PYTHOPT: A PROBLEM-SOLVING ENVIRONMENT FOR OPTIMIZATION METHODS

A Thesis Submitted to the
College of Graduate Studies and Research
in Partial Fulfillment of the Requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Krzysztof Voss

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
Saskatoon, Saskatchewan S7N 5A9

Abstract

Optimization is a process of finding the best solutions to problems based on mathematical models. There are numerous methods for solving optimization problems, and there is no method that is superior for all problems. This study focuses on the Particle Swarm Optimization (PSO) family of methods, which is based on the swarm behaviour of biological organisms. These methods are easily adjustable, scalable, and have been proven successful in solving optimization problems.

This study examines the performance of nine optimization methods on four sets of problems. The performance analysis of these methods is based on two performance metrics (the win-draw-loss metric and the performance profiles metric) that are used to analyze experimental data. The data are gathered by using each optimization method in multiple configurations to solve four classes of problems.

A software package `pythOPT` was created. It is a problem-solving environment that is comprised of a library, a framework, and a system for benchmarking optimization methods. `pythOPT` includes code that prepares experiments, executes computations on a distributed system, stores results in a database, analyzes those results, and visualizes analyses. It also includes a framework for building PSO-based methods and a library of benchmark functions used in one of the presented analyses.

Using `pythOPT`, the performance of these nine methods is compared in relation to three parameters: number of available function evaluations, accuracy of solutions, and communication topology. This experiment demonstrates that two methods (SPSO and GCPSO) are superior in finding solutions for the tested classes of problems. Finally, by using `pythOPT` we can recreate this study and produce similar ones by changing the parameters of an experiment. We can add new methods and evaluate their performances, and this helps in developing new optimization methods.

Acknowledgements

A sincere thank you to my supervisor, Dr. Raymond Spiteri, for providing me with the opportunity to be a part of the Numerical Simulation Laboratory. His guidance, time, patience, and funding was integral to the completion of my thesis. Thanks to Dr. Horsch, Dr. Eramian, and Dr. Steele for their contributions and advice. In addition, I thank the Department of Computer Science for their funding, resources, and encouragement. I have to make a special mention to Gwen Lancaster and Janice Thompson for their support, organization, and guidance towards the completion of my thesis.

I want to thank Eve Marie Johnson and her parents for their never-ending support. Eve Marie was able to show me a different perspective when I started having moments of doubt. She was my lifeline in what could sometimes feel like a bottomless tarpit, and gave me the necessary strength, and inspiration.

Thank you to my parents, and family, as they have always supported me in my interest and development in science. They also emphasized the importance of knowledge and fostered my curiosity.

Furthermore, thank you to my friends for helping me go through the process of writing this thesis, for their support, and for all the memories we made together. I need to directly mention Jordan Cooper, and Christianne Rooke for their contributions, reviews, corrections, suggestions, patience, and friendship. There are many others that I would also like to thank but I want to keep the acknowledgments to one page.

To Eve Marie. To parents. By me, for others.

Contents

List of Tables

List of Figures

List of Abbreviations

BP	Benchmark Problems
CDIWPSO	Chaotic Descending Inertia Weight PSO
DAPSO	Dynamic Adaptive PSO
DB	Database
DWPSO	Decreasing Weight PSO
GBPSO	Global Best PSO
GCPSO	Global Convergence PSO
GPF	Gold Particle Freezing
GWO	Grey Wolf Optimization
JSON	JavaScript Object Notation
NEP	Narrow Escape Problem
PSE	Problem-Solving Environment
PSO	Particle Swarm Optimization
RMD	Rational Material Design
SPSO	Standard PSO
TVACPSO	Time Varying Acceleration Coefficients PSO

List of Symbols

f	objective function
$\mathbb{D}$	search space
$\boldsymbol{x}$	decision vector
$\boldsymbol{x}^*$	optimal decision vector
$\boldsymbol{g}_=$	equality constraint function
$\boldsymbol{g}_\leq$	inequality constraint function
$\mathbb{F}$	feasible set
N	maximal number of evaluations

Chapter 1

Introduction

Optimization problems are common and appear in numerous areas of our lives. We solve one each time we pick the best route to a destination, we pick the best offer online, or when we schedule a day or a trip. Interesting and practical challenges in engineering and science can be expressed as optimization problems and solved with well-known methods. This approach is used for research in medicine, machine learning, financial markets, material design, electronic design, and many other fields.

Optimization problems may be difficult to solve. Solutions to some problems may not be found within given computational resources, and approximations have to suffice. There exist different approaches to solving these problems, and some optimization methods yield better approximations than others on a given problem. This wide spectrum of methods is a consequence of diversity of optimization problems, their characteristics, and available information. No single method outperforms all other optimization methods for every optimization problem as shown in the No Free Lunch theorem [49]. With limited computational resources, we try to devise methods that yield the best results in the least amount of time.

Selecting the most appropriate method for a given problem is a common difficulty. The selection is often based on the type of objective function or the type of constraints on the solutions. As a result, the selected method often comes from a subset of methods that are deemed reasonable for a certain problem. One can be confident in the choice made by comparing multiple solvers. The problem then reduces to the accessibility to numerous methods and the availability of metrics for measuring the performance of these methods.

This study evaluates an approach to selecting an optimization method for a class of problems. Real problems often have initial conditions that generate variations of a problem; i.e., we often deal with a class of similar problems. The right method should be capable of

solving most of the variations within the class better than other methods. Such a method should have a better chance of finding the solution to the given optimization problem.

1.1 Numerical optimization

Optimization methods help to find the minimal value of a function. The function is called the *objective function*, *fitness function*, or *cost function*. Formally, to solve an optimization problem is to find vector $\boldsymbol{x}^*$ such that

$$f(\boldsymbol{x}^*) \leq f(\boldsymbol{x}) \quad \text{for all } \boldsymbol{x} \in \mathbb{D},$$

where $\mathbb{D}$ represents domain of the objective function f. Vector $\boldsymbol{x}^*$ may represent the shortest path on a map, a financial investment, the structure of a protein, or a solution to an equation. This method of formulating problems shifts the focus to modelling a phenomenon of interest. Finding optimal decisions is then a matter of numerous evaluations of the model.

Many difficult problems have been successfully solved with optimization. For example, Kramer et al. [27] and Conn et al. [15] cite exemplary applications in biochemistry, aircraft design, hydrodynamics, medicine, earth sciences, automatic error analysis, and dynamic pricing. These examples reveal the variety of fields in which optimization is applied, but new applications are constantly emerging. The increase in the number of applications stresses the importance of efficient and robust optimization methods.

1.2 `pythOPT`

The `pythOPT` problem-solving environment (PSE) is computer software for performing performance analyses of optimization methods, and solving optimization problems. It facilitates persistence, parallel execution, the evaluation of different configurations of methods, and the preparation of analyses using available metrics. In turn, these analyses can be used to identify the most effective methods. These features foster the development of new optimization methods, and their fast evaluation.

The software includes a flexible implementation of a class of swarm-based optimization methods. These methods are highly adjustable. Changing parameters of the built-in methods

is the simplest way of creating new methods. The PSE can then be used to evaluate them and decide whether the changes are positive or not.

1.3 Contributions

Our main contribution is the `pythOPT` PSE that helps to identify the best optimization method for a given class of optimization problems. It takes a design of experiment on input, and it conducts the experiment, acquires and stores data, and generates data visualization. Our software features optimization methods and benchmark problems to provide a complete environment for development of new methods. It also allows for solving optimization problems. The research that we present is the output from using `pythOPT` with three different experiment designs.

Our research tests the performance of nine optimization methods (sixteen methods if we count major variations) on four sets of optimization problems. We identify superior performance of two methods (SPSO and GCPSO). The metrics that we use to analyse the results of our experiments emphasize the difference between methods that have faster convergence and those that are able to find better approximations over extended periods of time. Our results serve as a proof of concept for the efficacy of `pythOPT` in finding the most appropriate optimization method for a given class of optimization problems.

1.4 Overview

The second chapter contains background for the research. It defines optimization and presents optimization methods relevant to the thesis. The third chapter documents `pythOPT`, the problem-solving environment that has been developed for this research. The fourth chapter contains an analysis of the performance of optimization methods described earlier. It presents the methodology and four classes of optimization problems that are used as the basis for the analysis and the results. The fifth chapter summarizes these results and proposes future work.

Chapter 2

Numerical Optimization

Numerical optimization is a branch of applied mathematics and numerical analysis that offers methods for solving optimization problems. Sometimes a problem presents itself with little available information. In this case, the only way to find a solution is through numerical optimization.

This chapter presents the language used for modeling solutions for optimization problems, introduces the notation, expands on the most important concepts, and presents relevant numerical methods for solving optimization problems. Section 2.1 defines an optimization problem and what constitutes a solution to such a problem. Section 2.2 focuses on the objective function and methods for handling expensive function evaluations. Section 2.3 focuses on the search space and methods for handling constraints. Section 2.4 presents numerical methods for solving optimization problems that are relevant to the experiments presented in Chapter 4.

2.1 Optimization problems

Two fundamental concepts in optimization are the *objective function* $f : \mathbb{D} \to \mathbb{R}$ and the *search space* $\mathbb{D}$. An element of a search space is called a *decision vector* $\boldsymbol{x}$. Let a pair $(f, \mathbb{D})$ be called an *unconstrained optimization problem*. Let there exist a vector $\boldsymbol{x}^* \in \mathbb{D}$ such that

$$f(\boldsymbol{x}^*) \leq f(\boldsymbol{x}), \quad \forall \boldsymbol{x} \in \mathbb{D}. \tag{2.1}$$

A solution to an optimization problem is a value of $f(\boldsymbol{x}^*)$, called a *minimum*, and it is denoted by

$$f(\boldsymbol{x}^*) = \min_{\boldsymbol{x} \in \mathbb{D}} f(\boldsymbol{x}). \tag{2.2}$$

The inequality (2.1) allows multiple elements in $\mathbb{D}$ to be solutions. The level set of elements in $\mathbb{D}$ that yield the minimal value is represented by

$$\arg\min_{\boldsymbol{x}\in\mathbb{D}} f(\boldsymbol{x}) = \{\boldsymbol{x} \in \mathbb{D} : f(\boldsymbol{x}) = \min_{\boldsymbol{x}\in\mathbb{D}} f(\boldsymbol{x})\}. \tag{2.3}$$

Adding sets of $n_=$ *equality constraints*, $\boldsymbol{g}_= : \mathbb{D} \to \mathbb{R}^{n_=}$, and $n_\leq$ *inequality constraints*, $\boldsymbol{g}_\leq : \mathbb{D} \to \mathbb{R}^{n_\leq}$, defines a *feasible set* $\mathbb{F}$, a set of decision vectors that meet problem constraint

$$\mathbb{F} = \{\boldsymbol{x} : \boldsymbol{x} \in \mathbb{D},\ \boldsymbol{g}_=(\boldsymbol{x}) = \boldsymbol{0},\ \boldsymbol{g}_\leq(\boldsymbol{x}) \leq \boldsymbol{0}\}. \tag{2.4}$$

A pair $(f, \mathbb{F})$ is then called a *constrained optimization problem*. Constrained optimization problems take the form

$$\min_{\boldsymbol{x}\in\mathbb{D}} f(\boldsymbol{x}) \tag{2.5}$$

subject to

$$\boldsymbol{g}_=(\boldsymbol{x}) = \boldsymbol{0}, \tag{2.6}$$

$$\boldsymbol{g}_\leq(\boldsymbol{x}) \leq \boldsymbol{0}. \tag{2.7}$$

A solution to a constrained optimization problem is by definition an element in the feasible set $\mathbb{F}$.

2.2 Objective function

An objective function is one of two main components of an optimization problem. This section contains two illustrative optimization problems and a description of the structure of their objective functions. The section then moves on to a discussion about the objective function's properties and types. It concludes with a list of common classes of optimization problems.

An objective function defines a computation from search space $\mathbb{D}$ into a partially ordered set (often it is $\mathbb{R}$). The set has a minimum value if Eq. 2.1 is satisfied. The objective function f has two distinct names that relate to an element being sought – *cost* function for a minimum and *fitness* function for a maximum. A maximum of an objective function f is

a minimum of its negative,

$$\max_{\boldsymbol{x}} f(\boldsymbol{x}) \equiv \min_{\boldsymbol{x}} -f(\boldsymbol{x}).$$

These terms are synonyms, and either of them can be called an *optimum*.

At first glance, it may seem that f is merely a model, a simplified representation of a phenomenon of interest, but an objective function plays a more profound role, distinct from that of a model. Its structure is a part of a solution, and it can address some challenging aspects of a model. Examples of such aspects include:

- a high number of dimensions,
- a long evaluation time,
- no information about a model's derivatives,
- no information about a model's internals,
- multiple local optima, and
- no convergence of a model for elements in search space.

