, rounding, etc. This file will be the main thread, and the optimizer will be run as the auxiliary thread. Threading only makes sense if the threads are intended to run in parallel, that is why communication between the optimizer runner and the main thread can be done without queues. The main strategy is to include the delivery and retrieval of information inside the objective function. The optimizer needs to run it for every point; therefore, it will place and recover information at every run. This communication is done as described earlier, through queues. The information is delivered and extracted from the optimization runner by calling the main thread through functions that extract and deliver the needed information from it. The code uses two main queues and one containing auxiliary information. The two main queues contain the test points and the results respectively, and the auxiliary one contains information about whether the test points need to be evaluated or can be extracted from the cache. This auxiliary queue was the implemented method to identify when the value could be extracted from the cache or not.

To extract information about the combos the function “ask” from the main thread is used. This function initializes the auxiliary thread when it is called from the first time and contains the extraction from the test points queue, this call will block the main thread. As soon as “ask” is executed the optimizer starts running in parallel and will introduce the test points at the beginning of the execution from the objective function then it will use the .get() method over the result queue and will get blocked until results can be retrieved. When the ask function has been able to recover the data from the queue it will deliver this information to the optimization runner which will run the simulation and will parse the results to deliver the value for the objective function along the constraint values, the original objective value and other relevant information. This data is embedded into a dictionary which is sent to the main thread through the “tell” function. This step ends the iteration on the optimization runner. The iteration number is verified to be lower than the maximum number of iterations placed by the user. If it is a new set of trial points is requested leaving the optimization runner on standby otherwise the simulation is finalized. The “tell” function will verify if the obtained value was better to monitor how the stagnation counter increases and will place the results into the results queue. If the stagnation counter exceeds its maximum the function will place empty values into the results queue and will signal convergence. If not, the result will be placed in the results queue, and the optimizer will be able to use it to decide which new values to run and will once again place test values into the queue and repeat the process all over again. The flow chart in figure 3 summarizes this whole process.

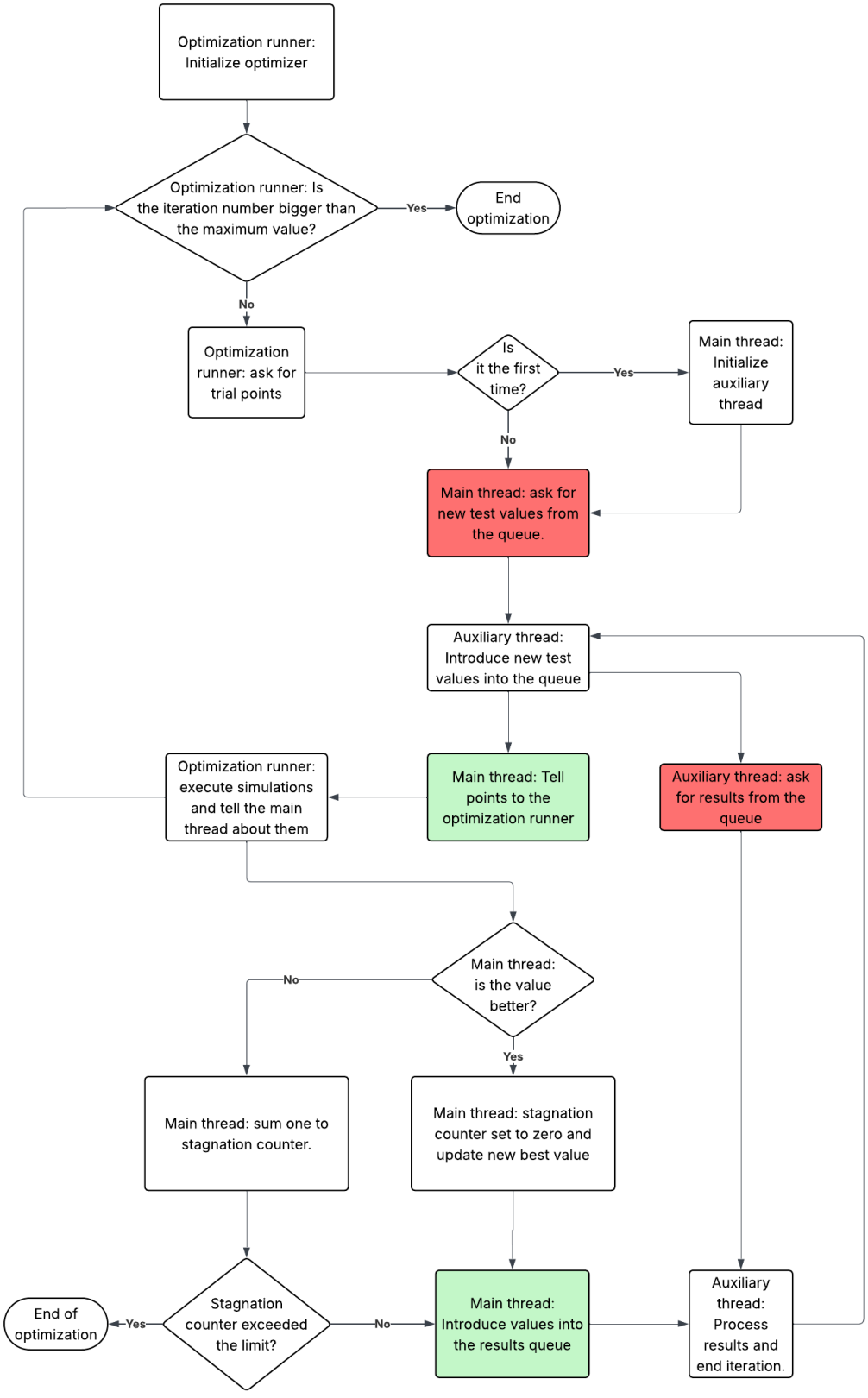

Figure 3. Flow chart depicting the interaction between the optimization runner and the file containing the optimizer

In the diagram the finalization routines are included along with a graphical representation of the points where the threads would stop (red squares) and continue (green squares). It is also evident that there exists a parallel relationship between the main thread and the auxiliary thread. One final thing is important to discuss to close this section and concerns the fact that other information is being moved back and forth between the main threat and the optimization runner. The cache is discussed first. As described earlier, it contains information about the points that were already simulated by using as keys the dictionaries characterizing these points and including a list containing the objective function value and the output directories as the values inside the dictionary. The initialization of the cache is done inside the main thread, but it is used on both hierarchies. On one hand the main thread is used to verify if the new test point has been evaluated before and to give this information to the optimization runner through the auxiliary thread. While on the other hand, the optimization runner uses it to extend it with new evaluations and to extract the results from the values simulated before. To send the cache back and forth two functions are implemented in the main thread. Both functions, like the ones used to send back and forth the data from the test points and the results, are executed on the optimization runner. The first function will deliver the cache to the optimization runner while the second one will assign the new version to the variable containing this information in the main thread.

As commented at the beginning of the discussion, queues are only exploited at their maximum when threading is employed. Threading only makes sense if multiple processes need to be executed in parallel. Keeping this in mind, the MADS algorithm did not need this basically because it can be employed sequentially since the acquisition function is decoupled from the objective function because it was especially implemented for the program. This means that there is no need to "trick" the optimizer to get the information from the acquisition function. This was one of the main reasons why the custom algorithm was implemented, a higher control of the mechanics of the optimizer was expected.

With this clarification done, it can be easily understood that there was no threading on this second algorithm. This simplified the problem and only two queues were kept, the test point queue and the auxiliary one. The queues were now more like dynamic lists where the data was eliminated after extraction, in this second algorithm they do not block any process. All the other principles for communication between the two hierarchies were exploited. Another relevant difference is the fact that the values inside the new test point queues were now lists of dictionaries including the whole set T_k.

3.3 THE OPERATION OF THE OPTIMIZATION RUNNER

The last section explained the relationship between the runner and the optimizer itself. Recall that one of the most important tasks of the optimization runner is to compile, execute and process the simulations that are carried out in Gasdyn. Up to now the parametric nature of the presented tool has not been elaborated. In this first part of the section this will be explained and in the next section the parameter creation and assignment will be explained inside the graphical user interface (GUI). The hierarchical performance described until now is only possible due to the clear existence of a differentiated set of tasks and enables the code to keep a friendly logic that will enable constant maintenance and future implementation of additional tasks.

The parameters can cover any numerical value inside the Gasdyn model. When assigned, internal subroutines can identify all the values bound to that single assignment. In the original MATLAB and Python optimizers depicted in [9] and [8] the compilation task only included the copying of the input files from the engine into the working directory, the generation of an additional input file, *input_optimice.dat,* and the inclusion of a special line inside the *input_gen.dat* file. This special line told the solver that it needed to look after the *input_optimice.dat* file and identify the variables that needed to be modified. For the solver to understand which variables to modify, special identifiers were generated for each of the possible variables that could be modified. Input valves, for example, used the identifier *AVO_VAR* along with the coordinates for the valve that was going to be modified.

The problem with this way to operate is that each class of optimizer needed to be developed from the beginning, the Gasdyn solver needed to be coded to understand a new identifier and a whole new python class needed to be developed to specifically manage that type of solver. The current method does not need an identifier. The current python code simply maps the parameter with its bindings and knows where to modify the assigned values. Additionally, the current compilation method directly modifies the input files from Gasdyn so there is no need to write the extra line inside *input_gen.dat*. This simplified the implementation of different types of optimization procedures that included, as commented earlier, any numerical variable that could be identified inside the model without the need of the direct modification of the solver.

One additional advantage was the ease the new mechanism had to execute multi-input problems. One of the main lines of work during the first phases of this project was to get the *engine_optimizer* framework to include this type of problem. They needed to be specifically implemented while in the new parametric framework this was not needed, the compiler understood what each parameter represented so it only needed to be told what values these parameters were taking. The main task of the compiler is this, copying the input files from the engine into the working folder and understanding where the parametric values will be used and to assign these values in the correct places. All of this is done by using a specialized class which was developed by different members of the Internal Combustion Engine group inside Politecnico di Milano (ICE group). This class is executed inside the optimization runner for achieving two things: the design of experiments and the optimization simulations. At the end of each compilation a class called job simulation info (JSI) is created and contains all the relevant information related to the job. To run a simulation the JSI can be delivered to the executer, and it will run the optimizer with the inputs inside the folder provided in the JSI and will place the results back in the same folder.

The executer, as the compiler, is an additional class that can execute all the possible tasks that are needed inside the program including the execution of simulations. This was provided by the ICE group as the compiler. The executer can take a list of JSIs and execute them, it is programmed to execute them in parallel if a list is provided to it.

With all of this defined the optimization of runner operation can be explained. The optimization runner will be launched for a single operating point, so it only manages one rpm. For multiple operating points, multiple runners need to be launched, each runner includes an optimizer of its own and they are launched in parallel through threading. The runner follows the logic presented in figure 3 to collect the point it will simulate. The compiler takes this point, which is a dictionary containing the names of the modified parameters along with their values, and compiles the input as

described earlier this leads to the generation of the JSI which is then executed using the executer and the final output is collected inside the working directory.

The output from a simulation is a series of csv files containing all possible relevant data that could have been collected throughout the simulation. These csv files need to be postprocessed to extract the needed value. To do this the Pandas library was used. This allows to easily transform the csv files into data frames and directly extract the needed values as numerical values. This can be done with any of the csv's which allows a generalization of the parsing method to use any of the outputs that the simulation provides. The compiler, executer and parser are the three things that form the functions $g(x)$ as described in the first section. They receive the testing point and extract the needed data for the calculation of the objective function and the constraints.

The next step is to generate the result that the optimizer uses to run its optimization loop. The result provided by the runner is a dictionary containing the values for the objective function, the objective itself, the amount of constraint violations, and the values for the constraint terms inside the objective function. This result is provided to the optimizer, and the simulated values are stored inside a class that stores the historical data from the simulation. Each entry inside the history class includes the iteration number, the point that was evaluated, the values from the results, the phase in which that specific value was generated (either DoE or optimization) and the output directory. After each iteration this class generates four csv files that record the optimization process. Two of them collect the information about the tested points and two of them about the best points. The reason why there are two is simply a matter of visualization. Two of the files contain copies of the entries from the regimes.csv files of all points. These extended versions can be used to have a more complete view of the simulations. The other files are simpler and allow the user to catch the most important information easily. These files can be used by the user to monitor the performance of the simulation, since as commented earlier, they are regenerated at the end of every simulation.

There are two runners in the code right now. The differences between the two of them are generated by the nature of their individual optimizers. The MADS algorithm generates a list of test points so the runner must be capable of managing parallel simulations while Scipy's differential evolution provides the runner with only one point. This is the only difference between the two of them and a generalization can be reached in future versions to further simplify the code. As seen up until here, the optimization framework here depicted presents a series of advantages that make it a robust tool. It offers a framework that is generalizable to any numerical variable inside the engine's model and allows the user to use both commercial optimization software and custom algorithms with an intuitive framework that can be understood by other developers to extend and improve the tool as it is. Furthermore, the framework allows for further generalization from its mechanics that can allow to enhance the benefits from it.

4. TOP LEVEL HIERARCHIES

This section will close the description of the code by introducing the final parts of the optimization framework. Two more hierarchies have been identified, the DoE Form and the graphical user interface from the whole program. The optimization runner related mathematics from the optimizers with the simulation environment; the DoE form relates the simulation environment with the user. This hierarchy includes the part of the graphical interface that enables the user to run the optimization problems that will be presented throughout the following sections of this document, but it also includes the logic that manages and launches the optimization runners along with the post processing capabilities that were implemented for this project. Additionally, it is also responsible for launching and managing the DoE. The first section will introduce the graphical user interface and will allow a visualization of most of the things discussed until now focusing mainly on the part regarding parametrization and optimization while the second one will discuss the logic behind the interface.

4.1 THE GRAPHICAL USER INTERFACE

The graphical user interface from the Gasdyn program was completely developed in python by the ICE group at PoliMi and was provided at the beginning of the project. The main library used for this task was *wxPython* a toolkit that allows the implementation of GUIs easily with open-source code. It is highly intuitive to use, and it presented no challenge at all to introduce the needed modifications to introduce the changes that were needed for the introduction of the logics beforementioned. The present document will focus on the section commented on the introduction of this section and most of the examples will be extracted from the optimization cases that were studied throughout the project. The starting point is an engine model that is intended to be optimized. The newest versions of Gasdyn include the parameter window as presented in figure 4. Here a table can be filled with the parameters that the user intends to optimize or simply to use in parametric design.

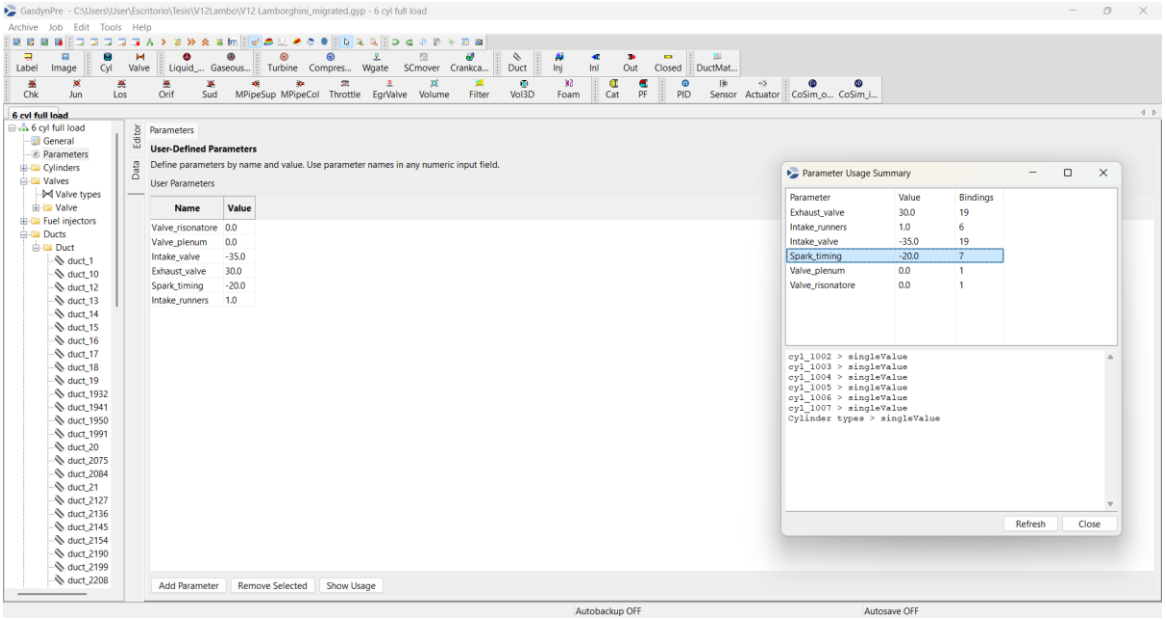

Figure 4. The Parameters Window with the show usage window

In the presented figure the variables represent binary variables representing the opening and closing of valves, the intake and exhaust valve timings from the cylinders in the engine, the spark timing and a scaling factor for the length of the intake runners. This is a clear example of the different types of variables that the users can use for the optimization tasks. These variables can be used as the user wishes. Throughout the following sections this will be depicted and the potential of the tool will be presented. In the lower part of the window the show usage button can be seen. This button allows the user to open the window observed in the right part of figure 4. This form can be used to observe all the sections in which a parameter is used. Keep in mind that depending on the parameter a single variation in the configuration might be bonded to many other parts of the model outside the section where it was assigned. With this tool these modifications can be tracked.

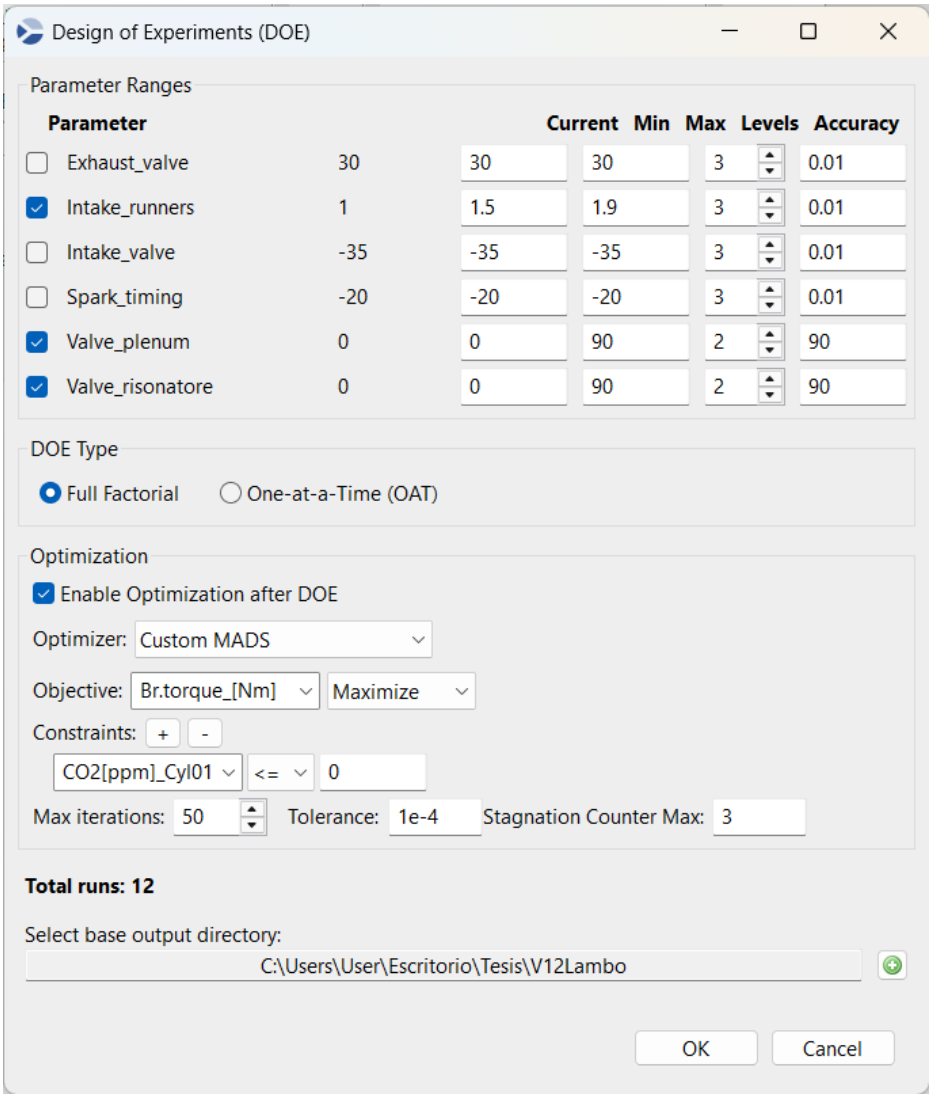

Figure 5. DoE form

The next important frame is the DoE form itself, and it is presented in figure 5. The first part of the form presents the list of parameters using seven columns from which five can be modified. The second and third columns present the name of the parameter given by the user and the original value. The first column is used to select the parameter for the optimization process, the fourth and

fifth columns present the lower and upper boundaries respectively. The sixth column represents the number of levels that the parameter will have. This parameter tells the logic of the form how many points are going to be extracted for that parameter in the DoE phase. Finally, the seventh column tells the logic the accuracy from the parameter. This column is fundamental to differentiate the type of parameter that is being used, since this is in practice what really differentiates them. The accuracy will be used by the general rounding method which is identical to the one described earlier for the optimization runner.

