Federal State Autonomous Educational Institution for higher professional education
National Research University
Higher School of Economics
International College of Economics and Finance

Glukhov Sergei Vladimirovich

Реализация метода MCS для задач глобальной оптимизации в финансовой математике

Implementation of the MCS Method for Global Optimization Problems in Financial Mathematics

BACHELOR'S GRADUATE QUAILIFICATION WORK
BACHELOR GRADUATION PAPER
for the study direction 38.03.01 ”Economics”
educational programme
“International Program in Economics and Finance”

Research supervisors:
Сухорукова Дарья Владимировна
Люлько Ярослав Александрович

Moscow, June 2025

Аннотация

Дипломная работа посвящена актуальной потребности в эффективных методах глобальной оптимизации для задач финансовой математики. Стандартные градиентные подходы зачастую демонстрируют низкую эффективность для вышеупомянутых задач, тогда как многие подходящие решения устарели технически.

Важнейший вклад данной работы заключается в создании новой высокопроизводительной версии алгоритма MCS, реализованной на языке Rust (Rust_MCS). Rust - современном языке системного программирования, отличающемуся надежностью и скоростью. Rust_MCS значительно превосходит предыдущие аналоги на MATLAB и Python, достигая ускорения в 500 раз и более. Для упрощенного взаимодействия пользователей разных уровней подготовлены интерфейсы интеграции с популярными языками программирования были созданы коннекторы для C++, Python3, а также современный веб-сервис, обеспечивающий доступ к функционалу даже пользователям без опыта программирования.

Польза разработанного инструмента (Rust_MCS) также иллюстрируется примерами практического применения в области финансов. Например, при формировании инвестиционного портфеля алгоритм позволяет достигать оптимального коэффициента Шарпа при нестандартных, сложных ограничениях, показывая лучшие результаты по сравнению с традиционными методами поиска решений на Python3. В рамках задачи калибровки финансовых моделей (модели Блэка–Шоллза и Хестона) Rust_MCS стабильно находит такие же, а иногда лучшие значения калибруемых параметров чем популярный способ из Python3, доказывая не только скорость и качество имплементации, но и полезность алгоритма как для практических, так и для научных задач.

Abstract

Global optimization is a fundamental pillar of financial mathematics, underpinning critical numerical methodologies for tasks such as model calibration, Sharpe ratio maximization, risk management, and valuation. These problems often involve objective functions that are nonconvex, noisy, computationally intensive, nondifferentiable, and of an inherently black-box nature. As a result, conventional gradient-based techniques (algorithms) prove either ineffective or impractical.

The Multilevel Coordinate Search (MCS) algorithm has proven to be a strong derivative-free method, performing effectively even with the complex financial landscapes.

This work addresses a need in global, derivative-free $\mathbb{R}^n \to \mathbb{R}$ optimizers by adapting MCS for modern computational environments.

The research aims to enhance numerical methods in quantitative finance by creating optimization tools that not only contribute to academic papers but also meet the industry's need for speed and reliability.

The most significant advancement is the development of Rust_MCS, a high-performance implementation of the MCS algorithm written in Rust. Rust, a modern systems programming language, offers exceptional reliability and speed, enabling Rust_MCS to achieve accelerations of 500 times or more compared to prior MATLAB and Python implementations, while preserving Matlab's correctness and functionality.

To ensure broad accessibility (given that most of financial libraries are in C++) I've developed interfaces for popular programming languages in finance (C++ and Python3) along with a modern web service. This web service allows users without programming experience to leverage Rust_MCS's capabilities.

The finance utility of Rust_MCS is demonstrated through its practical applications. For instance, in investment portfolio optimization, Rust_MCS efficiently identifies portfolios with optimal Sharpe ratio, even under complex constraints. It consistently outperforms traditional local search methods implemented in Python3. Furthermore, in the context of calibrating financial models (Black-Scholes and Heston models), Rust_MCS consistently finds similar, and sometimes better values for the calibrated parameters than the popular method from Python3, proving not only the speed and quality of the implementation but also the usefulness of the algorithm for both practical and scientific tasks.

Contents

1 Introduction

Global optimization lies at the heart of modern financial mathematics, providing the foundational numerical tools. These applications are not some theoretical constructs but practical necessities for ensuring accurate pricing, robust risk assessment, and strategic decision-making within financial institutions. The inherent complexities of markets and models, however, introduce significant challenges that often render traditional optimization approaches inadequate. Formally speaking, there is a necessity in global (not local) derivative-free optimizer $\mathbb{R}^n \rightarrow \mathbb{R}$ that can effectively deal with objective functions that are subject to common challenges of optimization problems in finance.

1.1 Challenges of Financial Optimization Problems

1.1.1 Non-Convexity

Objective functions in financial and ML models frequently exhibit non-convexity. This characteristic is not an abstract mathematical observation, but rather an intrinsic property directly resulting from the integration of real-world market complexities and practical limitations. For instance, the design of portfolio optimization models that account for higher-order moments, such as skewness and kurtosis, or real-world limitations (difficult constraints), intrinsically leads to non-convexity. This causes gradient-based optimizers to get "stuckked" in suboptimal local solutions.

1.1.2 Costly Function Evaluations and Non-Differentiability

In complex financial models the objective function is usually defined only via an expensive algorithm. In other words, computing f(x) itself can require heavy computation – e.g. implicitly stated (probably recursive) f(x), running a large Monte Carlo simulation or multi-step pricing routine– so each evaluation is costly.

First-order derivatives are not available (and algorithmic differentiation is often infeasible). Using finite differences would require on the order of n additional function calls to estimate a gradient and would suffer from severe numerical instability (step-size tradeoffs cause truncation or cancellation error). In a Monte Carlo evaluation Stochastic noise only makes matters worse: any numerical gradient estimate is likely to be dominated by sampling error. Reliable derivative information is effectively unattainable in these settings. One more example: inclusion of transaction costs in option pricing models can lead to non-linear partial differential equations or objective functions with piecewise linear or discontinuous cost structures. These pervasive non-differentiabilities and discontinuities are not minor inconveniences but fundamental structural features that render traditional gradient-based methods inherently inapplicable. Derivative non-zero optimizers rely on the existence and continuity of derivatives. When these properties are absent, such methods are not merely inefficient; they are inherently unsuitable. This justifies the requirement for zeroth-order (derivative-free) optimization methods.

1.1.3 Intrinsic Noise in Financial Data and Stochastic Models

Financial data inherently contains multiple noise components, which can broadly be classified into market microstructure noise and model-induced noise. Market microstructure noise arises from frictions in trading processes, such as bid-ask bounce, price discreteness, and asynchronous trading, leading to short-term "spikes" in high-frequency price series. Empirical studies have shown that this microstructure noise can significantly distort estimates, requiring specialized filtering techniques to recover "true" underlying price processes.

In addition to these data-driven sources, model-induced noise stems from numerical approximations and finite sampling. In Monte Carlo simulations the estimator for an expectation is itself a random variable. As a result, expressions for derivative prices (and "Greeks") can exhibit high variance.

Moreover, floating-point arithmetic introduces biases, particularly when summing thousands or millions of small increments (e.g., log-returns over high-frequency intervals). The error accumulates in long-running simulations or iterated recursions (as in some local volatility calibration routines), leading to noisy results if not properly controlled.

1.2 Role of Derivative-Free Global Optimizers in Quantitative Finance

Traditional gradient-based methods, despite their widespread applicability, struggle when faced with the characteristic features of financial objective functions listed above. These methods, which rely on iterative updates informed by first or second order derivatives, frequently become trapped in local minima and exhibit significant performance degradation under real-world financial mathematics challenges.

Consequently, there is a compelling need to adopt derivative-free global optimization techniques that do not assume smoothness or the availability of gradients. Among these, the Multilevel Coordinate Search (MCS) algorithm, introduced by Huyer and Neumaier in 1999, has demonstrated particular effectiveness. MCS systematically homes in on global minima without demanding gradient information. Its guaranteed convergence to the global optimum (under continuity assumptions near global minima) and modest memory requirements (storing only a coordinate tree) make it especially attractive for bound-constrained financial applications.

However, before detailing the specific developments and comparative performance analyses of MCS within financial mathematics, it is instructive to situate this work within the broader corpus of derivative-free global optimization techniques that have emerged over the past two decades. In particular, the subsequent literature overview will examine how seminal contributions—ranging from stochastic search heuristics (PSO etc) to deterministic partition-based methods (DIRECT etc) have been adapted to address the problems commonly encountered in finance stated above.

2 Literature Overview

Derivative-Free Global Optimization (DFGO), often referred to in literature as black-box optimization, is a field dedicated to finding optimal solutions within predefined solution spaces (box) for complex real-world problems, particularly where the analytical form of the objective function or its derivatives is unavailable or unreliable. This domain has a rich history and is currently undergoing significant expansion due to its wide applicability in science, engineering, and notably, financial mathematics and AI. Algorithms developed for this class of problems are broadly classified based on several characteristics. A primary distinction is made between direct methods, which determine search directions by computing values of the objective function directly, and model-based methods, which construct and utilize a surrogate (or approximate) model of the objective function to guide the search process[1].

Furthermore, algorithms are categorized as local or global. Local search algorithms aim to find an optimum within a specific region of the search space, whereas global optimization methods are designed to explore the entire feasible domain to identify the global optimum, often possessing mechanisms to arbitrarily refine the search domain. This distinction is particularly salient in financial mathematics, where objective functions (e.g., in model calibration or complex portfolio optimization) can be multimodal, exhibiting numerous local optima that are not the true global solution. Finally, algorithms are distinguished as stochastic or deterministic. Stochastic algorithms incorporate random elements or probabilistic decisions within their search steps. In contrast, deterministic algorithms follow a fixed sequence of operations and, given the same starting point and problem instance, will always produce the same trajectory and result. Deterministic methods, under specific conditions such as function continuity, can often provide theoretical guarantees of convergence to a global optimum.

The diversity of DFGO problems, especially in finance—characterized by non-linearity, non-convexity, non-smoothness, potential noise in function evaluations, and often computationally expensive objective functions—has led to a wide array of algorithmic approaches. Different classes of algorithms inherently possess strengths suited to particular problem structures. For instance, deterministic methods may offer convergence guarantees but can be rigid in their exploration, while stochastic methods excel at escaping local optima but typically lack such strong guarantees and can be computationally intensive [3]. This inherent variety and the lack of a universally superior algorithm are consistent with the implications of the “No Free Lunch” theorems in optimization, which suggest that no single algorithm can outperform all others across all possible problems.

It is important to note that many modern software implementations of these algorithms rely on hybrids, combining characteristics from more than one of these categories. The prevalence of hybrid algorithms proved the “No Free Lunch” theorem once more. Effective optimization often requires combining the strengths of different paradigms—for instance, the global exploration capabilities of deterministic methods with the local exploitation power of stochastic or model-based techniques—to successfully navigate the challenging landscapes encountered in financial mathematics. This hybridization is a key theme in modern DFGO research. Financial modelers must carefully weigh these trade-offs: for high-stakes decisions, such as those involving regulatory capital models, robustness and the assurance of finding a true global optimum might be prioritized over raw computational speed, potentially favoring hybrid methods with strong global search components or deterministic algorithms if the problem characteristics are suitable. Conversely, for exploratory analyses or less critical calibration tasks, faster stochastic or local methods might be deemed sufficient. This implies that algorithm selection itself becomes a crucial component of the financial modeling process, sometimes necessitating empirical testing or reliance on emerging algorithm selection tools[1, 5].

2.1 Deterministic Global Optimization Algorithms

Deterministic global optimization algorithms aim to find the global optimum of a function through a systematic and reproducible search process. These methods are often characterized by theoretical convergence guarantees.

2.1.1 Lipschitzian-Based Partitioning Methods: DIRECT and its Variants

The DIviding RECTangles (DIRECT) algorithm, introduced by Jones, Perttunen, and Stuckman (1993), is a prominent derivative-free global optimization method. It was developed to address key challenges associated with earlier Lipschitzian methods, such as the requirement for a known Lipschitz constant and the exponential growth in function evaluations often encountered [1].

DIRECT operates by systematically partitioning the search space, which is initially normalized to an n-dimensional unit hypercube, into smaller hyperrectangles. A core principle is the evaluation of the objective function solely at the center of these hyperrectangles, rather than at all extreme points. The algorithm employs a strategy of subdividing intervals (or dimensions of hyperrectangles) into thirds. This partitioning ensures that one of the resulting sub-partitions inherits the center of the parent interval, where the objective function value is already known, thus avoiding redundant computations.

The selection of hyperrectangles for further division is based on identifying so-called 'potentially optimal hyperrectangles' (POHs). A hyperrectangle j is deemed potentially optimal if it satisfies two main conditions. Firstly, its score, which balances its objective function value $f(c_j)$ at its center c_j against its size d_j (typically a measure like the distance from the center to its vertices or half the main diagonal), must lie on the lower-right envelope of the scores of all current hyperrectangles. This essentially means there must exist some positive rate-of-change constant $\tilde{K}$ (which DIRECT identifies implicitly, without requiring an explicit Lipschitz constant K) such that:

$$f(c_j) - \tilde{K}d_j \leq f(c_i) - \tilde{K}d_i, \quad \forall i \in S \tag{1}$$

where S is the set of all current hyperrectangles. Secondly, the selected hyperrectangle must offer a significant potential for improvement over the current best-found solution, f_{min}. This is formalized by the condition:

$$f(c_j) - \tilde{K}d_j \leq f_{min} - \epsilon|f_{min}| \tag{2}$$

where ϵ is a small positive tolerance parameter that balances local (exploitation around f_{min}) and global (exploration of larger, less evaluated hyperrectangles) search aspects.

The DIRECT algorithm is guaranteed to converge if the objective function is continuous in the neighborhood of a global optimum.

Despite its strengths, DIRECT has limitations. It cannot directly handle problems with infinite bounds and may exhibit slow convergence if the optimum lies on the boundary of the search domain, as its fixed division scheme does not evaluate points directly on the boundaries. Furthermore, while DIRECT is often quick to identify the basin of the global optimum, it can be slow in fine-tuning the solution to high accuracy[3]. The reliance on hyper-rectangle partitioning, though more efficient than exhaustive endpoint evaluation, still faces challenges with increasing dimensionality; the number of hyperrectangles to manage can grow rapidly, impacting overall efficiency, a phenomenon referred as a "curse of dimensionality" in geometric partitioning methods[5].

A critical issue for financial applications is DIRECT's normalization of variables to a unit hypercube. This geometric transformation implicitly assumes an isotropic search space, where the objective function exhibits similar sensitivity to changes in all normalized dimensions. However, financial models frequently involve parameters with vastly different scales and sensitivities. For instance, in option pricing, the objective function can be orders of magnitude more sensitive to a small change in volatility than to a proportionally similar change in an interest rate. Uniform scaling can therefore distort these intrinsic financial parameter relationships, leading to inefficient exploration. The algorithm might expend considerable computational effort refining dimensions of low sensitivity or misrepresent the relative importance of parameters, thereby misallocating its search budget.

The DIRECT-L variant, proposed by Gablonsky and Kelley (2001), attempts to address some of DIRECT's characteristics by introducing a bias towards local search[5]. This modification aims to enhance efficiency for functions with relatively few local minima[2]. However, this increased local bias, combined with the rigid hyperrectangle division strategy inherited from the original DIRECT (based on trisection), can lead to premature stagnation, particularly in flat regions of the objective function or near shallow local minima[6]. Such flat or shallow regions are not uncommon in financial calibration tasks, for example, when calibrating yield curve models or implied volatility surfaces where multiple parameter sets might yield similar objective function values. The algorithm may struggle to make substantial progress if the landscape does not offer clear descent paths that align well with its partitioning scheme. It is important to distinguish the DIRECT-L algorithm from "Direct Indexing," which is an unrelated financial investment strategy. The challenges posed by anisotropic financial landscapes underscore the need for either sophisticated pre-processing of financial problems before applying standard DIRECT, or the use of DIRECT variants or related algorithms (like MCS) that can handle scaling and partitioning more adaptively[2].

2.1.2 Deterministic Hybrid and Multistart Solvers

To enhance the performance of deterministic global optimization methods, hybrid approaches and multistart strategies are commonly employed. These often combine the strengths of global exploration techniques with the efficiency of local search algorithms.

OQNLP: OQNLP [22] is a multistart heuristic algorithm specifically designed to find global optima for smooth-constrained nonlinear programs (NLPs) and mixed-integer nonlinear programs (MINLPs). It operates by launching a local NLP solver from multiple, strategically chosen starting points. OQNLP is noted for its rapid convergence rates, particularly when the initial starting point for a local search is in proximity to the true optimum. This characteristic makes it effective for quickly obtaining “good enough” solutions, which can be valuable for tasks such as initial model calibration or rapid prototyping in financial modeling. However, OQNLP’s performance can degrade significantly when very high absolute precision is required. Furthermore, its effectiveness tends to diminish with very large evaluation budgets, and its ranking may deteriorate, especially for separable multimodal problems as the budget increases. This implies a critical trade-off: OQNLP is well-suited for rapid, approximate solutions but may be less appropriate for financial applications demanding extremely high accuracy, such as regulatory capital calculations or the pricing of complex derivatives that require tight error bounds.