The following examples present different approaches to tackling some of these challenges.

As the first example, we present a method of designing molecules that can potentially be new drugs. Let us assume that a molecule has the lowest free energy in its native, stable geometrical structure. The model function takes as the input a geometrical structure and returns a value of the corresponding free energy of a molecule. The result of minimizing the function is a solution to the original problem. The number of possible conformations is the main obstacle of this approach. The complexity grows exponentially with the number of atoms, and bigger molecules form a problem that is increasingly computationally infeasible. The solution proposed in [32] decomposes the problem into two phases:

- searching for initial conformations and
- refining the initial conformations.

The objective function takes as input an initial conformation and in fact refines it by internally solving an optimization problem of finding the best conformation in the neighbourhood of the initial conformation. The objective function in that last optimization problem evaluates free energy level of a molecule.

As mentioned, the computational complexity of models often poses a challenge in solving optimization problems. The second example introduces an approach used for designing helicopter rotor blades. A model divides a blade into ten segments and has a total of thirty-one parameters: ten masses, ten centres of gravity, and eleven single direction stiffnesses. The objective is to minimize the measure of vibration of a blade, namely the weighted sum of the first and the second harmonics of vibration of the blade. A simulation of a behaviour of a blade is computationally expensive and poses a challenge in a time-constrained environment.

The approach proposed by Booker et al. [10] uses a *surrogate* method, a method that controls evaluation of underlying models. Two models were used: an accurate model, which takes several hours to evaluate, and a less accurate model, which takes only a few minutes to evaluate. The surrogate uses both models to provide values that may be of lower accuracy but come at a lower computational expense. In this example, the structure of an objective function allows for more evaluations and ultimately a better approximation.

The above examples present two applications of numerical optimization and two objective functions that contain structures of evaluating models. These structures aim at improving accuracy of optimization. An objective function with such structure is not only a model; it may be considered a part of an optimization method.

2.2.1 Classification

Information about the class of an optimization problem helps in choosing an optimization strategy. Optimization problems are classified with respect to features of the components constituting a particular problem. For an objective function, these features include:

- structures/forms of the model and
- objective function type.

Convexity and Structure For a convex set $\mathbb{D}$, i.e., a set for which

$$\boldsymbol{x}, \boldsymbol{y} \in \mathbb{D} \Rightarrow \{\alpha\boldsymbol{x} + (1-\alpha)\boldsymbol{y} : 0 \leq \alpha \leq 1\} \subset \mathbb{D},$$

one can define a convex function f over it that has the following property:

$$\boldsymbol{x}, \boldsymbol{y} \in \mathbb{D} \Rightarrow f(\alpha\boldsymbol{x} + (1-\alpha)\boldsymbol{y}) \leq \alpha f(\boldsymbol{x}) + (1-\alpha) f(\boldsymbol{y}), \quad 0 \leq \alpha \leq 1.$$

If an objective function is a convex function and a search space is a convex set, then minimizing f over $\mathbb{D}$ is a *local* optimization problem. Otherwise, the problem is said to be *global*.

Certain objective functions have characteristics that can be used to solve a particular optimization problem efficiently. They appear commonly, and optimization problems involving these objective functions have their own names, e.g.,

- linear programming

  $$\min_{\boldsymbol{x}} \sum_{j=1}^{n} c_j x_j : \sum_{j=1}^{n} A_{ij} x_j \leq b_i,$$

  where $c_j, b_i, A_{ij} \in \mathbb{R}, i = 1, 2, \ldots, m$, and $j = 1, 2, \ldots, n$,

- linear least-squares problems

  $$\min_{\boldsymbol{x}} \sum_{i=1}^{m} \left(\sum_{j=1}^{n} A_{ij} x_j - b_i \right)^2$$

  where $b_i, A_{ij} \in \mathbb{R}$, and $i = 1, 2, \ldots, m, j = 1, 2, \ldots, n$, and

- quadratic programming

  $$\min_{\boldsymbol{x}} \sum_{j=1}^{n} (x_i^2 + c_j x_j) : \sum_{j=1}^{n} A_{ij} x_j \leq b_i, \quad i = 1, 2, \ldots, m.$$

Properties and forms of the functions involved in defining an optimization problem are not the only factors determining the category of an optimization problem. Objective functions can have different types.

Types of objective functions In defining an optimization problem, an objective function is often of type $\mathbb{D} \to \mathbb{R}$. This type represents a particular class of problems, namely *single-objective* optimization problems. Objective functions of type $\mathbb{D} \to \mathbb{R}^m$ form *multi-objective* optimization problems.

Optimization can also be classified with respect to types of variables that an objective function takes as the input. This classification distinguishes between *discrete* and *continuous* optimization. A continuous optimization problem is more general than a discrete problem; continuous optimization problems are a superset of discrete problems with an additional equality constraint that imposes on elements of a candidate decision vector, i.e., decision vectors proposed by an optimization method, to be integers. An example of such constraint is $\sin(\pi \boldsymbol{x}_i) = 0$ that allows only vectors of integer values in the feasible set. Another, more strict constraint is $\boldsymbol{x}_i(1 - \boldsymbol{x}_i) = 0$ that narrows possible values to the set $\{0, 1\}$, making it possible to represent discrete optimization problems. Although representing discrete optimization problems as continuous is formally possible, it is artificial, and methods for handling discrete problems are fundamentally different from those for approaching continuous problems.

These exemplary objective functions return a structure based on real numbers. It is a convenient choice because it enables usage of gradient-based methods or polynomial approximations to reduce the complexity of a problem. In general, an objective function maps its domain into a partially ordered set. A partially ordered set is a set of elements with a binary order relation $\leq$ such that for all elements in the set is reflexive ($a \leq a$), antisymmetric ($a \leq b, b \leq a \Rightarrow a = b$), and transitive ($a \leq b, b \leq c \Rightarrow a \leq c$).

In summary, objective functions are not merely models, and their evaluation is a part of an optimization method. An objective function has its own logic and state. It can use any number of models, modify them, or create new models. The type of optimization depends on factors including properties of a model, types of components of a decision vector, or a type returned from an objective function. The taxonomy of optimization is built around these types and properties.

2.3 Search space

If a decision vector needs to satisfy one or more properties, we define *constraints* on the domain of an objective function, and the problem becomes a *constrained* optimization problem. Many optimization problems in practice are constrained. This is often the case due to physical limitations imposed on particular models. Constraints may help resolve an ambiguity if more than one decision vector evaluates to the same value that is the identified minimum.

Commonly, one distinguishes between

- bound constraints,
- equality constraints, and
- inequality constraints.

This nomenclature is also presented in [33, p.123]. For a comprehensive survey of methods for implementing constraints, we refer to [14].

Bound constraints are simple linear inequality constraints that have the form of

$$\boldsymbol{x}_L \leq \boldsymbol{x} \leq \boldsymbol{x}_U, \tag{2.8}$$

where $\boldsymbol{x}_L$ and $\boldsymbol{x}_U$ are the lower and upper bounds, respectively. The bounds are imposed upon each component of a decision vector, $\boldsymbol{x} \in \mathbb{D}$. In the example of designing a helicopter rotor blade [9], bound constraints limit the weights of particular segments of a blade.

The optimization methods have information only about bound constraints. Also, they have an indirect view of the decision space. A method generates a decision vector that in turn is passed to an objective function. Equality constraints (2.6) that define the search space are strict. The likelihood that a candidate decision vector determined by an optimization method satisfies an equality constraint is low. To increase that probability and allow stochastic methods to evolve into regions that satisfy constraints of a given problem, the equality constraints are relaxed to inequality constraints

$$|\boldsymbol{g}_{=}(\boldsymbol{x})| - \boldsymbol{\epsilon} \leq \mathbf{0}, \tag{2.9}$$

where ϵ is a positive constant, and $|\boldsymbol{g}_{=}(\boldsymbol{x})|$ is a component-wise absolute value. One could reverse the problem and start with finding vectors that satisfy the constraints followed by optimization over the space of these vectors. However, solutions are only accurate to within some tolerance. For this reason, a numerical solution to a constrained optimization problem may not strictly be an element of the feasible set; it only satisfies equality constraints to within a tolerance $\boldsymbol{\epsilon}$.

The inequality constraints in Eq. 2.7 are the most general category of constraints. The previous categories can be expressed using them. Using Eq. 2.9, let us introduce a uniform notation for violations of constrains with a common index i:

$$G_i(\boldsymbol{x}) = \begin{cases} \max(0, |\boldsymbol{g}_{=,i}(\boldsymbol{x})| - \epsilon_i) & i = 1, 2, \ldots, n_{=}, \\ \max(0, \boldsymbol{g}_{\leq, i-n_{=}}(\boldsymbol{x})) & i = n_{=} + 1, n_{=} + 2, \ldots, n_{=} + n_{\leq}. \end{cases} \tag{2.10}$$

Vector $\boldsymbol{G}(\boldsymbol{x})$ represents violations of all constraints. The fist $n_{=}$ components of $\boldsymbol{G}(\boldsymbol{x})$ represent violations of the specific equality constraints, and the remaining $n_{\leq}$ components represent the violations of the specific inequality constraints. If it is not possible to make any assumptions about the constraints, then two common strategies for handling them are exclusion and penalization. These strategies are discussed further in the following subsection.

The above classes cover constraints that are known in advance. There are two separate categories of constraints for which an evaluation of an objective function must be performed first and then satisfaction of constraints can be determined. These are

- *hidden* or *physical* constraints and
- *policy* constraints.

Hidden constraints result from the inability to evaluate an objective function for certain values, e.g., a solver does not converge to a solution in a certain region. Policy constraints are imposed after a decision vector is evaluated, but the value is perceived as invalid.

2.3.1 Implementing Constraints

The domain of an objective function is a superset of a feasible set, so not all elements satisfy all constraints. In constrained optimization, a *candidate decision vector* is a vector

for which one does not know if all constraints are satisfied. Candidate decision vectors are generated by an optimization method, but before an objective function is evaluated for a candidate decision vector, the vector or the problem itself can be adjusted. Methods for these adjustments often fall into one of the following categories [44]:

- *repair methods* that modify a candidate decision vector and try to evaluate it,
- *problem-specific representations* that formulate a problem in a way so that all possible decision vectors satisfy the constraints,
- *penalty methods* that transform an optimization problem into a series of augmented problems that converge to the solution of the original problem.

Finding vectors that satisfy a given set of constraints within a certain tolerance may require a significant number of evaluations of $\boldsymbol{g}_=$ and $\boldsymbol{g}_\leq$. Effective methods for solving this problem are particularly necessary if any of the functions that define constraints takes a substantial time to evaluate.

Repair methods In many real-world optimization problems, the solution lies on the constraint boundary [44, p.501]. A repair method tries to fix an infeasible decision vector by moving it to the boundary of the feasible search space. For a simple bound constraint, the method can simply truncate the element of a decision vector that lies beyond the bound. For more complex constraints, the method can infer the new value of the element that violates a constraint by solving an equation with one unknown. In general, repair methods are problem-specific and can fail; however, they offer a simple approach of enforcing bound constraints that are a part of many optimization problems.

Problem-specific representations Depending on the problem, one may be able to turn more general constraints into bound constraints. For example, looking for an optimum within a circle can be either a constrained problem, or it can be bound constrained if we use polar coordinates [44, p.499].

Penalty methods Penalty methods emulate an unconstrained optimization problem with an augmented objective function θ of the following form

$$\theta(\boldsymbol{x}) = f(\boldsymbol{x}) + r\psi(\boldsymbol{x}), \tag{2.11}$$

where ψ is a function that incorporates information about constraints and r is a *penalty factor*. These methods divide into two categories:

- *interior point methods*, which do not evaluate infeasible points, and
- *exterior point methods*, which may evaluate infeasible points.

Interior point methods, also known as *barrier methods*, penalize candidate decision vectors that are already inside of the feasible set but only if they are close enough to the border (barrier) of the set. This way, the method keeps candidate decision vectors within the feasible set. The augmented objective function is called in this context a *barrier function*. For example, a *log-barrier function* has the following form:

$$\theta(\boldsymbol{x}) = f(\boldsymbol{x}) - r \sum_{i=1}^{n_= + n_\leq} \log G_i(\boldsymbol{x}), \quad \{i \colon G_i > 0\}. \tag{2.12}$$

Finding feasible decision vectors is often challenging. For this reason, interior point methods are not commonly used with constrained optimization problems. The algorithms that are discussed later rarely use them and instead use exterior point methods [44, p.484].