The next part of the section asks the user if the DoE will be executed by generating the permutations of all the sampled points from each dimension or if you will run each dimension independently. In both cases the simulation will be run in parallel.

The next section is the optimization section. The first block allows the user to choose between the Custom MADS or Scipy's differential evolution. Then the user can choose the objective from the optimization and whether it is going to be maximized, minimized or a target value wants to be reached. If this last option is chosen, an additional input box will appear to introduce the target. The objectives that can be chosen at this moment include volumetric efficiency, brake torque, efficiencies, power and brake specific fuel consumption. As shown, they can be either maximized or minimized as desired. The next part includes the constraints. Using the + and – buttons constraints can be added or eliminated as the user wishes. The constraints can be of the three types commented in section 2.1 and as commented in the same section the values in the *regimes.csv* file and in the *cylEmissWetFlow.csv* files are included for the user to constrain. The next three slots present the hyperparameters the user can choose for the optimizers, maximum number of iterations, the tolerance and the stagnations counter maximum. All of them have been explained in previous sections.

The final part of the form allows the user to choose the output dictionary which by default is the same as the one where the document with the engine configuration is stored. When the user finalizes and runs OK the DoE is launched and a message showing the advance of the process starts appearing. At first, it will only say it is running the DoE, when it finalizes and optimization begins this message will present the information about the iteration of the optimization you are currently in.

At the end of the optimization process the GUI will present a pop up presenting the result as shown in figure 6. This figure presents a multi regime optimization process where four operating points were run in parallel. Two values will appear as the maximum or minimums achieved. These two values were explained earlier in section 2. The first value presents the value respecting the tolerance provided by the user. The second one presents a better minimum that was achieved before ending the simulation but was not better under the tolerance provided. This is done to hint the user of a possible better minimum. However, in the engineering context this other minimum might be useless since an improvement in the third decimal, for example, might not be detectable. Remember, that lower tolerances make the optimization process slower, so a higher tolerance can be used if fast results are needed. This can be used for getting a first rough estimate for example.

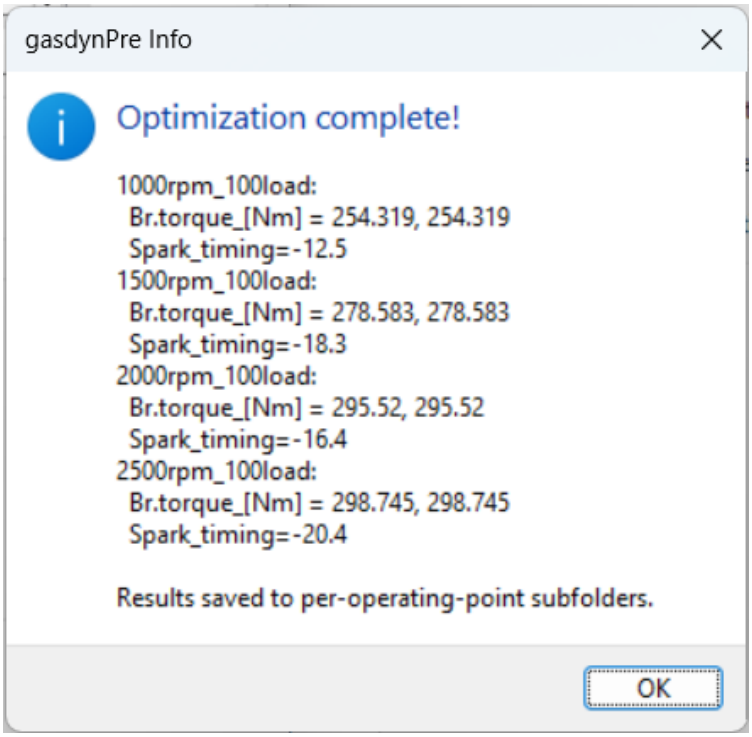

Figure 6. Pop up window presenting the result of a multi regime spark timing optimization process

4.2 THE LOGIC

As commented earlier this second section will comment on the logic existing in the DoE form file that enables the contact between the lower-level hierarchies with the graphical user interface. The software knows what points to simulate by checking what operating points are selected in the general window from the engine model. As was described earlier, an optimization runner, optimizer couple can only manage a single operating point. However, the user can select in the general window as many points as are needed. Therefore, the DoE form must be capable of generating multiple couples of this type and running them. The software can do it and runs all these threads in parallel through threading. Using the same nomenclature as earlier, the main thread here will be the DoE form while many threads executing the logic described in the previous sections operate as auxiliary threads. The DoE form will create, launch and manage all these threads.

To launch a single thread the DoE form verifies if all input information is consistent, collects the selected parameters and extracts the information it needs to determine the DoE points. In information is not consistent an error message will pop up, and the user will be sent back to the input section. The DoE points come from a uniform sampling of the design space. If n levels are chosen, the logic will divide the complete range in $n-1$ equal intervals and will take the limits from these intervals. These limits will then be rounded using the accuracy to guarantee they are feasible points and will be included in a list.

Then the DoE form will build the objective function. To do this it uses two predefined base classes, one for the objective and one for the constraints. The first one takes as input the key from the objective inside the *regimes.csv* file, whether it needs to be maximized, minimized or a target value is expected to be reached and the value for this objective if it exists. The second one uses as inputs

the key from the variable inside the constraint, the type of constraint and the threshold or objective value depending on the constraint type. These base classes include the possibility of evaluating these values and they were provided by the research group at the beginning of the project. Both can be evaluated and added to get the value from the objective function.

The next step is to simulate the points inside the list of DoE points. These points are compiled and executed using the same logic described in section 3. This is done for each of the operating points. The part described until here belongs to the first step of the optimization process and ends when all the simulations are done. In this step the optimization runner optimizer couples are generated too before launching. The next step is to initialize all the optimizers with the data from the DoE. These initializations include the generation of the cache and the assignment of the relevant variables for the optimizer such as the current best values for the objective function, the objective itself and the constraints along with the point generating those values. At this point the optimization is executed following the description previously given.

When the optimization is done, the optimization runner will provide the best point, the value for the objective at this point, the best value without considering the tolerance and the time of the execution. These values are displayed as presented in figure 6. When this is done postprocessing of the results is done. The postprocessing is done through an independent class and generates different figures depending on the type of optimization process that was carried out. The class was specially generated for this project and follows the modular logic of the source code as presented until now.

For single variable optimization processes two plots are generated. One focuses completely on the objective while the other one extends this information by presenting values for the volumetric efficiency, the brake torque and the brake specific fuel consumption. The first graph presents the DoE points and the points sampled for optimization along with a linear interpolation between these points to show a clean representation. The second one presents the same data but focuses on the three listed variables. For two variable optimization processes a contour plot is generated presenting the extrapolation of the data collected during the optimization process along with the tested points and the optimal point achieved. For multi regime single variable optimization a contour plot is also generated that presents the extrapolated contour plot using all of the data generated at the different regimes and the optimal point for each regime. The post processing class is also capable of generating a csv summarizing the optimal data for each regime in a multi regime optimization. These capabilities were developed following the example of the previous MATLAB OPTIMICE tool and the python optimizer framework from previous thesis done in the ICE group.

All data is saved into the corresponding working directories, and the optimization process finalizes. At the end, configuration returns to its original setup and the optimizer windows are closed. To wrap up this section, a final summary of the multi hierarchical nature of the code is given. At the top, the program has the main Gasdyn GUI which works as a main trunk connecting the different logics developed throughout the past years with the user. This thesis focuses on the Design of Experiments branch that was recently developed. This branch is the second hierarchy and focuses on the interaction between the user and the Gasdyn solver. It is also responsible for initializing the following hierarchy by executing the DoE and providing the data for the initialization of the optimization procedure. The third hierarchy is the optimization runner which focuses on the interaction between the Gasdyn solver and the mathematics and algorithms inside the optimizers. At the bottom the

optimizers are responsible for using data generated through simulation to generate new test points and check for convergence.

4.3 ORIGINAL CONTRIBUTIONS FROM THIS WORK

As commented throughout these past four sections, the main structure for the DoE branch was provided by the ICE group and was improved following a close study of the previous optimization tools developed by the group. The main introductions were focused on technical aspects that extended the applicability of the tool as it was and will be listed in this section. The initial version provided included only the SciPy's differential evolution algorithm along with all the logic described in section 3.2 and all the basic classes that were mentioned throughout the past sections. Two groups of contributions were included, aspects modifying the optimization loops and technical implementations that enhanced the result acquisition and visualization. They are listed as follows:

- Introduction and debugging of the cache: The introduction of the cache logic improved the execution times of the differential evolution solver which only received information through the direct execution of the test points.
- Introduction of parameter accuracy: The accuracy of parameters was included to enhance the physical meaning of results and enable a physical understanding of the problem; this was done by using a general round method that could use any accuracy provided by the user through a text box inside the GUI.
- Introduction of the stagnation counter: It was added to give more control over the termination criteria to the user.
- The custom MADS algorithm: The MADS algorithm was specifically customized for the applications that were observed throughout the thesis. It gave the user an alternative different to the evolutive algorithm that perfectly fitted many of the problems in internal combustion engine optimization which do not present objective function with hard topologies while keeping an alternative that could face problems presenting multiple minimums or plateaus such as the differential evolution algorithm. his enables faster and exact optimization in many application cases.
- Generalization of both optimizers to N dimensions: Debugging was needed to guarantee all optimizers were working correctly with any amount of input variables to ensure a smooth performance of the tool with the user.
- Use of the exterior penalty method to include passive constraints: Exterior penalty methods were introduced to guarantee any commercial algorithm, even those not including constraints from the beginning, could be used and further enrich the tool.
- Inclusion of environmental constraints: They were included to consider the challenges that the industry is currently facing and the importance of these constraints in the modern context of the power industry.
- Post processing capabilities: A module was generated to introduce postprocessing capabilities in the new tool.

The final contribution from this thesis is the detailed description of the working principles from the code here provided. This description is supposed to work as a guideline for future researchers involved with this code that can further enrich the tool while keeping the spirit of the code: modular

and general. The next sections will cover the validation through simulations that was done, showcasing the current capabilities of the tool.

5. VALIDATION OF THE FRAMEWORK

This section will cover the description of the different simulations that were carried out to guarantee the optimization tool was working correctly. The experiments were done over two engines. A one-cylinder engine and a V12 Lamborghini engine both provided by the ICE group from PoliMi.

The first model was used through the development phase of the code due to it being fast when simulating. One experiment done in this engine is showcased here. This experiment will cover an intake valve optimization including a constraint on the concentration of CO_2 in the flue gases.

The V12 Lamborghini engine model had been optimized previously, and the model was experimentally verified in past years. The idea was that the optimizer should have predicted similar values to what was in the configuration. 5 experiments were done on this engine. The first experiment concerns the use of two valves to extend the duct length depending on the rotational speed of the engine. These valves are either closed or open meaning that the variables describing their overture are binary. The next experiment will discuss the variable valve timing (VVT) strategy for the engine by introducing a two input multi regime problem and a classic problem in the internal combustion engine optimization world. Third experiment will focus on the spark timing strategy introducing a one input multi regime problem which is as classic as the VVT one. Finally, the fourth and fifth experiments will focus on the optimization of the intake runner's length and then on a complete duct length optimization by simultaneously optimizing the overture of the extension valves and the intake runner's length.

These experiments will showcase the capabilities of the tool by presenting its functionalities, demonstrating its capability to solve classic optimization problems in the context of internal combustion engine optimization and finally comparing the two algorithms to validate them together and present the importance of having this redundance. The experiments will first present a brief description of what was the intend of the specific experiment along with the theory concerning the experiment. Then the inputs for the optimizer will be presented and finally results and a discussion of them will be carried out. Due to the nature of the thesis, no detailed description of the underlying problems will be provided but key references and brief explanations will be present as commented earlier. All operating points are run with maximum load and the operating points are selected as 6000 rpm for the one-cylinder engine and between 1000 and 8000 rpm with steps of 500 rpm for the V12 Lamborghini engine.

5.1 TESTING THE BASICS

This first experiment covers a sample case on a one-cylinder engine whose model is presented in figure 7. This optimization will focus on the intake valve timing and will introduce a constraint on the concentration of CO_2 in the flue gases. Restraining the contaminants generated by the operation of internal combustion engines is a fundamental part of future developments in the field. Controlling combustion, improving after treatment systems or even introducing thermodynamic cycles different to the classic Otto cycle are some of the solutions that have been introduced throughout recent years. Right now, Gasdyn can provide the user with values about the concentration of these pollutants in the flue gases. This can be done both on dry and wet basis. Also, the flow of the contaminants can be retrieved from the output files. In this way engineers can collect data to limit the emissions using the developed framework. However, the regulations on emissions normally

constrain the g/km value for the emissions using test driving cycles in real and lab testing [14]. A future line of work is to generalize environmental constraining by including ways to include the constraints provided by regulatory agents.

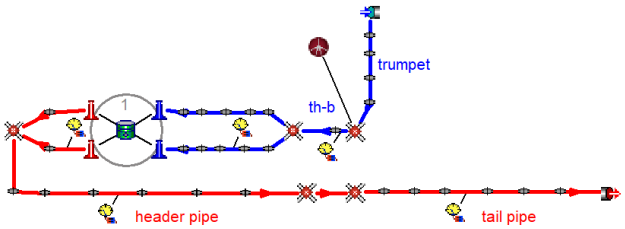

Figure 7. 1-cylinder engine model.

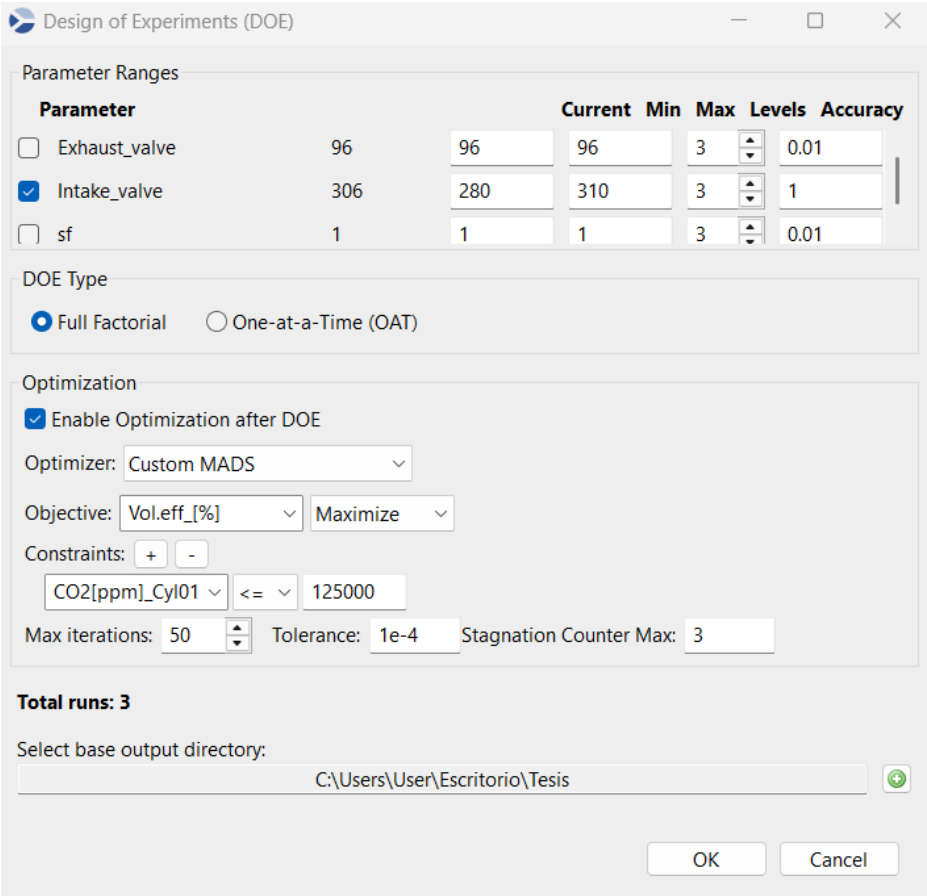

Figure 8. Inputs for the optimization problem

The idea of this experiment was to show the working principle of the optimizer and present a first example. Additionally, it works as a showcase of the use of the exterior penalty method to guarantee that the optimizer does not try to cross introduced boundaries. The intake valve timing uses an accuracy of one degree, and the upper and lower limits are set to be 310° and 280° respectively. The selected speed was 6000 rpm. A three level DoE was done. The optimized objective variable was the

volumetric efficiency, the optimization algorithm was the custom MADS, the stagnation counter maximum was set to 3 and the maximum number of iterations to 50. The configuration is presented in figure 8.

The first part of the experiment generated the results with a constraint that did not affect the selection of the optimum. This will work as benchmark to present the effect of the constraints and will present the type of graphs that are obtained for a single variable constrained problem. Figure 9 presents the graph showing the constraint that does not affect the selection of the optimum.

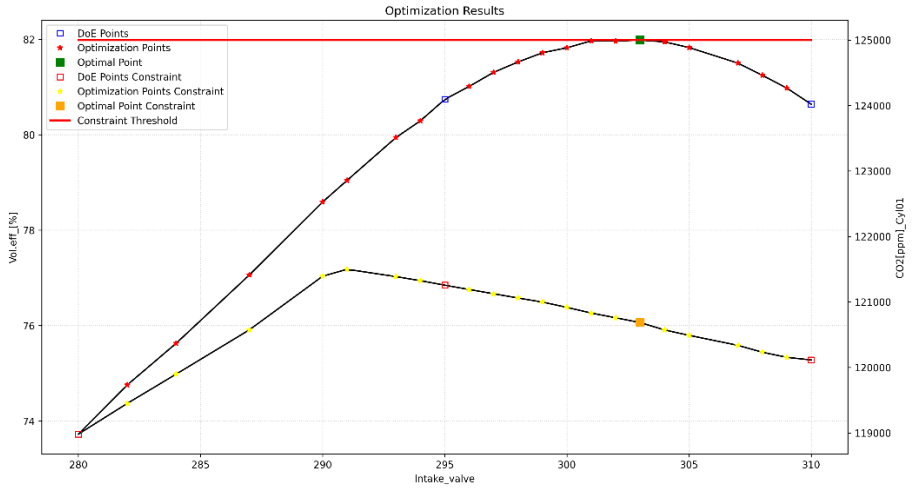

Figure 9. The unconstrained solution to the problem

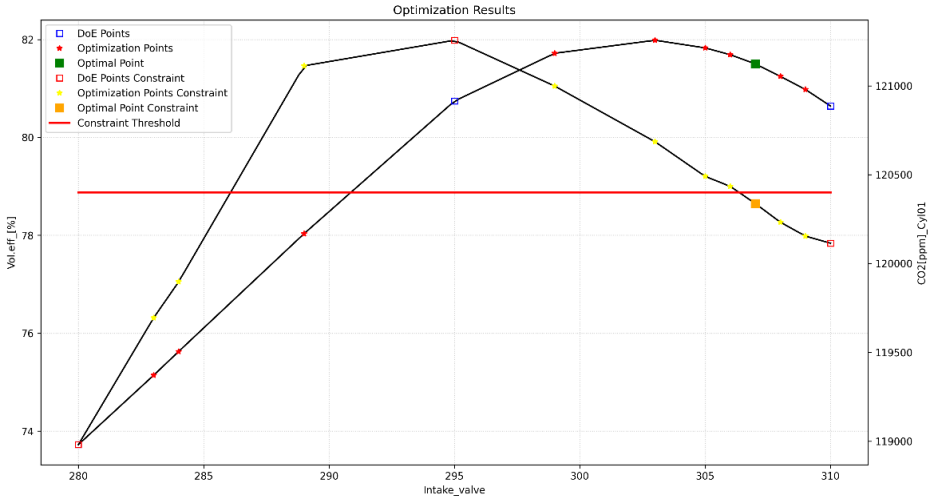

Figure 10. Constrained solution of the problem

Figure 9 presents both the objective and the constrained variable along with the boundary of the constraint denoted by the horizontal red line. It can be observed that the problem is concave down and presents a clear maximum. This is an ideal situation to use the MADS algorithm which hardly has any trouble with these situations. A recommendation is to keep the DoE levels as low as possible, this will guarantee the optimizer can explore correctly producing better results and keeping it from getting trapped in local minimums. Random evaluations are not enough to avoid these problems,

and extensive exploration provides the user with more information about the nature of the problem, providing an additional benefit.