StoGO (Stochastic Global Optimization): StoGO is a global optimization algorithm that systematically partitions the search space, which must be bound-constrained, into smaller hyper-rectangles using a branch-and-bound technique[2]. Subsequently, it searches these sub-regions using a gradient-based local-search algorithm, typically a BFGS (Broyden–Fletcher–Goldfarb–Shanno) variant, and can optionally incorporate some degree of randomness in its search[7]. A critical characteristic of StoGO is its explicit requirement for the gradient of the objective function to facilitate its local optimization phase, as evidenced by the “gr” argument in its R package documentation and NLopt implementations[7, 8]. The original development is attributed to Zertchaninov and Madsen[8].

Due to its reliance on gradient information for the local search component, StoGO is primarily designed for problems with smooth objective functions and may perform poorly when applied to non-smooth objectives, which are frequently encountered in financial modeling (e.g., optimization of Value-at-Risk (VaR), or functions involving path dependencies or early exercise features in derivatives)[2]. This fundamental dependency on gradients significantly restricts its applicability in quantitative finance, where analytical gradients are often unavailable or unreliable for complex, “black-box” models. The “availability gap” for gradients means that while StoGO’s branch-and-bound framework offers a global search perspective, its core refinement engine (BFGS) is hampered without accurate gradient information. Consequently, global search capability alone is insufficient; it must be coupled with genuinely derivative-free mechanisms to be broadly practical for many real-world financial optimization tasks.

2.2 Stochastic Global Optimization Algorithms

Stochastic global optimization algorithms employ probabilistic rules and random elements to explore the search space, often making them effective at escaping local optima in complex landscapes.

2.2.1 Population-Based algorithms

Population-based algorithms maintain and evolve a collection (population) of candidate solutions simultaneously, sharing information among them to guide the search towards promising regions.

Particle Swarm Optimization (PSO): PSO is a widely recognized metaheuristic global optimization paradigm inspired by the collective social behavior observed in flocking birds or schooling fish. It operates as a stochastic, evolutionary, population-based heuristic where individual candidate solutions, termed "particles," navigate the multidimensional search space. Each particle adjusts its trajectory based on its own best-discovered position and the best position found by the entire swarm, thus balancing individual exploration with collective exploitation. A key advantage of PSO is that it does not require any information about the derivatives of the objective function, relying solely on evaluating the objective function at different points in the search space[11].

In the context of financial mathematics, PSO has been extensively applied to portfolio optimization problems. In such applications, particles typically represent potential asset allocations, and the algorithm iteratively moves the swarm of these candidate solutions towards better regions of the search space to balance conflicting objectives, such as maximizing expected return while minimizing portfolio risk (variance or other risk measures)[10].

Despite its conceptual simplicity and ease of implementation, PSO presents several challenges. Its stochastic nature means that it cannot offer formal guarantees of convergence to the global optimum[12]. The performance of PSO is highly sensitive to the selection of its operational parameters, such as inertia weight, and cognitive (personal best) and social (global best) acceleration coefficients. Finding optimal settings for these parameters can be a difficult task and is often problem-dependent, potentially requiring a meta-optimization process itself[13]. Empirical studies have indicated that PSO may require a substantial number of iterations—for instance, one study noted around 331 iterations for portfolio weights to stabilize, although this figure can vary widely depending on the problem specifics and PSO variant [11]—and consequently, a large function evaluation budget. This can be a significant drawback when dealing with computationally expensive financial models where each function evaluation is time-consuming[12]. While attractive for its straightforward implementation, PSO's practical deployment in high-stakes financial environments is often hampered by this lack of theoretical convergence guarantees and the potentially high computational cost associated with achieving stable and reliable solutions. The need for extensive iterations to stabilize portfolio weights, for example, implies a high function evaluation budget, which might negate its initial "ease of implementation" advantage for computationally intensive financial models.

Controlled Random Search (CRS2_LM): The Controlled Random Search (CRS) algorithm, particularly the CRS2 variant with local mutation (CRS2_LM), is classified as a global optimization algorithm within the NLopt library [2]. It operates by maintaining a population of points in the search space and evolving these points using randomized heuristic rules. This evolutionary process, driven by random exploration and selection based on objective function values, is characteristic of global optimization strategies and is often compared to genetic algorithms due to their shared principle of exploring a broad region of the search space [14].

The random sampling nature inherent in CRS2_LM can become markedly inefficient as the number of dimensions increases (the "curse of dimensionality") [18]. This leads to a dramatic increase in the number of function evaluations required to adequately explore the search space, and the algorithm's computational cost tends to scale poorly with increasing problem size. Furthermore, the algorithm often exhibits slow and inconsistent convergence, a characteristic it shares with other multistart methods (like MLSL). The curse of dimensionality is particularly acute for algorithms that rely heavily on random sampling.

As the number of dimensions in financial models grows, the volume of the search space increases exponentially, making it practically impossible for random sampling to adequately cover the space with a reasonable computational budget. The “scalability wall” for pure random sampling methods like CRS2_LM means they are generally unsuitable for the majority of high-dimensional tasks.

2.2.2 Evolutionary Computation Algorithms

Evolutionary Computation (EC) are stochastic, nature-inspired algorithms specifically designed for optimization tasks. A key characteristic of these algorithms is their derivative-free nature, which makes them particularly well-suited for black-box optimization problems.

The study by Stripinis et al. (2025) found that when rotation is introduced in test instances, all prominent EC algorithms analyzed experience a significant drop in performance. On rotated separable problems, the AGSK algorithm was among those with the highest performance drop of approximately 16%. In the case of rotated non-separable problems, AGSK experienced the largest performance drops of approximately 9%, whereas EA4eig’s performance dropped by about 7% [3]. The impact of rotation is most pronounced in higher dimensions; for instance, AGSK’s performance on rotated separable problems can decrease by as much as 44%. This rotational variance is a critical concern for financial applications, as financial problems rarely exhibit perfectly axis-aligned parameter dependencies; correlations between financial variables effectively create a rotated or non-separable landscape. The direct application of these EC algorithms to many financial models may therefore yield suboptimal performance. This underscores the need for research into, or the application of, rotationally invariant EC methods or pre-processing techniques (like PCA) to unlock their full potential in finance.

When non-separable problems are introduced, the success rates of all these EC algorithms tend to decrease significantly, especially for higher-dimensional problems. Nevertheless, algorithms such as EA4eig, and AGSK still demonstrate relatively strong performance on non-separable problems, managing to solve a high percentage of such instances.

Another important consideration is the computational budget. To achieve performance levels comparable to the best deterministic methods, these top-tier EC algorithms typically require a substantial minimum evaluation budget. This “data-hungry” nature arises because EC algorithms are population-based, necessitating the evaluation of numerous solutions in each generation. Furthermore, the complex adaptive mechanisms embedded in advanced EC methods often require many function evaluations to effectively learn features of the optimization landscape and adapt their search parameters. This high sample complexity can be a challenge for computationally expensive financial models, such as those involving complex Monte Carlo simulations for derivative pricing or large-scale portfolio rebalancing with detailed transaction cost models. While EC methods offer robustness for complex, high-dimensional problems, their significant budget requirements mean they are best suited for financial problems where either the objective function is relatively inexpensive to evaluate, or where the potential impact of the optimization justifies extensive computational effort.

Specific EC algorithms highlighted for their performance include: EA4eig and AGSK.

2.2.3 Multistart and Local Search Methods with Randomization

This category includes methods that combine global exploration, often through randomized starting points, with local search procedures to refine solutions.

Multi-Level Single-Linkage (MLSL): MLSL is not a standalone optimizer but rather a framework that requires a separate local optimization algorithm to function effectively. Its primary objective is to locate all local minima within the search space, under the assumption that one of these will correspond to the global optimum. MLSL operates as a "multistart stochastic global optimization method," integrating several components: random sampling for broad exploration of the search space, clustering techniques to group promising points and crucially, to avoid repeated local searches in the basin of attraction of the same local optimum, and local search strategies for refinement around these potential minima. The derivative requirements of MLSL are thus dependent on the specific local optimization method it employs; it can utilize either gradient-based or derivative-free local solvers. Under certain conditions, MLSL offers theoretical guarantees of finding all local optima in a finite number of local minimizations [2].

While MLSL can be effective for low dimensional problems where its clustering mechanism is more efficient [2], its reliance on random restarts can introduce variability in convergence times and overall performance, which is also influenced by the choice of the local minimizer. A significant limitation arises in high-dimensional financial models: clustering methods, including single-linkage, generally become inefficient in high-dimensional spaces[17]. This algorithm is also subject to the "curse of dimensionality" problem. This illustrates a fundamental trade-off between the aspiration for exhaustive exploration and the practical constraints imposed by high-dimensional search spaces prevalent in finance. Furthermore, the effectiveness of MLSL is critically tied to the quality and efficiency of the chosen local optimizer; a suboptimal local solver will inevitably lead to poor overall MLSL performance.

Nelder-Mead (Simplex Method): The Nelder-Mead algorithm, introduced by Nelder and Mead (1965), is a well-known fast, derivative-free, direct local search method primarily used to find a local minimum of an objective function[21]. It does not guarantee finding the global minimum. Its core mechanism involves maintaining and iteratively updating a simplex (a geometric figure consisting of n+1 points in an n-dimensional space) by directly evaluating function values at its vertices, without requiring any derivative information[21]. These updates involve operations like reflection, expansion, contraction, and shrinkage of the simplex. This derivative-free nature makes it suitable for problems where the objective function is non-smooth, discontinuous, or where derivatives are difficult or impossible to compute[21]. Its applicability extends to parameter estimation and statistical problems where function values might be uncertain or subject to noise[23].

However, as a local search algorithm, Nelder-Mead's simplex updates struggle significantly with high-dimensional spaces[25]. The method is most effective for low dimensional problems [25] because as the number of dimensions grows, the simplex becomes more complex, and its volume and geometry can degenerate, leading to premature stagnation or very slow convergence. This illustrates a fundamental scaling limitation for direct search methods that rely on such geometric constructs. In high-dimensional problems, the simplex becomes unwieldy, leading to inefficient exploration and potential stagnation. This means Nelder-Mead is largely unsuitable for the high-dimensional optimization tasks, restricting its use to only very specific, low dimensional sub-problems or possibly as a final polisher if a good solution is already identified by a more robust global method, though its tendency to converge to non stationary points even for smooth convex functions is a concern[21].

Newton-CG (Newton Conjugate Gradient): Newton-CG is classified as a local optimization method, typically listed under local gradient-based optimization algorithms in libraries like NLopt[2]. It is a variant of Newton's method that utilizes a second-order Taylor series approximation of the objective function around the current iterate, thereby involving both the gradient (first-order derivative) and the Hessian matrix (second-order derivative) of the objective function[27]. While Newton-CG employs the conjugate gradient (CG) method to iteratively solve the Newton system ($H\Delta x = -g$), thereby avoiding the explicit inversion of the Hessian (which is particularly beneficial for large-scale problems), it still requires the computation of Hessian-vector products[28]. Its ability to detect and utilize negative curvature directions in the Hessian for non-convex functions further underscores its reliance on second-order derivative information[26].

The Newton-CG algorithm offers fast and consistent convergence for smooth functions for which the required derivatives are available[26]. However, the primary limitation for its use in financial mathematics is the stringent requirement for both first-order (gradient) and second-order (Hessian or Hessian-vector products) derivatives[27]. Having access to second-order derivatives is often too much to ask for in financial mathematics as stated earlier. While theoretically powerful and fast, Newton-CG's practical utility in quantitative finance is severely limited by the pervasive "black-box" nature and frequent non-differentiability of financial objective functions. Accurately approximating or computing second-order derivatives is often impossible or at least introduces heavy computational burden. For financial models this often negates the theoretical convergence speed.

2.2.4 Proprietary and Commercial Optimization Frameworks (NAG)

Closed-source global search minimizers, such as the robust NAG Optimization Modelling Framework available through the NAG Library and other proprietary solvers, are renowned in financial mathematics for their rigorous, thoroughly tested algorithms that deliver highly accurate and reliable solutions[35]. NAG Libraries offer powerful and reliable optimizers capable of solving large portfolio optimization and selection problems in the financial industry [36].

The advantages of using NAG's algorithms include exceptional numerical stability, extensive and meticulous documentation, and dedicated expert support that helps ensure models meet the strict accuracy and compliance requirements prevalent in the finance industry [37]. In practice, its solvers can efficiently explore enormous search spaces to identify globally optimal solutions, a critical feature when managing large-scale, high-stakes financial models [36]. NAG optimization solvers are highly flexible and callable from many programming languages, environments, and mathematical packages [37]. It is noteworthy that NAG implements the Multilevel Coordinate Search (MCS) algorithm (nag_glopt_bnd_mcs_solve), based on the work of Huyer and Neumaier (1999), which is recommended for low dimensional problems with simple bound constraints [39].

However, the closed-source nature of these tools limits transparency into the specific algorithmic implementations and restricts customizability, which can be a drawback when users need to audit the algorithms in detail or tailor them to niche financial applications. Additionally, the significant licensing costs associated with proprietary systems like NAG can restrict access for smaller firms or academic research groups when compared to more flexible, open-source alternatives[38]. The high cost and lack of full transparency of proprietary suites like NAG are frequently accepted as a necessary investment for the unparalleled numerical stability, accuracy, and expert backing required for real-world financial applications. This demonstrates a distinct cost-benefit calculus in industrial financial contexts compared to academic or open-source environments. The "trust vs. transparency" dilemma is often resolved in favor of trust when operational and regulatory risks are high, and the specialized nature of algorithms like NAG's MCS implementation, with recommendations for specific problem types, underscores that even commercial suites offer a portfolio of tools rather than a single universal solver.

2.3 Comparison of Optimizers in Different Financial Contexts

The diverse landscape of optimization algorithms presents a spectrum of choices for tackling problems in financial mathematics. Each algorithm possesses unique characteristics regarding its search strategy (global or local) and its reliance on derivative information. Table 1 provides a comparative summary of some optimizers discussed in this review.

Algorithm Name	**Global / Local**	**Derivative Order Needed**
MCS	Global	0 (Derivative-Free)
DIRECT	Global	0 (Derivative-Free)
MLSL	Global*	Varies (depends on local minimizer)
CRS2_LM	Global	0 (Derivative-Free)
StoGO	Global*	1 (Gradient)
Nelder-Mead	Local	0 (Derivative-Free)
Newton-CG	Local	2 (Gradient, Hessian)
PSO	Global	0 (Derivative-Free)
OQNLP	Global	0 (Derivative-Free)
EA4eig	Global	0 (Derivative-Free)
AGSK	Global	0 (Derivative-Free)
NAG Framework	Global*	Varies

Table 1: Optimization algorithms with their characteristics. Global* indicates that the algorithm has global search strategies, which often rely on local optimizers.

A fundamental axis of differentiation among derivative-free solvers is their performance relative to the available computational budget, measured in function evaluations. Financial models exhibit a vast range in computational cost: from near-instantaneous analytical formulas to computationally intensive simulations. This economic reality creates two distinct optimization regimes.

The Low-Budget Regime: For expensive functions where the number of evaluations is severely limited (fewer than a thousand evaluations for a moderately dimensional problem), deterministic and model-based methods demonstrate clear superiority. Algorithms like OQNLP and DIRECT-type hybrids are designed to be highly sample-efficient. They employ structured search patterns or build surrogate models that extract a maximal amount of information from each function evaluation. Their initial convergence is rapid, making them ideal for applications requiring quick, reliable results from complex models.

The High-Budget Regime: Conversely, when the objective function is computationally cheaper and a large number of evaluations is feasible, the performance landscape shifts dramatically in favor of advanced stochastic and evolutionary algorithms. Solvers like EA4eig often require a significant number of initial evaluations to initialize their populations and adapt their internal search parameters. During this ”warm-up” phase, they may underperform deterministic methods. However, once adapted, their sophisticated exploration and exploitation mechanisms enable them to navigate complex, high-dimensional landscapes more effectively, ultimately achieving superior solution quality. This makes them well-suited for tasks such as overnight risk model calibration, the backtesting of complex trading strategies, or academic research where computational resources can be extensively deployed to find the true global optimum.

The crucial insight for the practitioner is that the selection of an optimizer cannot be divorced from the operational context and computational economics of the financial task. There is no universally ”best” algorithm; there is only an algorithm best suited to the intersection of the problem's mathematical structure and its practical budget constraints.

Moreover, financial models are rarely isotropic; their parameters often exhibit vastly different scales and sensitivities. This creates a stretched, or anisotropic, search space. Furthermore, the presence of correlations—such as between asset returns in a portfolio or between factors in a multi-factor model—effectively rotates the problem's natural coordinate system.