Exterior point methods divide into *death penalty* and *non-death penalty* methods. The death penalty methods simply reject candidate decision vectors outside of the feasible set, thus entailing the generation of a new candidate decision vector. Non-death penalty methods, which are more commonly used, allow evaluation of any candidate decision vector. In that case, a penalty representing the magnitude of a violation of constraints is added for any candidate decision vector outside of the feasible set. With an increase in value of the penalty factor, the solution converges to the solution of the original problem.

The *static penalty method* [21] is another exterior point method. This method assigns a penalty to a violation of constraint i as follows

$$r_i(\boldsymbol{x}) = \begin{cases} R_{i1}, & \text{if } G_i(\boldsymbol{x}) \in (0, T_{i1}], \\ R_{i2}, & \text{if } G_i(\boldsymbol{x}) \in (T_{i1}, T_{i2}], \\ \vdots & \\ R_{in}, & \text{if } G_i(\boldsymbol{x}) \in (T_{in-1}, \infty), \end{cases}$$

where T_{in} is a constraint value threshold, R_{ij} is a penalty value for threshold j, and n is the total number of thresholds. We can use this methods with both equality and inequality constraints. The objective function is augmented to incorporate a violation of all constraints multiplied by a corresponding weight of each violation r_i. An example of a static penalty method is the *exact penalty method*, which is defined by

$$\theta(\boldsymbol{x}) = \begin{cases} f(\boldsymbol{x}), & \text{if } \boldsymbol{x} \in \mathbb{F}, \\ +\infty, & \text{otherwise.} \end{cases}$$

The above method uses values of a penalty factor independent from the generation number. In contrast, the *dynamic penalty methods*, first introduced by Joines and Houck [23], change the penalty factor value based on the current generation number. Let $M(\boldsymbol{x})$ be a weighted average of the magnitude of the constraint violation

$$M(\boldsymbol{x}) = \sum_{i=1}^{n_= + n_\leq} r_i G_i(\boldsymbol{x}).$$

The augmented objective function is then defined as

$$\theta(\boldsymbol{x}) = f(\boldsymbol{x}) + (cn)^\alpha M(\boldsymbol{x}),$$

where c and α are constants and n is the current generation number. The typical values for these constants are $c = 0.5$ and $\alpha = 2$; however, this penalty method heavily depends on the choice of these constants, and users should adjust them for their specific cases.

2.3.2 Variable encoding

The term *scaling* or *variable encoding* refers to transformations that are applied to components of a decision vector. These transformations can improve the effectiveness of an

optimization method by making variables have certain desirable properties, e.g., a similar magnitude.

A commonly used transformation is the *diagonal scaling* [20, p.274]. It encodes a decision vector $\boldsymbol{x}$ using a diagonal matrix with positive diagonal elements $\boldsymbol{D}$. These elements represent the order of magnitude of a particular decision variable. Consider

$$\boldsymbol{x} = \boldsymbol{D}\boldsymbol{x}',$$

where $\boldsymbol{x}$ represents a decision vector in the original units and $\boldsymbol{x}'$ represents a decision vector in the scaled units. The optimization problem is then solved in terms of $\boldsymbol{x}'$.

Another scaling method is able to represent the search space as a hypercube. The encoded vector, $\boldsymbol{x}'$, consists of values in the range $[-1, 1]$. The vector with the centres of the bounded search space is defined as $\boldsymbol{v} = \frac{1}{2}(\boldsymbol{x}_L + \boldsymbol{x}_U)$. Assuming that $v_i \neq 0$, we can encode decision vector, $\boldsymbol{x}$, as

$$\boldsymbol{x} = \text{diag}(\boldsymbol{v})\boldsymbol{x}' + \boldsymbol{v},$$

where $\text{diag}(\boldsymbol{v})$ is the diagonal matrix with elements of vector $\boldsymbol{v}$ on the diagonal.

However, it is typical for the variables to have different sensitivity in different parts of the domain, and applying the above methods may not be helpful. In such a case, a non-linear transformation may help in preserving the sensitivity of a variable. One such transformation that attempts to match the impact of changes of similar magnitude across a search space is given as

$$x_i = x_{Li} + (x_{Ui} - x_{Li})x_i'^2, \quad x_i' \in [0, 1].$$

The following non-linear transformation may help preserve the sensitivity if a range of a decision vector component spans a few orders of magnitude:

$$x_i = 10^a 10^{x_i'(b-a)}, \quad x_i' \in [0, 1],$$

yielding values of x_i within the range of $[10^a, 10^b]$.

2.4 Optimization methods

This work focuses on methods for solving non-linear global optimization problems. These methods can be categorized as

- *local methods* and
- *global methods*.

A local method requires an initial point and searches for an optimum within the neighbourhood of that point. A global method does not require an initial point and searches for an optimum over the entire search space $\mathbb{F}$. There is no perfect method for all optimization problems. The specific application and goal determine what trade-offs can be taken. Global optimization problems with a convex objective function and constraints can be solved with a local method efficiently. Non-convex objective functions or constraints pose a greater challenge and offer a meaningful choice between applying local and global methods. If the goal is to improve or refine an existing design or decision, then a local method may be better suited for such application. The initial point is provided by the currently used design, and only the vicinity of this design is to be considered. However, to find a ground-breaking design, which may be entirely different from the current design, a global method is often necessary.

The general strategy for solving optimization problems is iterative [36, p.8]. Each iteration is driven toward regions potentially containing the optimum by previous iterations. At the beginning, a set of initial candidates is generated and evaluated. That brings in new information about the problem, and a new set of candidates is generated. The procedure is repeated until a set of termination criteria is met. This process is briefly shown in Algorithm 1.

Algorithm 1 Iterative optimization algorithm

$\boldsymbol{x}^* \leftarrow$ INITIAL-CANDIDATE
repeat
  $\boldsymbol{x} \leftarrow$ CHOOSE-CANDIDATE
  if $f(\boldsymbol{x}) \leq f(\boldsymbol{x}^*)$ **then**
    $\boldsymbol{x}^* \leftarrow \boldsymbol{x}$
  end if
until MET-TERMINATION-CRITERION
return $(f(\boldsymbol{x}^*), \boldsymbol{x}^*)$

This section discusses each part of this procedure. Section 2.4.1 discusses methods for determining the initial points that are used in conjunction with global optimization methods. Section 2.4.2 presents a few conditions used as termination criteria used with local and global

methods. Section 2.4.3 shows the diversity of available optimization methods and briefly discusses differences between methods that are related to this work. A few well-known methods, which are the subject of the analysis demonstrated in Chapter 4, are presented throughout Section 2.5.1 to Section 2.5.3.

2.4.1 Initial candidates

Before an optimization method starts generating candidate decision vectors, it needs a set of initial candidates, which are evaluated in the first iteration. The choice of initial candidates may have a significant influence on the result of solving an optimization problem. A source of these candidates is often one of:

- a predefined set,
- a pseudo-random number generator, or
- expert knowledge.

A predefined set of initial candidates is used mostly with deterministic methods. Two implementations that are presented later in this section normalize a parameter space to a unit hypercube and choose the centre of the hypercube to be an initial point.

Using candidates generated with a pseudo-random number generator is the most popular approach. The basic version of this approach initializes component i of a candidate decision vector with a random value that follows a uniform distribution over the interval $[x_{Li}, x_{Ui}]$. The procedure is performed for all n initial candidates. The PSO-based algorithms that are implemented in the framework that we presented in Chapter 3 use this approach.

This approach has a few variants that may improve the quality of the final solution. Consider two variants that modify the basic approach on two distinct levels. The first one, instead of the uniform distribution, uses a distribution that emphasizes a potential area of interest. This variant utilizes a priori knowledge about the problem, e.g., the location of a promising part of the search space. The second one generates a larger number of candidate decision vectors. Each of these candidates is evaluated and the subset comprising the p^* best candidates becomes the initial set. Bhattacharya [8] generates and evaluates $5p^*$ initial

candidates. The number is large enough to also build a representative approximation used by a surrogate method.

The quality of an initial set can be improved with a gradient-based method. Each randomly generated candidate undergoes a local optimization. The results of this process yield the final initial set.

Expert knowledge is another source of initial candidates. It consists of empirical knowledge, results published in papers, or intuition. Other algorithms or domain-specific expert systems may also be used.

2.4.2 Termination criteria

The responsibilities of termination criteria in an optimization method are determining if the current solution is acceptable, avoiding false convergence, detecting unreasonably slow progress and avoiding unnecessary effort, checking availability of resources, and verifying input. The criterion that triggers termination is classified and communicated as either a success or a failure.

A successful termination means that an acceptable solution has been found within available resources. Testing whether an approximate solution is acceptable hinges on checking if sufficient conditions for optimality are met and whether the sequence of recent estimates converges. If no information about derivatives is available, then only few optimality conditions used in constrained optimization can be tested. Furthermore, convergence to the optimal solution is indistinguishable from either convergence to a non-optimal solution or a lack of progress.

It is important to note that a failure in termination does not imply an incorrect solution, nor does a successful termination imply a valid solution. A failure merely denotes that an acceptable solution has not been found; i.e., the current approximation is not within a tolerated distance to the assumed solution, which is rarely known. Ultimately, what is defined as the acceptable solution may not exist. In such a case, the optimization method evaluates new candidate vectors but never reaches an acceptable solution. The method is then terminated by satisfaction of another criterion.

Unfortunately, there is no set of termination criteria that is suitable for all optimiza-

tion problems [20, p.306]. According to characteristics listed by Boender et al. [4], good termination criteria depend on:

- a sample, i.e., values of an objective function with the corresponding locations,
- a problem, i.e., maximal use of information about a problem should be made,
- a method, i.e., properties of the used algorithm should be incorporated if possible,
- a loss, i.e., the gravity of the consequences of termination if the approximation is not good enough, and
- available resources and utilizing them with maximal efficiency.

An appropriate probabilistic model of the sampling information can be used to aid optimal stopping criteria. A probabilistic model is rarely known prior to sampling, but it can be built using statistical inference methods.

2.4.2.1 Optimality conditions

Both necessary and sufficient optimality conditions are defined in terms of derivatives. Given a function that is continuously differentiable in an open neighbourhood of $\boldsymbol{x}^*$, the first-order necessary conditions state that if $\boldsymbol{x}^*$ is a local minimum then:

$$\nabla f(\boldsymbol{x}^*) = \mathbf{0}. \tag{2.13}$$

This necessary condition, however, holds for any stationary point. The second-order necessary conditions state that if $\boldsymbol{x}^*$ is a local minimum of f and $\nabla^2 f$ exists and is continuous in an open neighbourhood of $\boldsymbol{x}^*$, then $\nabla f(\boldsymbol{x}^*) = 0$ and $\nabla^2 f(\boldsymbol{x}^*)$ is positive semi-definite; i.e., for any $\boldsymbol{p} \neq \mathbf{0}$,

$$\boldsymbol{p}^T \nabla^2 f(\boldsymbol{x}^*) \boldsymbol{p} \geq 0. \tag{2.14}$$

For any point $\boldsymbol{x}^*$ such that the first-order necessary conditions are met and $\nabla^2 f(\boldsymbol{x}^*)$ is positive definite, i.e., for any $\boldsymbol{p} \neq \mathbf{0}$,

$$\boldsymbol{p}^T \nabla^2 f(\boldsymbol{x}^*) \boldsymbol{p} > 0, \tag{2.15}$$

then $\boldsymbol{x}^*$ is a local minimum. These conditions are sufficient.

2.4.2.2 Testing convergence

Satisfaction of the following tests on a series of decreasing approximate solutions denotes a convergence of the series and thus a successful termination. Experiments included in Chapter 4 use a condition on the relative error of the form

$$\begin{cases} f(\boldsymbol{x}) - (1+\epsilon)f^* \leq 0, & \text{if } f^* \geq 0, \\ f(\boldsymbol{x}) - (1-\epsilon)f^* \leq 0, & \text{otherwise,} \end{cases} \tag{2.16}$$

where $\epsilon = 1\text{e}{-8}$ and f^* is the known optimum.

Avoiding stagnation Global optimization methods can get trapped in local optima that are otherwise indistinguishable from global optima. After a series of solutions converges, an optimization is further conducted for a number of iterations. With each of these additional iterations, the probability of further improvements decreases. In population-based algorithms, to limit the unnecessary computations but also not to miss a better optimum after a tolerable approximation has been found, the following termination criteria can be imposed for that stage of optimization:

- no improvement over the last n iterations,
- average population fitness values does not change,
- standard deviation of population fitness values does not decrease or drops below a tolerance.

Stagnation can cause termination or a change in the solver's strategy, i.e., to switch it from exploitation to exploration. Exploitation in the context of global optimization refers to the greedy strategy of changing a region of interest only if it is better than the current one. Exploration is another strategy wherein a region of interest may be changed to another region of a search space even if the new region shows little chance of an improvement. Global optimization algorithms often use both of these strategies, which are then labeled as a *local phase* and a *global phase*, respectively.