Figure 10 presents the results after toughening the constraint. This toughening excludes the original maximum, and a new one needs to be found. It can be clearly observed that the optimum falls into the right side, as expected, without crashing with the constraint. Notice the constraints do not limit exploration and that this guarantees the user can correctly observe the trend of the different values. Some of the points presented better performance regarding the objective but even values close to the boundary are not considered. This is caused by a correct implementation of the exterior penalty method which guarantees the not compliance of the constraint leads to enormous positive values of the objective function discarding them numerically. Notice the shape of the design space looks slightly different from the one observed in figure 9. This difference highlights the importance of the stagnation counter as an exploration enabler to enhance the knowledge of the design space. However, remember that more exploration means more execution time. figure 11 presents an additional constraining of the problem to further validate the constraining method.

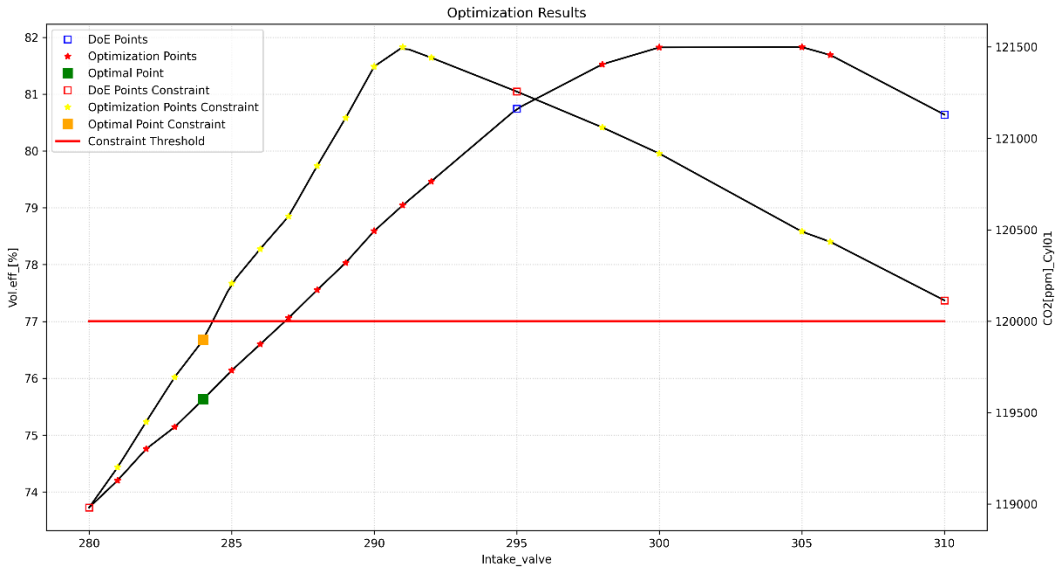

Figure 11. Further constrained solution of the problem

It can be observed that the constraint works as expected. This is the only validation of the constraints that are made throughout the present document. Constraints are a fundamental part of optimization problems so further verification would be highly beneficial.

Now the operation of the MADS optimizer is briefly discussed. The denser areas of exploration are around the optimal values along with points that are far away and simply provide an idea of the behavior of the constraint and the objective in the sections away from the optimal region. Higher maximum stagnation counter values will lead to additional random evaluations that could enhance the mapping of far away regions. The cache, even if existing, is not used for these problems since the sampling space σ was never empty.

This simple simulation worked as a final validation of the performance of the custom MADS algorithm and was part of the tuning of the algorithm. Once again, it is important to highlight the

importance of environmental restrictions for the future of internal combustion engines design and to promote this part of the optimization tool.

5.2 BINARY PROBLEMS AND CACHE VERIFICATION

This second optimization problem focuses on the opening and closing of two valves that modify the length of the ducts in the intake system of a V12 Lamborghini engine which is presented in figure 12. The reason why these systems are implemented in a high-performance engine like the one being optimized is because the geometry of the intake system becomes variable throughout the speed range. This is fundamental to tune the dynamic effects of the engine throughout the complete speed range for it. Correct tuning leads to enhanced filling coefficient, thus to better volumetric efficiency λ_v. This last parameter is directly proportional to the engine's brake mean effective pressure. Therefore, it is directly proportional to the power the engine will produce. The following equation presents this relationship:

$$P = BMEP * V\left(\frac{n}{\varepsilon}\right) = \eta\left(\frac{LHV}{\alpha}\right)\lambda_v \rho_a * V\left(\frac{n}{\varepsilon}\right)$$

In this equation η represents the engine's efficiency, LHV is the lower heating value from the fuel, α is the air fuel ratio, λ_v is the volumetric efficiency, ρ_a is the density of air, V is the total displacement of the engine, n is the rotational speed of the engine and ε is an integer that changes value depending on whether a four stroke or two stroke engine is being analyzed. The most important part here is to show the importance of volumetric efficiency as a fundamental factor for reaching high power levels.

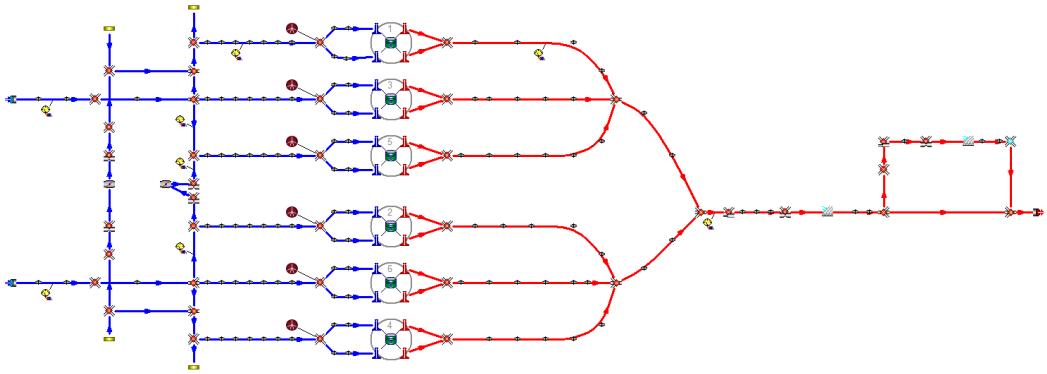

Figure 12. Gasdyn model for a V12 Lamborghini engine

Dynamic effects are the second relevant part in this theoretical discussion. An engine is subject to two main dynamic effects in its intake system, inertia effects and wave effects. Inertia effects are related to the fact that the air mass behaves as a spring-mass system. The resonance frequency of this spring-mass system was modelled by Helmholtz. The idea is that if the natural frequency of the equivalent system is equal to twice the frequency of the engine it will promote the engine's filling even after Bottom Death Center (the end of the intake stroke), where there is no pressure gradient to push the air in, because the system will be oscillating towards the cylinder [15]. The equation describing the optimal rotational speed for a given intake system is:

$$n_{op_{in}} = \frac{a}{4\pi}\sqrt{\frac{A}{LV_m}}$$

In this equation a is the speed of sound, A is the cross-sectional area from the pipe, L is the length of the pipe and V_m is the engine's displacement. The second dynamic effect is called the wave effect. When the filling of the engine begins, the cross-sectional area across the poppet valve is small, in this situation choke can be generated and a shock wave will appear in the intake system. This shock wave will move at the speed of sound and oscillate inside the pipe. If tuned correctly it can lead to the generation of a pressure gradient that can push more air into the cylinder even after suction has disappeared from the cylinder complementing the inertial effect. The optimal speed is given by the equation:

$$n_{opt,w0} = \frac{a}{8L}$$

There is a second group of wave effects which occur when the valve is closed. These pressure systems oscillate inside the duct. If the oscillation is tuned to achieve a positive pressure peak for the next opening it will promote the filling of the engine, if they promote a pressure depression they will lead to constraining the motion of the air towards the cylinder. To check if the effect will be beneficial or detrimental the following equation quantifying the number of half periods during the closing interval is used:

$$k = \frac{3}{4}\left(\frac{a}{nL}\right)$$

In this equation a represents the sound speed, n the rotational speed from the engine and L the duct length. If k is even the consequence of this wave effect will be beneficial. Notice in all the provided equations the duct length plays a primary role. This discussion enabled the understanding of the importance of the duct length and will be relevant when verifying the final optimization steps.

The next part of the discussion focuses on what will be identified on this optimization. In the previous section a large sampling space σ was used. This keeps the MADS algorithm from using the cache, this second problem will constrain it to use it since the design space is small being formed by the combination of the opening closed state of the two valves leading to a total number of four states. The second thing that is desired is to explore how the program can manage binary variables. This is possible due to an intelligent use of the accuracy and the general rounding method, which can take any numerical accuracy given. Gasdyn can take a value of either zero or ninety for the opening of any of these valves, for the optimizer to manage the variable like this, the lower boundary is given as zero and the upper boundary as ninety with an accuracy for the variable of ninety. This last parameter guarantees that only multiples of ninety inside the range are allowed, therefore, only zero and ninety. The optimization was run with two DoE levels, the custom MADS algorithm, the optimization objective was the volumetric efficiency, fifty iterations were allowed, the tolerance was set to 1e-4, and the stagnation counter maximum was set to one. The selection of the stagnation counter value was done to keep the optimization as fast as possible and will guarantee a single iteration is done with values collected from the cache since all the design space was simulated during DoE. The input for the optimizer is shown in figure 13.

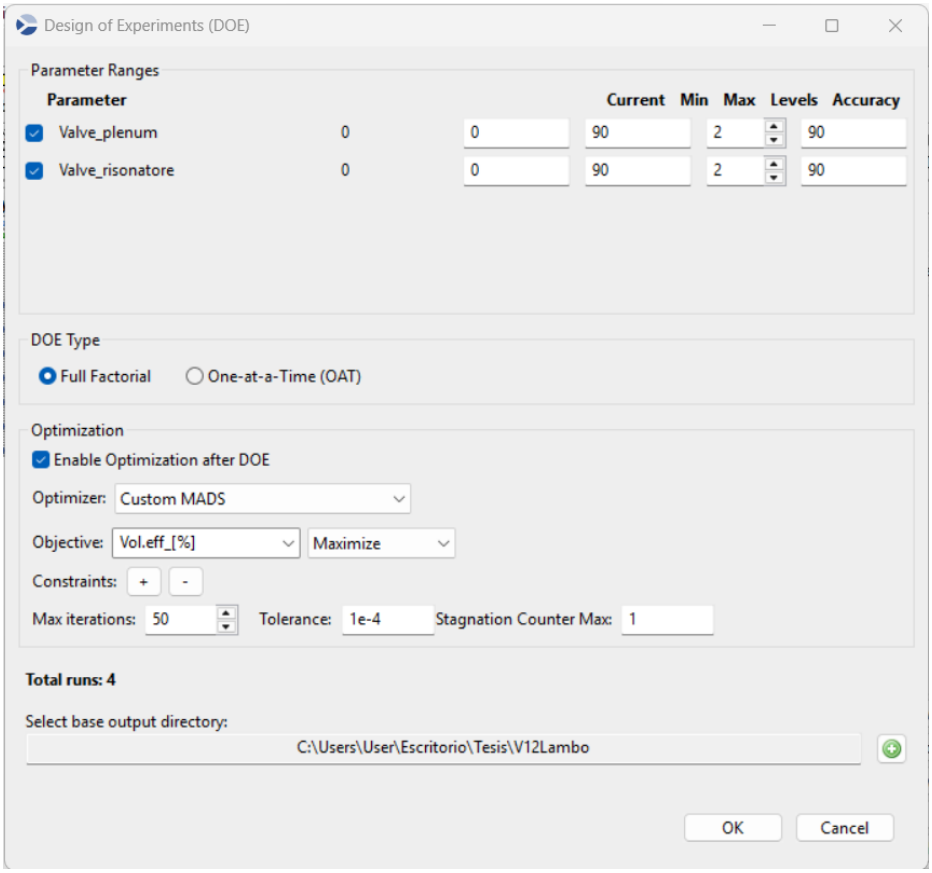

Figure 13. Optimizer input for variable duct length system.

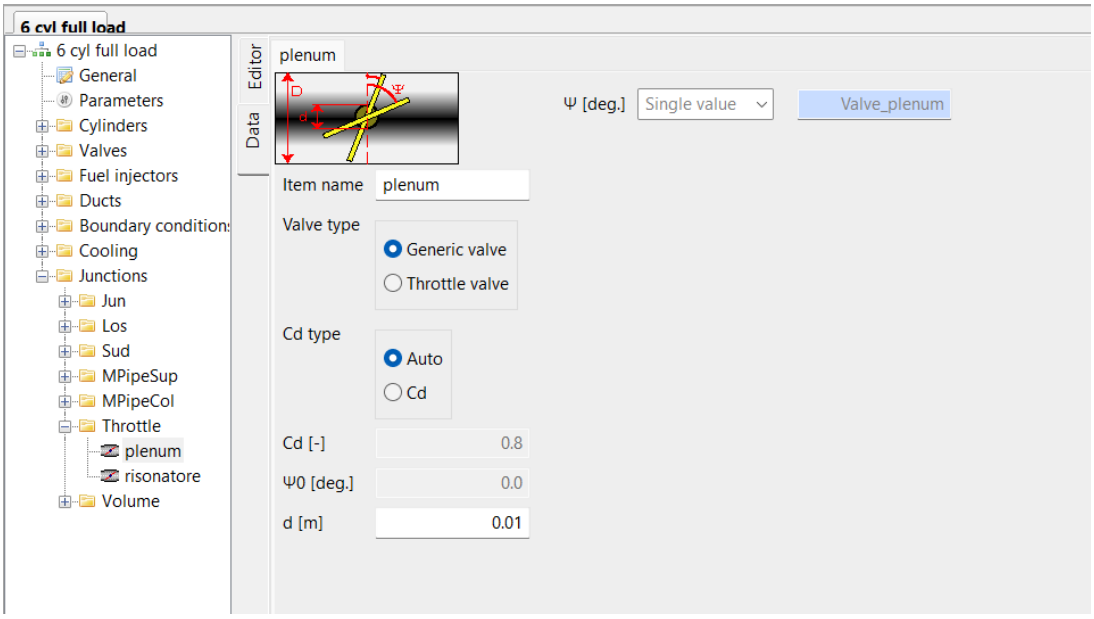

Figure 14. Example of how to generate a multi regime optimization where values for a table will be found

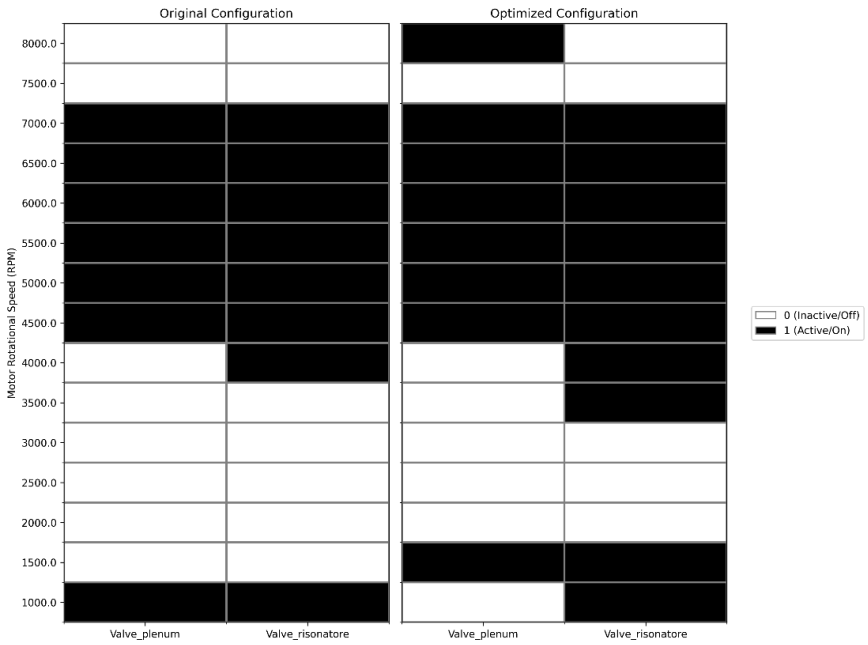

Figure 15. Comparison between the original and the optimal configuration for the valves

Another thing that needs to be commented on before the results to be shown and discussed is the fact that this optimization is multi-regime and will replace the values inside a table that contains the values for opening and closing for all the engine's speeds. This can be carried out by assigning a single value input to the valve value, Ψ, and placing the parameter name in the slot. Then all the needed rpm values are selected in the general window and the optimizer is launched. As shown in figure 6 the optimizer will return the optimized values for each independent regime. Figure 14 presents the way in which the parameter can be assigned based in one of the two valves.

The collected results are two. The first one is presented in figure 15 and compares the original configuration with the optimal one. The second result compares the optimal volumetric efficiencies achieved with both configurations and is presented in figure 16. From the first figure it can be observed that configurations are not modified a lot. This is directly related to the fact that this engine has already been optimized in the past to reach maximum performance. The quantification of these changes is observed in the second graph, and it can be clearly noticed that the original configuration was nearly optimal, excluding only the value for 3500 rpm where a 5% increase on the volumetric efficiency was achieved. Besides this change no additional enhancement was achieved.

Regarding the verifications on the operation of the optimizer, this phase was fundamental for the verification of the cache for the MADS algorithm which as commented before had not been really used due to the existence of large sampling spaces σ. During this optimization, cache was verified to work correctly and an additional step eliminating out of bound points inside P_k was included to avoid crashes when using the cache. Even if it is not expected to be used constantly in future operation, it was important to guarantee its functioning in case modifications of the custom MADS are implemented. Results were satisfactory. One final highlight must be done. The following experiments used the modified version of the engine with the optimized configuration for the valves since the experiments were understood as a sequential process to emulate how a real optimization procedure could be carried out with the optimization tool.

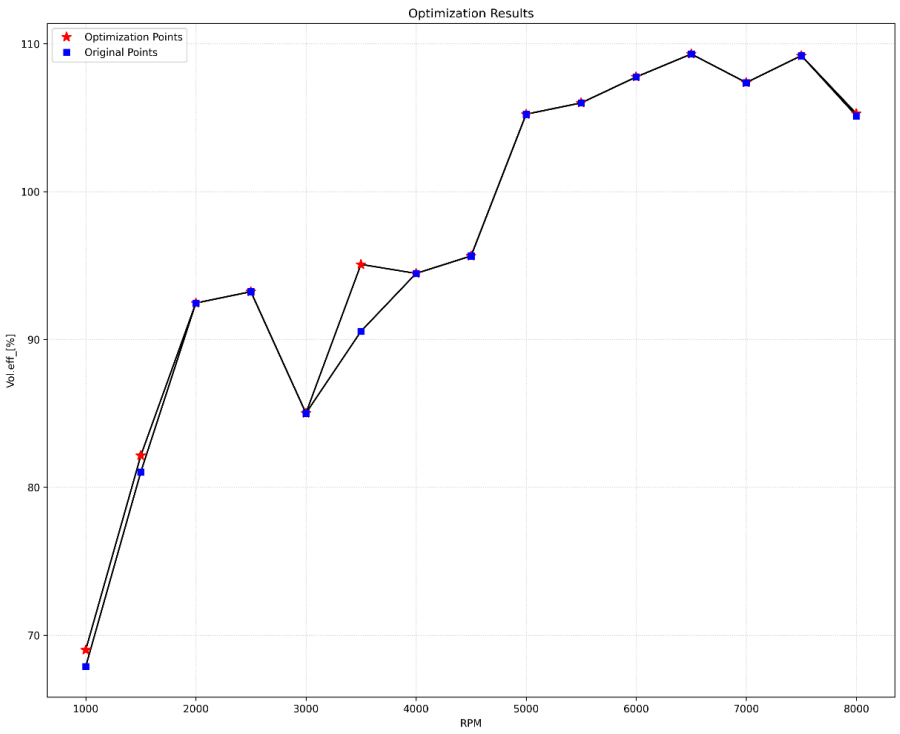

Figure 16. Comparison of the volumetric efficiency for both configurations

5.3 THE VARIABLE VALVE TIMING STRATEGY

The model for the four stroke engine normally implies that the exhaust valve should open at the end of the expansion stroke (Bottom Death Center 1, BDC1) and close at the end of the exhaust stroke (Top Death Center 1, TDC 1) while the intake valve should open at the beginning of the intake stroke (TDC 1) and close at the end of it (Bottom Death Center 2, BDC1). This is highly idealized and real engines do not follow this rigid pattern. Valves open following specific timing that answers special requirements [1]. This first part of the section will discuss these opening logics.

For exhaust valve, the opening is usually done between 40 and 60 degrees before BDC1. This will decrease the expansion work done by the engine since the piston is depressurized but it also reduces the amount of work that the engine must do to expel the exhaust gases. Opening the exhaust valve before the end of the expansion stroke guarantees the existence of a pressure gradient between the piston and the ducts that can be used to generate a “blow down”, a rapid flow of the exhaust gases from the cylinder. The optimal exhaust valve opening will guarantee a correct tradeoff between these two competing behaviors.

The intake valve closing follows a different logic; it is done around 40 to 80 degrees after Top Death Center. Beforehand, it can be thought that no air will enter and that it would backflow since the cylinder is pushing the air towards the intake valve. However, due to the dynamic effects discussed in section 5.2 the existence of pressure oscillations is known. These pressure oscillations, if tuned correctly, will push more air inside the cylinder even with the piston opposing their behavior. Remember, the tuning can only be done effectively for one duct length, or in the best scenario, for some discrete values using valves modifying the geometry. Short lags are better for slow speeds where some degrees represent more than enough time for the gas exchange process to occur

correctly. Long lags are better for high speeds since each degree represents a shorter time. Notice a single lag value cannot optimize all speeds.