This geometric reality poses a significant challenge for many optimizers. Empirical studies robustly demonstrate that many powerful evolutionary algorithms, while highly effective on axis-aligned (separable) test problems, suffer a dramatic loss of performance when the problem is rotated. This rotational variance is a critical vulnerability. The direct application of a high-performing EC solver to a financial problem with correlated parameters may lead to inefficient search and suboptimal results, as the algorithm's search operators are not aligned with the landscape's true structure.

In contrast, certain algorithmic classes exhibit greater robustness to this rotation. DIRECT-type methods, for instance, partition the space in a way that, while not explicitly rotation-invariant, can be less susceptible to performance degradation than some EC methods that have strong directional biases. The Covariance Matrix Adaptation Evolution Strategy (CMA-ES), and hybrids incorporating it, are specifically designed to learn and adapt to the correlation structure of the objective function, effectively deforming their search distribution to align with the landscape's geometry. This makes them theoretically well-suited for the correlated, anisotropic problems common in finance. The key takeaway is that an understanding of a model's parameter dependencies and correlations is vital for selecting an appropriate optimizer. Ignoring the geometric properties of the financial landscape can lead to a severe misapplication of even the most powerful algorithms.

Effective optimization almost invariably requires a balance between two distinct phases: a global search to ensure the correct region of the solution space is identified, and a local search to refine the solution to high precision within that region. No single algorithm perfectly masters both.

Global partitioning methods, such as those based on the DIRECT framework, excel at the exploration phase. They are designed to systematically rule out vast regions of the search space and are reliable in locating the basin of attraction of the global optimum. However, their fixed partitioning schemes can make them inefficient at the final convergence stage, often requiring many evaluations to achieve high numerical precision.

Conversely, local search algorithms are exceptionally efficient at local refinement. When initiated within the correct basin of attraction, they can converge rapidly to a highly precise solution. The challenge, of course, is that they are blind to the global landscape and will quickly become trapped in the first local minimum they encounter.

This inherent duality strongly motivates the use of hybrid algorithms, which have become the state-of-the-art. These hybrids formalize the two-phase approach:

- **Deterministic Hybrids:** These explicitly combine a global partitioning algorithm (like DIRECT) with a dedicated local search routine. The global method identifies promising regions, from which the local solver is launched to perform efficient refinement. The MCS algorithm is here.
- **Stochastic Hybrids:** The most successful evolutionary algorithms are themselves hybrids. They combine population-based global exploration with sophisticated adaptive mechanisms that act as powerful local search engines, learning the landscape's local curvature and adapting the search accordingly.

For many financial applications, particularly in pricing and risk management, this two-phase paradigm is not just advantageous but essential. A global search is required to avoid model misspecification from calibrating to a local optimum, while a high-precision local refinement is necessary because small errors in calibrated parameters can lead to significant and unacceptable pricing.

While theoretical performance is often measured in function evaluations, practical financial applications are constrained by wall-clock time and the need for operational reliability.

Execution Time: A critical finding from comprehensive benchmarks is that some of the most sample-efficient deterministic algorithms can be computationally intensive internally. The overhead of managing complex data structures (like partition lists in DIRECT-type methods) can lead to situations where the algorithm is slower in practice than a stochastic method, even if it uses fewer function evaluations.

Reliability: In a production environment, an algorithm must be robust. A solver that is brilliant on average but occasionally fails on seemingly easy problems is a significant operational risk. Empirical studies that track such failures reveal that some otherwise powerful algorithms can exhibit this brittleness. In contrast, other solvers, often well-tested hybrids, demonstrate exceptional reliability by consistently solving the vast majority of problems thrown at them, even if they are not always the fastest.

2.4 Conclusion

This review of the Derivative-Free Global Optimization (DFGO) algorithms reveals a vast and diverse ecosystem of algorithms, each founded on distinct mathematical principles. However, a critical analysis through the lens of financial mathematics' problematic leads to conclusion: a significant majority of these general-purpose algorithms are inadequately suited for the unique and demanding challenges inherent in financial applications. The defining characteristics of financial optimization problems—such as non-convex and non-smooth objective functions, the frequent unavailability of analytical derivatives (the ”black-box” problem), severe parameter anisotropy, and the rotational effects induced by correlated variables—collectively act as a rigorous filter, disqualifying many widely-known methods.

Specifically, entire classes of algorithms are rendered impractical. Gradient-based local optimizers like Newton-CG and even global methods requiring first-order information like StoGO are fundamentally incompatible with the predominantly non-differentiable nature of complex financial models. Classic local search methods such as Nelder-Mead, while simple and derivative-free, are unreliable due to their inability to escape local optima—a critical flaw when model risk and accuracy are paramount—and their poor scaling with dimension. Similarly, many stochastic methods, including population-based algorithms like PSO and random-sampling techniques like CRS2_LM, often struggle with the ”curse of dimensionality” and require vast computational budgets that are infeasible for expensive financial functions, all while offering no guarantee of convergence. Even highly sophisticated Evolutionary Algorithms (EA4eig, AGSK), which demonstrate formidable performance on academic benchmarks, exhibit a critical vulnerability: a significant degradation in efficiency when faced with the rotated, non-separable problem geometries that are the norm in finance due to parameter correlations.

The MCS algorithm, the focus of this work, emerges from this review as a great fit! Its proven effectiveness is underscored by its implementation in industrial-grade, proprietary libraries such as the NAG Optimization Framework, which are trusted for high-stakes financial modeling.

Ultimately, the choice of an optimizer in finance is a multi-faceted decision that weighs the trade-offs between sample efficiency and internal computational overhead, speed and operational reliability, and, most critically, the alignment between an algorithm's intrinsic assumptions and the geometric reality of the financial problem. This literature review firmly establishes that the path to successful optimization in financial mathematics lies not in applying generic tools, but in identifying and implementing algorithms from the small class of specialized, structurally-aware methods capable of navigating these complex and unforgiving landscapes. But once the ”right” algorithm is identified, its implementation starts playing the leading role.

3 Detailed Description of the MCS Algorithm

This section synthesizes explanations of the algorithm from [1, 38, 40] with my own commentary.

The Multilevel Coordinate Search (MCS) algorithm is classified as a derivative-free global optimization algorithm. It belongs to the family of Lipschitzian approaches. While some Lipschitzian techniques necessitate prior knowledge of a Lipschitz constant, MCS is specifically engineered to perform effectively even when this constant is unknown.

MCS is a deterministic algorithm, a characteristic that differentiates it from stochastic methods that incorporate random or probabilistic elements into their search strategies. The deterministic nature of MCS provides a significant advantage in terms of solution reproducibility and theoretical guarantees when compared to stochastic methods. This characteristic renders MCS particularly appealing for applications where rigorous validation and consistent outcomes are paramount, such as critical engineering design, financial modeling, or scientific discovery, where results must be consistently repeatable and verifiable.

3.1 Historical Context: Inspiration from the DIRECT Algorithm and MCS's Innovations

The MCS algorithm is conceptually aligned with the DIRECT (DIviding RECTangles) method for global optimization, originally proposed by Jones *et al.* in 1993. Both algorithms share the fundamental principle of combining global exploration with local exploitation through a multilevel partitioning approach.

However, MCS was developed to address several practical limitations inherent in the original DIRECT algorithm:

- **Handling Infinite Box Bounds.** The original DIRECT algorithm normalizes a finite search box to a unit hyper-cube, which inherently restricts its application to problems with predefined finite bounds. MCS overcomes this constraint by allowing for problem formulations with either finite or infinite box bounds, significantly broadening its applicability.
- **Convergence at Boundaries.** DIRECT can exhibit unnecessarily slow convergence when the global minimizer is located precisely at the boundary of the search space, as its partitioning scheme may struggle to reach the boundary directly. MCS improves upon this by allowing for more irregular splitting procedures and by explicitly placing division points at the boundaries of the hyper-rectangles.
- **Splitting Procedure.** While DIRECT typically subdivides a box along multiple coordinates simultaneously, MCS generally performs splitting along a single coordinate in each step. This allows for a more focused and adaptive exploration strategy based on coordinate-specific information derived from sampled points.

3.2 Formal Problem Formulation: Bound-Constrained Optimization (with finite or infinite bounds)

The MCS algorithm is specifically designed to solve the general bound-constrained global optimization problem:

$$\min_{x\in[u,v]} f(x).$$

In this formulation:

- $f(x)$ represents the multivariate objective function to be minimized. It is assumed to be a black-box function, implying that its analytical expression or derivatives are not available.
- x denotes an n-dimensional vector of decision variables, where $x \in \mathbb{R}^n$.
- $[u, v]$ defines the search space as a rectangular box (hyper-rectangle). Formally,

  $$[u, v] := \big\{ x \in \mathbb{R}^n \mid u_i \le x_i \le v_i,\ i = 1, \dots, n \big\}.$$

  The vectors u and v are n-dimensional vectors whose components can be finite real numbers or extend to $\pm\infty$. A fundamental condition is that $u_i < v_i$ for all $i = 1, \dots, n$, which ensures a non-empty search interval for each dimension.

Only points with finite components are considered valid elements of the box $[u, v]$, even when its bounds are infinite. In the special case where $u_i = -\infty$ and $v_i = +\infty$ for all i, the problem effectively reduces to an unconstrained optimization problem.

The explicit accommodation of infinite bounds in the problem formulation is a critical design choice that significantly broadens MCS's applicability. This feature allows MCS to tackle problems without requiring users to impose arbitrary finite boundaries. In many real-world scenarios—especially during initial problem formulation or when the optimal region is not tightly localized—defining precise finite bounds can be challenging or might introduce artificial constraints that exclude the true global optimum.

3.3 Core Principles and Algorithmic Architecture of MCS

3.3.1 Fundamental Philosophy of Multilevel Coordinate Search

The central philosophy of MCS is to iteratively refine the search for a global minimizer by systematically partitioning the multi-dimensional search space into progressively smaller hyper-rectangles ("boxes"). Each hyper-rectangle contains a distinguished point, known as the *base point*, for which the objective function value is known. A key distinction from the DIRECT algorithm is that this base point is not necessarily the midpoint of the box but can be located anywhere within it, frequently at its boundary.

The partitioning strategy employed by MCS is not uniform across the entire search space. Instead, MCS prioritizes the subdivision of regions that are anticipated to contain lower function values, thereby focusing computational effort on more promising areas. This adaptivity in base-point placement and the non-uniform partitioning strategy represent deliberate design choices to enhance the algorithm's responsiveness to the objective landscape. Such dynamic allocation of computational resources allows MCS to adjust its search intensity and focus, potentially leading to more efficient convergence in complex or asymmetric objective landscapes. A rigid, midpoint-based strategy might be inefficient if the function's minimum is consistently located near boundaries or if the function exhibits highly irregular behavior within subregions. Non-uniform partitioning enables a more intelligent allocation of computational resources. This adaptive approach accelerates convergence by concentrating search efforts where they are most likely to yield significant improvements, making MCS more robust across diverse function landscapes.

MCS can be conceptualized as a "branch without bound" method. It employs interval subdivision techniques akin to those found in traditional branch-and-bound methods, but it distinguishes itself by not relying on explicit lower bounds to prune less promising subregions from the search space.

3.3.2 The Multilevel Strategy: Hyper-rectangle Levels (s) and the Maximum Level (s_{max})

The core mechanism for balancing global exploration and local exploitation in MCS is its multilevel approach. Each hyper-rectangle is assigned a *level*, denoted by s, where $s \in \{0, 1, \ldots, s_{\text{max}}\}$. This level serves as a rough measure of how many times a particular box has been processed or subdivided.

- $s = 0$: This level indicates that the hyper-rectangle has already been divided and is no longer considered for further subdivision in the current sweep.
- $s = s_{\text{max}}$: This represents the maximum permissible level. Hyper-rectangles reaching this level are considered too small for further division, effectively setting a minimum granularity for the search.
- $0 < s < s_{\text{max}}$: These are the active levels. When a box within this range is selected and split, its current level is reset to 0. Its newly generated child boxes (descendants) are then assigned a new level of either $s + 1$ or $\min(s + 2,\ s_{\text{max}})$.

This hierarchical level system implicitly correlates with the size of the hyper-rectangle: boxes with smaller s values (excluding $s = 0$) generally correspond to larger boxes that have undergone fewer subdivisions. The dynamic assignment of levels—particularly the $\min(s+2,\ s_{\text{max}})$ rule for descendants—is a sophisticated heuristic for controlling the search's granularity. The $s + 2$ jump implies that some children are immediately considered for a finer level of search, rather than just the next sequential level. Concurrently, the s_{max} cap prevents indefinite subdivision. This non-linear progression of levels allows the algorithm to accelerate the refinement of promising regions by giving them a higher level, thus making them eligible for more detailed scrutiny sooner, while ensuring that the overall search depth is bounded by s_{max}. This dynamic level management is a key design feature for optimizing the exploration-exploitation trade-off.

3.3.3 The Sweep Mechanism: Balancing Global Exploration and Local Exploitation

The MCS algorithm orchestrates its search through a series of *sweeps* that are performed iteratively after an initial setup phase. This sweep mechanism is fundamental to how MCS balances its global and local search objectives.

The global search component of MCS is primarily realized by the systematic nature of its sweeps. In each sweep, the algorithm prioritizes processing boxes starting from the lowest available levels, which correspond to the largest, least-explored hyper-rectangles. This ensures a broad and systematic investigation of the entire search space, preventing the algorithm from prematurely focusing on a single region.

Within each level during a sweep, MCS employs a local exploitation strategy. It selects the box that possesses the lowest objective-function value for further processing. This criterion directs the search toward regions that have already shown promise, facilitating the refinement of potential minima. The sweep mechanism represents a sophisticated implementation of the exploration-exploitation trade-off: it ensures comprehensive global coverage by systematically revisiting broader regions, while simultaneously optimizing local refinement by focusing on the most promising areas within each level. This dual focus is critical for efficient global optimization. The algorithm first decides where to search (globally, by level) and then what to prioritize within that selected region (locally, by best observed function value). Such a design prevents the algorithm from becoming trapped in a local optimum too early by forcing continued exploration of broader territories, while ensuring that promising areas are refined with high efficiency.

3.3.4 Initialization of Sweeps and the Record List (b_s)

The MCS algorithm commences with an initialization procedure designed to generate an initial set of hyper-rectangles and their corresponding function values. This process typically involves evaluating the objective function f at an initial point x_0, which is then designated as the current best point, $x^* = x_0$. Subsequently, for each coordinate $i \in \{1, \ldots, n\}$, f is evaluated at $L_i - 1$ additional points within the search space $[u, v]$. These points are chosen such that they agree with x^* in all other coordinates, thereby forming L_i pairs $(x_{i,\ell}, f_{i,\ell})$ for each dimension.

A *record list* is maintained throughout each sweep. For every active level s (where $0 < s < s_{\text{max}}$), this list contains a label b_s. The label b_s points to the hyper-rectangle that currently holds the lowest function value among all non-split boxes at that specific level s. If no boxes exist at a particular level s, then b_s is set to zero. To initiate a sweep, the level index s is set to the lowest level for which $b_s \neq 0$, ensuring that the algorithm begins its exploration from the broadest, least-processed regions.

3.4 Algorithmic Flow and Splitting Rules

3.4.1 Initialization Procedure

The MCS algorithm begins with a comprehensive initialization procedure that sets the stage for the subsequent iterative search. Unlike the DIRECT method, MCS allows a base point to belong to multiple boxes, and the base point of a box is typically not its midpoint but a point on its boundary (often a vertex). Each box is defined by a base point x and an "opposite point" y, forming a box $B[x, y]$.

The initialization process generates an initial set of boxes by performing the first split along each coordinate. This is done at three or more user-defined values $x_{i,\ell}$, where function values are computed, and some adaptively chosen intermediate points, resulting in at least four subboxes. Specifically, for each dimension $i \in \{1, \ldots, n\}$, a set of ordered points

$$u_i \;\leq\; x_{i,1} \;<\; x_{i,2} \;<\; \cdots \;<\; x_{i,L_i} \;\leq\; v_i \quad \text{with} \quad L_i \geq 3$$

is defined. The objective function f is first evaluated at an initial point x_0, which is set as the current best point x^*. Then, for each coordinate i, f is evaluated at $L_i - 1$ points in $[u, v]$ that match x^* in all other coordinates, yielding L_i pairs $(x_{i,\ell}, f_{i,\ell})$. The point with the smallest function value is then designated as x^* before proceeding to the next coordinate.

From the initialization list and corresponding function values, an initial set of boxes is constructed. The root box is $B[x, y] = [u, v]$, with $x = x_0$ as the base point and y as one of the corners of $[u, v]$ farthest from x. For each coordinate i, the current box is split into $2L_i - 2$, $2L_i - 1$, or $2L_i$ subintervals—depending on whether two, one, or none of the $x_{i,\ell}$ lie on the boundary. This involves additional splitting points

$$z_{i,\ell} \;=\; x_{i,\ell-1} + q\,(x_{i,\ell} - x_{i,\ell-1}), \quad \ell = 2, \ldots, L_i, \quad q = \tfrac{1}{2}(5 - \sqrt{5})$$

with $k = 1$ or 2 chosen so that the part with the smaller function value receives a larger fraction of the interval.