2.4.3 Categories of methods

Optimization algorithms are categorized according to many criteria, e.g., assumptions they make about an optimization problem, information they require, mechanisms they use or avoid using internally, use of random variables, or inspiration behind a method. A classification considers a particular aspect of a method and as a result a concrete method lies at an intersection of a few classes. For example, Newton's method for optimization:

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_n - [\nabla^2 f(\boldsymbol{x}_n)]^{-1} \nabla f(\boldsymbol{x}_n)$$

is a deterministic, local, and gradient-based method. This subsection presents categories relevant to the methods described and analyzed in following subsections of this chapter.

Strong and weak methods This notion classifies methods with respect to the strength of a relation between a method and a problem. A *strong* method is customized to solve a specific problem. Such methods make strong assumptions about a problem and can exploit its properties. A *weak* method can be more widely applied, however, with lesser expectations about its efficiency. The examples presented in Chapter 4 focus on application of weak methods to non-linear optimization problems.

Gradient methods Exploiting information about derivatives is a base for many optimization algorithms, e.g., Newton's method or quasi-Newton methods. These methods are efficient, and if such information can be obtained then a gradient-based method is preferred [27]. Problems that do not make any assumptions about derivatives may be solved with different algorithms.

A solution to an optimization problem can be defined in terms of derivatives. If first-order necessary conditions are met, then a candidate decision vector may be a local optimum. Finding such candidate decision vectors reduces to finding roots of $\nabla f(\boldsymbol{x})$.

Derivative-free methods Derivative-free methods are weak methods that require no prior information about an objective function and thus are applicable to black-box objective functions. Derivative-free methods can be further divided into two more-specific groups:

- methods based on interpolation and

- methods without the notion of derivative.

In this work, we focus on the second group of methods, also known as *direct search* methods. Hooke and Jeeves define them as methods that have an order relation between two decision vectors [22]. Originally, direct search referred to what is now known as *pattern search*. Torczon et al. [45] further discuss definitions of direct search. Direct search methods do not use information about derivatives to select new candidates. Instead, these methods use only comparisons.

Direct search methods were proposed and started being used in the 1960s. Due to the lack of a coherent mathematical analysis, direct methods lost the focus of the mathematical programming community by early 1970s; however, the methods remained in use. Parallel computing brought a significant momentum to many algorithms that had fallen out of favour before and revived analysis of direct search methods [26].

Stochastic methods Stochastic methods involve randomness in solving an optimization problem. The randomness can be present at any stage of an optimization process. However, if an objective function or any constraint contains a stochastic component, then they comprise a stochastic optimization problem. The class of stochastic methods is numerous. Some of the most common methods are presented in Figure 2.1.

Evolutionary methods Evolutionary programming was first proposed by L. Fogel in a series of studies conducted in the early 1960s. The idea was to have a population of competing algorithms whose performance was improved through a simulated evolution of their logic. In principle, this description also applies to genetic algorithms and, in fact, the border between genetic algorithms and evolutionary algorithms is vague. The basic difference between the two classes is that genetic algorithms are classified as problem independent, whereas evolutionary algorithms require adjustments to a particular problem [33, pp.289–290]; e.g., evolutionary algorithms can use some knowledge for discriminating candidates.

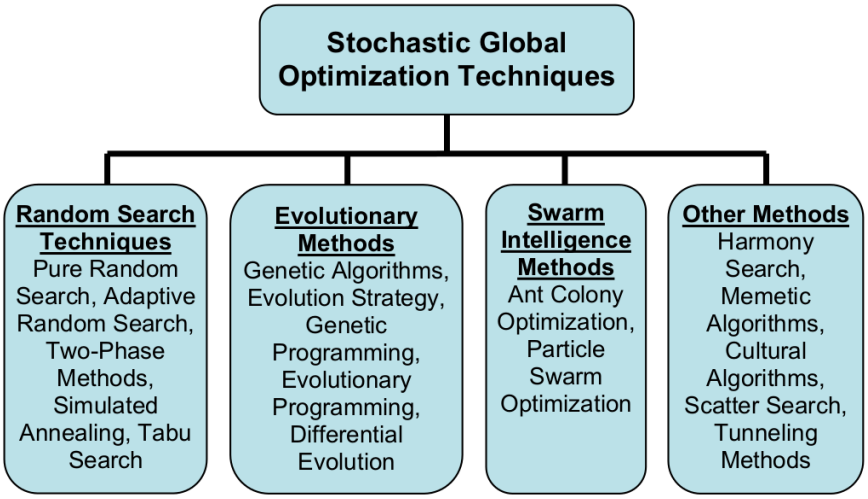

Figure 2.1: Classification of stochastic global optimization methods

Swarm Intelligence Swarm intelligence methods are a class of nature-inspired methods based on the *evolutionary computation* framework. In these methods, a population of *agents* cooperates over a series of iterations to attain an acceptable solution [37]. The communication between agents is defined by the topology of a communication network. Only the closest nodes in the network can communicate. Swarm intelligence also refers to the problem-solving behaviour that emerges from the interactions of such agents [17].

Swarm intelligence methods use a population of agents and follow four principles [34]:

- proximity: population carries elementary space-time computations,
- quality: population responds to quality factors in environment,
- diverse response: population does not work in narrow channels, and
- stability (or adaptability): environmental changes do not change the population mode every time.

2.5 Selected methods

This section demonstrates the optimization methods (or algorithms) that provide a basis for our research. All of them are global and derivative-free methods. Initially, DIRECT and

Grey Wolf Optimization algorithms are presented. Next, the research of Particle Swarm Optimization (PSO) is discussed, and then we move on to explore the selected seven PSO variants.

2.5.1 DIRECT algorithm

`DIRECT`, due to Jones et al. [24], is a global optimization method that is deterministic, i.e., it uses no randomness for selecting new decision vectors for evaluations. It is an example of a direct search method; but its name comes from the description of the method: `DIvide RECTangles`. The algorithm is a generalization of Shubert's algorithm [43], and it has improvements over some of its predecessor's drawbacks. It addresses problems with emphasis on global search and on an expensive initialization phase on multidimensional optimization problems. Despite being conceived two decades ago, the algorithm is a subject of an ongoing research [31].

The algorithm works on a normalized search space, $\mathbb{D}$, that forms a hypercube. It evaluates an objective function at the centroid of the hypercube and at two other points for each dimension of $\mathbb{D}$. The coordinates of the two points share all but one value with the centre point. This one coordinate differs by one third of the hyperrectangle's length in the given dimension and is both added to and subtracted from the value of the centre point. The results of these evaluations determine the scheme that is used for dividing the hypercube into hyperrectangles. The results are also added to a data structure that keeps track of all hyperrectangles and their potential for containing an optimum. In the next phase, the method selects a subset of *potentially optimal hyperrectangles* from that data structure. At this stage, the first iteration is complete. The next iteration repeats all but the first of the above steps for each of the selected hyperrectangles.

Two aspects of the above procedure require clarification. The first is the partitioning scheme. A potentially optimal hyperrectangle is divided into thirds along the dimensions that have the maximum side length. If there are multiple dimensions matching this criterion, then the dimension along which the minimal value is located is used. Figure 2.2 shows typical partition schemes in two and three dimensions. The three-dimensional example uses colours to encode values: green denotes a low value, blue denotes a high value, and red denotes a

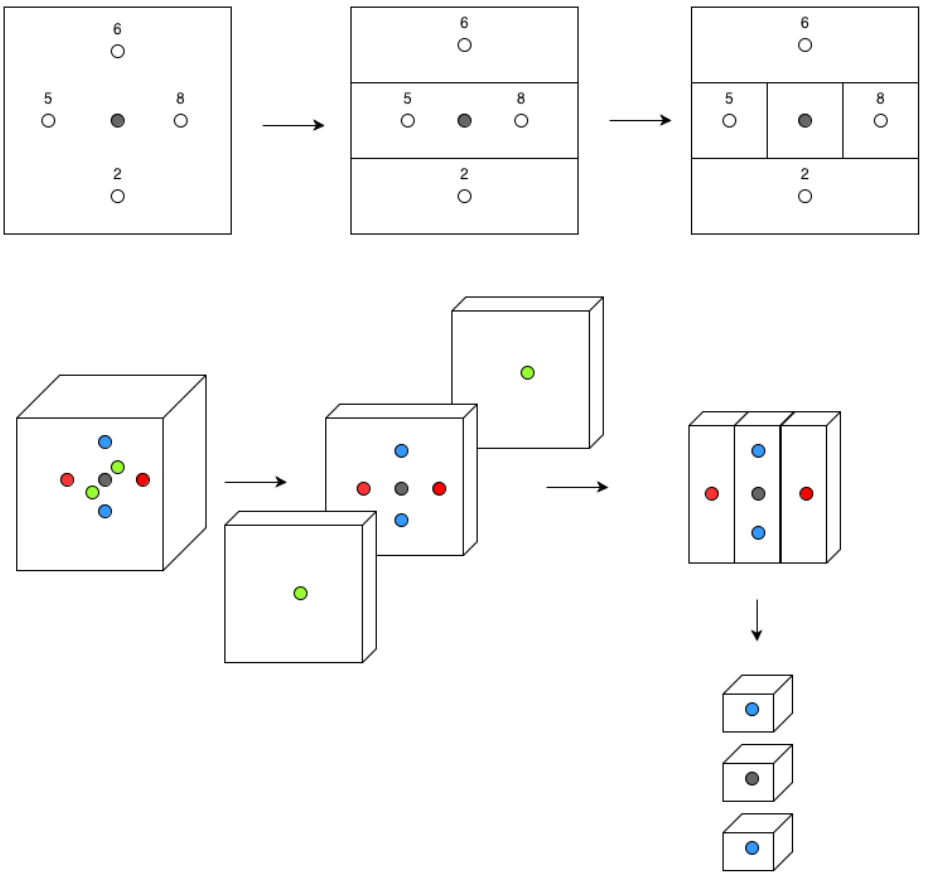

Figure 2.2: DIRECT space partitioning scheme in 2D and 3D

value in between the other two. Formally, the method divides a hyperrectangle along the dimension i that minimizes w_i such that

$$w_i = \min\{f(\boldsymbol{c} + \delta_i \hat{\boldsymbol{e}}_{\boldsymbol{i}}), f(\boldsymbol{c} - \delta_i \hat{\boldsymbol{e}}_{\boldsymbol{i}})\}. \tag{2.17}$$

where $\boldsymbol{c}$ is the centroid of a hyperrectangle, δ_i is the third of the length of the dimension i, and $\hat{\boldsymbol{e}}_{\boldsymbol{i}}$ is a unit vector in that dimension. This approach favours the hyperrectangle that contains the lowest of evaluated values to be explored more thoroughly. In the next step, the method divides the longest dimension of the selected hyperrectangles into thirds. The second widest dimension is divided next. This scheme continues until the selected hyperrectangle is divided in all dimensions.

The second of the aspects is identification of the potentially optimal hyperrectangles. This part most resembles Shubert's algorithm. Let $\boldsymbol{c}_i$ be the centre point of the hyperrectangle i, and $\boldsymbol{d}_i$ be the maximal distance between $\boldsymbol{c}_i$ and the sides of the hyperrectangle i. The

potentially optimal hyperrectangle j satisfies the following conditions:

$$f(\boldsymbol{c}_j) - \tilde{K} d_j \leq f(\boldsymbol{c}_i) - \tilde{K} d_i, \quad \text{for all } i = 1, 2, \ldots, m, \tag{2.18}$$

$$f(\boldsymbol{c}_j) - \tilde{K} d_j \leq f^* - \epsilon |f^*|, \tag{2.19}$$

for constant $\epsilon > 0$ and rate-of-change $\tilde{K} > 0$. Identifying such rectangles reduces to finding a lower convex hull of points that represent hyperrectangles, where the x-axis represents d_j values and the y-axis represents a value at the point $\boldsymbol{c}_j$.

`DIRECT` does not have a distinct global and local phase. Instead, a mix of both strategies is used over the iterations of this method. An iteration comprises of evaluating the objective function at the centres of a few hyperrectangles. The size of a hyperrectangle, in relation to the size of the entire domain, determines the bias toward a global phase or a local phase.

The `DIRECT` algorithm that is described in the original paper [24] has been implemented in Python and integrated into `pythOPT`, which is the framework presented in Chapter 3. VTDirect95 is a widely used `FORTRAN` package that implements a modified version of the algorithm. That implementation is robust, and it has been used to solve optimization problems in multiple disciplines. The source code of VTDirect95 is publicly available.