The intake valve opening and the exhaust valve closing are designed to achieve an overlap of the two processes. The main use of this strategy is to use the momentum of the exhaust gases to pull in the intake flow and enhance the filling. This optimal behavior is only observed in special operating conditions. When the overlap becomes too long some of the intake gases go into the exhaust manifold and generate a loss. In gasoline engines with port fuel injections or a carburetor this means air fuel mixture is lost and pollutants include unburned hydrocarbons. In throttled gasoline engines, at partial loads it is possible to observe the backflow of exhaust gases which will go through the cycle all over again. This can help with the reduction of NO_x emissions by reducing the temperature of the combustion process. However, if too many flue gases dilute the charge, combustion will be incomplete and other pollutants will appear.

To fully exploit the timing strategy, it is important to modify the valve timing depending on the engine speed and load. In that way the benefits from a correct strategy can be obtained throughout the operation of the engine. Due to costs, reliability and simplicity conventional engines use single values for their timing making the engine vulnerable to the problems listed earlier. To avoid this variable valve timing systems, exist. These systems have been developed throughout the years to achieve a continuous and flexible variation of the valve timings and in some cases of the lift profiles. The third generation of these systems achieve total flexibility. These third-generation systems are divided depending on the working principle that they employ as mechanical, hydraulic or electrical. Their benefits, besides the ones already mentioned generated by having an optimal timing strategy, include load control making it possible to reduce the losses introduced by the throttle in throttled engines.

This discussion highlighted the importance of correct control of the valve timing strategy and the possibility that there exists to do it in real designs through the third generation of variable valve timing systems. This is why it was described as a “classical” optimization process during the introduction of this chapter. The optimization process that was carried out in the project only modifies the timing for the beginning of a predefined valve lift profile. To achieve a total optimization system that reassembles what can be done with a third-generation system the lift profile needs to be modified too. This could also be a future line of work along with the possibilities it could include.

This problem is a multi-regime two variable problems. The strategy for the parametric design is the same as the one depicted in the previous section. A single value, instead of a structured grid, will be introduced by taking the value of the parameter that will be varied. In this way the optimizer will independently optimize the intake and exhaust valve openings for each regime. This is presented in figure 17 for the intake valve timing and in figure 18 for the exhaust valve timing.

The optimization objective was volumetric efficiency, the bounds for the intake valve opening lag were placed as -90 to 0 degrees and the bounds for the exhaust valve opening lag were placed as 5 and 95 degrees. The accuracy for both variables was set to one degree. Three DoE levels were selected for each one of the parameters. The custom MADS algorithm was used to execute optimization using 50 iterations, a tolerance of 1e-4 and a stagnation counter of 3. The tolerance was chosen because there is no need to get fast results, exactitude was valued over time expense for these verifications. Figure 19 presents the input values for this simulation.

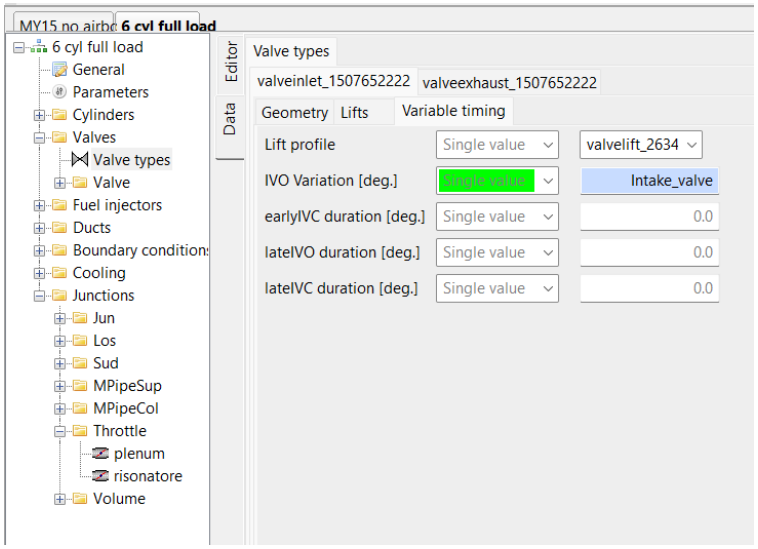

Figure 17. Image depicting the parametric setup for the intake valve timing optimization

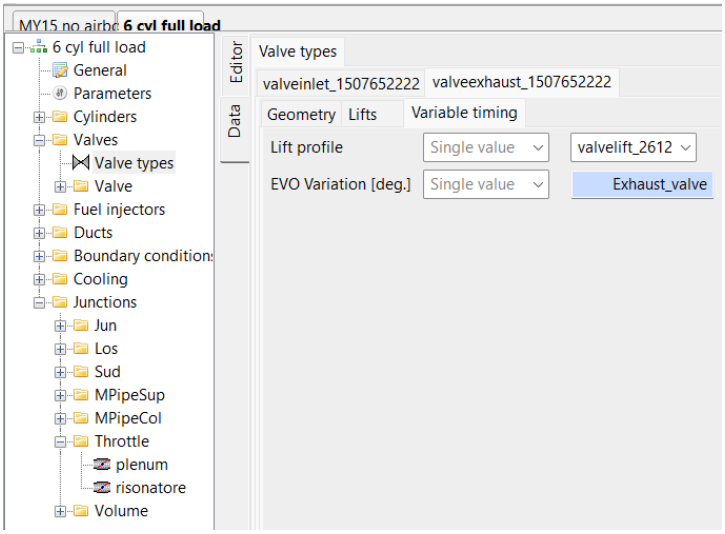

Figure 18. Image depicting the parametric setup for the exhaust valve timing optimization

The results obtained include contour plots for each of the operating points optimized. For example, figure 20 presents the plot for 7500 rpm. Here it can be observed that the function is concave down, which as explained earlier is an ideal situation for the MADS algorithm because it will not have trouble falling int other minimums. The region presents a plateau, but since that is the place where a higher density of evaluations will be done, it is not a problem for the algorithm to find the maximum.

Regarding the number of evaluations. The algorithm did in average 9.47 iterations which represent less than 80 evaluations for each simulation (remember the number of simulations done per iteration is equal to $3^n - 1$ where n is the number of input variables). Considering that the design space had 8100 possibilities (90 possibilities in each axis) it is clearly observed that the evaluations are marginal compared to the idea of simulating the whole design space. The maximum number of iterations done was eighteen and the minimum one was four.

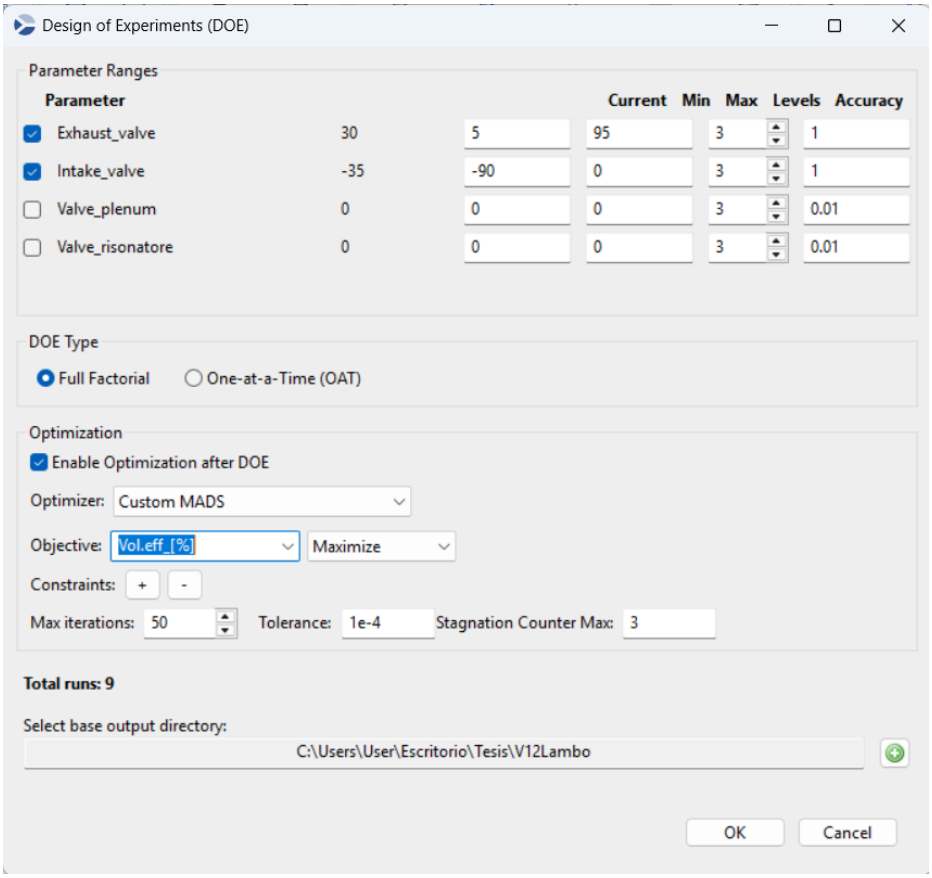

Figure 19. Inputs for the VVT strategy simulation

The maximum value of iterations was achieved for the simulation at 1000 rpm. The initial maximum value reached through the DoE used exhaust valve opening lag equal to 50 degrees and intake valve opening lag equal to -45 degrees. The optimal value was found to have an exhaust valve opening lag of 27 degrees and an intake valve opening lag of -76 degrees. There is a considerable distance between the two points. This generally leads to large exploration. To showcase this difference in exploration figure 21 presents the contour plot for this optimization problem. A larger detail of the plot is found which can be further analyzed to explore other possible sections with the algorithm.

The minimum number of iterations were found at the optimization process for 2500 rpm. Here the initial value was considerably closer to the optimum which led to a lower number of iterations. 4 iterations indicate that it was not directly found during the DoE, but it was one of the points surrounding the optimum found in the initial phase. To further verify the results the comparison between the optimal configuration reached in the previous section and the optimized one is presented in figure 22 while the numerical results can be seen in appendix A It can be observed that the original configuration did not have a strong VVT strategy, so this optimization showcases considerable improvements. The average increase in volumetric efficiency was found to be 5.64% with a maximum improvement of 21.7% reached for 1000 rpm and a minimum improvement of 0.3% achieved for 6000 rpm.

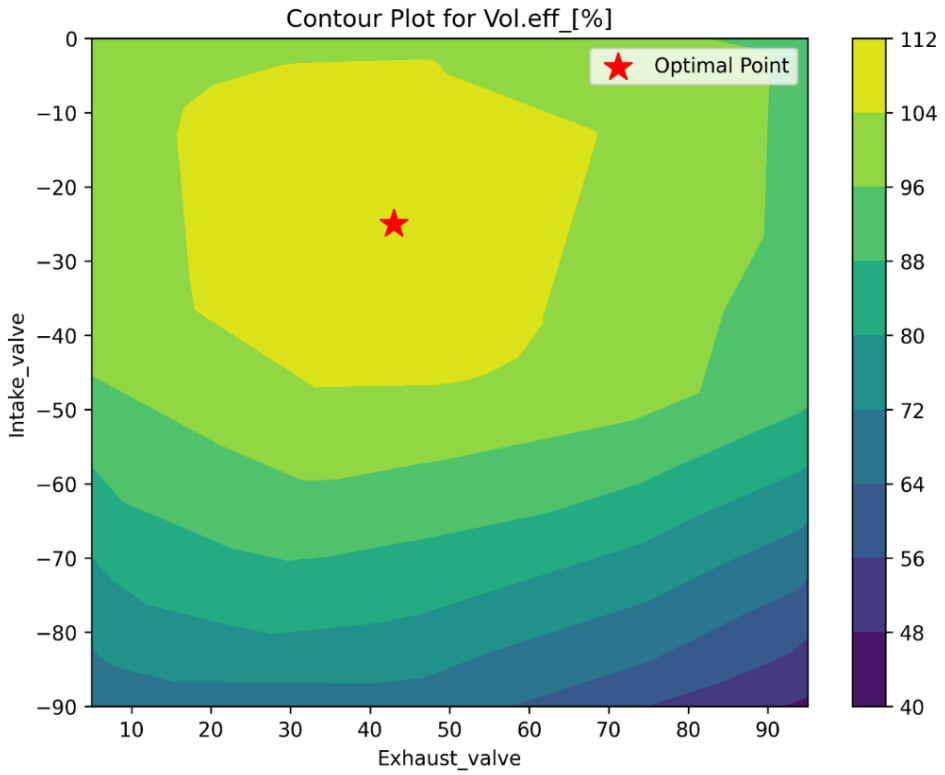

Figure 20. Contour plot for VVT optimization at 7500 rpm

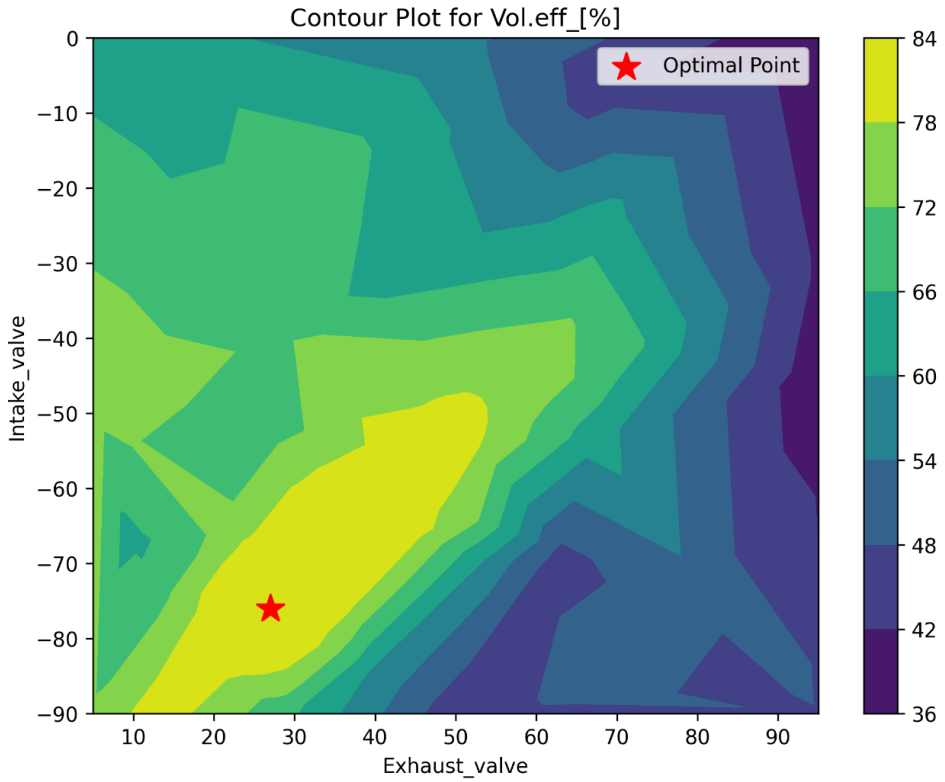

Figure 21. Contour plot for VVT at 1000 rpm

In general, this simulation presented the capacity that the current simulation tool has of facing the classical optimization problem of variable valve timing in the context of internal combustion engines. The proposed strategy must be verified to guarantee its validity by checking the operating limits of the selected variable valve timing system that would be introduced. It must be said that these simulations were generated through batches, using only 4 or 5 regimes each. It was observed by doing these process that the compilation of the simulations is done sequentially which makes the simulations expensive in terms of time. A way to enhance optimization times is to do the compilation for the simulation in parallel, since this was noticed to be the most time-consuming task when the optimization was being carried out.

To summarize, this experiment fully showcased the capability of the tool to find optimal intake and exhaust valve openings at different rotational speeds, it showcased the MADS algorithm as totally functional and capable of carrying out correct optimization procedures and was used to guarantee post processing was done correctly. To enhance the capabilities of the tool valve lifting should be evaluated as an additional optimization problem and compilation of the input variables should be done in parallel to reduce execution times for each of the simulations. Achieving optimization of the lifting profile and the valve timing simultaneously would allow the deployment of the full capabilities provided by third generation variable valve timing systems.

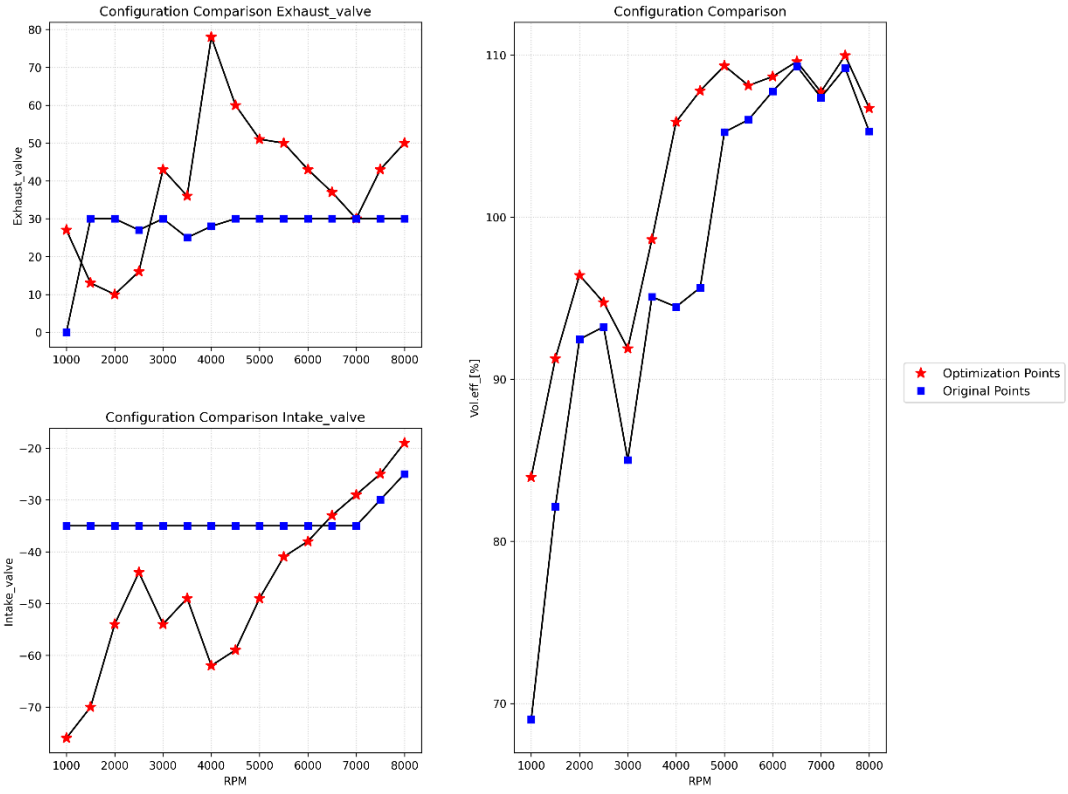

Figure 22. Configuration comparison between the optimal configuration and the one only including the optimal strategy for the valves

5.4 SPARK TIMING STRATEGY

Spark timing is fundamental to guarantee a spark ignition engine can transform the highest amount of thermal energy released by the fuel into mechanical work. In the normal combustion process, the beginning of the process is completely controlled by the spark. It is important to recall that the combustion process of gasoline engines is formed by three phases. At the beginning the spark will ignite the air fuel mixture around it and will induce the generation of a laminar flame front that will slowly turn turbulent. When the flame front is completely turbulent a rapid propagation of it is achieved and a fast increase in pressure and temperature occurs. A short time after the pressure peak the flame front reaches the walls and the combustion process is completed and then extinguishes. The three phases are depicted in figure 23.

As can be observed in figure 23 the combustion process takes time, and this timing must be optimized to distribute the process close to the Top Death Center. The idea, as commented earlier, is to achieve the highest transformation of thermal energy into mechanical energy. To explain how this is done two opposing situations can be imagined depicting the logic, a highly advanced spark plug ignition, and a highly retarded spark plug ignition. In the first case, the pressure peak will be achieved closer to the Top Death Center which will lead to a higher average pressure during the end of the compression phase. This higher pressure translates into more work being done by the piston to reach Top Death Center which reduces the mechanical efficiency of the engine. Still, the pressure peak will be higher since it will benefit more from the compression work introduced by the piston. A higher peak will lead to more force being exerted on the piston head during the expansion stroke. Thus, to higher work extraction during this phase. In the second case, the mechanical loss during the compression stroke that was commented on earlier will be reduced. The maximum pressure value will also drop in this case, thus decreasing the work extraction during the expansion stroke. There is a tradeoff between the two situations, and the spark timing selection needs to specifically answer to this requirement.