The resulting subboxes acquire new base points x' (obtained by changing x_i to the boundary point $x_{i,\ell}$) and opposite points y' (obtained by changing y_i to the other end of the interval). This information allows for defining priorities on coordinates. For each i, the variability of f with respect to that coordinate is estimated using quadratic interpolation through three consecutive $(x_{i,\ell}, f_{i,\ell})$ pairs. Coordinates with higher variability are assigned higher priority, stored in a ranking vector π_i.

The root box is assigned level $s = 1$. When a box of level s is split, the boxes corresponding to the smaller fraction of the golden-section split receive level $\min(s + 2,\ s_{\text{max}})$, while the others receive level $s + 1$. Consequently, after initialization, non-split boxes have levels in $\{2, \ldots, n + 2\}$, implying that s_{max} should be significantly larger than n.

3.4.2 The Sweep Process (Detailed Steps)

After the initialization procedure, the branching process in MCS proceeds through a series of sweeps. Each sweep is defined by three iterative steps:

Step 1: Scan and Define Record List (b_s) The algorithm scans the list of all currently non-split boxes. For each active level s (where $0 < s < s_{\text{max}}$), a record list is constructed.

This list contains a label b_s that points to the box with the lowest function value among all boxes at that specific level s. If no boxes exist at a particular level s, then $b_s = 0$. The sweep then initializes its current level index s to the lowest level for which $b_s \neq 0$. This ensures that the algorithm prioritizes exploration of the broadest, least-processed regions first.

Step 2: Candidate Splitting and Updates The box identified by the label b_s (the box with the lowest function value at the current level s) is considered a candidate for splitting.

- If the box is *not* split (based on rules detailed in Section 3.3), its level is incremented by one. Subsequently, the b_{s+1} entry may be updated if this newly leveled box now represents a better candidate at level $s+1$.
- If the box *is* split, it is marked as "split" and its newly generated child boxes (descendants) are inserted into the system. The record list is then updated if any of these children yield a strictly better function value at their respective new levels.

Step 3: Level Advancement and Sweep Control The current level index s is increased by one.

- If s reaches s_{max}, the current sweep is considered complete, and a new sweep is initiated from Step 1.
- If $s < s_{\text{max}}$ but $b_s = 0$ (meaning there are no non-split boxes at level s), the algorithm proceeds directly to Step 3 again, effectively skipping over empty levels.
- Otherwise, if $s < s_{\text{max}}$ and $b_s \neq 0$, the algorithm returns to Step 2 to process the next candidate box at the new level s.

This structured, iterative process ensures that MCS systematically explores the search space while continuously prioritizing the most promising regions for refinement.

3.4.3 Splitting Rules: Division by Rank vs. Division by Expected Gain

The MCS algorithm employs two primary splitting rules for hyper-rectangle division: *Division by Rank* and *Division by Expected Gain*. These rules determine how a candidate hyper-rectangle is subdivided to explore the function's landscape more effectively.

Division by Rank This rule is applied when a box has reached a sufficiently high level, indicating it has been processed multiple times.

- **Condition for Application.** A hyper-rectangle is always divided by rank if its level s satisfies
$$s > 2\,n\left(\min_j n_j + 1\right),$$
where n is the dimension of the problem and n_j is the number of times coordinate j has been split in the history of the box. The term $\min_j n_j$ represents the minimum

division count across all coordinates. This condition ensures that even if a box does not show immediate promise for improvement, it will eventually be split along all its coordinates to ensure thorough exploration of the search space.

- **Splitting-Index Selection.** When splitting by rank, the algorithm selects the coordinate i that has been split the fewest times (i.e., $n_i = \min_j n_j$). Among those coordinates with the smallest n_i, the one with the lowest priority π_i (highest variability rank determined during initialization) is chosen as the splitting index.
- **Division Process.**
  - If $n_i = 0$ (coordinate never divided): The splitting is performed according to the initialization list, using predefined values $x_{i,\ell}$ and golden-section split points. New base points and opposite points are defined, and the resulting subboxes are assigned levels. Boxes corresponding to the smaller fraction of the golden-section split receive level $\min(s+2,\, s_{\text{max}})$, while the others receive $s+1$.
  - If $n_i > 0$ (coordinate previously divided): The i-th component of the box ranges between x_i and y_i. The splitting value z_i is chosen as

    $$z_i \;=\; x_i \;+\; \tfrac{3}{2}\left(\xi'' - x_i\right), \quad \text{where} \quad \xi'' = \text{subint}(x_i,\, y_i).$$

    This split primarily aims at reducing the size of a large interval rather than finding an optimal function value. The box is divided into three parts using z_i and the golden-section split point, requiring only one additional function evaluation. The smaller golden-section fraction receives level $\min(s+2,\, s_{\text{max}})$, and the other two parts get $s+1$. The base points of the children are set accordingly, with the first child's base point being z_i, and the second and third children's base points being x' (obtained by changing the i-th coordinate of x to z_i).

  The function $\text{subint}(x, y)$ is crucial for managing splits within potentially unbounded intervals, ensuring that new components are not excessively large and that descendants shrink appropriately. It is defined as:

  $$\text{subint}(x,y) \;:=\; \begin{cases} \text{sign}(y), & \text{if } 1000\,|x| < 1 \text{ and } |y| > 1000, \\ 10\ \text{sign}(y)\,|x|, & \text{if } 1000\,|x| < 1,\ |y| > 1000\,|x|, \\ y, & \text{otherwise.} \end{cases}$$

  This function helps define a smaller interval within which the new component must lie, preventing splits from occurring too close to the base point and ensuring that descendants of both bounded and unbounded boxes shrink sufficiently fast over time.

Division by Expected Gain This rule is applied when the hyper-rectangle's level s is relatively small, indicating a larger unexplored territory, and the algorithm seeks to prioritize splits likely to lead to a significant reduction in the function value.

- **Condition for Application.** This rule is considered if

  $$s \le 2\,n\left(\min_j n_j + 1\right).$$

At this stage, a local separable quadratic model of the function f is constructed to estimate the expected gain along each coordinate.

- **Local Separable Quadratic Model.** The model is represented as

$$e(\xi) \;=\; f(x) \;+\; \sum_{i=1}^{n} e_i(\xi_i),$$

where each $e_i(\xi_i)$ is a quadratic function fitted using interpolation at the base point x and $2n$ additional points. These additional points are obtained by tracing back the history of the box for coordinates that have been split, or from the initialization list for coordinates that have not.

- **Expected Gain Calculation** ($\hat{e}^i$)**.** For each coordinate i, the expected gain $\hat{e}^i$ in function value is calculated when a new point is evaluated by changing only that coordinate in the base point:
  - If $n_i = 0$ (coordinate never divided): The expected gain is

$$\hat{e}^i \;=\; \min_{\ell=1,\ldots,L_i} \{f_{i,\ell}\} \;-\; f(x_{i,\ell}).$$

  - If $n_i > 0$ (coordinate previously divided): A quadratic partial correction function

$$e_i(\xi_i) \;=\; \alpha_i\,(\xi_i - x_i) \;+\; \beta_i\,(\xi_i - x_i)^2$$

    is used. The splitting value z_i is chosen from the interval $[\xi',\, \xi'']$ (defined by the subint function), and

$$\hat{e}^i \;=\; \min_{\xi_i \in [\xi',\, \xi'']} e_i(\xi_i).$$

- **Splitting Condition.** The box is split if the expected best function value,

$$f_{\text{exp}} \;:=\; f(x) \;+\; \min_{1 \le i \le n} \hat{e}^i,$$

is less than the current best function value f_{best} (including values from local optimization). If this condition is violated, the box is not split, and its level is simply increased by one.

- **Division Process.**
  - If $n_i = 0$: The splitting follows the initialization list, and the assignment of new base points, opposite points, and levels is the same as in Division by Rank when $n_i = 0$.
  - If $n_i > 0$: The splitting value z_i is used. The box is split at z_i (if $z_i \neq y_i$) and at the golden-section split point, resulting in two or three parts. The larger fraction of the golden-section split receives level $s + 1$, and the smaller fraction receives min$(s + 2,\, s_{\text{max}})$. If a third part exists and is larger than the smaller golden-section fraction, it receives level $s + 1$; otherwise, it receives min$(s + 2,\, s_{\text{max}})$. Base points and opposite points are updated similarly to the Division by Rank rule.

3.5 Local Search Enhancement and Management of Local Optima

3.5.1 The Role of Local Search in MCS

The local search enhancement in MCS is designed to accelerate convergence by initiating focused searches from promising points. This is particularly beneficial once the global search component of the algorithm has identified a region likely to contain a global minimizer. The local search algorithm within MCS primarily involves building local quadratic models and performing line searches.

3.5.2 Triple Search for Quadratic Model Construction

The *triple search* is a fundamental procedure for constructing a local quadratic model, which serves as an approximation of the objective function $f(x)$ around a current best point x_{best}. This quadratic model is represented as

$$q(x) \;=\; f \;+\; g^\top(x - x_{\text{best}}) \;+\; \tfrac{1}{2}\,(x - x_{\text{best}})^\top G\,(x - x_{\text{best}}),$$

where f is the function value at x_{best}, g is an approximation of the gradient, and G is an approximation of the Hessian matrix. The primary goal of the triple search is not only to build this model but also to reduce the function value.

To construct the quadratic model, the triple search utilizes $n(n+1)/2$ function values. The process involves evaluating the function at specific points derived from three vectors $x_\ell < x_m < x_r$. For each coordinate i, points are chosen such that their ith component is one of $x_{i,\ell}$, $x_{i,m}$, $x_{i,r}$, while other components match x_{best}.

The procedure for building the quadratic model and updating x_{best} is iterative:

1. **Initialization.** The current best function value is set as $f = f(x_{\text{best}})$.
2. **Diagonal Elements (g_i and G_{ii}).** For each coordinate $i = 1, \dots, n$:
   - Compute $f(x_{(i,1)})$ and $f(x_{(i,2)})$, where $x_{(i,1)}$ and $x_{(i,2)}$ are points obtained by changing only the ith coordinate of x_{best} to the other two values in $\{x_{i,\ell}, x_{i,m}, x_{i,r}\}$.
   - Fit a quadratic polynomial

     $$p(\xi) \;=\; f(x_{\text{best}}) \;+\; g_i\,(\xi - x_{\text{best},\,i}) \;+\; \tfrac{1}{2}\,G_{ii}\,(\xi - x_{\text{best},\,i})^2$$

     that interpolates at $(x_{(i,j)},\, f(x_{(i,j)}))$ for $j = 1, 2$. Use these values to compute approximations for g_i and G_{ii}. If $\min\{f(x_{(i,1)}),\, f(x_{(i,2)})\} < f(x_{\text{best}})$, store a new best point $x_{\text{new\_best}}$ without immediately updating x_{best}.
3. **Off-Diagonal Elements (G_{ik}).** For each $k = 1, \dots, i-1$:
   - Compute $q(x_{(k,1)})$ and $q(x_{(k,2)})$ using the current quadratic model.
   - Compute $f(x_{i,k})$, where $x_{i,k}$ is a point obtained by changing the ith and kth coordinates of x_{best} to the ones with smaller $q(x_{(i)})$ and $q(x_{(k)})$ values from the current quadratic model.
   - Update $x_{\text{new\_best}}$ if $f(x_{i,k})$ yields a strict improvement.

- Compute G_{ik} (and G_{ki}) such that the quadratic model interpolates at $x_{i,k}$.

4. **Update** x_{best}. After completing the loops for i and k, if $x_{\text{new\_best}} \neq x_{\text{best}}$, then update $x_{\text{best}} \leftarrow x_{\text{new\_best}}$, and re-expand f and $g_{1:i}$ around the new x_{best}.

This method ensures that the constructed quadratic model is the unique quadratic interpolant to $n(n+1)/2$ distinct points. An optional input parameter allows for handling sparsity patterns in the Hessian, which can reduce the number of function evaluations. A "diagonal triple search" is a simplified version that only computes diagonal elements of the Hessian, taking off-diagonal elements from the previous iteration, requiring only $2n$ additional function values.

3.5.3 Coordinate Search for Evaluation Points

The *coordinate search* procedure is employed to determine the x_ℓ, x_m, x_r points for the triple search. It is based on a line search routine:

1. **Initial Line Search.** A line search (with a parameter $s^{\text{ls}}_{\text{max}}$ limiting the number of points) is performed along each coordinate, starting with the candidate point for the "shopping basket."

2. **Point Selection.** After the line search along the first coordinate, the best point found and its two nearest neighbors (or two nearest neighbors on one side if boundary conditions apply) are selected as $\{x_{1,\ell}, x_{1,m}, x_{1,r}\}$.

3. **Subsequent Line Searches.** For subsequent coordinates ($i > 1$), the line search starts from the current best point obtained from the previous line search. The set $\{x_{i,\ell}, x_{i,m}, x_{i,r}\}$ is chosen to include:

   - The best point found so far,
   - The starting point of the current line search (if different),
   - If possible, the nearest neighbor of the best point on the other side.

   It is crucial that the ith coordinate of the old x_{best} remains within $\{x_{i,\ell}, x_{i,m}, x_{i,r}\}$ to preserve the integrity of previously fitted surfaces.

3.5.4 Overall Local Search Algorithm Steps

The local search algorithm in MCS integrates the triple search and coordinate search into a multi-step process:

1. **Initial Exploration** Starting with a candidate from the "shopping basket," a full triple search is performed. The points x_ℓ, x_m, x_r for this triple search are determined by a coordinate search. This step yields an updated point x, its function value f, and approximations of the gradient g and Hessian G.

2. **Quadratic Model Minimization and Line Search**

- Define an initial search box as

$$d \;:=\; \min\big(v - x,\; x - u,\; 0.25\,(1 + |x - x_0|)\big),$$

where x_0 is the absolutely smallest point in $[u, v]$.

- Minimize the quadratic function

$$q(h) \;=\; f \;+\; g^\top h \;+\; \tfrac{1}{2}\,h^\top G\,h$$

over the box $[-d,\; d]$ to find a promising search direction p.

- Perform a line search along $x + \alpha p$, using $\alpha = 0$ and $\alpha = 1$ as input. Update x and f accordingly.

- Compute the ratio

$$r \;=\; \frac{f_{\text{old}} - f}{f_{\text{old}} - f_{\text{pred}}},$$

where f_{old} is the function value at the beginning of Step 2, and f_{pred} is the predicted function value from the quadratic model at $\alpha = 1$. This ratio r indicates the predictive quality of the quadratic model.

3. **Stopping Criteria** Check termination conditions:

   - If a limit on the number of visits to Step 3 (per local search) or the total function calls has been exceeded, stop.
   - If no components of the current x are at the boundary, the last triple search was a full one, and a specific stopping criterion (e.g., small approximated gradient) is met, stop.

4. **Boundary Handling** If some components of x are at the boundary and the stopping criterion is met, perform line searches along these boundary coordinates. If no improvement in function value is achieved, stop.

5. **Adaptive Triple Search**

   - If $|r - 1| > 0.25$ (indicating poor quadratic model prediction) or the stopping criterion in Step 3 was met, perform a full triple search.
   - Otherwise (indicating good quadratic model prediction), perform a diagonal triple search to save function evaluations.
   - These triple searches are only performed for coordinates where x_i is not at the boundary.
   - The points $\{x_{i,\ell}, x_{i,m}, x_{i,r}\}$ are chosen as x_i, max$(x_i - \delta, u_i)$, and min$(x_i + \delta, v_i)$ (or two neighbors at distance δ and 2δ if x_i is at the boundary), with $\delta = 3\,\varepsilon$ (machine accuracy).
   - Update x, the reduced gradient g, and the reduced Hessian G.

6. **Adaptive Box Minimization and Line Search**

   - Adjust the size of the minimization box:

$$\text{if}\, r < 0.25,\; d := \frac{d}{2}; \qquad \text{if}\, r > 0.75,\; d := 2\,d.$$

- Minimize the same quadratic function q over the adjusted box

$$\big[\max(-d,\, u-x),\ \min(d,\, v-x)\big].$$

- Perform a line search along $x + \alpha p$ (where p is the solution to the minimization problem), using $\alpha = 0$ and $\alpha = 1$.
- Recompute the ratio r and return to Step 3.

The stopping criterion for local optimization is met if the function value has not improved or if the approximated gradient is small, defined by

$$|g|^\top \max\big(|x|,\ |x_{\text{old}}|\big) \ <\ \gamma\,(f - f_0),$$

where γ is an input parameter and f_0 is the smallest function value from initialization.

3.6 Adaptive Triple Search

When moving to the coordinate-wise triple search, the algorithm distinguishes two cases based on the quality of the quadratic model (measured by r) or whether the stopping criterion was already met in Step 3:

1. If $|r-1| > 0.25$ (indicating poor quadratic model prediction), or if the stopping criterion was met in Step 3, then a *full triple search* is performed.
2. Otherwise (indicating a good quadratic model prediction), a *diagonal triple search* is performed to save function evaluations.