2.5.2 Grey Wolf Optimization

Grey Wolf Optimization (GWO) is a stochastic optimization algorithm that implements leadership hierarchy and the hunting mechanism of the grey wolf (Canis lupus) [35]. It evolves a population of agents, in this context referred to as *wolves*, that represent candidate solutions chosen according to a model of the behaviour of a pack of grey wolves. The wolves form a hierarchy with the fittest solutions at the top, and the least fit at the bottom. The locations of the three fittest wolves $(\boldsymbol{x}_\alpha, \boldsymbol{x}_\beta, \boldsymbol{x}_\gamma)$ approximate the location of the global optimum. All wolves in the pack move towards that approximated location using additionally stochastic and linearly decreasing factors. The algorithm is based on three ideas that are presented in the following paragraphs.

Encircling prey The encircling step is modeled as

$$\boldsymbol{x}_{n+1} = \boldsymbol{x}_g - \boldsymbol{v} \circ (\boldsymbol{c} \circ \boldsymbol{x}_\alpha - \boldsymbol{x}_n), \tag{2.20}$$

where n represents the iteration number, $\boldsymbol{x}_n$ is the position of a wolf, $\circ$ denotes the component-wise vector multiplication, and $\boldsymbol{x}_g$ represents an approximated location of the global optimum. Vectors $\boldsymbol{v}$ and $\boldsymbol{c}$ are stochastic coefficients that control the behaviour of the algorithm as it shifts from exploration to exploitation. They are defined as

$$\boldsymbol{v} = 2w(n)\boldsymbol{\epsilon}_1 - w(n), \tag{2.21}$$

$$\boldsymbol{c} = 2\boldsymbol{\epsilon}_2, \tag{2.22}$$

where $w(n)$ represents a coefficient that linearly decreases from two to zero, and $\boldsymbol{\epsilon}_1$ and $\boldsymbol{\epsilon}_2$ are random vectors in the range $(0, 1)$.

Hunting The approximated location of the global optimum is determined by the locations of the three fittest wolves $\boldsymbol{x}_\alpha, \boldsymbol{x}_\beta, \boldsymbol{x}_\gamma$. The rest of the population moves toward the randomized combination of vectors directed toward these three fittest locations. The location of each wolf is updated once per iteration as follows

$$\boldsymbol{x}_{n+1} = \frac{1}{3} \sum_{\psi \in \{\alpha,\beta,\gamma\}} \boldsymbol{x}_{n,\psi} - \boldsymbol{v} \circ (\boldsymbol{c} \circ \boldsymbol{x}_{n,\psi} - \boldsymbol{x}_n), \tag{2.23}$$

where $\boldsymbol{x}_n$ denotes the location of a wolf in iteration n, and $\boldsymbol{x}_{n,\alpha}, \boldsymbol{x}_{n,\beta}, \boldsymbol{x}_{n,\gamma}$ are the locations of the first, the second, and the third best wolves in that iteration, respectively.

Attacking Attacking in the context of the GWO is the exploitation phase of the algorithm. GWO uses a linearly decreasing factor $w(n)$ that is a function of the current iteration number. As the algorithm progresses the value $w(n)$ decreases from two to zero. If $|w(n)| < 1$, then wolves move toward the approximated position of the global optimum; otherwise, they move away from that position. Because the attacking is the final step of a hunt, when the final value of $w(n)$ is zero, this assures that wolves only explore their vicinity regardless of any other wolves in the pack.

2.5.3 Particle Swarm Optimization

Particle Swarm Optimization (PSO) is a stochastic method for optimizing non-linear functions. PSO evolved from what was initially a simplified social model and developed into a

method for optimization of continuous non-linear functions. At its foundation lies the hypothesis that social sharing of information gives an evolutionary advantage [25]. The general idea of the method is to evolve a number of agents over a number of iterations. In the context of PSO-based methods, agents are called *particles*. Each particle is driven through a search space toward a randomized combination of its individual best position and a socially shared best position. More specifically a particle's velocity and position evolve as follows:

$$\boldsymbol{v}_i = \boldsymbol{v}_i + c\epsilon_1(\boldsymbol{p}_i - \boldsymbol{x}_i) + c\epsilon_2(\boldsymbol{p}_g - \boldsymbol{x}_i) \tag{2.24}$$

$$\boldsymbol{x}_i = \boldsymbol{x}_i + \boldsymbol{v}_i, \tag{2.25}$$

where $\boldsymbol{x}_i$ denotes position of the particle i, $\boldsymbol{v}_i$ is its velocity, c is a constant factor, $\boldsymbol{p}_i$ and $\boldsymbol{p}_g$ are individual and global best positions, and ϵ_1 and ϵ_2 are random variables that are uniformly distributed in the $(0, 1)$ range.

Over the years following the inception of PSO, the research community devised many improvements to the original method. PSO became an ambiguous term without a precise baseline that could be used in comparison against new methods. The improvements rendered the original version unrepresentative, and newer versions were being used in comparisons. This issue was addressed by Bratton and Kennedy [11] who defined the standard PSO (SPSO) variant. The standard incorporates the major improvements that have been introduced to PSO before publication of the paper in 2007. These improvements include introduction of communication topologies within a swarm and introduction of parameters for inertia weight and constrictions. Moreover, the standard proposes a methodology for evaluating and comparing performances between methods.

The original paper [25] presents the organic evolution of the method through a series of shifts in the paradigm that lie at the foundation of PSO. The first paradigm uses nearest neighbour and Euclidean distance to determine which particles communicate their velocities. The final paradigm of the paper uses the concept of *global neighbourhood*; i.e., each particle communicates with all the others. Subsequent papers investigated other topologies more closely. Any topology that is different from the global neighbourhood is a *local topology*. The SPSO uses the *ring topology*; i.e., each particle communicates with exactly two other particles (see Figure 2.3). This approach makes the convergence rate lower and thus better

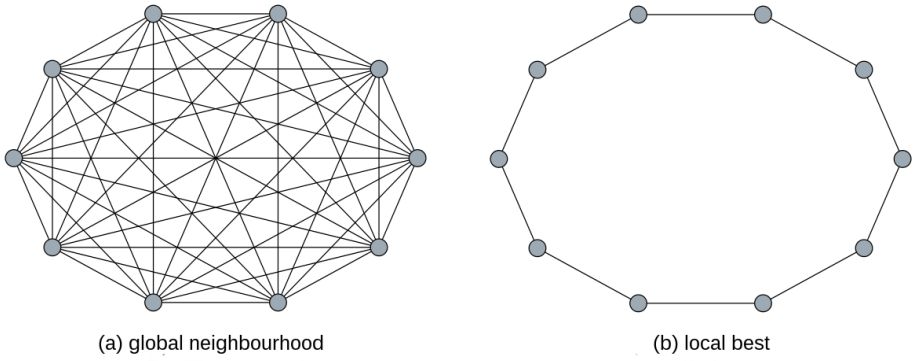

Figure 2.3: Communication topologies within PSO

prevents premature convergence, resulting in better quality of the final approximation at the expense of a higher number of objective function evaluations. The ring topology is part of the SPSO definition. In the general case, both approaches should be evaluated.

Early versions of the original PSO algorithm had a possibility of particle velocities quickly rising to unreasonably high values (state of *explosion*) that caused instability. The original version solves this issue by introducing a parameter v_{max} that clamps the maximal velocity of a particle and partially solves the problem. However, choosing this parameter appropriately may be difficult. Shi and Eberhart [42] proposed to replace v_{max} with an *inertia weight* parameter, w, that is applied directly to the velocity of a particle. They also proposed to use two constants, c_1 and c_2, to differentiate emphasis on local and global phase:

$$\boldsymbol{v}_i \;=\; w\boldsymbol{v}_i + c_1\epsilon_1(\boldsymbol{p}_i - \boldsymbol{x}_i) + c_2\epsilon_2(\boldsymbol{p}_g - \boldsymbol{x}_i). \tag{2.26}$$

Inertia, w, is a positive constant or a function. A positive linearly decreasing function provides a balance between global and local search.

The approach used in SPSO uses a *constriction factor* χ:

$$\boldsymbol{v}_i \;=\; \chi\left(\boldsymbol{v}_i + c_1\epsilon_1(\boldsymbol{p}_i - \boldsymbol{x}_i) + c_2\epsilon_2(\boldsymbol{p}_g - \boldsymbol{x}_i)\right). \tag{2.27}$$

It is a special case of the method proposed along with the inertia weight. Let $\varphi = c_1 + c_2$ from Eq. 2.27; then the value of χ is calculated as follows:

$$\chi = \frac{2}{|2 - \varphi - \sqrt{\varphi^2 - 4\varphi}|}.$$

Both approaches were developed around the same time; however, the paper introducing the constriction factor was published four years later by Clerc and Kennedy [13]. Their analysis found that for $\varphi < 4$ the particles take on a spiral trajectory around found minima, and the trajectory is not guaranteed to converge, whereas for $\varphi > 4$, the convergence is quick and guaranteed. According to [11], most implementations of constricted PSO use $\varphi = 4.1, c_1 = c_2 = 2.05$, resulting in $\chi \approx 0.72984$.

The above improvements add parameters to the original method that can significantly improve the method's efficiency. A number of studies have been conducted to find an optimal set of parameters for the algorithm, and a few promising configurations have been devised. Further information can be found in [46] and [19, p.60]. Bratton and Kennedy proposed the Standard PSO (SPSO) algorithm that utilizes some of these studies to provide a reference PSO method to be used in benchmarks [11]. The algorithm is defined in terms of the Eq. 2.27 where $\chi \approx 0.72984$ and $c_1 = c_2 = 2.05$. It uses $p = 50$ particles and the ring topology. We show in Chapter 4 that it, in fact, performs best out of the presented PSO variants in nearly all test cases, what makes it a good reference method.

2.5.3.1 PSO Variants

The previous section presents SPSO, a variant that is used as the reference PSO method. This subsection describes other PSO variants that are implemented in `pythOPT`. The `pythOPT` problem-solving environment is the subject of Chapter 3. For a brief overview of over one hundred PSO variants and a taxonomy of PSO methods, we refer to [41]. The paper indicates the wide spectrum of approaches to improving performance of PSO-based methods.

Global Best PSO Global Best PSO (GBPSO) is the original PSO method as introduced by Kennedy and Eberhart [25]. The equations

$$\boldsymbol{v}_i = \boldsymbol{v}_i + 2\epsilon_1(\boldsymbol{p}_i - \boldsymbol{x}_i) + 2\epsilon_2(\boldsymbol{p}_g - \boldsymbol{x}_i), \tag{2.28}$$

$$\boldsymbol{x}_i = \boldsymbol{x}_i + \boldsymbol{v}_i, \tag{2.29}$$

govern the evolution of a swarm. The stochastic factors are multiplied by two, which makes the average weight for the *social* and the *cognition* part equal to one. As a result, particles

statistically contract the swarm to the current global optimum [42]. If the velocity factor was not present, then the behaviour of such a method would resemble a local search.

GBPSO has two problems that need to be addressed in an implementation. The first of them is a possibility of reaching a state of *explosion* in which velocities approach unreasonably high values. To mitigate this behaviour, the original method employs a parameter, v_{max}, that defines a velocity limit imposed on all particles. The second is the particles that move outside of the search space. The simplest strategy for dealing with them is to leave their velocity and infeasible position unchanged and skip the evaluation step [11].

Decreasing Weight PSO The Decreasing Weight PSO (DWPSO) introduces the inertia weight parameter, $w(n)$ (see Eq. 2.32) that linearly decreases the velocity magnitude in the following

$$\boldsymbol{v}_i = w(n)\boldsymbol{v}_i + 2\epsilon_1(\boldsymbol{p}_i - \boldsymbol{x}_i) + 2\epsilon_2(\boldsymbol{p}_g - \boldsymbol{x}_i), \tag{2.30}$$

$$\boldsymbol{x}_i = \boldsymbol{x}_i + \boldsymbol{v}_i, \tag{2.31}$$

The original analysis found that the method on average has the best performance for a constant w in the range of $[0.9, 1.2]$ [42]. A higher value of the parameter decreases dependence on the initial position of particles in a swarm and promotes exploration of the search space; i.e., it promotes a global search.

The second improvement employs this relation between w and the character of a search to gradually turn the initial global search into a local search. The original variant linearly decreases the value of w with respect to the number of the current iteration as in:

$$w(n) = w_s - (w_s - w_e)\frac{n}{N-1},$$

where N is the maximal number of iterations, n is the current iteration number, and w_s and w_e are the initial and final values of the parameter, respectively. The analysis of this variant found that linearly decreasing w from 1.4 to 0.0 outperformed using any of the previously tested constant values.