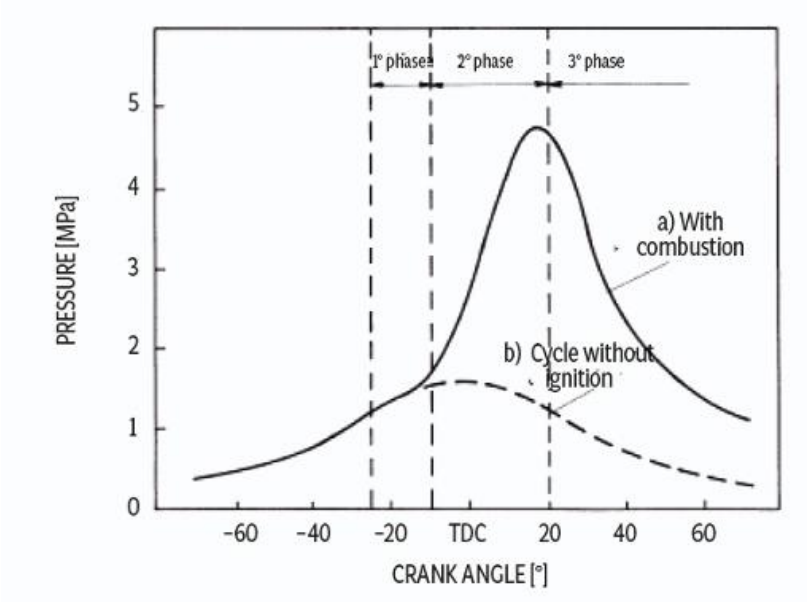

Figure 23. Development of the pressure profile through the engine operation [2]

There is a second challenge related to the selection of spark timing, abnormal combustion modes. There are two groups of abnormal combustion, pre ignition and knock. In both cases parts of the fuel will be burned by something different than the flame front. In the first case a hot spot will deliver the energy necessary for the ignition of part of the charge prior to the spark timing [16]. In the second case the charge will self-ignite, generating a local increase of pressure and a system of pressure oscillations. The first type is not affected by spark timing, but the second one is highly related. There is a relevant value characterizing how fast a certain fuel will self-ignite under specific temperature and pressure conditions, the self-igniting time τ_a. This time is measured experimentally and characterized by the time a certain fuel needs to self-ignite at given thermodynamical conditions. The higher the temperature and the pressure, the shorter the time. That is why retarding the spark, a situation like the second one used to explain the importance of the optimized parameter, helps to reduce the risk of knock. To recall, the retardation of the spark leads to a lower maximum pressure and temperature which enlarges τ_a guaranteeing the flame front can reach all the fuel before it can self-ignite. Knock is detrimental for the structural integrity of the engine and should be avoided as much as possible.

The optimization process that will be here discussed concerned a multi regime single variable optimization focused on the spark timing strategy. The objective variable was the brake torque which was maximized. The parameter was placed using the same methodology as with the other multi regime processes here depicted. Figure 24 presents where in the model the parameter was assigned and depicts the procedure. A single value entry was selected instead of a structured grid for the spark advance value from the cylinder model and the engine speeds needed for simulation were chosen inside the general tab of the model.

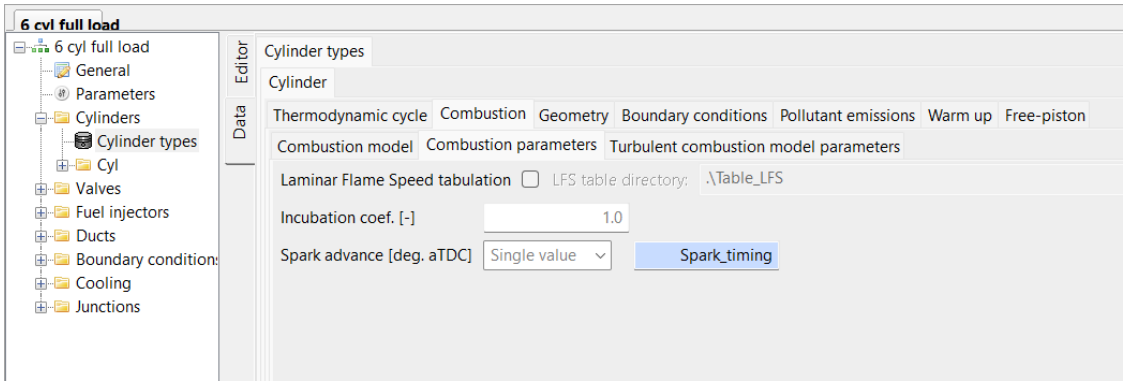

Figure 24. Depiction of where the parameter was placed inside the Gasdyn model

The optimization was carried out with four DoE levels, upper and lower boundaries equal to 0 and --45 degrees after Top Death Center respectively. The MADS algorithm was employed, with a maximum number of iterations equal to 50, a tolerance of 1e-4 and a maximum for the stagnation counter of 5. The tolerance for the parameter was selected to be 0.1 after looking at the original values for the spark timing in the structured table. The inputs are shown in figure 25. When compared with the previous optimization processes, the number of DoE levels and the maximum stagnation counter are larger. This was done after noticing that the problem presented a complex topology with multiple local maxima concentrated in a single region.

The results that will be presented will be divided into three sections. The first section will present the results from an engineering perspective. In this section the results are going to be discussed on light of what was discussed at the beginning of this section. The second section will discuss the performance of the MADS algorithm and will specifically present a real example of how the algorithm explores and advances through the design space looking for the optimum. The third section will compare the MADS algorithm with the SciPy's differential evolution algorithm to present the differences in hyperparameters, number of simulations and the importance of the cache.

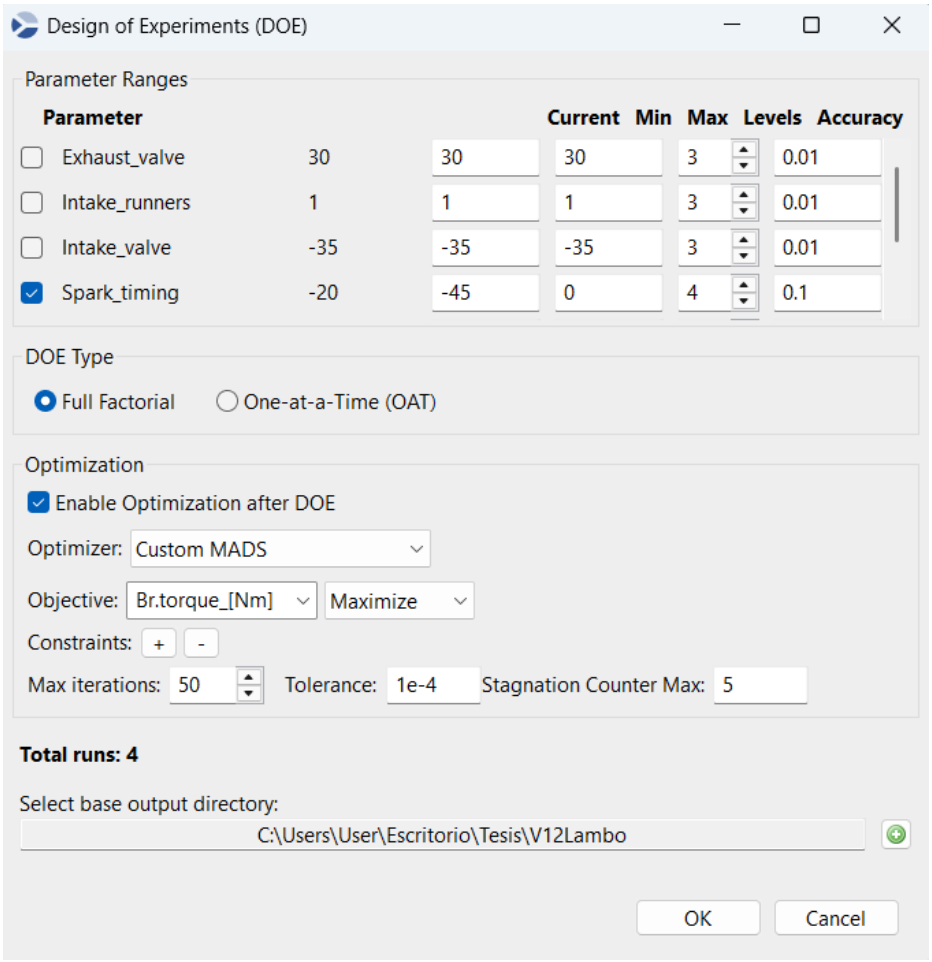

Figure 25. Inputs for the spark timing optimization

5.4.1 THE RESULTS FROM AN ENGINEERING PERSPECTIVE

Figure 26 presents the contour plot that a user will normally get in an optimization of this type. This contour plot shows the optimal result achieved for each of the optimized rotational speeds along with the interpolation of the data collected during the exploration done at each one of the regimes studied. The numerical values for the values presented in the figure can be found in appendix B. In general terms it can be observed that this engine was optimized for operation at around 5000 rpm. This complements the information from figure 22 which shows maximum values for volumetric efficiency are reached around this rotational speed. It can also be observed that the optimizer suggests higher spark advance at high speeds. This is related to the fact that as velocity increases

the time that each degree represents is lower and that there is an enhanced turbulence of the engine at higher speeds. The lower time per degree translated into the need for more degrees to achieve near optimal combustion. This by itself would suggest a constant increase in the spark timing with rotational speed. However, it is clearly observed in figure 26 that the value remains somewhat constant when reaching high speeds. This is related to the enhanced turbulence the engine achieves at this range. Flames with a higher turbulence propagate faster, making the second phase of the combustion process shorter assuring the pressure peak to be in its optimal position without the need for extra time. In other words, the extra turbulence makes the combustion process faster so there is no need for additional time.

Figure 27 presents the comparison between the optimized configuration and the optimal reached in section 5.3. The first thing that can be observed is that for the low velocity range the configurations are close with just some slight variations but always follow the same trend. This somewhat guarantees the correct operation of the optimizer. Then the configuration diverges, suggesting additional spark advance for the highest velocities. On the light of the ideas previously presented, these higher advances would be related to a higher risk of seeing knock. The additional advancement will allow the fuel to reach higher temperatures and pressures reducing the value for τ_a and allowing the fuel to ignite before being reached by the flame front. This presents one major supposition for the current optimization; the risk of knock was not considered. To consider it, additional constraints need to be included. In future work this can be applied to further enhance the capacity of the optimizer to solve this classical problem. Some type of index can be calculated and included inside the constraint tab to guarantee this would not happen. This would also work to verify the constraint behavior of the optimization tool, which as commented during section 5.1, needs further verification.

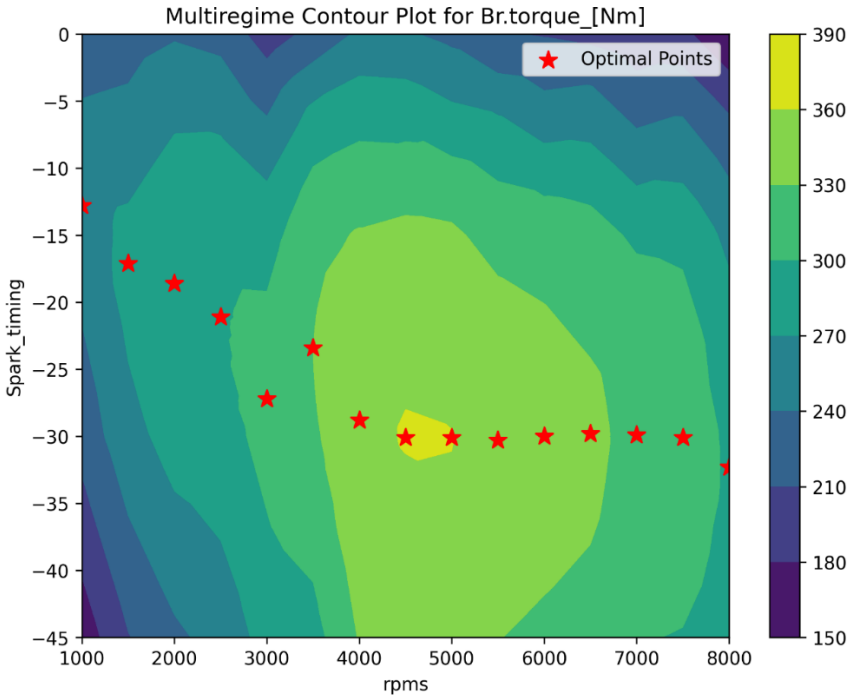

Figure 26. Multi regime contour plot for the optimization of the spark timing

Regarding the improvement achieved, the average increase was 0.5%, which can be seen as marginal compared with the improvement achieved in previous simulation phases. This is further evidence of the fact that the configuration provided was near optimal. The maximum increase was achieved by 8000 rpm and had a value of 0.9%. This increase is somewhat misleading because it is achieved with a valve advancement that can even lead to knock due to how extreme it is compared with its counterpart in the original configuration. The increase for the optimal point identified earlier was the second highest with a 0.8% increase.

The importance of tolerance here is also depicted. The selection of a tolerance of the order of 1e-4 made the algorithm sensitive to these small changes and was capable of suggesting these slight enhancements. From an engineering perspective, this should be compared with the real capability that there exists to measure effectively this difference (2.6 Nm in the best improvement) and how willing the design team of risking knock for this slight difference is. This is commented to highlight the importance of results analysis. This tool suggests concrete values that may be employed, but the users still need to make wise usage of these values.

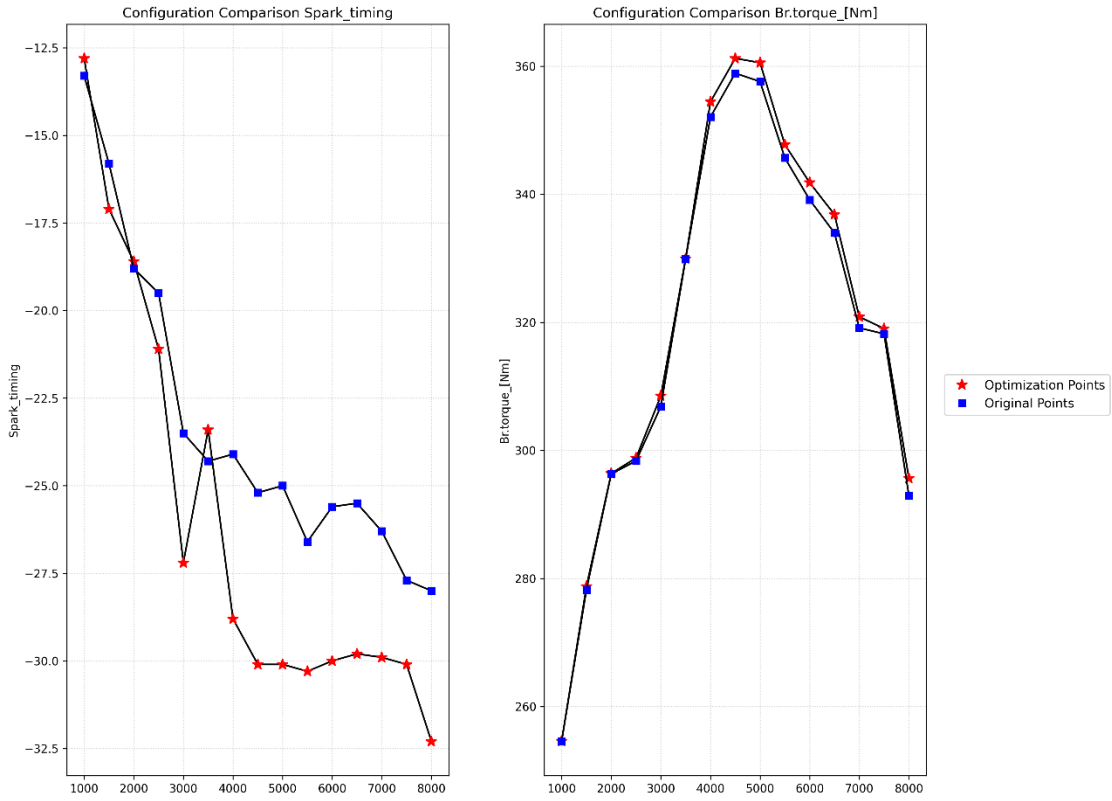

Figure 27. Comparison of the optimized configuration with the optimal configuration reached in the previous section.

5.4.2 OPERATION OF THE MADS ALGORITHM

This section will depict what stated in section 2.2 by showing the data of one of the regimes evaluated during the spark timing optimization. Additionally, the main data related to the evaluations will be commented on. The selected operation point was 2000 rpm because it was the point in which more iterations were executed, a total amount of sixteen. Table 2 presents the data for this optimization process. The third column describes the points that belong to the set P_k, set of

poll step trial points, while the points in the fourth column describe the points in T_k, set of final trial points. The following paragraphs will describe the step-by-step process to analyze the real application of the generalizations commented in the previous sections. This process will also showcase the weaknesses of the custom optimizer and suggest the importance of having alternative solvers for the user to employ in these cases.

Table 2. Optimization data for the optimization of spark timing at 2000 rpm using MADS algorithm

iteration	phase	Spark_timing _poll [deg ATDC]	Spark_timing [deg ATDC]	Br.torque [Nm]
0	doe	-	-45	234.13645
0	doe	-	-30	281.46545
0	doe	-	-15	294.20184
0	doe	-	0	237.3235
1	opt	-15.1	-15.1	294.31382
1	opt	-14.9	-14.9	294.21504
2	opt	-15.3	-15.3	294.48396
2	opt	-15.5	-15.5	294.68273
3	opt	-15.9	-15.9	295.14739
3	opt	-16.3	-16.3	295.35373
4	opt	-17.1	-17.1	295.91653
4	opt	-17.9	-17.9	296.23388
5	opt	-19.5	-19.5	296.16141
5	opt	-21.1	-21.1	295.84387
6	opt	-18.7	-18.7	296.28723
6	opt	-17.1	-40.4	249.98708
7	opt	-18.8	-18.8	296.31858
7	opt	-18.9	-18.9	296.30985
8	opt	-19	-19	296.29077
8	opt	-19.2	-19.2	296.27238
9	opt	-18.7	-37.6	259.21351
9	opt	-18.9	-36.9	261.51672
10	opt	-18.7	-38.1	257.59339
10	opt	-18.9	-29.6	282.62682
11	opt	-18.7	-24.9	291.85668
11	opt	-18.9	-18.6	296.39858
12	opt	-18.7	-41.2	247.23288
12	opt	-18.5	-18.5	296.17741
13	opt	-18.7	-43.6	239.08451
13	opt	-18.5	-0.4	239.24006
14	opt	-18.7	-24.7	292.11579
14	opt	-18.5	-38	257.86694
15	opt	-18.7	-4.7	261.12865
15	opt	-18.5	-19.6	296.2894
16	opt	-18.7	-42.2	243.81919
16	opt	-18.5	-33	273.29372

The first phase, as expected, concerns the execution of the 4 levels of the DoE. It can be clearly observed that these points are uniformly distributed inside the range given to the parameter in the inputs presented in figure 25. The first iteration explores the surroundings of the first point with a mesh size equal to one. This leads to the evaluation of the points -15.1 and -14.9. These two points are found in σ, the sampling space, so there is no need for random extractions and P_k=T_k. Then the first trait about the custom algorithm can be observed. Since the next best point is -15.1, the algorithm will begin a directional search in this direction to explore rapidly the region where the optimal point is hypothesized to be. Since a better point was found, the mesh size will be also increased, and the suggested points will be -15.3 and -15.5. This directional search will continue until the fifth iteration, with a maximum mesh size of 16. At this iteration no better point is found and the directional search is over.

In the sixth iteration, the set of polling directions will be D_1, direction set exploring the immediate surroundings, and the surrounding area of the optimal point is verified after a reduction of the mesh size. One of the points suggested by the poll step, -17.1 had already been evaluated during the fourth iteration. This means it was removed from σ during this step and must be replaced by a random extraction from σ. The point is selected to be -40.4. In this step a new best point is found. This triggers the reduction of the mesh size to its minimum value since a directional search will be done after a verification of the surroundings (condition $D_{k+1} \neq D_k \cup D_k = D_1$ occurs). The directional search goes on during two iterations and then it falls back again to check the surroundings.