These triple searches are only performed for coordinates i such that x_i is not at the boundary of its interval $[u_i,\, v_i]$. The three points

$$\{\, x_i^l,\ x_i^m,\ x_i^r \,\}$$

are chosen as

$$x_i^m \ =\ x_i, \quad x_i^l \ =\ \max\big(x_i - \delta,\ u_i\big), \quad x_i^r \ =\ \min\big(x_i + \delta,\ v_i\big),$$

where $\delta = 3\,\varepsilon$ (machine accuracy). If x_i is at a boundary (u_i or v_i), then the neighbors are taken at distances δ and $2\,\delta$ from x_i, still clipped to $[u_i,\, v_i]$.

After selecting $\{x_i^l,\, x_i^m,\, x_i^r\}$, the point x, the reduced gradient g, and the reduced Hessian G are all updated based on the outcome of the triple search on coordinate i.

3.7 Adaptive Box Minimization and Line Search

- Adjust the size of the minimization box:

$$\text{if } r < 0.25,\ d := \frac{d}{2}; \qquad \text{if } r > 0.75,\ d := 2\,d.$$

- Minimize the quadratic function q (obtained in Step 2) over the adjusted box
$$\big[\max(-d,\, u-x),\ \min(d,\, v-x)\big].$$
- Perform a line search along $x + \alpha p$ (where p is the minimizer of q over that box), using $\alpha = 0$ and $\alpha = 1$.
- Recompute the ratio r and return to Step 3.

The stopping criterion for local optimization is met if
$$|g|^\top \max\big(|x|,\ |x_{\text{old}}|\big) \ < \ \gamma\,(f - f_0),$$
where γ is an input parameter and f_0 is the smallest function value from initialization.

3.8 The "Shopping Basket" Mechanism

The "shopping basket" is a collection of "useful" points—specifically, the base points and function values of boxes that have reached the maximum level (s_{max}). Local searches are initiated from these candidate points at the end of each sweep. The purpose of the shopping basket and its associated procedure is to avoid redundant local optimizations and to identify truly new local optima.

3.8.1 Process for Managing Shopping Basket Candidates

1. Sort Candidates All base points of s_{max}-level boxes collected in the current sweep are sorted in ascending order of their function values.

2. Check for Redundancy For each candidate point x in sorted order:

- If a local search has already been performed from x, skip to the next candidate.
- Otherwise, for each point w already in the shopping basket with $f(w) \le f(x)$, perform the following monotonicity checks:
  1. Evaluate the function at
  $$x' \ = \ x + \tfrac{3}{1}\big(w - x\big) \ = \ x + 3\,(w - x).$$
  If $f(x') > f(x)$, then x is not in the basin of attraction of w. Move to the next w.
  2. Evaluate the function at
  $$x'' \ = \ x + \tfrac{3}{2}\big(w - x\big) \ = \ x + \tfrac{3}{2}\,(w - x).$$
     - If $f(x'') > \max\big(f(x'),\, f(w)\big)$ and $f(x') < f(x)$, then set $x := x'$ and move to the next w.
     - If $\min\big(f(x'),\, f(x'')\big) < f(w)$, then all four points $\{\,x,\, x',\, x'',\, w\}$ are in the same valley. Replace x by whichever of x' or x'' has the smaller function value, and move to the next w.
     - Otherwise (if the function values are monotone between x and w), discard x for local search, and move to the next candidate x.

3. Initiate Local Search If a candidate x (or its updated version after the monotonicity checks) survives all redundancy checks, initiate a local search from x.

4. Add to Shopping Basket After the local search completes, apply the monotonicity checks described above to the local minimizer. Only if the result of the local search yields a truly new point (not in the basin of any existing basket point) is it added to the shopping basket.

This procedure ensures that local searches are only performed from points likely to produce new or improved local minimizers, thus avoiding unnecessary function evaluations and improving overall efficiency.

3.9 Theoretical Convergence and Practical Performance

3.9.1 Theoretical Convergence Properties

MCS without Local Search If the maximum number of levels s_{max} tends to infinity, MCS is guaranteed to converge to the globally optimal function value, provided the objective function f is continuous in a neighborhood of a global minimizer. In this setting:

- The set of base points sampled by MCS becomes dense in the search domain $[u, v]$. Hence, for any point $x \in [u, v]$ and any $\varepsilon > 0$, there exists some future sweep in which MCS samples a point within distance ε of x.
- Each level eventually becomes empty: in each sweep, one box leaves the lowest nonempty level and no new box is added at that level. Since the "splitting by rank" rule ensures levels move upward by at most two during splitting, any box with a sufficiently high level is guaranteed to be split repeatedly along each coordinate direction.
- Safeguards against overly narrow splits prevent infinite refinement without sampling density, so boxes containing any given x shrink to arbitrarily small size after enough splits.

Formally, let $x^* \in [u, v]$ be a global minimizer and assume f is continuous around x^*. Then for any $\varepsilon > 0$, there exists a level s_0 such that for all $s_{\text{max}} \geq s_0$, MCS eventually samples a base point x with

$$f(x) < f(x^*) + \varepsilon.$$

The worst-case upper bound on the number of sweeps required to sample within ε of x^* is

$$\frac{p^{s_0-1} - 1}{p-1},$$

where p is an upper bound on the number of boxes generated by a single splitting step. This exponential bound is consistent with the NP-hard nature of global optimization.

MCS with Local Search Under the same assumptions as above, plus the following idealized conditions:

1. The local search algorithm always converges to a local minimizer in a finite number of steps when started within that minimizer's basin of attraction.
2. The function values at any non-global local minimizer y satisfy

$$f(y) > f(x^*) + \varepsilon, \qquad \text{for some } \varepsilon > 0.$$

3. There is no accumulation of non-global local minimizers arbitrarily close to x^*, and no global minimizer lies "at infinity."

Then there exist finite integers L and S such that for any $s_{\text{max}} \geq L$, MCS with local search finds a global minimizer after at most S sweeps. In other words, convergence occurs in a finite number of sweeps under these conditions, which is strictly stronger than the "eventual" convergence guarantee when local search is omitted.

3.9.2 Practical Performance: Advantages and Disadvantages

Literature on benchmarking the MCS against other global optimization methods on large test suites. The main findings are summarized below.

Advantages

Problem-Specific Strengths:

- *Convex smooth problems:* Solved 76% within tolerance (2nd best); achieved best solution on 76% of instances
- *Nonconvex smooth (limited budget):* Found global solutions on >70% up to 800 evaluations; best solution on 71% of tests
- *Nonconvex nonsmooth:* Solved 40% of cases despite high difficulty
- *Low dimensions:* Solved 90% (1-2 vars) and 78% (3-9 vars) of problems

Operational Benefits:

- *Starting point improvement:* 100% improvement rate for convex smooth problems at very low τ (10^{-6}); consistently top-performing across all problem types
- *Low-budget efficiency:* High performance for nonconvex smooth problems with small evaluation budgets
- *Minimal solver inclusion:* Part of minimal solver set for nonconvex smooth problems (≤ 500 evaluations), solving 66%
- *Parameter robustness:* Performs well with identical tuning across diverse problems

Disadvantages

Scalability Issues:

- *Dimension sensitivity:* Success rates drop significantly with dimension—<64% (10-30 vars), <28% (31-300 vars)
- *Loves constraints:* Performs poorly in an unconstraints case ($u \in (-\infty, \ldots, -\infty), v \in (+\infty, \ldots, +\infty)$).

- *Computational intensity:* Often hits 10-minute CPU limit for >30 variables in benchmarks.
- *Rotation sensitivity:* approximately 9% performance drop on rotated separable problems; substantial degradation on nonseparable problems in higher dimensions

Algorithmic Limitations:

- *Fixed initialization:* Always starts from box center $[u, v]$, ignoring user-specified initial guesses. This is an issue of my implementation and can be easily foxed in the future.
- *Hard parallelization:* Modern CPU and GPU (especially) focuses on many parallel computations. This algorithm turned out to be not easy to parallelize and therefore leverage full potential of modern hardware.
- *Local minima traps:* May miss global minimum for $n \geq 4$ unconstrained problems when encountering low-lying local minimizers.
- *Multimodal challenges:* Can fail on hard problems with many local minima within fixed high function call limits
- *Implementation overhead:* MATLAB version suffers from non-vectorized loops; Rust implementation expected to reduce overhead drastically

4 Implementation and Comparative Results

4.1 Limitations of Existing Implementations (MATLAB/Python)

While the Multilevel Coordinate Search (MCS) algorithm offers significant mathematical robustness and algorithmic advantages for derivative-free global optimization, its practical utility in financial environments is also heavily dependent on its underlying implementation. The choice of programming language plays a pivotal role in transforming a theoretically sound algorithm into a production-ready solution.

Existing open-source implementations of the MCS algorithm in MATLAB and Python3, though functional for academic exploration and prototyping, often fall short of meeting the performance demands of modern financial applications. These interpreted languages inherently introduce performance bottlenecks due to runtime overhead, which significantly limits execution speed for computationally intensive tasks.

Furthermore, these languages typically exhibit inefficient memory management and type conversions. Interpreted languages like Python rely on automatic garbage collection, which, despite advancements, can introduce unpredictable pauses and higher memory consumption due to its runtime overhead. Similarly, MATLAB's memory management may prove less efficient for very large-scale, computational heavy optimization problems common in finance.

Beyond performance, these existing solutions are "not production ready" due to suboptimal code quality, poor documentation, and the use of obsolete techniques. These limitations often stem from the inherent characteristics of the chosen languages, coupled with design choices that may not prioritize production-grade robustness or maintainability.

Notably, the Python implementation, even after preliminary corrections (e.g., changing `np.Inf` to `np.inf`), proved non-functional in my tests due to persistent runtime errors. Consequently, it was excluded from the direct performance comparison. The primary errors encountered were:

1. A `TypeError` within the `lsnew.py` module:

```
Traceback (most recent call last):
File "XXXX\runmcs.py", line 27, in <module>
xbest,fbest,xmin,fmi,ncall,ncloc,flag = mcs(fcn,u,v,smax,nf,stop,iinit,local,gamma,hess)
File "XXXX\mcs.py", line 310, in mcs
xmin1,fmi1,nc,flag,nsweep,nsweepbest = lsearch(fcn,x,f1,f0min,u,v,nf-ncall,stop,local,gamma,hess,nsweep,nsweepbest)
File "XXXX\lsearch.py", line 192, in lsearch
alist,flist,nfls = gls(fcn,u,v,xmin,p,alist,flist,nloc,small,smaxls)
File "XXXX\gls.py", line 72, in gls
alist,flist,abest,fbest,fmed,up,down,monotone,minima,nmin,unitlen,s,alp,fac = lspar(func,nloc,small,sinit,short,x,p,
      alist,flist,amin,amax,alp,abest,fbest,fmed,up,down,monotone,minima,nmin,unitlen,s)
File "XXXX\lspar.py", line 45, in lspar
alist,flist,alp,fac = lsnew(func,nloc,small,sinit,short,x,p,s,alist,flist,amin,amax,alp,abest,fmed,unitlen)
File "XXXX\lsnew.py", line 60, in lsnew
dist = [max(i,j,k) for i,j,k in zip([i-abest for i in alist[1:s]], [abest-i for i in alist[0:s-1]], (unitlen*np.ones(
      s-1)).tolist())]
TypeError: '>' not supported between instances of 'complex' and 'float'
```

2. An `IndexError` within the `split_func.py` module:

```
Traceback (most recent call last):
File "XXXX\runmcs.py", line 27, in <module>
xbest,fbest,xmin,fmi,ncall,ncloc,flag = mcs(fcn,u,v,smax,nf,stop,iinit,local,gamma,hess)
File "XXXX\mcs.py", line 244, in mcs
xbest,fbest,xmin,fmi,ipar,level,ichild,f,flag,ncall1,record,nboxes,nbasket,nsweepbest,nsweep = split(fcn,i,s,smax,par
      ,n0,u,v,x,y,x1,x2,z[:,par],xmin,fmi,ipar,level,ichild,f,xbest,fbest,stop,prt, record,nboxes,nbasket,nsweepbest,
      nsweep)
File "XXXX\split_func.py", line 190, in split
```

```
ipar[nboxes],level[nboxes],ichild[nboxes],f[0,nboxes] = genbox(par,s+1,3,f[1,par])
IndexError: index 10000 is out of bounds for axis 0 with size 10000
```

These errors highlight potential deep-seated issues within the Python codebase. In fact, I have found several bugs in the Python implementation while building my Rust one.

4.2 Advantages of Rust for Financial Computing

Rust offers a compelling set of features that directly address the limitations of interpreted languages and the demanding requirements of quantitative finance.

Rust is a systems programming language renowned for its unparalleled performance and robust safety guarantees. A core tenet of Rust's design is zero-cost abstractions, meaning that high-level programming constructs do not incur any runtime overhead, leading to execution speeds comparable to highly optimized C or C++ code. This predictability, safety and high performance is absolutely crucial for building reliable optimizer.

Rust achieves its memory safety without garbage collection through a unique compile-time enforced ownership model, complemented by concepts like borrowing and lifetimes. This innovative approach eliminates entire classes of memory-related bugs (e.g., null pointer dereferencing, use-after-free errors) before the program even runs, all without the need for a garbage collector. The absence of garbage collection pauses ensures highly predictable performance and significantly reduced memory overhead, a critical advantage over languages that rely on runtime garbage collection.

Rust's unique compile-time guaranteed memory safety directly mitigates a common and insidious class of subtle, hard-to-debug errors prevalent in high-performance, concurrent financial systems, thereby enhancing overall system reliability and trustworthiness beyond mere speed.

Rust's design philosophy, which prioritizes fine-grained control over system resources, raw performance, and robust security, makes it an ideal choice for systems programming within the financial sector. It is increasingly being adopted as the language of choice for building sophisticated algorithms for quantitative analysis, developing secure and scalable trading platforms, and constructing other high-performance financial infrastructure (especially in a crypto world).

Rust ensures that if code compiles, it is free from certain classes of errors—an essential guarantee for global optimizers that may run for hours and otherwise risk crashing or entering an invalid state. Rust prevents such failures by design.

4.3 My Own Implementation of the Algorithm: `Rust_MCS`

To overcome the limitations of existing implementations and to create a production-ready solver, I developed `Rust_MCS`, a new implementation of the Multilevel Coordinate Search algorithm written entirely in Rust. The complete source code is available in the GitHub repository: https://github.com/SergeiGL/Rust_MCS. This implementation not only meets the high-performance requirements of financial mathematics but also significantly improves code maintainability and correctness, largely due to Rust's advanced type system.

Rust's static and strong type system is a cornerstone of its ability to outperform dynamically typed languages like Python and MATLAB. Unlike these languages, where type checking occurs at runtime, Rust verifies all types at compile time. This eliminates a whole class of runtime errors and removes the performance overhead associated with dynamic type checking. ”zero-cost abstractions” like enums enhance code readability and maintainability and are compiled down to highly efficient machine code without any runtime penalty. This is in stark contrast to MATLAB and Python, where abstractions often introduce significant performance costs. The result is a codebase that is both easier to maintain and faster in execution.

Justification of Crates

The `Rust_MCS` leverages carefully selected external libraries, or ”crates”:

- **`nalgebra`**: This crate is fundamental to the implementation. The MCS algorithm is inherently mathematical, relying heavily on linear algebra (vector operations, LU decomposition etc). `nalgebra` provides a rich, efficient, and ergonomic API for these operations. By using `nalgebra`, `Rust_MCS` can perform complex mathematical computations with low-level performance while maintaining clear and readable syntax. Also nalgebra is the fastest crate (faster than the ndarray) to deal with small dimensional matrices (ideal fit: recall that MCS performance in high dimensional problems is very limited)
- **`criterion`**: To empirically validate the performance claims, benchmarking is essential. `criterion` is a powerful benchmarking framework for Rust that provides statistically rigorous analysis of performance. It was used as a development dependency to systematically compare the execution speed of `Rust_MCS` against other implementations and to guide optimization efforts, ensuring that the final implementation is verifiably fast.
- **`approx`**: When dealing with floating-point arithmetic, direct comparison for equality is fraught with peril due to precision issues. The `approx` crate provides macros for asserting the approximate equality of floating-point numbers. This was crucial in the development-testing phase to verify the functional correctness of `Rust_MCS` by comparing its outputs against the results from the trusted MATLAB implementation.