Time-Varying Acceleration Coefficients PSO Time-Varying Acceleration Coefficients PSO (TVACPSO) [38] is an improvement on the DWPSO. In addition to introducing the

inertia weight, DWPSO uses two other parameters c_1, c_2 that replace the original constants of 2. They remain constant in the original DWPSO method. TVACPSO linearly decreases these parameters according to

$$\boldsymbol{v}_i = w(n)\boldsymbol{v}_i + c_1\epsilon_1(\boldsymbol{p}_i - \boldsymbol{x}_i) + c_2\epsilon_2(\boldsymbol{p}_g - \boldsymbol{x}_i), \tag{2.32}$$

where

$$c_1(n) = c_{1s} - (c_{1s} - c_{1e})\frac{n}{N-1}, \tag{2.33}$$

$$\text{and} \quad c_2(n) = c_{2s} - (c_{2s} - c_{2e})\frac{n}{N-1}. \tag{2.34}$$

The results presented in [38] show that the best strategies are to decrease c_1 from 2.5 to 0.5 and to increase c_2 from 0.5 to 2.5.

Chaotic Descending Inertia Weight PSO Chaotic Descending Inertia Weight PSO (CDIWPSO) and Dynamic Adaptive PSO (DAPSO) are two variants of the DWPSO algorithm [5]. CDIWPSO [18] avoids getting stuck in local optima by using chaotic numbers, z_k, that are sensitive to initial conditions:

$$z_{k+1} = \mu z_k(1 - z_k),$$

where $\mu = 4$. This equation generates values in the range $(0, 1)$ if z_0 is in $(0, 1)$ and $z_0 \notin \{0, .25, .5, .75, 1\}$

The function that defines inertia weight $w(n)$ is given as:

$$w(n) = (w_e - w_s)\frac{N-n}{N} + w_s z_{k+1},$$

where w_s and w_e represent limits on the value of the inertia parameter, N is the limit of iterations, and n is the current iteration number. The rest of the method is the same as DWPSO.

Dynamic Adaptive PSO Dynamic Adaptive PSO (DAPSO) [50] is an algorithm that uses DWPSO as a framework. DWPSO suffers from the premature convergence due to the lack of momentum of particles in the exploitation state [5]. DAPSO, similarly to DWPSO,

modifies the value of the inertia weight, and additionally, it factors in the diversity of solutions in the population. Experiments presented in [50] show that DAPSO is more effective than DWPSO.

The function that defines inertia weight $w(n)$ is given as

$$w(n) = w_b + (w_e - w_b)\, F(n)\phi(n),$$

where w_b and w_e represent constant minimal and constant maximal value of inertia parameter. $F(n)$ and $\phi(n)$ are factors that depend on the distribution of solutions within the population. The adjustment function ϕ is defined as

$$\phi(n) = \mathrm{e}^{-\frac{n^2}{2\sigma^2}},$$

where $\sigma = N/3$ and N represents the limit on total number of iterations. The diversity function $F(n)$ is defined as:

$$F(n) = 1 - \frac{2}{\pi}\arctan(E),$$

where the group fitness, E, is

$$E = \frac{1}{p}\sum_{i=1}^{p}\left(f(\boldsymbol{x}_i) - f_{avg}\right)^2,$$

where p is the number of particles and f_{avg} is the average fitness value of the population.

Global Convergence PSO If a particle updates a swarm's best position, then its velocity for the next iteration depends solely on its previous velocity. If that velocity is close to zero, then the particle gradually converges to that (possibly suboptimal) solution while attracting other particles. This may lead to a premature convergence of a swarm, or so-called *stagnation*. Global Convergence PSO (GCPSO) introduces a *search radius* parameter, ρ, that scales a random factor that is added to the velocity of the particle that most recently updated swarm's best position [47]. This strategy mitigates the stagnation phenomenon and increases local convergence.

GCPSO updates the velocity, $\boldsymbol{v}$, of the particle that most recently updated the best position in a population as follows

$$\boldsymbol{v} = \boldsymbol{p}_g - \boldsymbol{x} + w\boldsymbol{v} + \rho(1 - 2\epsilon),$$

where ϵ is a uniformly distributed variable within range $[0, 1]$ and w is a constant inertia weight. The initial value of ρ is 1. It is updated according to the following formula

$$\rho(n+1) = \begin{cases} 2\rho(n), & \text{if } \sigma(n+1) \geq \sigma_c, \\ \frac{1}{2}\rho(n), & \text{if } \varphi(n+1) \geq \varphi_c, \\ \rho(n), & \text{otherwise,} \end{cases}$$

where σ and φ represent numbers of consecutive successes and failures:

$$\sigma(n+1) = \begin{cases} 0, & \text{if } \varphi(n+1) > \varphi(n), \\ \sigma(n)+1, & \text{otherwise,} \end{cases}$$

$$\varphi(n+1) = \begin{cases} 0, & \text{if } \sigma(n+1) > \sigma(n), \\ \varphi(n)+1, & \text{otherwise,} \end{cases}$$

and σ_c and φ_c are threshold values. For problems with a high number of dimensions, $\sigma_c = 15$ and $\varphi_c = 5$ are recommended [47].

Chapter 3

The pythOPT PSE

The software package `pythOPT` is a problem-solving environment (PSE) for solving optimization problems and evaluating the performance of optimization methods. A PSE is a software framework that provides the computational facilities necessary to solve a target class of problems [39]. The `pythOPT` PSE provides an interface for defining and solving optimization problems and an implementation of the optimization methods described in Chapter 2. It also provides tools for evaluating the performance of optimization methods via a set of popular benchmark problems and a method for randomizing these problems. This combination of functionality allows for a rapid evaluation of new methods and an efficient adjustment of already existing ones for a given set of problems.

This chapter is broken into five sections, which are structured to follow the IEEE1016-2009 standard [3] for Software Design Descriptions (SDD). Section 3.1 lists the functional and non-functional requirements expected of the PSE, and it presents the PSE's actors and use cases. Subsequent sections communicate the design of `pythOPT`. Section 3.2 presents services provided by `pythOPT` with reference to their more detailed description. Section 3.3 delineates the structure of `pythOPT` and dependencies between its components. Section 3.4 focuses on the data types and the interfaces defined in the `pythOPT` library. Section 3.5 describes the design of the persistence layer of `pythOPT`. Section 3.6 describes the mechanism for distributing work and performing it concurrently. Section 3.7 presents in more detail how optimization methods are abstracted as software objects.

3.1 Requirements

The `pythOPT` PSE users can generally be divided into two groups: those interested in solving optimization problems and those interested in comparing optimization methods. The first group requires an interface to define an optimization problem and a method that provides a solution to the problem. The overhead that arises from using the interface should be minimal. The second group requires an ability to select solvers and adjust their behaviour through available parameters or by replacing parts of a solver. Also, it requires an interface to build a test suite that consists of a collection of solvers, a collection of problems, and a metric for interpreting results of using these solvers on all problems in the suite. The `pythOPT` PSE provides all the necessary elements to complete both of these use cases.

These use cases and feature requests that arose in applying `pythOPT` to problems presented in Chapter 4 resulted in the following list of functional and non-functional requirements for `pythOPT`. The functional requirements include an ability to

- define and solve unconstrained optimization problems,
- add modifications to existing PSO-based methods,
- use the solvers VTDirect95 and LGO,
- perform a robust comparison of optimization methods, and
- reproduce solutions acquired through the framework.

The non-functional requirements include

- scalability of performing benchmarks,
- maintainability of code base, and
- a simple interface with little boilerplate code.

3.2 Use cases

Users who want to solve an optimization problem import `pythOPT` into their script and define an instance of a problem by providing a reference to an objective function and information about bound constraints of a search space. Subsequently, they pass the name of a selected solver as an argument to a method of the problem instance. After the solver terminates, it returns the computed solution, which is returned to a calling function.

Users who perform research in optimization use a different work flow. They often solve one problem multiple times with the same solver, but they adjust the solver's settings before each use. Then, they analyze how these adjustments influence the quality of a resulting solution. Research might involve comparing multiple solvers on the same problem, a set of problems, or in the case of this study, a set of benchmark problems. For these analyses to be meaningful, the resources allocated to each of the solvers, such as a limit on the number of objective function evaluations, time constraint, etc., should be similar.

3.3 `pythOPT` package structure

The structure of `pythOPT` is decomposed into five packages; see Figure 3.1. The package `pythopt.engine` contains modules that define interfaces, data types, and common operations. An appropriate interface must be implemented by an object, for it to be compatible with the rest of the system, e.g., the `pythopt.engine.solver.Solver` interface must be implemented in all optimization methods. Some data structures are used in many solvers, e.g., the `pythopt.engine.geometry.Space` class abstracts a search space and provides methods for manipulating a position in that search space including enforcement of boundaries. Other modules are responsible for reading `pythOPT`'s configuration file, managing files, compiling objectives from source, and other common operations.

The `pythopt.engine.solution.Solution` instance holds information about any finished optimization process. This includes the identified minimum and its position, all the information provided by a solver, e.g., the number of times `Objective` was evaluated, the exit code of a solver, complete output of a solver (for `FORTRAN` solvers), and solver settings.

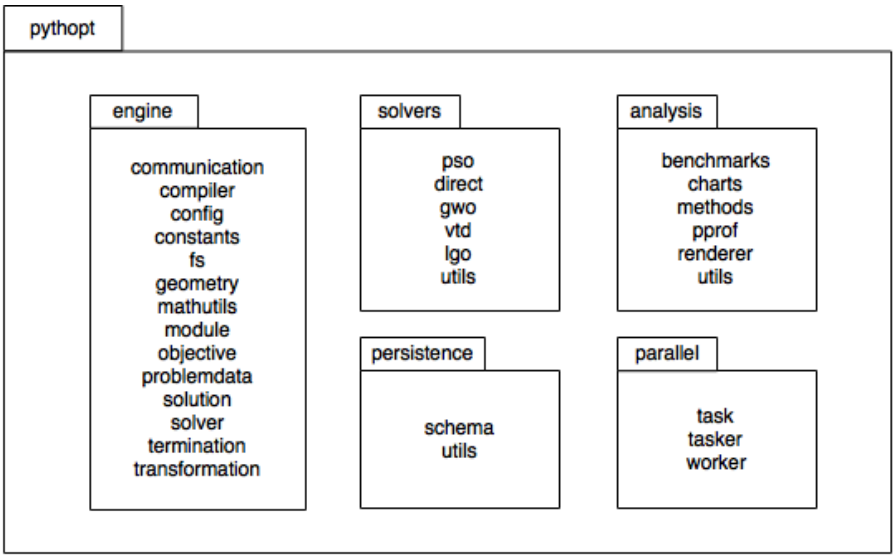

Figure 3.1: `pythOPT` package structure

Modules implementing optimization methods are located in the `pythopt.solvers` package. Each method must implement the `Solver` interface. The `pythOPT` PSE provides implementations of the algorithms described at the end of Chapter 2; i.e., seven PSO-based algorithms, GWO, and DIRECT.

The `pythopt.solvers.pso.PSO` is decomposed and implemented using the template method pattern. The PSO solver base class implements the original algorithm as described in [25]. Implementations of the variants of that algorithm override the template methods.

There exist many software packages that implement optimization methods, e.g., `SciPy`, `pyOpt`. To use a method implemented in another package, a user must implement an adapter for that method. The `pythopt.solvers` package contains two exemplary adapters for `FORTRAN` solvers: `LGO` and VTDirect95. LGO (Lipschitz Global Optimization) is a closed-source commercial solver suite [1]. It provides three methods for global optimization and one for local optimization:

- adaptive partition and search (branch-and-bound),
- adaptive global random search (single-start),
- adaptive global random search (multi-start), and

- constrained local optimization (reduced gradient method).

The adapter for this solver is located in the `pythopt.solvers.lgo` module. The other `FORTRAN` solver, VTDirect95, is the original implementation of the DIRECT algorithm [2], which is described in Chapter 2. The adapter for this implementation is given in the `pythopt.solvers.vtd` module.

The package `pythopt.analysis` contains modules for analyzing the performance of optimization methods and definitions of popular benchmark functions. It also contains utilities for randomizing behaviour of the benchmark functions to prevent bias toward known solutions. The available performance analysis methods are presented in Chapter 4. These methods include

- comparisons based on number of wins, draws, and losses against other solvers,
- performance profiles.

The package `pythopt.persistence` contains modules that provide operations on the persistence layer of `pythOPT`. The layer is implemented with a relational database (DB). The `pythopt.persistence.schema` module defines the schema for use by the DB, and the `pythopt.persistence.utils` module provides helper functions to access the DB. The package also contains initialization scripts to populate the DB with predefined solvers and analyses.