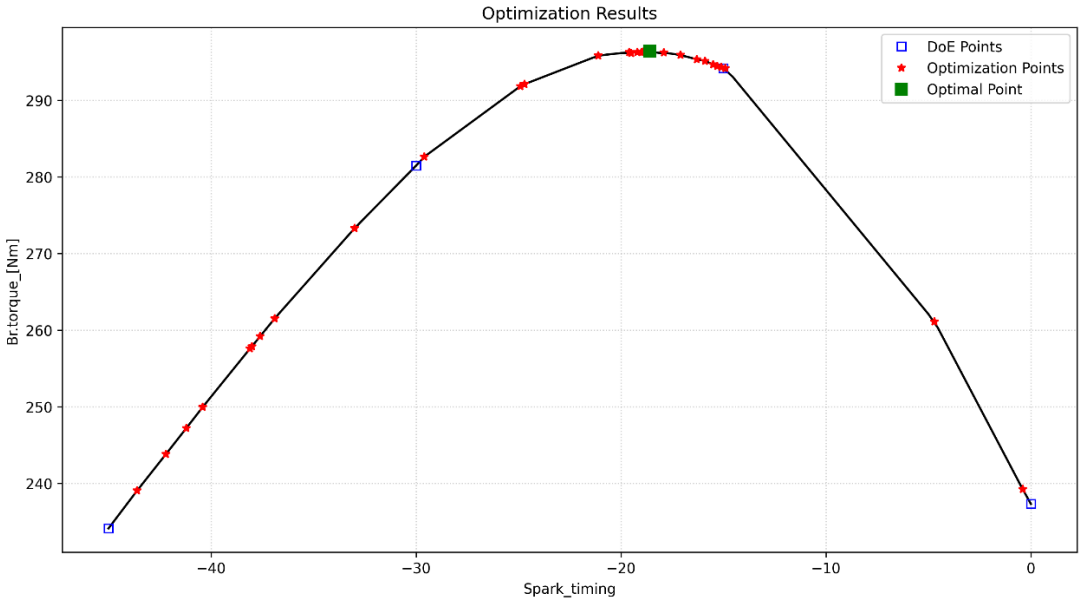

Figure 28. Objective plot for spark timing optimization at 2000 rpm using MADS algorithm

At this point, the immediate surrounding of the optimal point does not appertain to σ anymore. The poll step becomes completely ineffective and random evaluations are done. This is observed by noticing that the poll step continuously suggests the same points during the following iterations and the set T_k does not include any of them. In iteration 11 one of these random points ends up being a new best -18.6 which was close to the previous optimum but not in the immediate surroundings so it was ignored. The condition $x_k \notin P_k$ is triggered and the mesh size is set to 1 along with a search

on the surroundings. The reduction of the mesh size is not really observed because the mesh size was already at its minimum value. After this iteration the immediate surroundings of the point are verified, and no better value is found. This leads to random evaluations until the stagnation counter maximum is reached and the optimization process is terminated. Figure 28 presents graphically the optimization process.

There is one major problem in the fact that the optimal was found through a random evaluation, this obtention was done through chance not through the systematic use of the algorithm. The fact that there were two local maxima in such a concentrated region represents a major challenge for the MADS algorithm which can easily get stuck in local maxima. Random evaluations are a good way to solve the problem, but they are random and they might be ineffective when trying to smoothen the optimization process. Figure 28 presents a region with clear maximum and in general the other optimization problems here discussed have shown the same. Suggesting that complex topologies are rather an exception in optimization of internal combustion engines. However, this emphasizes the importance of counting with alternative algorithms that can manage better this kind of challenges, such as the evolutive algorithm that was implemented. One final comment is regarding the numerical difference between the values. It was just a 0.03% increase for a difference of 0.1 degrees. From an engineering perspective this is practically null. The improvement still needs to be verified during an experimental campaign where a final decision will be made, and these numerical differences might become worthless due to the accuracy of the instruments. Therefore, the performance of the optimizer is still believed to be satisfactory, and emphasis is done on exploration to provide information to engineers to run the needed verification experiments to reach optimal configurations. The trend is believed to be more important than the direct optimal value since it provides more information for inference and further exploration in advanced design phases.

As commented earlier, the maximum number of iterations was 16 and it was carried out during the evaluation of the 2000 rpm operation point. This adds up to a total number of 32 simulations during the optimization phase. With the selected accuracy and bounds for the parameters it can be deduced that the design space included 450 values. Therefore, less than 10% of the design space was explored during the optimization process. This reduced reliance on excessive simulations is a desired characteristic since each simulation is time intensive. The average number of iterations was 9.7, so less than 20 simulations were done during the optimization phase on average. The minimum number of iterations was 6 and was achieved for the operation points of 4500, 5000, 7000 and 7500 rpm. For this point the optimum was found close to the DoE phase, at the first optimization step. Emphasis wants to be made for the 6000-rpm operation point. Table 3 presents the optimization data for this optimization process.

It can be observed that the first iteration of the optimization phase was ineffective. However, the generalization for the mesh size portrayed in section 2.2 considers the fact that finding the optimal in DoE phase leads to low exploration of the surrounding region. The algorithm increases the mesh size for up to 3 consecutive iterations to achieve a better exploration of the region and depicts the tendency around the possible optimum. In table 3 it can be observed that it was during this exploration that a better point was found. With this explanation all the different custom behaviors implemented in the algorithm have been depicted. In future work it is important to try modifications

like the ones here depicted. They may lead to enhancement of the performance of the optimizer. One of the main things that can be worked on for the current version of the optimizer is to enhance the search step. Random extractions work for exploring the design space, but maybe complex topologies can be better faced if local explorations are done during the search step. A higher understanding of the optimal region by the user will lead to better use of the results obtained.

Table 3. Optimization data for the optimization of spark timing at 6000 rpm using MADS algorithm

iteration	phase	Spark_timing_poll [deg ATDC]	Spark_timing [deg ATDC]	Br.torque [Nm]
0	doe	-	-45	324.454
0	doe	-	-30	341.81206
0	doe	-	-15	315.23483
0	doe	-	0	208.09742
1	opt	-30.1	-30.1	341.80557
1	opt	-29.9	-29.9	341.77823
2	opt	-30.2	-30.2	341.79011
2	opt	-29.8	-29.8	341.84788
3	opt	-29.7	-29.7	341.82777
3	opt	-29.6	-29.6	341.77512
4	opt	-29.9	-17.7	324.14723
4	opt	-29.7	-17.5	323.63964
5	opt	-29.9	-37.1	337.2601
5	opt	-29.7	-41.3	331.1165
6	opt	-29.9	-21.1	332.22704
6	opt	-29.7	-34	340.46454
7	opt	-29.9	-26.2	339.66048
7	opt	-29.7	-36.4	338.10876

5.4.3 COMPARISON OF THE OPTIMIZATION PROCESSES

As commented earlier, this problem presented a challenging topology because multiple local maxima were concentrated in a single area and separated by small valleys making it hard for the MADS algorithm to identify them. This section will compare the performance of the two algorithms. First the number of simulations will be compared along with the amount of cache hits that the evolutive algorithm had. It is expected to have far more evaluations with the evolutive algorithm. Then, the numerical results will be verified and commented on to verify how different they are between one another. The SciPy's differential evolution was run with the same input values as the optimization with the MADS algorithm but with a different maximum stagnation counter. A value of 15 was chosen for this hyperparameter. Figure 29 presents the inputs for this second optimization process.

This optimization process executed on average 24.9 simulations versus the 20 simulations that the MADS algorithm did. The higher number of executions was expected. What was not expected was the reduced number of cache hits. The maximum was 3 cache hits while the average was 0.6. This was possibly due to the reduced value of the maximum stagnation counter. For the evolutive algorithm to correctly explore the design space, the "fitter" members of the population need to be

capable of becoming predominant. This can only be achieved with a number of generations large enough, which for this case was not given. In a case where the population would be large enough, it will be observed that members of new generations will group around the optimal region and future generations will concentrate here. When this type of accumulation is observed, cache hits become frequent. It will be observer later that high cache efficiencies are observed when this occurs.

Regarding the numerical results obtained, figure 30 depicts the comparison between the two optimization processes. In general, it can be observed that the optima found are different but in general the torque values are similar. The evolutive algorithm was capable of improving the optimal values for 1000 rpm and 3500 rpm. For 1000 rpm the suggest value varied by 0.8 degrees and achieved an improvement of 0.02%. For 3500 rpm the difference was 0.6 degrees, and the improvement achieved was 0.003%. In both cases it is completely marginal, but it also depicts the fact that the topology of the problem was rather complex with multiple maximums close to each other. For 4000 rpm both optimizers achieved the same result. For all the other values the MADS algorithm achieved a better performance. The maximum improvement was 0.3% and was achieved at 8000 rpm where the differential evolution algorithm failed to explore lower values of spark timing.

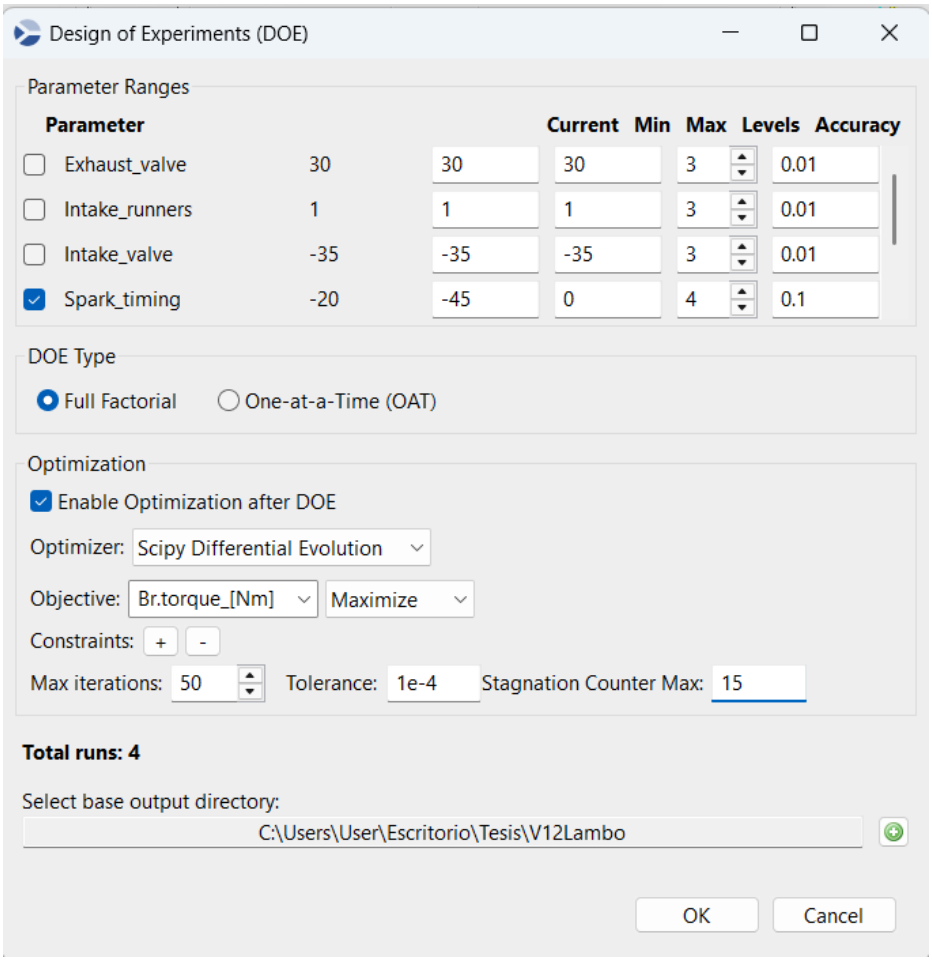

Figure 29. Input values for the optimization with SciPy's differential evolution

It was observed that the evolutive algorithm had a logic that favors exploration of the design space. The problem with this is that for it to be effective, the design space must be constrained to small

regions, or a high number of maximum iterations and stagnation counter maximum values must be given. Normally, the constraining of the limits defining the design space cannot be done, since there is no certainty of the region where the optimum might be. A possible procedure for optimization problems where a difficult topology is expected is to use the MADS algorithm to identify the region where the optimum will be and then apply a second optimization step using differential evolution to identify the maximum inside this rather complex region. The first optimization step can also be carried out with differential evolution, but as observed, the MADS algorithm keeps the simulations at a low value when running regular optimization problems like the ones discussed in this document. Finally, large emphasis must be put on the fact that higher stagnation counter maximums are recommended when running optimization problems that use the evolutive algorithm to allow it to run its complex but exploration driven acquisition function. Table 4 summarizes the data regarding the amount of Gasdyn calls that each algorithm made.

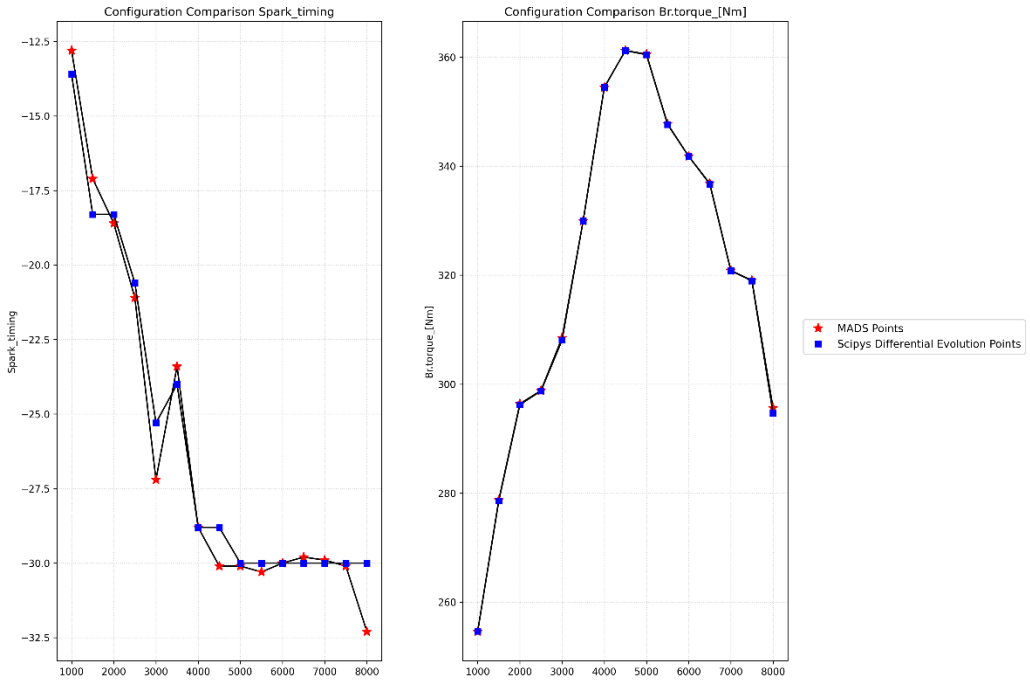

Figure 30. Result comparison for both spark timing optimization executed with both algorithms

Table 4. Summary table depicting the difference on executed simulations

Algorithm	Average Gasdyn Calls
MADS	20
SciPy's Differential Evolution	24.9

5.5 DUCT LENGTH OPTIMIZATION

This optimization procedure along with the following are based on what was described in section 5.2. Thus, no additional explanation regarding the theory behind the optimization process is described. The most important conclusion from that section was that the length of the intake duct is fundamental for the tuning of the dynamic effects. This tuning can be done for a single speed if no variable geometry systems are included. In the current optimization step, the valves used to vary the

length of these ducts have been fixed, therefore the only tunable parameter is the length of the intake runner. In figure 12 the model for the optimized engine was presented. The duct that will be optimized is the one connecting the intake manifold with the two ducts going directly to the valves, this duct is called the intake runners.

It can be observed that the model has six identical ducts of this type, one per cylinder. The parametric configuration for this variable is different from the one that has been done in previous cases. First, the optimization will be done for a single rpm, 5000, which was identified in the previous discussions as the operating point with maximum power for this engine. Inside the configuration for each of the ducts a value called the scaling factor length is included, this is a factor that multiplies the length of the individual parts of the duct. Therefore, to prepare the simulation the parameter must be placed inside the slot representing this scaling factor for each of the intake runners.

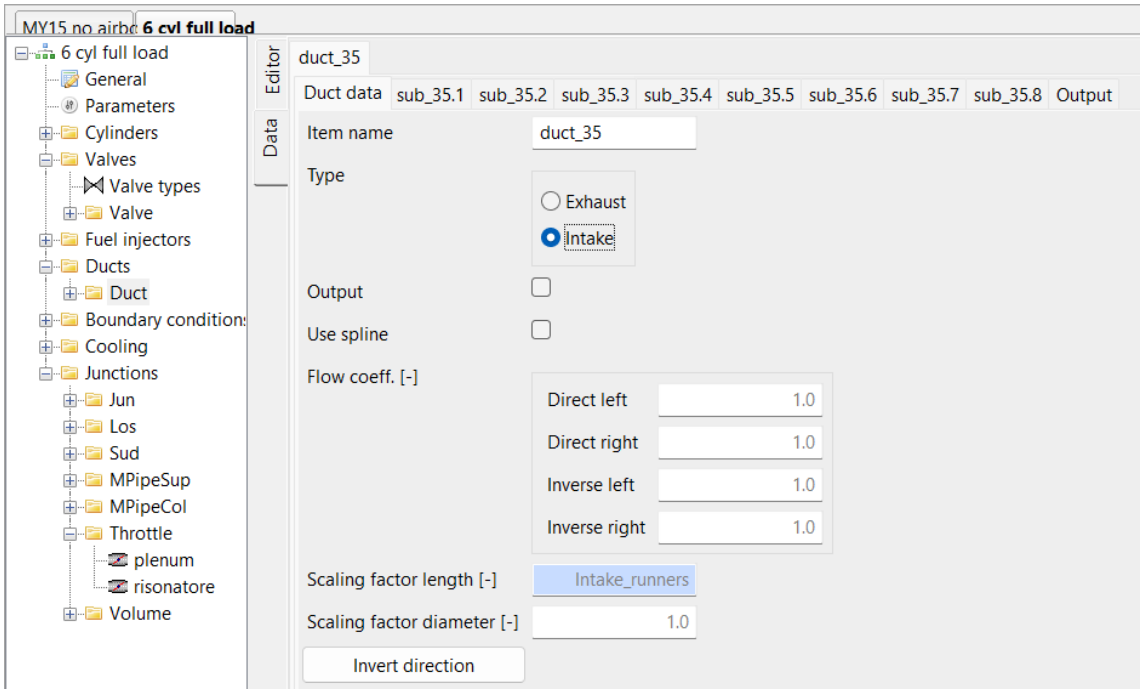

Figure 31. Parametric configuration for the intake runner length optimization

In total this process is repeated six times to optimize simultaneously the six intake runners. The accurate of this non dimensional parameter was set to be 0.01. The objective parameter was selected to be the brake torque and the operating point, as commented earlier, was chosen to be 5000 rpm. The upper and lower bounds were selected to be 2 and 1 respectively and a DoE with three levels was executed. The optimization problem was executed using both algorithms. For the MADS algorithm the maximum number of iterations was set to be 50 and a stagnation counter maximum of 5 was used. Figure 32 presents the used configuration. For SciPy's differential evolution 100 iterations were allowed and the stagnation counter maximum was set to be 120, therefore this termination mechanism was practically deactivated. Figure 33 presents the inputs for this optimization procedure. In both cases, the tolerance was set to be 1e-4.

The selection of the larger maximum stagnation counter was due to expected hard topography since the small accuracy was believed to be related to phenomena like the ones noticed during the spark

timing optimization described in the previous section. The bounds for the parameter were determined after running simpler optimization problems before running the final optimization. In this way the optimizer was already somewhat close to the optimal point. The objective of this optimization was to close the logic that is supposed to depict a normal optimization procedure for an engine. When all the other parameters are selected, the length of the runners is the final parameter engineers can modify to optimize the performance of a single point. One consideration must be made when optimizing the engine for a single operation point detrimental behavior introduced at other operating points. This is related to problems in the tuning of the dynamic effects from these other operating points. This generated the idea of introducing a multi regime optimization tool capable of simultaneously optimizing the same objective across different operating points. For this particular case it would enable the engineers to avoid the introduction of detrimental effects, but it can also be used to optimize single value configurations for any of the optimization processes described earlier. This can work as an additional generalization of the optimization tool to enhance its applicability in the industry.

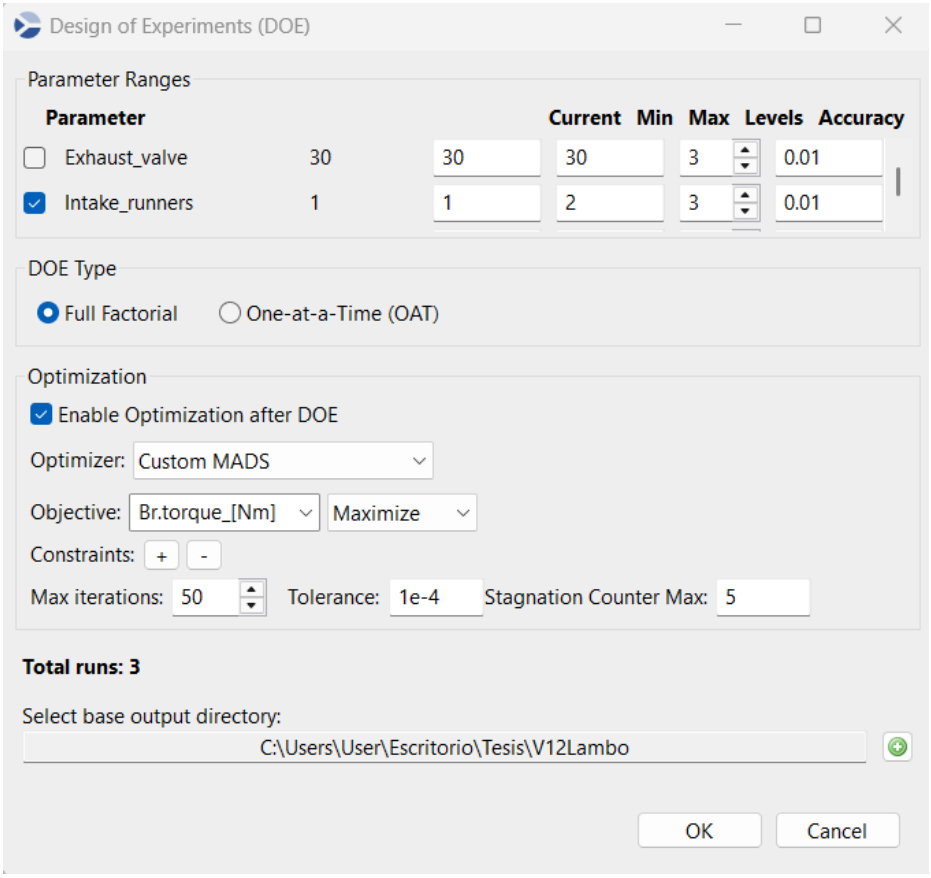

Figure 32. Input values for optimization using MADS algorithm

The results that will be discussed during this section follow the same structure as before. First the result will be analyzed from an engineering perspective and then the way in which the algorithms operated and a comparison of them will be offered. Notice that with the configuration depicted in figure 33 the evolutive algorithm will have a higher chance of fully exploiting its acquisition function to enhance exploration and use the cache appropriately to avoid excessive simulations. Also, it will

work as a direct comparison for both algorithms. The low tolerance makes it hard for the evolutive algorithm to reach convergence based on this criterion since it needs all the values in a generation to be extremely close, almost equal. This means that here convergence will not be observed either, the algorithm will terminate the simulation when the maximum number of iterations is reached.