API Design and Constant Generics

The public API of `Rust_MCS`:

```
let (xbest, fbest, xmin, fmi, ncall, ncloc, ExitFlag) =
mcs::<N>(func, &u, &v, nsweeps, nf, local, gamma, smax, &hess).unwrap();
```

A key feature of this API is the use of constant generics (denoted by `<N>` in this example). This allows dimension of the optimization problem, `N`, to be specified as a compile-time constant. The Rust compiler can then leverage this information to perform even more powerful optimizations. For instance, loops over the dimensions of input vectors can be unrolled, and memory for small, fixed-size vectors and matrices can be allocated on the stack instead of the heap. This avoids the overhead of dynamic memory allocation and results in a significant performance boost (especially in working with complex matrix operations as here). As discussed above, Rust's strong type system is already a huge boost, but constant generics usage gives Rust even greater edge over Python and MATLAB, where array (vectors and matrices) sizes are determined at runtime.

Functional Correctness

A primary goal for `Rust_MCS` was to ensure its results are reliable. To this end, the implementation was rigorously tested. Rust_MCS includes ≈ 400 automated tests (utilizes cargo test) with $\approx 90\%$ code coverage. The ”ground truth” values for Rust tests were taken from the original MATLAB implementation. This verifies that my code in line with the original Matlab version. High coverage minimizes hidden bugs and unexpected behavior, ensuring robustness and stability in production. Moreover, these tests serve as a safety net for future development and refactoring, allowing developers to optimize performance and introduce new features confidently while not breaking the existing functionality. The fact that Rust_MCS passes all tests demonstrates the functional correctness of my implementation.

Performance Monitoring

To support computationally intensive tasks, Rust_MCS employs Rust's built-in benchmarking tool (criterion) along with flamegraphs. Benchmarks provide precise, quantitative (multiple passes -> p-value) measurements of execution speed, enabling the detection of regressions and the assessment of performance improvements. Flamegraphs visualize CPU usage across the call stack, highlighting hotspots where optimization yields the greatest benefit. While the benchmarking tool answers the question ”How slow we are”, the profiling tool tries to identify ”Why are we so slow”.

4.4 Performance Benchmark: MATLAB vs. Rust

To empirically validate the performance benefits of Rust for the MCS algorithm, a series of benchmark tests were conducted comparing a Rust implementation against the established MATLAB version.

Test Configuration:

- **CPU:** 12th Gen Intel(R) Core(TM) i7-12700KF
- **RAM:** 32 GB DDR5 @ 6000 MHz (2 slots)
- **Operating System:** Windows 11 Pro, Version 24H2 (Build 26100.4202), 64-bit
- **MATLAB Version:** R2024a
- **Rust Version:** rustc 1.87.0 (17067e9ac 2025-05-09), stable-x86_64-pc-windows-msvc toolchain

MATLAB Code Used for Testing:

```
clearvars;
clear global;

path(path,'jones'); % Assumes 'jones' folder with MCS code is in path
fcn = "feval";      % Function evaluation method
data = "hm6";       % Test problem identifier (Hartmann 6-dimensional function)
prt = 0;            % Print level (0 = no output)
iinit = 0;          % Initialization method (Simple initialization list)
u = [0; 0; 0; 0; 0; 0]; % Lower bounds for variables
v = [1; 1; 1; 1; 1; 1]; % Upper bounds for variables
% smax, nf, stop, local, gamma are varied in the tests below
hess = ones(6,6);  % Placeholder Hessian matrix for hm6 (6x6)

% Example of a single run's parameter setup:
% smax = 150;      % Maximum number of levels
% nf = 1000000;    % Maximum number of function evaluations
% stop = [2000];   % nsweeps: Number of sweeps for stopping criterion
% local = 2000;    % Number of points for local search
% gamma= 2e-16;    % Parameter for stopping criterion

format long g;
tic;
[xbest,fbest,xmin,fmi,ncall,ncloc,flag] = mcs(fcn,data,u,v,prt,smax,nf,stop,iinit,local,gamma,hess);
time_ms = toc * 1000;
fprintf('MCS execution time: %.2f ms\n', time_ms);
```

Note (MATLAB): In newer versions of MATLAB, the function name `split.m` (part of the original MCS distribution) is reserved. To run the MCS algorithm, it is necessary to rename this file and its corresponding function calls to `split_.m` (or a similar non-reserved name).

Rust Code Used for Testing:

My Rust implementation of the MCS algorithm can be downloaded here [43].

The default `cargo bench` configuration. The exact Rust benchmark code and further technical specifications can be found on the project's GitHub page.

nsweeps	nf	local	γ	smax	MATLAB (ms)	Rust (ms)	**Speedup**
1000	1 mil.	100	2.00×10^{-10}	100	4 599.07	131.46	**34.98x**
1000	1 mil.	100	2.00×10^{-10}	200	68 086.88	376.45	**180.87x**
1000	1 mil.	100	2.00×10^{-10}	300	237 058.16	630.58	**375.94x**
1000	1 mil.	2000	2.00×10^{-14}	150	30 906.09	249.46	**123.89x**
2000	1 mil.	2000	2.00×10^{-14}	150	128 364.08	834.44	**153.83x**
1000	1 mil.	10	2.00×10^{-14}	500	697 629.47	1 323.60	**527.07x**

Table 2: Comparative timing results for MCS in MATLAB vs. Rust for the HM6 function

Comparative Results: the data demonstrates the substantial performance advantage of the Rust implementation. Across all tested configurations, Rust consistently outperformed MATLAB by orders of magnitude, with speedups exceeding 527x. This dramatic improvement underscores Rust's suitability for computationally intensive financial modeling tasks. Moreover, the algorithm guarantees convergence if and only if smax $\to \infty$; however, when smax > 1000, MATLAB tests can take hours to run, while Rust easily handles smax > 3000 within seconds. This indicates that my Rust_MCS implementation can handle inputs that were previously practically impossible to compute!

5 Unlocking Accessibility: Rust_MCS Ecosystem

Beyond the core algorithm, Rust_MCS extends its reach through supportive repositories designed to lower entry barriers (both for programmers and non-programmers).

Given that the majority of core financial quantitative libraries are implemented in C++, effective C++ integration was a key consideration for Rust_MCS. This, along with other accessibility features, is discussed in this chapter.

5.1 Rust_MCS_Cpp

The `Rust_MCS_Cpp` [45] repository demonstrates how to integrate Rust_MCS into existing C++ projects via a foreign function interface (FFI). This bridge preserves performance and safety advantages without requiring undertanding of Rust concepts. The Rust_MCS_Cpp repository allows C++ developers to incrementally adopt Rust_MCS within their established infrastructures and potentially improve results of their research papers.

5.2 Rust_MCS_Python

The `Rust_MCS_Python` [46] repository exposes Rust_MCS functionality to python3 users. By supporting zero-copy NumPy arrays and well-documented interoperability, this wrapper lets Python developers leverage Rust_MCS's speed and safety in just one import statement! Thousands of projects written in python can now integrate the optimizer into an existing codebase without any Rust knowledge. As a result, there is an easy to use way to call the Rust optimizer from Python3.

5.3 Rust_MCS_Web

The `Rust_MCS_Web` [44] repository offers a pretty in-browser interface that requires only Docker for setup. By packaging the optimizer and its dependencies into a container, users can launch and interact with Rust_MCS directly in their web browser—no programming knowledge or Rust installation needed - the process of installing Rust, retrieving the latest Rust_MCS version, compiling and presenting results is automated. Important note is that everything runs locally, on your PC. This repository approach democratizes access, enabling researchers, analysts, and students to experiment with the optimizer in a self-contained environment.

6 Economic Applications

6.1 Portfolio Construction using MCS for Sharpe Ratio Optimization

6.1.1 Problem Formulation

The goal is to construct a portfolio of 206 MOEX stocks (adjusted for dividends and splits) that maximizes the Sharpe ratio

$$S = \frac{E[R_p] - R_f}{\sigma_p},$$

where $E[R_p]$ is the portfolio's expected return, R_f is the risk-free rate, and σ_p is portfolio volatility. Realistic constraints are imposed:

- **Total Weight:** $\sum_{i=1}^{206} w_i = 1$.
- **Individual Weights:** For each stock i, either $w_i = 0$ or $0.01 \le w_i \le 0.10$.

These ensure full investment, liquidity considerations, ease of trading (hard to buy and maintain weight that is too close to 0), diversification and regulatory compliance, while promoting sparsity.

6.1.2 Methodological Approaches

1. Python Two-Step procedure (SciPy SLSQP)

1. *Stage One:* Solve
$$\max_{0 \le w_i \le 0.10} S(w_1, \ldots, w_{206})$$
under continuous bounds $[0, 0.10]$. Drop (set w_i = 0) stocks with $w_i < 0.01$. Record the subset $\mathcal{S}_1$ with $w_i \ge 0.01$ and their cumulative weight.
2. *Stage Two:* Restrict optimization to $\mathcal{S}_1$. Enforce $0.01 \le w_i \le 0.10$. Other stocks' w remain zero.

2. Rust Approach (Rust_MCS)

Define a single objective that incorporates all constraints via an iterative adjustment of a raw weight vector $\widetilde{w} \in [0, 0.10]^{206}$:

1. **Thresholding:** If $\widetilde{w}_i < 0.01$, set $w_i = 0$.
2. **Normalization:** Rescale nonzero w_i so $\sum w_i = 1$.
3. **Capping:** If any $w_i > 0.10$, cap at 0.10 and redistribute the excess proportionally among other nonzero weights.
4. **Iterate:** Repeat the above until each w_i is either 0 or in $[0.01, 0.10]$.

Note: this function does not work for any weights. However, I got assertions in both versions, so I am sure it works just fine in this case.

Compute the Sharpe ratio on the adjusted w and pass this black-box function to the Rust_MCS global optimizer. The MCS search efficiently explores the space, inherently enforcing sparsity and bounds in one pass.

3. Python Replica of Rust

Translate the Rust weight-adjustment logic into Python. Perform a single-stage optimization on the replicated objective (as the constraint is now trivial).

6.1.3 Results and Discussion

Approach	**Achieved Sharpe Ratio**
Rust_MCS	0.1256
Python Two-Step (SLSQP)	0.1253
Python (Rust Replica)	0.0599

Rust_MCS attains the highest Sharpe ratio (≈0.1256), slightly above the Python two-step (≈0.1253). In contrast, the Python replica with a local optimizer converges to a suboptimal solution (≈0.0599). In fact the optimzier cannot go any further than the initial point, illustrating that a powerful global search is essential when the objective includes complex, non-smooth adjustments. This example confirms that Rust_MCS delivers both computational efficiency and superior economic performance in constrained Sharpe ratio optimization.

Full code for replication of my results can be found here: [47]

6.2 Model Calibration

In this section, two canonical calibration problems in financial mathematics will be tested: determining the implied volatility in the Black-Scholes model and calibrating the more complex Heston stochastic volatility model. The performance and results of the Rust_MCS global optimizer will be compared against standard, commonly used gradient-based local optimizers available in Python's SciPy library. The market data for a single European call option on JPMorgan Chase & Co. (Ticker: JPM), expiring on June 13, 2025, is used for both case studies.

6.2.1 Black-Scholes Model Calibration

The Black-Scholes model provides a foundational framework for option pricing. Its single unobservable parameter is the volatility of the underlying asset, σ. Calibrating the model to a market price involves finding the *implied volatility*—the value of σ that equates the theoretical Black-Scholes price to the observed market price.

The optimization problem is to minimize the squared error between the model price and the market price:

$$\min_{\sigma>0} \left(\mathrm{BS}(S_0, K, T, r, \sigma) - C_{\text{market}}\right)^2$$

where BS$(\cdot)$ is the Black-Scholes formula for a European call option. For a single liquid option, this objective function is typically well-behaved and unimodal, making it amenable to both local and global optimization methods.

I performed the calibration using two distinct minimizers:

1. **Python Minimizer:** SciPy's 'minimize_scalar' function, a robust local search method well-suited for one-dimensional problems.
2. **Rust MCS Minimizer:** My Rust_MCS implementation.

Both optimizers successfully converged to the same implied volatility:

$$\sigma_{\text{implied}} = \mathbf{0.2063} \quad (20.63\%)$$

This result serves as an important validation, confirming that for a simple, well-behaved problem, the Rust_MCS implementation correctly identifies the global minimum, which coincides with the solution found by the specialized local optimizer.

Table 3: Parameters and Result for Implied Volatility Calculation

Input Parameters	
Parameter	Value
Underlying Price (S)	\$261.95
Strike Price (K)	\$262.50
Time to Maturity (T)	0.0192 years
Risk-Free Rate (r)	5.00%
Market Call Price (C_{market})	\$2.84
Calculation Result	
Implied Volatility (σ_{implied})	**20.63%**

6.2.2 Heston Model Calibration

The Heston model extends Black-Scholes by treating volatility as a stochastic process that mean-reverts. It is defined by the following system of stochastic differential equations:

$$\begin{aligned} dS_t &= rS_t dt + \sqrt{\nu_t} S_t dW_t^1 \\ d\nu_t &= \kappa(\theta - \nu_t)dt + \xi\sqrt{\nu_t} dW_t^2 \end{aligned}$$

where the Wiener processes are correlated such that $E[dW_t^1 dW_t^2] = \rho dt$. Calibrating the Heston model requires estimating five parameters:

- ν_0: The initial variance.
- κ: The speed of mean reversion of the variance.
- θ: The long-run average variance.
- ξ: The volatility of variance ("vol of vol").
- ρ: The correlation between the asset's and variance's stochastic processes.

The objective is to find the parameter vector $\vec{p} = (\nu_0, \kappa, \theta, \xi, \rho)$ that minimizes the squared relative error, subject to box constraints and the Feller condition ($2\kappa\theta > \xi^2$), which ensures the variance process remains non-negative.

$$\min_{\vec{p}\in\mathcal{D}} \log\left[\left(\frac{\text{Heston}(\vec{p}) - C_{\text{market}}}{C_{\text{market}}}\right)^2\right]$$

This calibration is a far more challenging, non-convex optimization problem. When calibrating to a single instrument, the problem is ill-posed, meaning multiple parameter sets can yield the same model price.

I compared SciPy's 'L-BFGS-B' (a quasi-Newton local optimizer) with my 'Rust_MCS' global optimizer. The calibration objective was to minimize the log of squared relative error between the Heston model price and the observed market price of a European call option.

Python L-BFGS-B Optimizer

The `L-BFGS-B` optimizer was run three times with perturbed initial guesses to mitigate the risk of converging to a local minimum. The results of these runs were as follows:

- Run 1: Objective function value of -33.2865, converged successfully.
- Run 2: Objective function value of -37.2651, abnormal termination.
- Run 3: Objective function value of -35.3837, abnormal termination.

The parameters from the Run 2 were selected for analysis, as per the script's logic. Despite the "abnormal termination" message, which can occur for various numerical reasons, the optimizer found a parameter set that replicates the market price with the highest precision. The calibrated parameters are detailed in Table 4.

Rust MCS Optimizer

The global optimizer `Rust_MCS` was executed ones as the optimizer is global. The algorithm terminated after reaching the maximum number of sweeps (`StopNsweepsExceeded`). It consistently identified the same global minimum, yielding an objective function value of -43.5493. The resulting parameters are presented in Table 4.

Table 4: Comparison of Heston model parameters calibrated using L-BFGS-B and Rust_MCS.

Parameter	**L-BFGS-B (Python)**	**Rust_MCS**
Objective Function Value	-37.2651	-43.5493
Calibrated Parameters		
Initial Variance (v_0)	0.044977	0.005000
Mean Reversion Speed (κ)	2.9547	7.0093
Long-term Variance (θ)	0.035879	0.376250
Volatility of Volatility (σ_v)	0.2399	0.5890
Correlation (ρ)	0.2010	-0.1743
Price Comparison		
Market Price	\$4.42	\$4.42
Heston Model Price	\$4.42	\$4.42
Absolute Error	\$0.0000	\$0.0000
Relative Error	0.00%	0.00%

Interpretation of Results

Although both parameter sets can accurately reproduce the input market price, they imply significantly different market dynamics, as summarized in Table 5. The Rust_MCS optimizer found a solution with a lower objective function value, suggesting a better fit to the objective function's landscape.

Table 5: Interpretation of calibrated Heston model dynamics.

Metric	**L-BFGS-B (Python)**	**Rust_MCS**
Initial Volatility ($\sqrt{v_0}$)	21.2%	7.1%
Long-term Volatility ($\sqrt{\theta}$)	18.9%	61.3%
Volatility Regime	High vol, expected to decrease	Low vol, expected to increase
Mean Reversion Half-life	0.23 years	0.10 years
Correlation Implication	Volatility rises with price	Leverage effect
Feller Condition ($2\kappa\theta/\sigma_v^2$)	3.68	15.20

The solution from L-BFGS-B indicates a high-volatility environment (21.2%) expected to revert downwards to a long-term average of 18.9%. The positive correlation ($\rho = 0.2010$) is somewhat wrong (atypical) for equity markets as the leverage effect suggests the opposite. In contrast, the Rust_MCS solution implies a state of very low current volatility (7.1%) with an expected reversion to a much higher long-term level of 61.3%. The negative correlation ($\rho = -0.1743$) aligns with the well-known leverage effect in equities, where volatility tends to increase as prices fall. Furthermore, the Feller condition ($2\kappa\theta > \sigma_v^2$) is strongly satisfied in both cases, ensuring a non-negative variance process, though it is significantly higher for the Rust_MCS parameters. The substantial difference in the objective function values (-37.27 vs. -43.55) suggests that the Rust_MCS optimizer located a more profound minimum, avoiding a local optimum that constrained the L-BFGS-B search. And this is not a coincidence: Rust_MCS is designed to identify multiple basins of attraction, and in my tests, it was capable of finding the same solution as L-BFGS-B in addition to other, deeper minima, making it extremely suitable for model calibration.