The package `pythopt.parallel` contains modules that are used to create, distribute, and perform computations (instances of solving an optimization problem using a specific solver configuration).

3.4 The logical viewpoint

This section presents relations between software classes in `pythOPT` and concepts in mathematical optimization. The most fundamental classes of the system are depicted in Figure 3.2. The rest of the section describes each of these classes.

The `ProblemData` class instances represent optimization problems and provide a `solve` method that initiates an optimization process and returns the computed solution. As men-

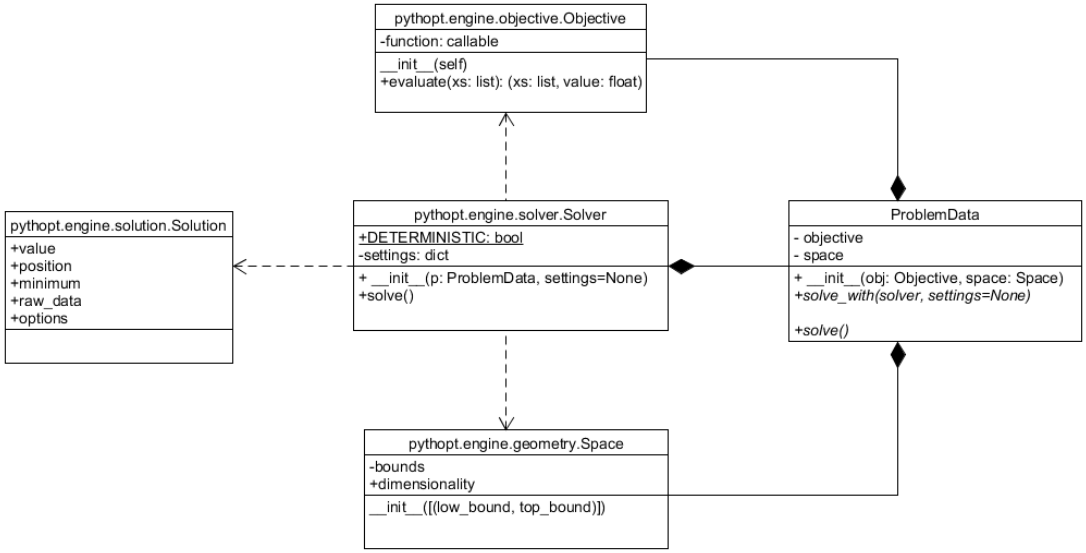

Figure 3.2: `pythOPT` core classes

tioned in Chapter 2, an optimization problem consists of a search space and an objective function. For a given problem, these concepts are represented by instances of `Space` and `Objective` classes, respectively. A `ProblemData` instance holds references to both of these instances.

An instance of the `Space` class represents a search space. The class provides methods for generating and modifying decision vectors. The decision vectors that are returned by these methods remain within bounds of a search space. The constructor takes a list of pairs to determine dimensionality and boundary constraints of a problem. The `CubeSpace`, a subclass of `Space`, is a special case that has the same bound constraints for all dimensions.

The `Objective` instance represents an objective function. The class has a public method `Objective.evaluate` that evaluates an objective function. Because the method may return a value for an argument that is different from the original argument, the method returns a pair consisting of a value and an argument for which an evaluation was made.

The `FortranObjective`, a subclass of `Objective`, stores all the information necessary to compile and execute code of an objective function implemented in `FORTRAN`. The constructor requires a list of files or a single filename that contains the source of the objective.

Each optimization method implemented in `pythOPT` implements the `Solver` interface. A solver's constructor accepts an optional dictionary of settings so that a behaviour of a solver

can be adjusted during instantiation. A call to `solver.solve` returns a `Solution` instance.

A `Solution` instance contains an answer and a dictionary of outputs from the solver that has been used to acquire the solution. It also contains a field with the original, unparsed output from the solver.

3.5 Data persistence

This section describes the schema of the `pythOPT` DB that is depicted in Figure 3.3. The schema provides an abstraction for representing an optimization problem, its problem space, a complete solver configuration, and a solution, and it provides an abstraction for binding these elements into a unit of computation.

The DB also provides an abstraction for binding a set of optimization problems, a set of solver configurations, and common termination criteria to an analysis instance. Together, these abstractions allow for a distributed evaluation of solver configurations and an organization of solver analyses.

The DB can be easily populated with problems that are available in `pythOPT` and were used in experiments described in Chapter 4. A distribution of `pythOPT` comes with a `Makefile` file that defines a target named `db`. The recipe for that target prepares a DB schema and initializes the DB to reproduce all results presented in this work, with the exception for the Rational Material Design project presented in Section 4.3.3.

3.6 Distributed workflow

The software package `pythOPT` allows for the solution of optimization problems concurrently. This feature is especially useful in performing performance analyses of multiple optimization methods. To generate data for an analysis, each of the analyzed optimization methods is run with multiple problems. In general, there is no data dependency that would prevent a user from running a solver over all problems or all solvers over all problems at the same time. The package exploits this property and uses multiple computers simultaneously.

The modules in the `pythopt.parallel` package are the backbone of the distributed work-

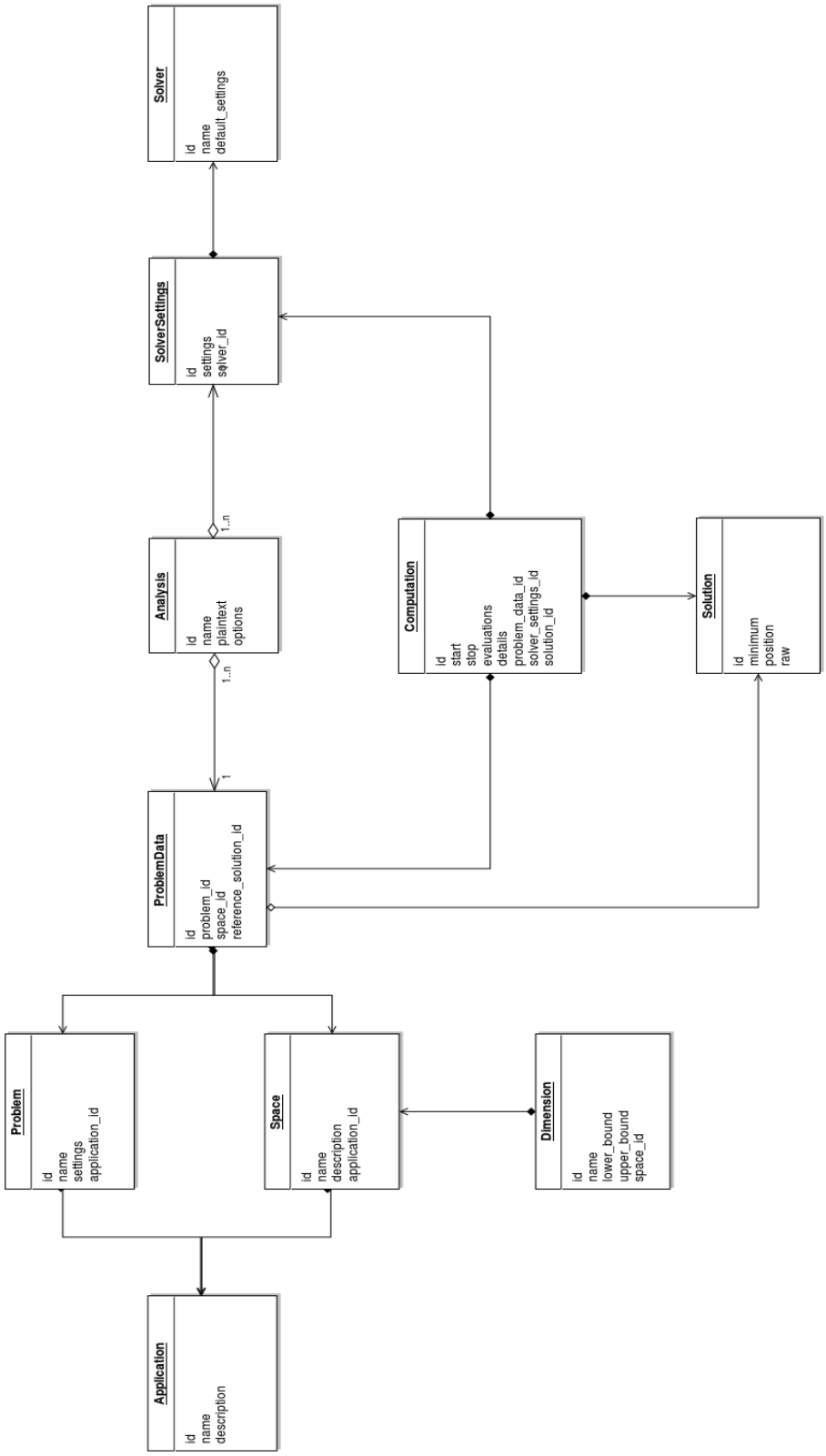

Figure 3.3: `pythOPT` database schema

flow in pythOPT. The pythopt.parallel.task.Task class defines an interface that all optimization problems that are saved to a DB must implement. The pythopt.parallel.tasker module creates all the necessary objects in a DB for a given Analysis instance. The pythopt.parallel.worker module polls a DB for pending computations, performs them, and stores the solutions. The tasker and worker modules are to be run as standalone services that constantly monitor the DB and update its state.

An Analysis instance is represented as a JSON object in a DB. It contains information about optimization problems that form a test suite and a dictionary of solver group names to lists of solver settings. Using this representation, we can group multiple solver configurations under one name; e.g., if we compare GBPSO with DWPSO using ten random seed values for each of the solvers, then we have two groups ('PSO' and 'DWPSO'), and each of them has a list of ten configurations that differ only in values of the random number generator seed.

The pythopt.parallel.tasker module queries a DB periodically for Analysis objects that do not have any associated solver configurations or problem data. If such an object is found, the module creates SolverSettings instances in a DB for all solver configurations used in the analysis. After that, the module adds computations for solving all of the problems in a test suite with all of the newly added solver configurations.

The pythopt.parallel.worker module polls a DB for objects that represent pending computations, i.e., the Computation objects that have no start time set. Once it identifies a pending computation, it fetches all the necessary information to configure a solver and to build a ProblemData instance. It runs the solver and saves its output to the DB as a Solution object.

Optimization problems in the DB are organized into Application instances. Each application has to define its own Task subclass that references two other classes: one for handling a search space description and the other for handling an objective function description. The worker module checks which Task subclass to use for a given Computation instance and initializes that subclass with this Computation instance. After that subclass is initialized, worker runs the solver and stores the Solution.

The worker process is the main execution unit in pythOPT. A user can run arbitrarily many of them, but if the number is too high, then the DB (which is used as the task queue)

may become a bottleneck.

3.7 Implementation

This section focuses on implementation of solvers and objectives in `pythOPT`.[1] The information in this section is helpful to users who wish to add a new solver and make it compatible with the rest of the system. It presents the implementation of the PSO solvers and a structure of `FORTRAN` adapters.

3.7.1 Solver structure

All `Solver` subclasses have two class attributes called `DETERMINISTIC` and `settings`. The `DETERMINISTIC` attribute is a boolean that allows a client code to decide how to perform an analysis of a solver. The `settings` attribute is a dictionary that holds default parameter values for a particular method. Names of the parameters and the associated default values are taken from the publications that introduce a given method or from an implementation that is adapted by a specific solver.

Solvers also support a set of universal settings that control stopping criteria and a random number generator. These additional settings have been introduced to unify the interface and operation on commonly changed attributes, e.g., maximal number of function evaluations. The supported termination settings are presented in Table 3.1.

Table 3.1: Universal settings for termination criteria

Name	Description
`eval_lim`	maximal number of objective function evaluations
`niter`	maximal number of iterations
`minimum`	a target value for a solver to attain
`max_noimprov`	a number of evaluations yielding no improvement that halts the algorithm.

Every solver must be initialized with `ProblemData` and optionally a dictionary of parameters that adjusts its behaviour. Among these parameters are the universal settings

[1] In this section, the words *solver* and *objective* refer to concrete classes that implement the `Solver` and `Objective` interface, respectively.

for termination criteria (see Table 3.1) that are supported by all solvers. After a solver is initialized, a user can call the `solve` method that returns a `Solution` instance.

3.7.2 PSO implementation

The behaviour of PSO can be adjusted using PSO–specific settings that are presented in Table 3.2. Solvers derived from PSO may require additional settings, and they may turn a previously optional setting into an obligatory one.