The graph presenting the one input optimization results for the MADS algorithm is presented in figure 34 while the one presenting the results for SciPy's differential evolution is presented in figure 35. The first difference that arises is the mapping done by both algorithms. As commented earlier the acquisition function for the evolutive algorithm is exploration driven, is this capability what gives it the ability to identify other minimums in complex topologies. This is reflected in a far better mapping of the objective function presented in figure 35. The topology for this specific problem, as shown in the previously mentioned figure, includes three local maximums in the selected interval. These maximums have appreciable distance between each other and present large differences in their values of the objective function. This generates that neither of the two algorithms had any problem identifying the maximum. This is also promoted by the fact that the objective function presents a clear polynomial shape without plateau like zones like the one observed in figure 28 which introduces a challenging condition to the MADS algorithm.

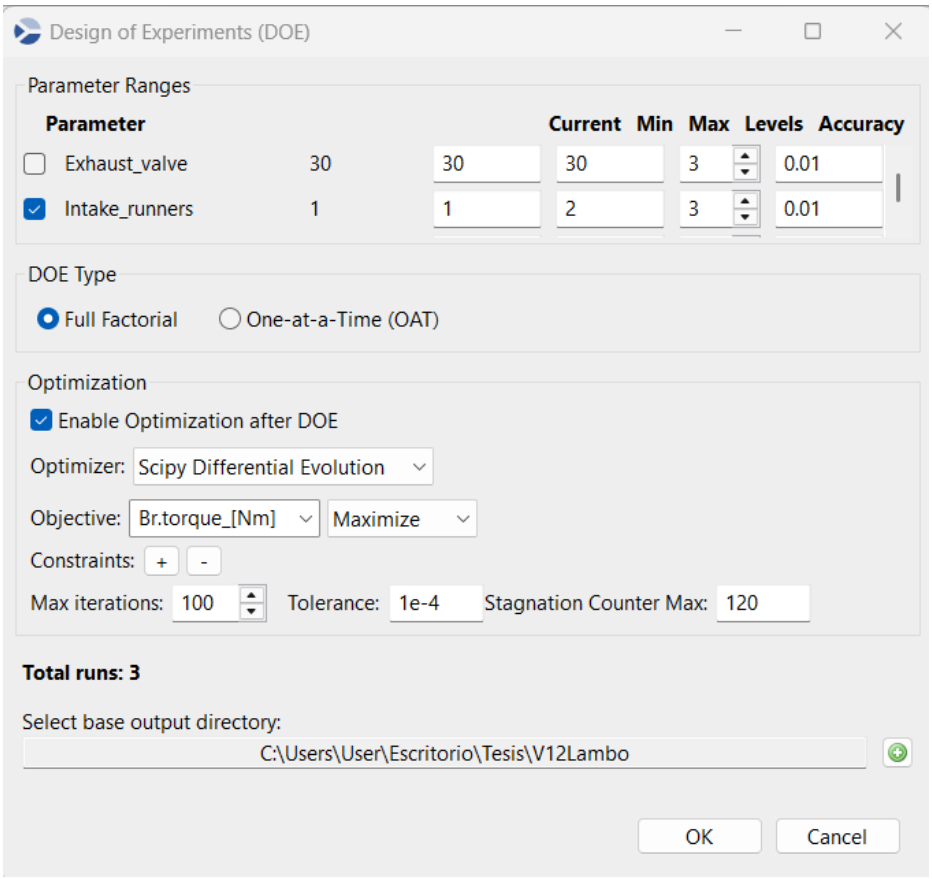

Figure 33. Input values for optimization using SciPy's differential evolution

From the graphs it can also be seen that the evolutive algorithm executed a large number of simulations. To be exact 54 simulations were executed versus 18 that were done with the MADS algorithm. It has been constantly commented that having this amount of simulations is good for the

exploration of the design space. However, the time expense that this introduces makes the MADS algorithm highly attractive when the objective function is smooth and does not present any complex topologies. Regarding the time expense, it is also important to consider that the differential evolution algorithm executes these simulations in series which further increases the time consumption that the process takes. Keep in mind that great exploration of complex objective functions can be achieved by employing this algorithm.

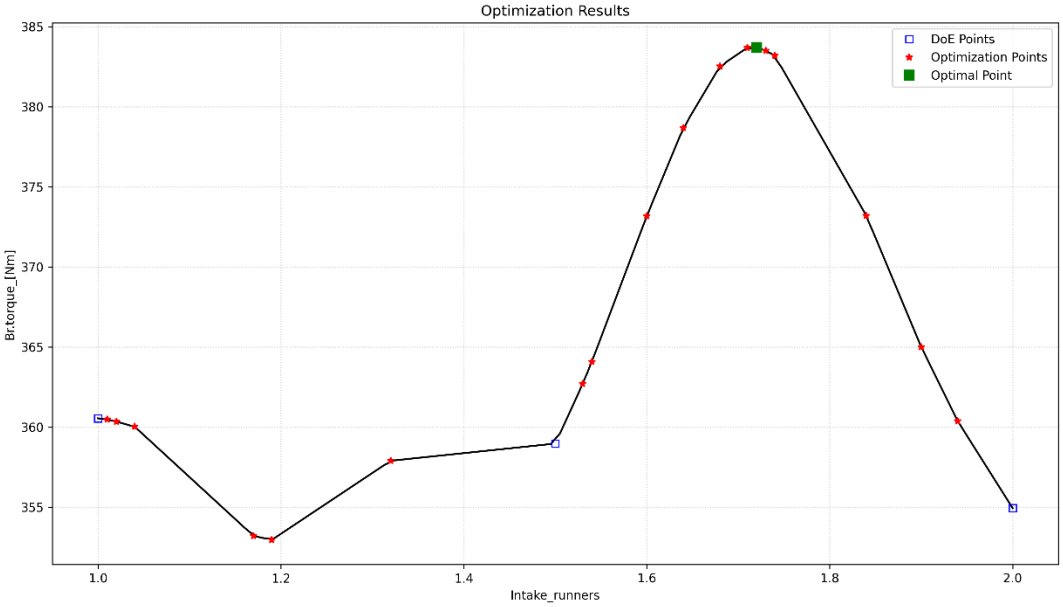

Figure 34. Optimization results using the MADS algorithm

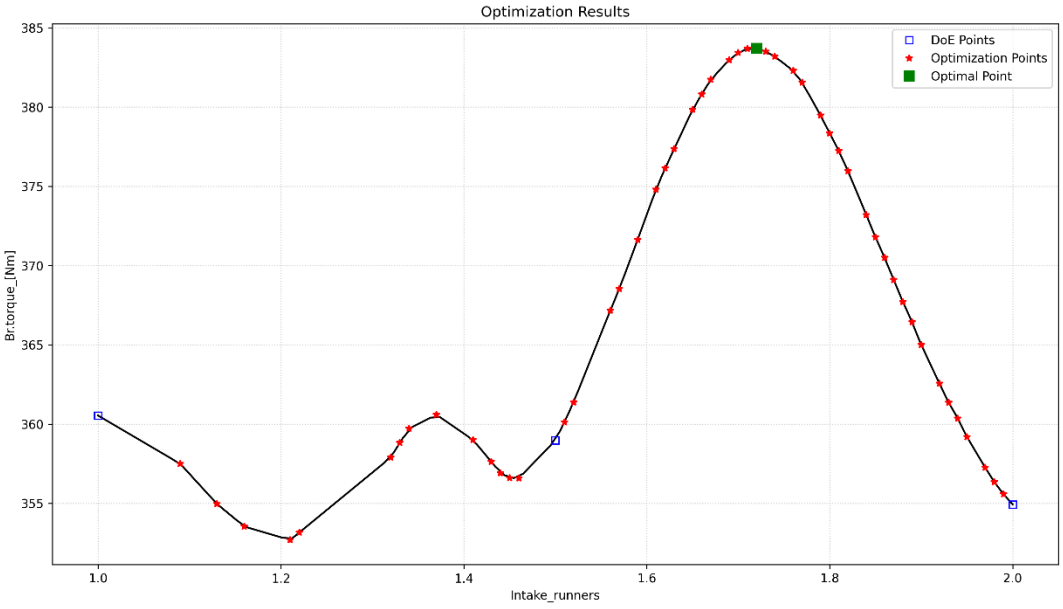

Figure 35. Optimization results using SciPy's differential evolution algorithm

Another fundamental metric is the obtained optimal value. Both algorithms arrived to 1.72 as optimal scale factor for the duct length. The torque achieved was $383.7\ Nm$ at the optimized point.

However, the fact that this optimization might have had detrimental behavior on other operating points was analyzed by comparing the improvement from the original configuration for this optimization case and the one just before. Figure 36 presents the comparison between the original configuration and the one achieved after the spark timing optimization while figure 37 presents the comparison between the fully optimized configuration and the original one.

The graphs present the three most relevant variables for understanding the performance of the engine and they will be discussed individually and then a final discussion comparing the results will be done. For the first comparison it was observed that the volumetric efficiency could be enhanced, especially in the initial region. This has to do with the implementation of the VVT strategy since the original configuration did not count with a strong use of this enhancement mechanism. The average increase achieved was 6% with a maximum increase of 23.6% achieved at 1000 rpm which confirms the improvement mentioned in section 5.3. After including the intake runner length optimization the improvement was only 21.7% which presents evidence of the strong synergy that exist between the different parameters in the engine, the spark timing seems to be responsible for an almost 2% increase at this rpm. If we verify the average increase it passed from 5.64% to the 6% here presented further enforcing the argument here presented. The minimum increase was 0.1% and was achieved at 6500 rpm a change from the minimum change observed earlier of 0.3% and 6000 rpm.

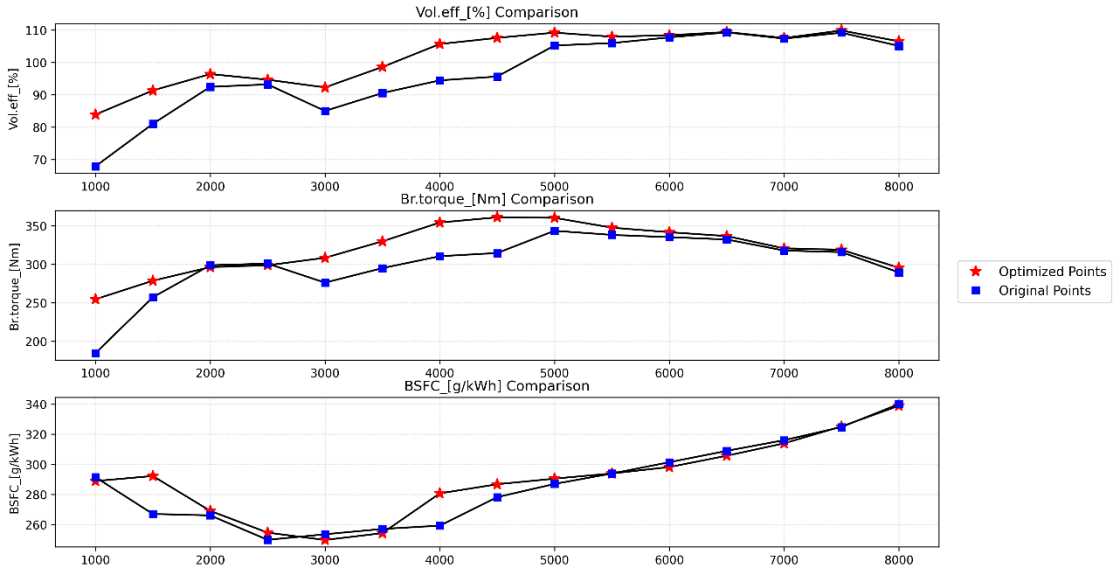

Figure 36. Configuration comparison without intake runner optimization

Regarding the torque, the average increase was 7.5% with a maximum value of 38% achieved at 1000 rpm. As commented in section 5.2 the power of the engine, and thus the torque, has a proportional relationship with the volumetric efficiency. Therefore, it makes sense that the increase of the volumetric efficiency discussed earlier lead to a similarly large increase in the brake torque. The minimum value for the change in performance was -0.7% and was observed at 2500 rpm. Here the previously presented argument is not as clear, since there was a 1.5% increase in the volumetric efficiency but then the torque actually decreases. This is part of the synergy with other parameters that were probably changed when modifying the optimizable parameters and these changes cannot be clearly seen. It is also important to say that the change is quite little. When checking the results

presented figure 27 it is clearly observed that this point along with 2000 rpm was near optimal in the original configuration, that explain why the variation between the two torques is marginal.

The final relevant parameter is the brake specific fuel consumption. For this specific variable the average change was an increase of 1.2%, which is not a beneficial value because it means that the engine becomes less economical. Depending on the application, this might be relevant, considering that the optimization was done for a car where performance is above everything, this result loses relevance. The maximum increase was achieved, a 9.4% increase and was observed at 1500 rpm while at 3000 rpm the maximum decrease was observed with a reduction of 1.5% from the original value. At 3500 rpm the maximum absolute reduction was observed.

The next step is to repeat this comparison with the data presented in figure 37. The first thing that can be observed is that the engine achieved a higher maximum brake torque and maximum volumetric efficiency. It passed from 361.22 Nm and 109.89% to 383.7 Nm and 116.16%. This enhancement is completely achieved by the correct tuning of the dynamic effects by using a correct duct length.

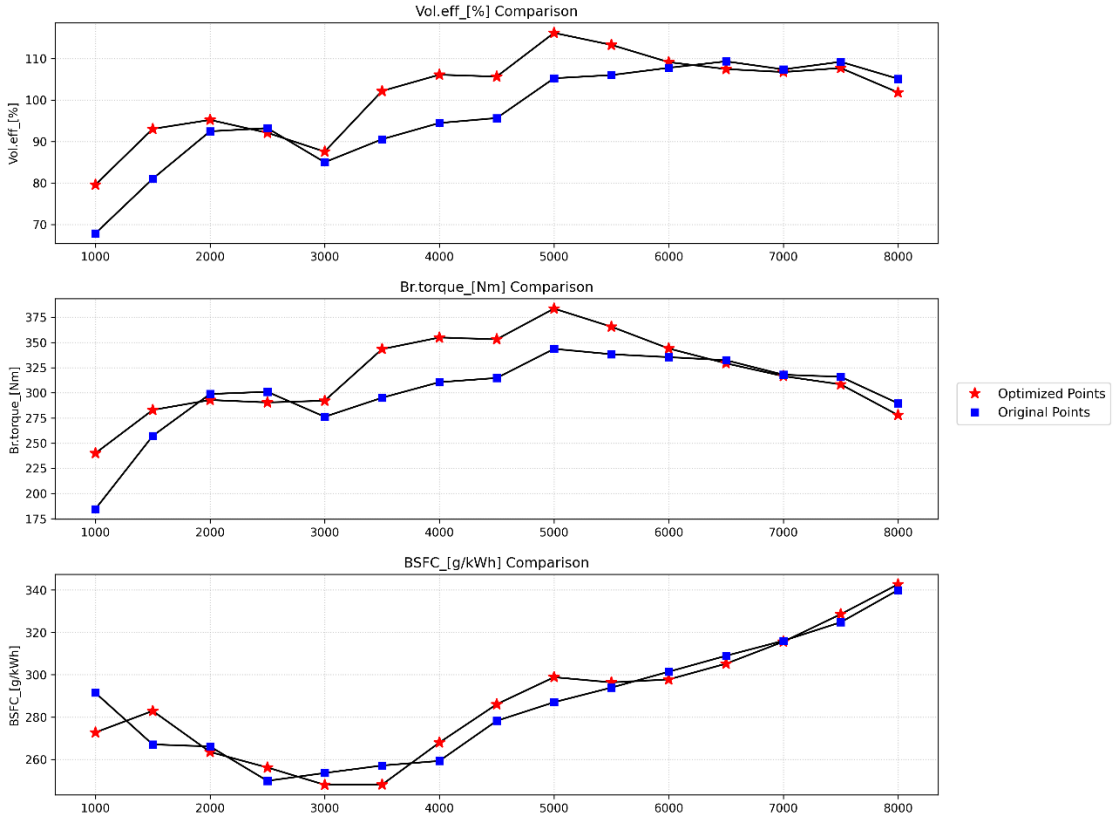

Figure 37. Configuration comparison with intake runner optimization

Focusing on volumetric efficiency the average change was 5.6% with a maximum increase of 17.2% still achieved at 1000 rpm. The reduction in the average change and in the maximum value is related to the introduction of a poorer tuning of the dynamical effects at other rpms. This showcases the effect of this strong optimization for a single rpm and the importance of having variable intake duct

length systems. The maximum reduction was 3.1% and occurred at 8000 rpm. Regarding the optimized point, 5000 rpm, the enhancement passed from 3.8% to 10.4%.

Regarding torque, the trend is kept, the average change dropped from 7.5% to 6.5%. At the optimized point the enhancement increased from 4.9% to 11.6%. The maximum increase was achieved at 1000 rpm with a value of 30%, 8 points less than what observed in the previous optimization step. The maximum decrease was observed to be 4% at 8000 rpm. Notice this is a large decrease when compared with the 0.7% maximum decrease observed earlier. These changes indicate that if the reductions of performance at 8000 rpms are limited, it is possible to achieve a more balanced optimized configuration. That is why the multi regime optimizer was suggested since it will enable this type of loss performance to be considered when tuning parameters that can only take a single value.

For brake specific fuel consumption, the trend is modified. The average increase was 0.3%, one quarter of what was seen with the previous results. The maximum increase was 5.9% at 1500 rpm and the maximum reduction was 6.4% at 1000 rpm. This leads to the conclusion that this optimization step was positive when considering this parameter.

In conclusion, the full optimization procedure was capable of generating a configuration that favors the operation at a single rpm. This is great for applications where the engine does not have to constantly vary its load and speed. This is not the case with land vehicles, and it might not be recommended to use an engine with this configuration. Still, the increase in torque and volumetric efficiency is impressive and shows the importance of correctly tuning this parameter but to better do it the detrimental effects introduced at other operating points need to be considered. This last comment suggests the importance of extending the optimization tool to include constraints on different performance parameters from regimes different from the one being optimized. This multi regime configuration is also fundamental to extend the usage of the optimization tool for engines where strategies offering different values for the engine parameters through the operating range are not employed and single values for the parameters must be employed.

5.6 THE EXTENDED DUCT LENGTH OPTIMIZATION

This final section will optimize the length of the intake runners along with the opening and closing of the valves discussed in section 5.2. This optimization problem is especially challenging because it will involve a continuous variable (that even if discretized can be thought of as continuous) and two binary variables. This problem falls into multiple integer nonlinear problems (MINLP) based on the definition provided in [4] since it handles discrete and continuous variables along with nonlinear system dynamics. Belotti et. al. suggests these problems are extremely challenging.

Most of the optimization problems up until now have been of this type but now the challenge resides in the fact that multiple variable types are going to be mixed. Without any relaxation, the intake runner length is completely continuous while the overture of the valves is discrete. The first simplification that we have applied is the discretization of the continuous space to provide physical sense to our results. Then we use a black box optimization method by running simulations to get information of the objective function, we have never modeled it mathematically. These strategies, embedded in the optimization tool, guarantee that these hard problems can be faced by the optimization tool here presented.

For this specific optimization the evolutive algorithm was selected as the best possibility. MADS algorithm follows the logic of exploring its surroundings as commented before. This leads to the generation of 26 trial points during the poll face. The problem is that due to the discretization of space, there are not 26 trial points around a single value leading to excessive executions from the algorithm which as commented earlier does not use the cache but replaces the not feasible values with random extractions from the sampling space σ. Therefore, the evolutive algorithm was chosen to face this challenging problem. Based on the analysis that had been done earlier, the idea was to have a large maximum number of iterations along with a high stagnation counter maximum even a value that practically deactivated this termination method.