It is worth to emphasize that parameters found by Rust_MCS (although maybe a bit strange) are mathematically-valid.

The Rust_MCS optimizer showed its ability to efficiently navigate in complex 5d environments and its high degree of independence from the starting point, making it a great choice for model calibration!

The code: Python3 Code for Models Calibration.

7 Future work

Future work could concentrate on parallelizing the Rust_MCS algorithm to fully exploit the capabilities of modern hardware architectures, thereby significantly boosting its performance. Initially, efforts could focus on CPU-based parallelization, with a subsequent consideration for GPU implementation (as GPUs have thousands of cores VS tens on CPU).

Additionally, the original repository contains several ”TODO” comments. These indicate areas that, while currently functional, have potential for further optimization and refinement.

Furthermore, a promising avenue for future research involves integrating this Rust_MCS implementation with existing financial models. For instance, the research presented in ”Stochastic Mean-Reverting Trend (SMART) Model in Quantitative Finance” [48] utilized an MCS method implemented in NAG for calibrating a stochastic trend model (distinct from volatility models like Heston) in the context of algorithmic trading. A valuable extension would be to replace the NAG-based MCS with the present Rust implementation, leveraging the Rust_MCS C++ interface described above [45]. The primary benefit of switching to Rust_MCS (through its C++ interface) is that it’s free, open-source and provide safety guarantees.

8 Conclusion

This diploma has thoroughly investigated the implementation of the Multilevel Coordinate Search (MCS) method for global optimization problems within financial mathematics. Challenges inherent in financial optimization were considered and why these characteristics often render traditional gradient-based optimization techniques ineffective or impractical, underscoring the need for robust derivative-free global optimizer.

A comprehensive literature review contextualized MCS within the broader landscape of derivative-free global optimization algorithms, comparing its deterministic, Lipschitzian-based approach with other deterministic, stochastic, and hybrid methods. This review revealed that few algorithms possess the combination of unique requirements of financial applications: true derivative-free operation, robustness to complex financial landscapes and practical computational efficiency. Proprietary solutions like the NAG, which includes an MCS implementation, are often closed-source and offered at a significant price tag. The analysis confirmed that no single algorithm is universally superior, aligning with the ”No Free Lunch” theorems, and emphasized the pragmatic adoption of hybrid approaches or thoroughly vetted commercial solutions in high-stakes financial environments.

A significant contribution of this diploma is the development and benchmarking of a new MCS implementation in Rust. My empirical results demonstrated a huge performance advantage of the Rust implementation over its MATLAB and Python counterparts, achieving over 527x speedup in some cases, while carefully insuring the functional correctness. The Rust_MCS ecosystem, including web, C++ (very important as most of financial software is in C++) and Python connectors, significantly enhances the algorithm's accessibility and ease of integration into diverse computational environments, from academic research to industrial production systems.

Finally, I showcased the real world economic applicability of the Rust_MCS framework through two case studies: Sharpe ratio maximization in portfolio construction and models' calibration (Black-Scholes and Heston).

The Rust_MCS is a valuable tool for quantitative finance, capable of tackling challenged common in finance with superior efficiency and reliability. Rust_MCS could be a top choice for many scientific and industry-based financial tasks!

References

[1] Derivative-Free Global Optimization Algorithms: Bayesian Methodand Lipschitzian Approaches. https://arxiv.org/abs/1904.09365

[2] NLopt algorithms - NLopt Documentation. https://nlopt.readthedocs.io/en/latest/NLopt_Algorithms/

[3] Algorithms for Noisy Problems in Gas Transmission Pipeline Optimization - ResearchGate. https://www.researchgate.net/publication/226623769_Algorithms_for_Noisy_Problems_in_Gas_Transmission_Pipeline_Optimization

[4] Sensitivity analysis - Wikipedia. https://en.wikipedia.org/wiki/Sensitivity_analysis

[5] Linas Stripinis, Jakub Kudela: Benchmarking Derivative-Free Global Optimization Algorithms Under Limited Dimensions and Large Evaluation Budgets https://ieeexplore.ieee.org/document/10477219

[6] Luis Miguel Rios, Nikolaos V. Sahinidis: Derivative-free optimization: a review of algorithms and comparison of software implementations https://link.springer.com/content/pdf/10.1007/s10898-012-9951-y.pdf

[7] stogo: Stochastic Global Optimization in nloptr: R Interface to NLopt - rdrr.io. https://rdrr.io/cran/nloptr/man/stogo.html

[8] stogo function - Stochastic Global Optimization - RDocumentation. https://www.rdocumentation.org/packages/nloptr/versions/2.0.3/topics/stogo

[9] nlopt 0.8.1 - Docs.rs. https://docs.rs/crate/nlopt/latest/source/src/lib.rs

[10] Particle Swarm Optimization Approach for the Portfolio Optimization Problem - Science UTM. https://science.utm.my/procscimath/wp-content/uploads/sites/605/2023/10/107-111-Ooi-Chong-Eu_Nur-Arina-Bazilah-Aziz-.pdf

[11] (PDF) Optimizing Stock Portfolio Using the Particle Swarm Optimization Algorithm and Assessing PSO and Other Algorithms - ResearchGate. https://www.researchgate.net/publication/386494169_Optimizing_Stock_Portfolio_Using_the_Particle_Swarm_Optimization_Algorithm_and_Assessing_PSO_and_Other_Algorithms

[12] Empirical Study of Particle Swarm Optimization - ResearchGate. https://www.researchgate.net/publication/3810335_Empirical_Study_of_Particle_Swarm_Optimization

[13] Convergence analysis of particle swarm optimization algorithms for different constriction factors - Frontiers. https://www.frontiersin.org/journals/applied-mathematics-and-statistics/articles/10.3389/fams.2024.1304268/full

[14] crs2lm: Controlled Random Search in nloptr: R Interface to NLopt - rdrr.io. https://rdrr.io/cran/nloptr/man/crs2lm.html

[15] NLopt algorithms - NLopt Documentation. https://nlopt.readthedocs.io/en/stable/NLopt_Algorithms/

[16] Some variants of the controlled random search algorithm for global optimization - WIReDSpace. https://wiredspace.wits.ac.za/bitstreams/df7775ed-bed1-4f11-bf6c-f336631659ad/download

[17] Overcoming the Curse of Dimensionality When Clustering Multivariate Volume Data - SciTePress. https://www.scitepress.org/papers/2018/65419/65419.pdf

[18] The Challenges of Clustering High Dimensional Data - College of Science and Engineering - University of Minnesota. https://www-users.cse.umn.edu/~kumar/papers/high_dim_clustering_19.pdf

[19] What are the best practices for clustering high-dimensional data? - GeeksforGeeks. https://www.geeksforgeeks.org/what-are-the-best-practices-for-clustering-high-dimensional-data/

[20] Clustering in high dimensions: distance metrics, binary vs continuous, statistical tests for number of clusters / noise points [closed] - Cross Validated. https://stats.stackexchange.com/questions/329084/clustering-in-high-dimensions-distance-metrics-binary-vs-continuous-statistic

[21] Nelder–Mead method - Wikipedia. https://en.wikipedia.org/wiki/Nelder%E2%80%93Mead_method

[22] USER'S GUIDE FOR TOMLAB/OQNLP1 https://tomopt.com/docs/TOMLAB_OQNLP.pdf

[23] A novel hybrid genetic algorithm and Nelder-Mead approach and it's application for parameter estimation. - F1000Research. https://f1000research.com/articles/13-1073

[24] Analysis and simplification of the winner of the CEC 2022 optimization competition on single objective bound constrained search. https://direct.mit.edu/evco/article-abstract/doi/10.1162/evco.a.27/130817/Analysis-and-simplification-of-the-winner-of-the?redirectedFrom=fulltext

[25] Effect of dimensionality on the Nelder–Mead simplex method - ResearchGate. https://www.researchgate.net/publication/250889865_Effect_of_dimensionality_on_the_Nelder-Mead_simplex_method

[26] Optimizers - optimagic - Read the Docs. https://optimagic.readthedocs.io/en/latest/algorithms.html

[27] Newton's method in optimization - Wikipedia. https://en.wikipedia.org/wiki/Newton%27s_method_in_optimization

[28] Clement W. Royer Michael ONeill, Stephen J. Wright: A Newton-CG Algorithm with Complexity Guarantees for Smooth Unconstrained Optimization https://optimization-online.org/wp-content/uploads/2018/03/6513.pdf

[29] Adaptive gaining-sharing knowledge-based variant algorithm with historical probability expansion and its application in escape maneuver decision making https://link.springer.com/article/10.1007/s10462-024-11096-4

[30] Nonlinear conjugate gradient method - Wikipedia. https://en.wikipedia.org/wiki/Nonlinear_conjugate_gradient_method

[31] Conjugate gradient methods - Cornell University Computational Optimization Open Textbook. https://optimization.cbe.cornell.edu/index.php?title=Conjugate_gradient_methods

[32] General reference - NLopt Documentation - Read the Docs. https://nlopt.readthedocs.io/en/stable/NLopt_Reference/

[33] Numerical Optimization (Springer Series in Operations Research and Financial Engineering) - Amazon.com. https://www.amazon.com/Numerical-Optimization-Operations-Financial-Engineering/dp/0387303030

[34] Numerical Optimization https://link.springer.com/book/10.1007/978-0-387-40065-5

[35] Software | School of Mathematics - The University of Edinburgh. https://maths.ed.ac.uk/research/data-decisions/optimization-and-operational-research/software

[36] Mixed Integer Linear Programming (MILP) Solver - nAG. https://nag.com/mixed-integer-linear-programming/

[37] NAG Numerical Library - Wikipedia. https://en.wikipedia.org/wiki/NAG_Numerical_Library

[38] nAG Library. https://nag.com/nag-library/

[39] e05 Chapter Introduction : NAG C Library, Mark 26. https://support.nag.com/numeric/cl/nagdoc_latest/html/e05/e05intro.html

[40] MCS: Global Optimization by Multilevel Coordinate Search - Arnold Neumaier. https://arnold-neumaier.at/software/mcs/

[41] e05 Chapter Introduction : NAG C Library, Mark 26. https://support.nag.com/numeric/nl/nagdoc_latest/

[42] A Stochastic Approach to Global Optimization - ResearchGate. https://www.researchgate.net/publication/38009236_A_Stochastic_Approach_to_Global_Optimization

[43] The main Rust_MCS repository written by me: https://github.com/SergeiGL/Rust_MCS

[44] Web interface for the Rust_MCS written by me: https://github.com/SergeiGL/Rust_MCS_web

[45] C++ connector for Rust_MCS written by me: https://github.com/SergeiGL/Rust_MCS_Cpp

[46] Python3 connector for Rust_MCS written by me: https://github.com/SergeiGL/Rust_MCS_Python

[47] Sharpe ration example for Rust_MCS: https://github.com/SergeiGL/Rust_MCS_Sharpe_Ratio

[48] Stochastic Mean-Reverting Trend (SMART) Model in Quantitative Finance: https://link.springer.com/article/10.1134/S1995080224601565

```python
import yfinance as yf
import numpy as np
import scipy.stats as si
from scipy.optimize import minimize_scalar
from datetime import datetime
import rust_mcs_python
import scipy.optimize as optimization
from functools import partial
from scipy.optimize import minimize
import scipy.stats as si
from scipy.integrate import quad
from scipy import stats
import warnings
warnings.filterwarnings('ignore')


TICKER = 'JPM'

RISK_FREE_RATE = 0.05 # example risk free

stock = yf.Ticker(TICKER)

underlying_price = stock.history(period='1d')['Close'].iloc[0]

options_dates = stock.options
print(options_dates)

expiry_date_str = options_dates[1]
options = stock.option_chain(expiry_date_str)
calls = options.calls

# Select a specific option contract (e.g., a near-the-money call)
# We'll find a call option with a strike price close to the current stock price
near_the_money_call = calls.iloc[(calls['strike'] - underlying_price).abs().argsort()[0]]

strike_price = near_the_money_call['strike']
market_price = near_the_money_call['lastPrice'] # The observed market price

# 4. Calculate time to expiration (T) in years
time_to_expiration = (datetime.strptime(expiry_date_str, '%Y-%m-%d') - datetime.now()).days / 365

print(f"--- Option Data for {TICKER} ---")
print(f"Underlying Price (S): ${underlying_price:.2f}")
print(f"Strike Price (K): ${strike_price:.2f}")
print(f"Market Price of Call: ${market_price:.2f}")
print(f"Time to Expiration (T): {time_to_expiration:.4f} years")
print(f"Risk-Free Rate (r): {RISK_FREE_RATE:.4f}")
print("---------------------------------")

# ('2025-06-06', '2025-06-13', '2025-06-20', '2025-06-27', '2025-07-03', '2025-07-11', '2025-07-18', '2025-08-15',
      '2025-09-19', '2025-10-17', '2025-11-21', '2025-12-19', '2026-01-16', '2026-03-20', '2026-06-18', '2026-09-18',
      '2026-12-18', '2027-01-15', '2027-12-17')
# --- Option Data for JPM ---
# Underlying Price (S): $261.95
# Strike Price (K): $262.50
# Market Price of Call: $2.84
# Time to Expiration (T): 0.0192 years
# Risk-Free Rate (r): 0.0500
# ---------------------------------


class BlackScholesModel:
"""
Implementation of the Black-Scholes model for pricing European options.
"""
def __init__(self, S, K, T, r, sigma):
self.S = S         # Underlying asset price
self.K = K         # Option strike price
self.T = T         # Time to expiration in years
self.r = r         # Risk-free interest rate
self.sigma = sigma  # Volatility of the underlying asset

def d1(self):
return (np.log(self.S / self.K) + (self.r + 0.5 * self.sigma ** 2) * self.T) / (self.sigma * np.sqrt(self.T))

def d2(self):
return self.d1() - self.sigma * np.sqrt(self.T)

def call_option_price(self):
return (self.S * si.norm.cdf(self.d1(), 0.0, 1.0) - self.K * np.exp(-self.r * self.T) * si.norm.cdf(self.d2(), 0.0, 1.0))

def put_option_price(self):
return (self.K * np.exp(-self.r * self.T) * si.norm.cdf(-self.d2(), 0.0, 1.0) - self.S * si.norm.cdf(-self.d1(), 0.0, 1.0))


def calculate_implied_volatility(S, K, T, r, market_price, option_type='call', minimizer="python"):
"""
Calculates the implied volatility (sigma) for a given option.

This function uses the scipy.optimize.minimize_scalar method to find the
volatility that minimizes the difference between the Black-Scholes
model price and the observed market price.
"""

# Objective function to minimize: the squared difference between
# the Black-Scholes price and the market price.
def objective_function(sigma):
model = BlackScholesModel(S, K, T, r, sigma)
if option_type == 'call':
```

```
model_price = model.call_option_price()
else:
model_price = model.put_option_price()
return (model_price - market_price)**2

if minimizer=="python":
print("Python minimizer")
solution = minimize_scalar(
objective_function,
bounds=(1e-7, 2.0),  # Volatility is typically between 0% and 200%
method='bounded'
).x
else:
print("Rust MCS minimizer")
solution = rust_mcs_python.mcs_py(
n=1,
func=objective_function,
u=np.array([1e-7]),
v=np.array([2.0]),
nsweeps=10,
nf=1000,
local=10,
gamma=1e-10,
smax=10,
hess=np.ones([1,1])
)
solution = float(solution[0])

return solution


for optimizer in ["python", "Rust_MCS"]:
implied_vol = calculate_implied_volatility(
S=underlying_price,
K=strike_price,
T=time_to_expiration,
r=RISK_FREE_RATE,
market_price=market_price,
option_type='call',
minimizer=optimizer
)
print(f"  Calculated Implied Volatility (Python)  (): {implied_vol:.4f} ({implied_vol*100:.2f}%)")

# Python minimizer
#   Calculated Implied Volatility (Python)  (): 0.2063 (20.63%)
# Rust MCS minimizer
#   Calculated Implied Volatility (Python)  (): 0.2063 (20.63%)

class HestonModel:
"""
Implementation of the Heston stochastic volatility model for pricing European options.

The Heston model assumes that volatility follows a mean-reverting square-root process:
dS_t = r * S_t * dt + sqrt(V_t) * S_t * dW1_t
dV_t = kappa * (theta - V_t) * dt + sigma_v * sqrt(V_t) * dW2_t

Parameters:
- S: Current stock price
- K: Strike price
- T: Time to expiration
- r: Risk-free rate
- v0: Initial variance (volatility squared)
- kappa: Mean reversion speed of variance
- theta: Long-term variance level
- sigma_v: Volatility of variance (vol of vol)
- rho: Correlation between stock and variance processes
"""

def __init__(self, S, K, T, r, v0, kappa, theta, sigma_v, rho):
self.S = S
self.K = K
self.T = T
self.r = r
self.v0 = v0        # Initial variance
self.kappa = kappa  # Mean reversion speed
self.theta = theta  # Long-term variance
self.sigma_v = sigma_v  # Vol of vol
self.rho = rho      # Correlation

def characteristic_function(self, phi, j=1):
"""
Heston characteristic function for option pricing.
j=1 for P1, j=2 for P2 in the option pricing formula.
"""
# Model parameters
S, K, T, r = self.S, self.K, self.T, self.r
v0, kappa, theta, sigma_v, rho = self.v0, self.kappa, self.theta, self.sigma_v, self.rho

# Avoid numerical issues
if abs(phi) < 1e-12:
phi = 1e-12

# Parameters for different integrals
if j == 1:
u = 0.5
b = kappa - rho * sigma_v
else:  # j == 2
u = -0.5
b = kappa
```

```

# Complex calculations
i = complex(0, 1)  # imaginary unit

# Discriminant
d = np.sqrt((rho * sigma_v * i * phi - b)**2 - sigma_v**2 * (2 * u * i * phi - phi**2))

# g factor
g = (b - rho * sigma_v * i * phi + d) / (b - rho * sigma_v * i * phi - d)

# Handle numerical stability
exp_dT = np.exp(d * T)
if abs(exp_dT) > 1e10:  # Prevent overflow
exp_dT = 1e10 * np.sign(exp_dT.real)

# C and D functions
term1 = 1 - g * exp_dT
if abs(term1) < 1e-12:
return complex(0, 0)

C = (r * i * phi * T +
(kappa * theta / sigma_v**2) * ((b - rho * sigma_v * i * phi + d) * T -
2 * np.log(term1 / (1 - g))))

D = ((b - rho * sigma_v * i * phi + d) / sigma_v**2) * ((1 - exp_dT) / term1)

# Characteristic function
cf = np.exp(C + D * v0 + i * phi * np.log(S))

return cf

def call_option_price(self):
"""
Calculate call option price using Heston model via Fourier inversion.
"""
def integrand1(phi):
"""First integral for P1"""
try:
if phi == 0:
return 0
cf = self.characteristic_function(phi, j=1)
integrand = (np.exp(-1j * phi * np.log(self.K)) * cf / (1j * phi)).real
return integrand
except:
return 0.0

def integrand2(phi):
"""Second integral for P2"""
try:
if phi == 0:
return 0
cf = self.characteristic_function(phi, j=2)
integrand = (np.exp(-1j * phi * np.log(self.K)) * cf / (1j * phi)).real
return integrand
except:
return 0.0

# Calculate the two probabilities P1 and P2
try:
# Use adaptive integration with better error handling
P1_integral, _ = quad(integrand1, 1e-10, 50, limit=100, epsabs=1e-8, epsrel=1e-6)
P1 = 0.5 + (1/np.pi) * P1_integral

P2_integral, _ = quad(integrand2, 1e-10, 50, limit=100, epsabs=1e-8, epsrel=1e-6)
P2 = 0.5 + (1/np.pi) * P2_integral

# Ensure probabilities are in [0,1]
P1 = max(0, min(1, P1))
P2 = max(0, min(1, P2))

# Heston call option price formula
call_price = self.S * P1 - self.K * np.exp(-self.r * self.T) * P2

return max(call_price, 0)  # Ensure non-negative price

except Exception as e:
print(f"Integration failed: {e}")
# Fallback to Black-Scholes if integration fails
return self._black_scholes_fallback()

def put_option_price(self):
"""Calculate put option price using put-call parity"""
call_price = self.call_option_price()
put_price = call_price - self.S + self.K * np.exp(-self.r * self.T)
return max(put_price, 0)

def _black_scholes_fallback(self):
"""Fallback Black-Scholes pricing"""
sigma_equiv = np.sqrt(self.v0)
if sigma_equiv <= 0:
return max(self.S - self.K * np.exp(-self.r * self.T), 0)

d1 = (np.log(self.S/self.K) + (self.r + 0.5*sigma_equiv**2)*self.T) / (sigma_equiv*np.sqrt(self.T))
d2 = d1 - sigma_equiv*np.sqrt(self.T)
return self.S * stats.norm.cdf(d1) - self.K * np.exp(-self.r*self.T) * stats.norm.cdf(d2)


def estimate_initial_volatility(S, K, T, r, market_price, option_type='call'):
"""Estimate initial volatility using Black-Scholes implied volatility"""
```

```
def bs_price(vol):
d1 = (np.log(S/K) + (r + 0.5*vol**2)*T) / (vol*np.sqrt(T))
d2 = d1 - vol*np.sqrt(T)
if option_type == 'call':
return S * stats.norm.cdf(d1) - K * np.exp(-r*T) * stats.norm.cdf(d2)
else:
return K * np.exp(-r*T) * stats.norm.cdf(-d2) - S * stats.norm.cdf(-d1)

def objective(vol):
if vol <= 0:
return 1e6
try:
return (bs_price(vol) - market_price)**2
except:
return 1e6

# Try to find implied volatility
from scipy.optimize import minimize_scalar
result = minimize_scalar(objective, bounds=(0.01, 5.0), method='bounded')

if result.success:
return result.x
else:
# Rough estimate based on option moneyness
moneyness = S / (K * np.exp(-r * T))
if moneyness > 1.1:  # Deep ITM
return 0.15
elif moneyness < 0.9:  # Deep OTM
return 0.35
else:  # ATM
return 0.25


def calibrate_heston_model(S, K, T, r, market_price, option_type='call',optimizer='python'):
"""
Calibrate Heston model parameters to match market option price.

Returns: Dictionary with calibrated parameters
"""

# Estimate initial volatility
implied_vol = estimate_initial_volatility(S, K, T, r, market_price, option_type)
initial_variance = implied_vol**2

def objective_function(params):
"""
Objective function to minimize: squared difference between model and market price.
"""
try:
v0, kappa, theta, sigma_v, rho = params

# Parameter constraints to ensure model stability
if v0 <= 1e-6 or kappa <= 0.1 or theta <= 1e-6 or sigma_v <= 1e-6:
return 1e6
if abs(rho) >= 0.99:
return 1e6
if 2 * kappa * theta <= sigma_v**2:  # Feller condition
return 1e6

# Create Heston model instance
model = HestonModel(S, K, T, r, v0, kappa, theta, sigma_v, rho)

# Calculate model price
if option_type == 'call':
model_price = model.call_option_price()
else:
model_price = model.put_option_price()

# Check for reasonable price
if model_price <= 0 or model_price > S:
return 1e6

# Return squared relative error
relative_error = abs(model_price - market_price) / market_price
return np.log(relative_error**2)

except Exception as e:
return 1e6

# Better initial parameter guesses based on market data
initial_guess = [
max(0.01, initial_variance),  # v0: initial variance
2.0,     # kappa: mean reversion speed
max(0.01, initial_variance * 0.8),  # theta: long-term variance (slightly lower)
0.2,     # sigma_v: volatility of volatility
-0.3     # rho: correlation (typically negative for equity)
]

# Tighter parameter bounds
bounds = [
(0.005, 0.5),
(0.1, 8.0),         # kappa
(0.005, 0.5),       # theta
(0.01, 1.0),        # sigma_v
(-0.95, 0.95)       # rho
]

# Optimization with multiple attempts
print("  Calibrating Heston model parameters...")
```

```python
print("   This may take a few moments due to numerical integration...")

best_result = None

# Try multiple starting points
for attempt in range(3):
if attempt > 0:
# Perturb initial guess for subsequent attempts
perturbed_guess = [
initial_guess[0] * (0.5 + np.random.random()),
initial_guess[1] * (0.5 + np.random.random()),
initial_guess[2] * (0.5 + np.random.random()),
initial_guess[3] * (0.5 + np.random.random()),
initial_guess[4] * (0.5 + 0.5 * np.random.random()) * np.sign(initial_guess[4])
]

# Ensure bounds
for i, (lower, upper) in enumerate(bounds):
perturbed_guess[i] = max(lower, min(upper, perturbed_guess[i]))
else:
perturbed_guess = initial_guess

if optimizer == "python":
print("Python Optimizer")
result = minimize(
objective_function,
perturbed_guess,
method='L-BFGS-B',
bounds=bounds,
options={'maxiter': 5000, 'ftol': 1e-16}
)
print(f"minimum: {result}")
v0, kappa, theta, sigma_v, rho = result.x
else:
print("Rust MCS optimizer")
result = rust_mcs_python.mcs_py(
n=5,
func=objective_function,
u=np.array([0.005, 0.1, 0.005, 0.01, -0.95]),
v=np.array([0.5, 8.0, 0.5, 1.0, 0.95]),
nsweeps=200,
nf=5000,
local=20,
gamma=1e-16,
smax=100,
hess=np.ones([5,5])
)
print(result[-1])
print(f"minimum: {result[1]}")
v0, kappa, theta, sigma_v, rho = result[0]

return {
  'v0': v0,
  'kappa': kappa,
  'theta': theta,
  'sigma_v': sigma_v,
  'rho': rho,
  'success': True,
  'optimization_result': best_result,
  'implied_vol': implied_vol
}


def run_heston_calibration(underlying_price, strike_price, time_to_expiration,
risk_free_rate, market_price, option_type='call', optimizer="python"):
"""Main function to run Heston calibration with proper error handling"""

print("  Starting Heston model calibration...")
calibration_result = calibrate_heston_model(
S=underlying_price,
K=strike_price,
T=time_to_expiration,
r=risk_free_rate,
market_price=market_price,
option_type=option_type,
optimizer=optimizer
)

if calibration_result['success']:
# Extract calibrated parameters
v0 = calibration_result['v0']
kappa = calibration_result['kappa']
theta = calibration_result['theta']
sigma_v = calibration_result['sigma_v']
rho = calibration_result['rho']

print(f"\ n Heston Model Calibration Results:")
print(f"   Initial Variance (v0):     {v0:.6f}  ( = {np.sqrt(v0)*100:.2f}%)")
print(f"   Mean Reversion Speed  ():  {kappa:.4f}")
print(f"   Long-term Variance  ():    {theta:.6f} (long-term   = {np.sqrt(theta)*100:.2f}%)")
print(f"   Vol of Vol  ():           {sigma_v:.4f}")
print(f"   Correlation  ():           {rho:.4f}")
print(f"   Implied Vol (Black-Scholes): {calibration_result['implied_vol']*100:.2f}%")

# Test the calibrated model
calibrated_model = HestonModel(
S=underlying_price,
K=strike_price,
T=time_to_expiration,
r=risk_free_rate,
v0=v0,
kappa=kappa,
theta=theta,
sigma_v=sigma_v,
rho=rho
)

heston_price = calibrated_model.call_option_price() if option_type == 'call' else calibrated_model.put_option_price()

print(f"\ n Price Comparison:")
print(f"   Market Price:              ${market_price:.2f}")
print(f"   Heston Model Price:        ${heston_price:.2f}")
print(f"   Absolute Error:            ${abs(heston_price - market_price):.4f}")
print(f"   Relative Error:            {abs(heston_price - market_price)/market_price*100:.2f}%")

# Model interpretation
print(f"\ n Model Interpretation:")
current_vol = np.sqrt(v0) * 100
long_term_vol = np.sqrt(theta) * 100
print(f"   Current implied volatility: {current_vol:.1f}%")
print(f"   Long-term volatility:       {long_term_vol:.1f}%")

if current_vol > long_term_vol + 1:
print(f"      Volatility expected to decrease (high vol regime)")
elif long_term_vol > current_vol + 1:
print(f"      Volatility expected to increase (low vol regime)")
else:
print(f"      Volatility near long-term level")

print(f"      Mean reversion half-life: {np.log(2)/kappa:.2f} years")

if rho < -0.1:
print(f"      Negative correlation: volatility increases when price falls (leverage effect)")
elif rho > 0.1:
print(f"      Positive correlation: volatility increases when price rises")
else:
print(f"      Low correlation between price and volatility")

# Feller condition check
feller_condition = 2 * kappa * theta / (sigma_v**2)
print(f"      Feller condition: {feller_condition:.2f} (should be > 1 for well-behaved variance)")

return calibrated_model, calibration_result

else:
print("  Heston model calibration failed")
print("   This could be due to:")
print("   • Market price inconsistent with no-arbitrage bounds")
print("   • Insufficient liquidity in the option")
print("   • Numerical integration difficulties")
print(f"   • Try using Black-Scholes implied vol: {calibration_result['implied_vol']*100:.1f}%")

return None, calibration_result

for optimizer in ["python", "Rust_MCS"]:
run_heston_calibration(
underlying_price=float(underlying_price), # Current stock price
strike_price=near_the_money_call["strike"],
time_to_expiration=time_to_expiration,
risk_free_rate=RISK_FREE_RATE,
market_price=near_the_money_call["lastPrice"],
option_type='call',
optimizer=optimizer
)


#   Starting Heston model calibration...
#   Calibrating Heston model parameters...
#    This may take a few moments due to numerical integration...
# Python Optimizer
# minimum:   message: CONVERGENCE: RELATIVE REDUCTION OF F <= FACTR*EPSMCH
#   success: True
#    status: 0
#       fun: -33.28652381319492
#         x: [ 4.447e-02  1.999e+00  3.667e-02  1.979e-01 -3.022e-01]
#       nit: 6
#       jac: [-1.628e+06  4.996e+04 -1.370e+07  2.468e+05  2.083e+05]
#      nfev: 414
#      njev: 69
#  hess_inv: <5x5 LbfgsInvHessProduct with dtype=float64>
# Python Optimizer
# minimum:   message: ABNORMAL:
#   success: False
#    status: 2
#       fun: -37.26513321662305
#         x: [ 4.502e-02  2.530e+00  3.613e-02  2.746e-01  2.437e-01]
#       nit: 4
#       jac: [ 5.194e+08  3.903e+05 -1.887e+08  7.271e+06  2.042e+06]
#      nfev: 444
#      njev: 74
#  hess_inv: <5x5 LbfgsInvHessProduct with dtype=float64>
# Python Optimizer
# minimum:   message: ABNORMAL:
#   success: False
#    status: 2
#       fun: -35.38369165525719
#         x: [ 4.498e-02  2.955e+00  3.588e-02  2.399e-01  2.010e-01]
#       nit: 3
#       jac: [ 3.047e+08  1.576e+05 -6.478e+07  2.463e+06  6.971e+05]
#      nfev: 372
#      njev: 62
#  hess_inv: <5x5 LbfgsInvHessProduct with dtype=float64>

#  Heston Model Calibration Results:
#    Initial Variance (v0):     0.044977  ( = 21.21%)
#    Mean Reversion Speed  ():  2.9547
#    Long-term Variance  ():    0.035879 (long-term   = 18.94%)
#    Vol of Vol  ():           0.2399
#    Correlation  ():           0.2010
#    Implied Vol (Black-Scholes): 19.86%

#  Price Comparison:
#    Market Price:              $4.42
#    Heston Model Price:        $4.42
#    Absolute Error:            $0.0000
#    Relative Error:            0.00%

#  Model Interpretation:
#    Current implied volatility: 21.2%
#    Long-term volatility:       18.9%
#       Volatility expected to decrease (high vol regime)
#       Mean reversion half-life: 0.23 years
#       Positive correlation: volatility increases when price rises
#       Feller condition: 3.68 (should be > 1 for well-behaved variance)
#  Starting Heston model calibration...
#  Calibrating Heston model parameters...
#    This may take a few moments due to numerical integration...
# Rust MCS optimizer
# StopNsweepsExceeded
# minimum: -43.549289157610396
# Rust MCS optimizer
# StopNsweepsExceeded
# minimum: -43.549289157610396
# Rust MCS optimizer
# StopNsweepsExceeded
# minimum: -43.549289157610396

#  Heston Model Calibration Results:
#    Initial Variance (v0):     0.005000  ( = 7.07%)
#    Mean Reversion Speed  ():  7.0093
#    Long-term Variance  ():    0.376250 (long-term   = 61.34%)
#    Vol of Vol  ():           0.5890
#    Correlation  ():           -0.1743
#    Implied Vol (Black-Scholes): 19.86%

#  Price Comparison:
#    Market Price:              $4.42
#    Heston Model Price:        $4.42
#    Absolute Error:            $0.0000
#    Relative Error:            0.00%

#  Model Interpretation:
#    Current implied volatility: 7.1%
#    Long-term volatility:       61.3%
#       Volatility expected to increase (low vol regime)
#       Mean reversion half-life: 0.10 years
#       Negative correlation: volatility increases when price falls (leverage effect)
#       Feller condition: 15.20 (should be > 1 for well-behaved variance)
```

Листинг 1: Python3 Code for Models Calibration