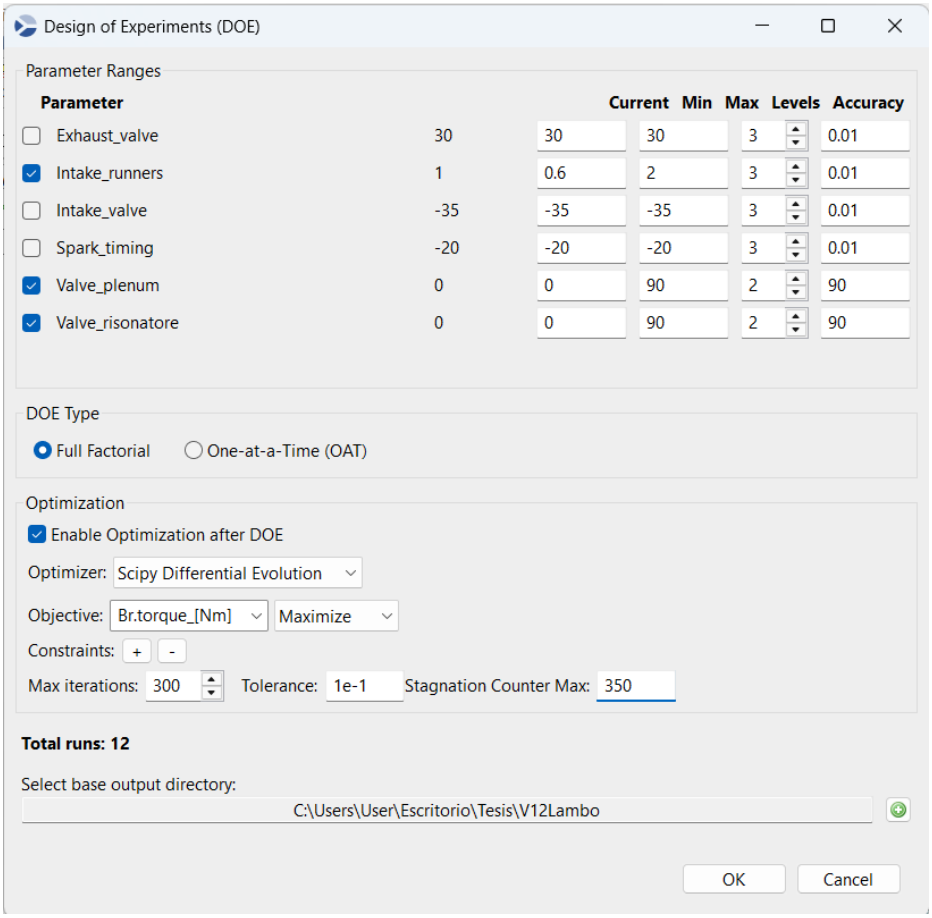

Figure 38. Input parameters for the rough optimization step

To facilitate the problem two optimization steps were taken. The first one covered the whole possible design space and used a rough tolerance, and the second one used the data collected in the first one to reduce the region being searched in the design space. The input parameters used for the rough optimization step are shown in figure 38. It can be observed that the parameters for the overture are configured as in section 5.1 while the parameter for the scaling factor of the length of the intake runners is bounded between 0.6 and 2. The maximum number of iterations was 300 and the stagnation counter was set to 350, the tolerance was set to 1e-1 to favor convergence. The selection of the stagnation counter provided a practical deactivation of this termination method. The input parameters for the detailed optimization step are presented in figure 39. The main differences are

the bounds for the intake runners which were modified to 1.5 and 1.9 using the data from the first optimization step, the reduction of the maximum number of iterations to 200, the reduction of the maximum of the stagnation counter to 250 and the reduction of the tolerance to 1e-4. The iterations were reduced because it was expected that in a smaller area of the design space it would be easier for the optimizer to converge, however the value of the tolerance kept this from happening.

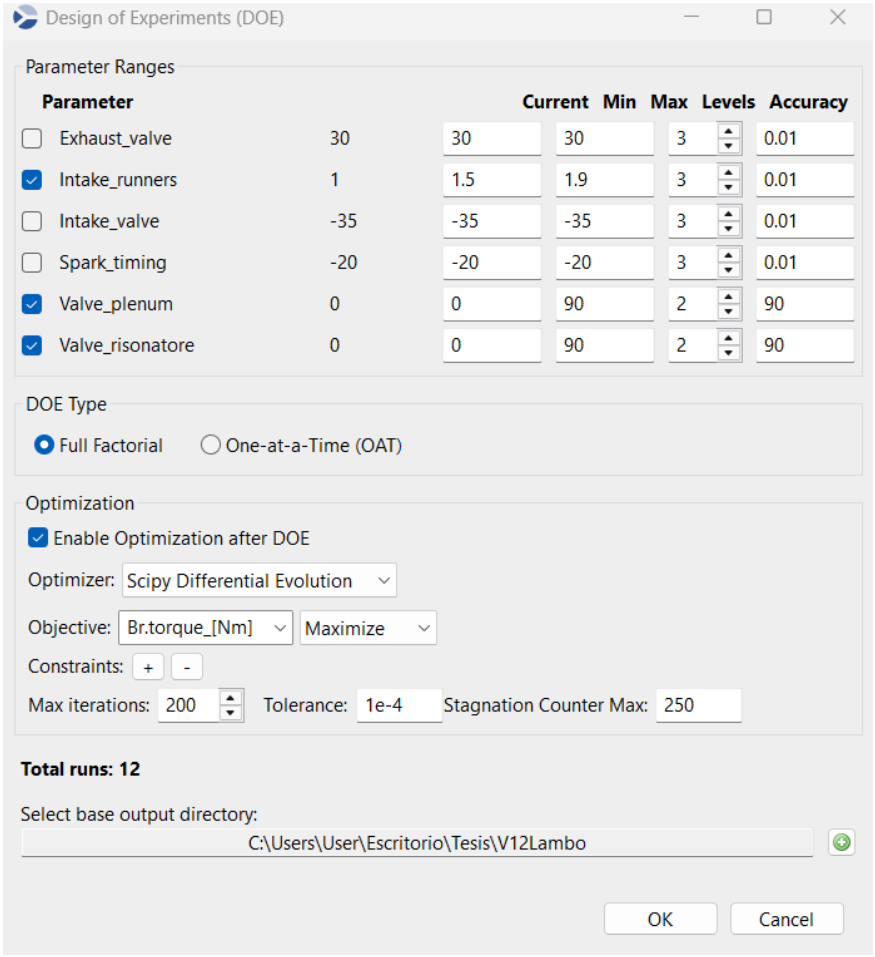

Figure 39. Inputs for the detailed optimization step

The first optimization process converged after only ninety-four iterations. From these iterations sixty were actual evaluations and thirty-six were cache hits. This means the algorithm had a cache efficiency of 38.3%. Thinking about what was commented earlier about the MADS algorithm, this amount of simulations would have been made in only three iterations with most of the data simulated being completely random. The value found was 1.75 with both valves opened. From the previous optimization procedures, it was found that the optimal value used both valves opened with an optimal shape scaling factor of 1.72. The torque value was $382.85\ Nm$ compared to the optimal of $383.7\ Nm$. The difference between these two values is larger than the tolerance which means the algorithm still could have identified this better value. However, keep in mind that the tolerance for the evolutive algorithm is used more of a threshold dispersion of the population so it works differently from what expected.

In the second optimization step the optimizer executed two hundred iterations, therefore it did not converge following the intrinsic logic from SciPy. The design space here was formed by one hundred and twenty possible values from which eighty-six were simulated. In total there were one hundred and fourteen cache hits, a 57% cache efficiency. Eighty-six simulations would have been done in four iterations from the MADS algorithm in this problem, therefore the evolutive algorithm is believed to be the right choice in this complex problem. It finally reached the same optimal point: 1.72 with both valves opened, reaching a satisfactory result.

This problem allowed the demonstration of the capabilities of the optimization tool which was capable of managing a complex problem of this characteristic thanks to the mechanics it uses and overall, thanks to the different optimizers it counts on. Thanks to the general way in which optimization algorithms have been implemented it is possible to implement more of them to fit different situations that could present different complexities, for example the multi regime optimization. There is one thing that has not been commented on related to the optimization of the intake runners. It is fundamental to keep in mind that the duct cannot be infinitely large or little, this can be constrained by using the limits of the scaling factor for the length which further demonstrates the general applicability of the current tool.

6. CONCLUSIONS

The document began by doing a mathematical description of the lower hierarchy found inside the tool, the optimizer. A large explanation was made all along section 2. Large emphasis was put on the MADS algorithm which was one of the largest contributions from the current work because it was custom fitted for the optimization of internal combustion engines.

The next section focused on describing the functioning principle of the code which is highly hierarchical. Four main hierarchies were identified: the optimizer, the optimization runner, the DoE form and the main graphical user interface. These elements related mathematics with Gasdyn (optimization runner), and Gasdyn with the user (DoE form and main GUI). The code was presented as highly modular, which allows for large modifications and easy maintenance of all its individual parts making it durable and ready to be manipulated by future coders. The ability to implement both commercial and custom optimization solutions is believed to be one of the main traits from the optimization tool since it will enable the integration of more algorithms that can face in the best possible way different optimization problems. Additionally, the parametric nature it has makes it possible to implement many different optimization problems without extensive modifications of the source code. One of the biggest problems observed in the previous optimization tools which due to the method they used to enable the modification of the engine parameters a parameter identifier needed to be included, along with all of the logic to systematically modify it, in the Gasdyn solver for it to become optimizable.

The final part of the thesis validated the tool by showing the way in which it manages constraints, how it faces classical optimization problems and by comparing the different functionalities it has. By focusing on a classical calibration method for an engine the tool could achieve average improvement of 5.6% for volumetric efficiency and of 6.5% for torque when the full tuning trajectory was implemented and of 6% for volumetric efficiency and of 7.5% for torque when the length from the intake runners was excluded from the optimization process.

Even if it was not the main objective of the thesis, it was possible to discuss the results from a theoretical perspective that validates much of the theory that exists around internal combustion engines. The discussions inside each of the sections dedicated to the experiments elaborate the theoretical validity of the results here observed. One of the most important results on this front was the relevance of the intake duct length as a parameter that can maximize performance for a single operating point but can also reduce the performance from the other operating points. The spark timing advance also presented large theoretical validity by presenting how as the engine speed is increased the optimal configuration suggests larger spark advancements to have more time for combustion completion. Furthermore, it also presented clearly how the increase on turbulence due to higher rotational speeds made this increase in the advancement not continously necessary. The optimizer was shown to be able to consider real physical limits on the different variables using user provided accuracy and the existence of experimental limits due to resolution through user given tolerances that are considered when comparing values through the optimization procedures.

The stated theory during section 2 for the MADS algorithm was verified by doing a close analysis of the optimization trajectories for two optimization processes in section 5.4.2 demonstrating the real application from the modifications that were implemented on the algorithm proposed originally in [11]. Additionally, its main weakness was found and analyzed by comparing it with SciPy's differential

evolution algorithm in a problem with a complex topology like the one observed when optimizing torque by modifying the spark timing. It was demonstrated that the two algorithms validate each other and do not lead to largely different results; however, the differential evolution heuristic uses an acquisition function based on exploration that offers two benefits. First, it provides a better mapping of the objective function and second it enables the optimization tool to correctly optimize the function even if challenging topologies exist. The tradeoff is the number of evaluations which were observed to increase. Due to this reason a cache mechanic was implemented that with enough iterations became highly effective reaching cache efficiencies of 38.3% and 56%. When simulations were limited, the cache was basically useless. It was also implemented in the MADS algorithm but is not frequently used because exploration was preferred over velocity when programming the algorithm.

SciPy's differential evolution was shown to be highly flexible in section 5.6 where the optimization problem was extended to become a MINLP that combined discrete and continuous variables with complex nonlinear dynamics. It was evidenced that it can face the problem thanks to the mechanics that have been embedded into it through the tool. The MADS algorithm proved to be perfect for simpler optimization procedures, but it can also face this problem, however it is not recommended in its current configuration because it will lead to a excessive number of simulations since the grid for this optimization problem is not traditional at all.

The validation of the results also emphasized the capabilities introduced by the post processing module which was responsible for the generation of all the data and graphs here presented. In general, the performance of the optimizer and the different mechanics that were implemented was satisfactory. To summarize the main contributions of the work a list was provided in section 4.3. Besides what listed there the validation and discussion of the results was done. The final parts of this section are devoted to the discussion of different additions that can be done to the optimization tool to enrich it.

6.1 FUTURE WORK

The different future lines of work are presented following the order in which they were mentioned throughout the document. They are the product of the reflections the author made throughout the development of the project. The first one commented regarded an extension in the environmental constraints. Even if it is great to have the chance to constrain the concentration and the flow of the pollutants, these are not the constrained values on the different regulations. As commented during section 5.1 the regulations constrain the values of g/km, therefore there is no certainty about achieving these limits with the configuration provided. It would be highly positive if the designers could simulate this by considering the test cycles and can even include some type of index that can be constrained. This is part of accepting the fact that the future of optimization in this application will be highly related to the reduction of pollutants and the compliance with the limits placed by regulatory agents. The Gasdyn software is capable of simulating transient trajectories, optimization over these trajectories to comply with regulatory requirements is a great chance to further demonstrate the capabilities from the optimization tool.

Then, during the VVT simulation it was demonstrated that the optimizer can manage the valve timing. However, it is known that modern third generation VVT systems can modify the lifting profiles from the valves to achieve load control or even modify the underlying thermodynamic cycles among

other possibilities to enhance performance [1]. It is believed that enhancing the optimization tool by allowing it to optimize the lift profiles from the valves will increase the robustness of the tool by allowing it to achieve the optimal configuration for the most modern third generation VVT systems. It is believed that the parametric nature of the current optimizer will help with making this possible in future versions of the optimization tool.

During this VVT optimization it was also observed that simulation is not the task that is taking the longest time for complex engine models, it is the compilation. The more complex models use larger amounts of data as input and the compilation step must modify and copy all this information in the simulation folder. This process is done in series which makes it highly time consuming. This opens a clear window for the improvement of the execution times of every optimization process.

The following experiment, optimization for spark timing, presented results that could lead to problematic combustion processes since they are not considering the increased risk of knock that is induced with the optimized configuration presented in figure 26. Introducing constraints verifying the chance of having abnormal combustion will enrich the optimization tool. For this to be done a feasible index must be included inside the constraints and must be tested to verify its use. This will also provide additional validation about the capability of the current optimization tool to manage constraints.

The final things concern the modification of optimization logic and the introduction of new optimization strategies. In general, it was observed that for complex nonconvex topologies the MADS algorithm got stuck. This is quite normal and these optimization problems are known to have a special complexity as commented in [4]. The introduction of the genetic algorithm offers a direct alternative that can help to face these problems; however, it is believed that there exists a way to modify the current MADS implementation to enhance performance. The decision of using a search step that randomly chooses values to replace previously evaluated ones was done to enhance the exploration that the algorithm was doing and provide the user with an idea of the trend from the objective function. Nevertheless, it is believed that even if it is important to evaluate areas away from the optimum, the algorithm also should take values close to the found candidate for best point with a certain probability. Some type of homologous method to the epsilon greedy method from reinforcement learning could be implemented which would take with a certain probability a value close to the optimal, otherwise it would take a random value as it is currently done [17]. This is a heuristic that can allow a search step that does not only explores the design space as a whole but also exploits the information about where it should explore to try and find better optimums.

Two completely different optimization systems are needed to achieve an optimization tool that can make the optimization tool highly robust. First, the multi regime optimization methods are believed to be relevant to make the tool usable in low budget designs and extend its applicability to other problems. Determining single value parameters that can balance performance along the whole operating range is relevant when there are no complex mechanisms to change the values at the different operation points or when a certain parameter that is fixed is going to be optimized, like the duct lengths. This new optimizer must be capable of introducing constraints at different regimes from the one that is being optimized and should optimize an objective function including performance metrics from all these individual regimes. The second optimization system that is

needed is for multi-objective optimization. A tool capable of generating the Pareto fronts from these optimization types is needed.

7. REFERENCES

[1] G. Ferrari, G. D'Errico and A. Onorati, "2.4 Flows through poppet valves," in *Internal Combustion Engines*, Bologna, Società Editrice Scolapio s.r.l., 2022, pp. 70-82.

[2] G. Ferrari, G. D'Errico and A. Onorati, "10.2 Spark ignition combustion of a homogeneous charge mixture," in *Internal Combustion Engines*, Bologna, Società Editrice Esculapio s.r.l., 2022, pp. 414-427.

[3] J. B. Heywood, "Modeling Real Engine Flow and Combustion Processes," in *Internal Combustion Engine Fundamentals*, New York, NY, USA, McGraw-Hill, 1988, pp. 748-822.

[4] P. Belotti, C. Kirches, S. Leyffer, J. Linderoth, J. Luedtke and A. Mahajan, "Mixed-Integer Nonlinear Optimization," *Acta Numerica,* vol. 22, pp. 1-131, 2012.

[5] X. Yu, L. Zhu, Y. Wang, D. Filev and X. Yao, "Internal combustion engine calibration using optimization algorithms," *Applied Energy,* vol. 305, 2022.

[6] S. Patnaik, N. Khatri and E. R. Rene, "Fueling the future: Exploring the synergy of artificial intelligence-based algorithms and the use of biofuels in engine development," *Taiwan Insititute of Chemical Engineering,* vol. 177, 2024.

[7] G. Boccardo, A. Piano, A. Zanelli, M. Babbi, L. Cambrigli, S. Mosca and F. Millo, "Development of a virtual methodology based on physical and data-driven models to optimize engine calibration," *Transportation Engineering,* no. 10, 2022.

[8] N. Murugavel, *Development of an optimization framework integrating Gasdyn for tuning internal combustion engine parameters to enhance performance,* Milano: Scuola di Ingegneria Industriale e dell'Informazione, Politecnico di Milano, 2025.

[9] E. Stringhetti, *Numerical optimization of naturally aspirated spark ignition engines operating with Atkinson cycle,* Milan: Scuola di Ingegneria Industriale e dell'Informazione, Politenico di Milano, 2018.

[10] S. Rao, "Exterior Penalty Function Method," in *Engineering Optimization Theory and Practice*, Hoboken NJ, USA, John Wiley & Sons, 2020, pp. 406-410.

[11] C. Audet and J. E. Dennis, "Mesh Adaptive Direct Search Algorithms for Constrained Optimization," *Society for Industrial and Applied Mathematics,* vol. 17, no. 1, pp. 188-217, 2006.

[12] A. Nelson, "scipy.optimize.differential_evolution," 2014. [Online]. Available: https://docs.scipy.org/doc//scipy-1.9.1/reference/generated/scipy.optimize.differential_evolution.html.

[13] Python Software Foundation, "queue — A synchronized queue class," [Online]. Available: https://github.com/python/cpython/blob/main/Doc/library/queue.rst?plain=1. [Accessed 5 6 2026].

[14] G. Montenegro, "12.3.4 Test Cycles," in *Internal Combustion Engines*, Bologna, Società Editrice Esculapio s.r.l., 2022, pp. 544-548.

[15] G. Ferrari, G. D'Errico and A. Onorati, "Intake and Exhaust Systems," in *Internal Combustion Engines*, Bologna, Società Editrice Escolapio s.r.l., 2022, pp. 133-155.

[16] G. Ferrari, G. D'Errico and A. Onorati, "10.5 Abnormal forms of combustion," in *Internal combustion engines*, Bologna, Società Editrice Esculapio s.r.l., 2022, pp. 440-454.

[17] R. S. Sutton and A. G. Barto, "2.2 Action-Value Methods," in *Reinforcement Learning an Introduction*, Cambridge, Masachussets, MIT, 2015, pp. 33-36.

APPENDIX A: FINAL RESULTS FOR VVT STRATEGY

Table 5. Final results for VVT strategy

Rotational Speed [rpm]	Intake Valve Opening [deg ATDC]	Exhaust Valve Opening [deg BBDC]
1000	-76.0	27.0
1500	-70.0	13.0
2000	-54.0	10.0
2500	-44.0	16.0
3000	-54.0	43.0
3500	-49.0	36.0
4000	-62.0	78.0
4500	-59.0	60.0
5000	-49.0	51.0
5500	-41.0	50.0
6000	-38.0	43.0
6500	-33.0	37.0
7000	-29.0	30.0
7500	-25.0	43.0
8000	-19.0	50.0

APPENDIX B: FINAL RESULTS SPARK TIMING STRATEGY

Table 6. Final results spark timing strategy

Rotational Speed [rpm]	Spark Timing [deg ATDC]
1000	-12.8
1500	-17.1
2000	-18.6
2500	-21.1
3000	-27.2
3500	23.4
4000	-28.8
4500	-30.1
5000	-30.1
5500	-30.3
6000	-30.0
6500	-29.8
7000	-29.9
7500	-30.1
8000	-32.3