Table 3.2: PSO settings

Name	Description
`nparticles`	a number of particles in a swarm
`inertia`	a pair of the initial and the final inertia values
`c1`	a pair of the initial and the final global weights
`c2`	a pair of the initial and the final local weights
`seed`	a seed for a random number generator

The basic PSO solver does not vary any of the `inertia`, `c1`, or `c2` values over iterations, and so it uses only the first elements of the pairs that represent the range of these parameters. The DWPSO and TVACPSO solvers decrease the inertia linearly across all iterations. To determine the rate of change of this setting, the solvers require the `niter` setting to be defined.

The PSO implementation follows the template method design pattern. The original algorithm is implemented in the base class, `PSO`. The implemented variants overwrite selected hook methods. The hook methods are called by the `solve` method in the base class. The sequence diagram of the `solve` method is presented in Figure 3.4. The method creates a list of `Particle` instances that are initialized with `ProblemData`. Then, it starts the evolution of particles. The evolution consists of particle evaluation, an update of the solver's state, and the main evolution loop. Each iteration of the loop updates the weights used by the particles, evolves particles, updates an answer and counters for termination criteria, and checks if any termination criteria are satisfied.

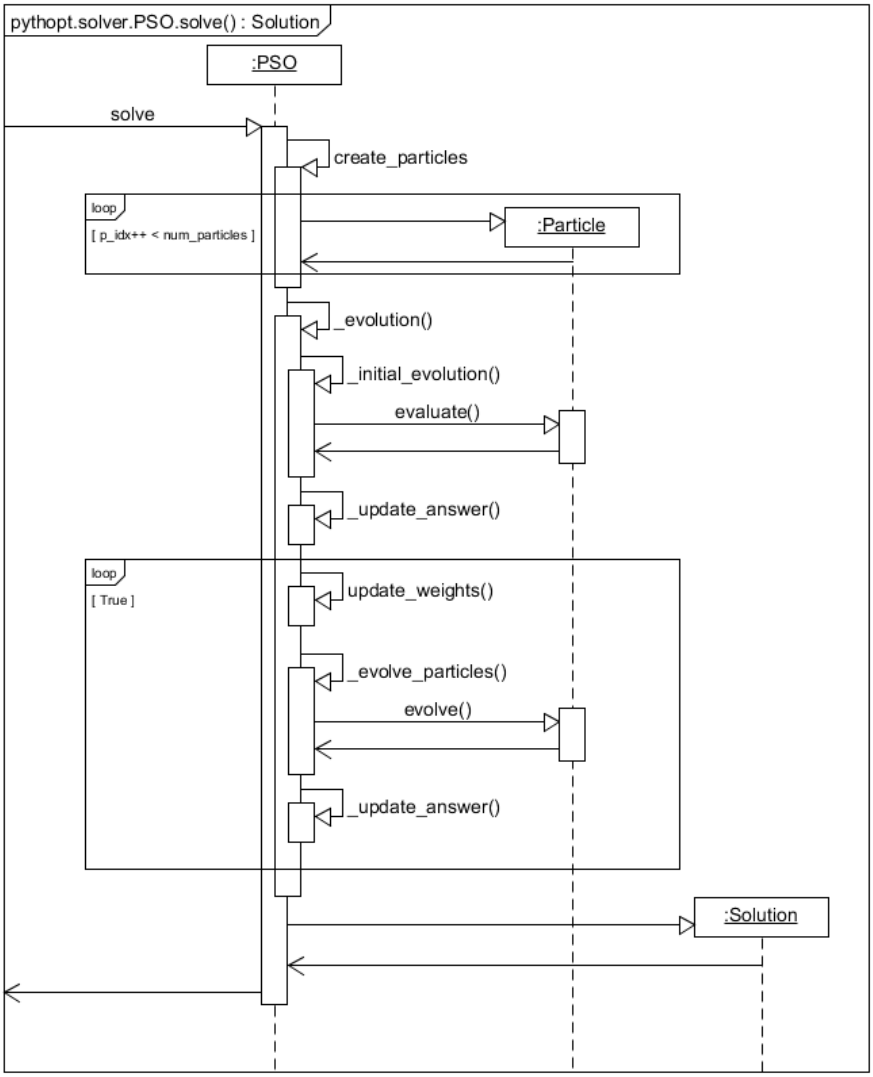

Figure 3.4: PSO: UML sequence diagram

3.7.3 `FORTRAN` Adapters

There are two interfaces for `FORTRAN` objects in `pythOPT`: one for objective functions and the other for solvers. Both interfaces have common responsibilities, but some of their parts differ significantly. The differences stem from the limited interaction that `FORTRAN` solvers offer, e.g., the inability to examine the state of a solver. Similarities and differences between these two interfaces are described below.

Solvers depend on the interface provided by the `Objective` class. To get a value associ-

ated with a candidate decision vector, a solver passes the vector to the `evaluate` method of an `Objective` instance. However, instantiating an `Objective` requires a Python object that can be used as a function, i.e., a type that is a functor / Python callable. If the implementation of an objective function is provided in `FORTRAN`, then a user needs to make it possible to import that function into Python. This is commonly referred to as *extending* Python, and there are a number of ways to approach it.

One method of extending Python with `FORTRAN` code is to use `F2PY`. `F2PY` is a tool within `numpy` package that compiles `FORTRAN` source code (using an external `FORTRAN` compiler) to generate a Python module. It requires a signature file that defines an interface that is to be exposed by the module. The `Objective` class defines the interface that a module must implement, so the signature file can be generated from a template. The only element that must be provided is a `FORTRAN` driver that calls the original objective code and implements the following interface:

```
SUBROUTINE DRIVER (POSITION, DIMENSIONS, FITNESS)
    REAL*8, INTENT(IN)  :: POSITION(*)
    INTEGER, INTENT(IN) :: DIMENSIONS
    REAL*8, INTENT(OUT) :: FITNESS
END SUBROUTINE
```

If an objective implements this interface, then no driver code is necessary.

The approach is used to implement the `FortranObjective` class. A user only needs to supply a list of `FORTRAN` source files that implement the objective. On the first call to the `evaluate` method of such an objective, the necessary files are generated, compiled, linked, and imported into Python. The following calls can skip the module generation, and they access the objective function directly. The class diagram of classes involved in this flow is presented in Figure 3.5.

Solvers implemented in `FORTRAN` use a similar mechanism. Their drivers are more complicated because they perform initialization of a solver and integrate the `FortranObjective` objects. This procedure varies from one solver to another, so it is natural to implement one adapter for each solver. Each adapter implements the `Solver` interface; i.e., it provides the

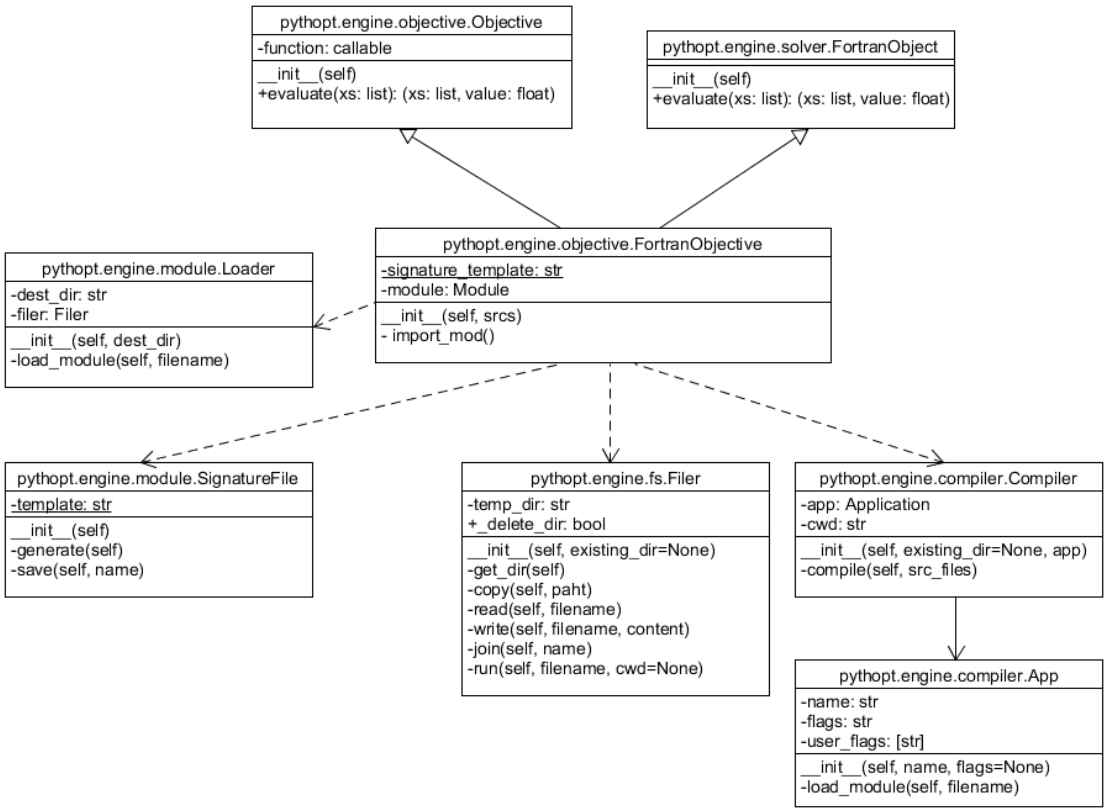

Figure 3.5: Structure of a `FORTRAN` objective

`solve` method. On the first call to the method, it generates a driver from a template, compiles and links a solver and a `FortranObjective` instance, and runs the resulting executable in a separate process. The standard output of the process is captured and parsed by an adapter and returned to a caller as a `Solution` instance. The adapters currently available in `pythOPT` generate drivers that print results in a `JSON` format, which makes parsing easier.

Chapter 4

Numerical results

This chapter describes an approach for choosing a method for solving a class of optimization problems. Section 4.1 presents the metrics used to analyze the performance of the optimization methods available in `pythOPT`. It also introduces the methodology and the structure of experiments that were conducted on problems described in the subsequent sections. Section 4.2 details the application of the methodology to a set of standard benchmark problems and discusses results of that application. Section 4.3 contains another three sets of experiments that were performed on optimization problems that emerged in three research projects. The experiments are organized in subsections, each of which studies performance of optimization methods on one set of problems. The first two experiments use the same methodology that was used to analyze optimization methods performance on benchmark problems in Section 4.2.1. Experiments on the third project were conducted in a simplified way. Section 4.4 contains the summary of the experiments and the results presented in the previous sections.

4.1 Methodology

We demonstrate performance analyses of sixteen solvers on four sets of optimization problems. This section presents the methodology that was used for generating experimental data for these analyses. It contains descriptions of performance metrics used to analyze experimental data, techniques used for controlling bias in the data, and the design of the experiments that are presented in the subsequent sections.

A performance metric is a method of analyzing experimental data. It provides the relative performance of a set of optimization methods. Two metrics, win-draw-loss and performance

profiles, were used to analyze the results of numerical experiments. The experiments are presented in Sections 4.2 and 4.3.

Stochastic methods are often tested multiple times with different seeds from a random number generator. Deterministic solvers do not have such a parameter and have to be randomized in an alternative way. In our tests, we randomize the problem domain instead of solver settings. The procedure for that process is presented in Section 4.1.3.

Performance metrics were generated for a few sets of termination criteria and solver settings. Each set makes certain assumptions about resources available to any given solver. The assumptions, the list tested solvers, and their settings are described in Section 4.1.4.

4.1.1 Win-Draw-Loss

The win-draw-loss (WDL) metric presents the number of problems on which a solver achieved the best solutions. This metric characterizes each method with three scores. If one method has the best result on a given problem, it scores a win. If the same best result is attained by more than one method, then each of the methods scores a draw. In such cases, the remaining methods, regardless of how far behind the best score their results were, score a loss.

The results used for this analysis are floating-point numbers. They are the optimal values identified by solvers on a given problem. Comparison of these numbers is performed with a tolerance expressed in terms of the number of significant digits.

4.1.2 Performance Profiles

The interpretation and analysis of large data sets generated in experiments can quickly become overwhelming. The typical approach for small data sets uses tables that contain selected performance metrics per solver per column and one benchmark problem per row. Two options for addressing larger data sets are averages or cumulative totals over all problems. Such a form of presenting results is highly sensitive to a small number of difficult tests, obfuscating results obtained from simpler tests.

A *performance profile* is a metric in which solvers are compared by the ratio of their run-time on a given problem to the best run-time achieved by any solver [16]. This approach

is insensitive to the difficulty level of a benchmark problem, and it is viable for analyzing large data sets.

The metric is based on the amount of time that a solver needs to find a solution to a problem. For each problem, we measure the time that it takes for a certain solver to solve the problem. The timings of all the solvers, in the context of a particular problem, are then divided by the minimum time any solver needed to solve the problem. A performance profile of a solver presents the percentage of problems solved within a given factor of the best metric, e.g., time or number of function evaluation, among all tested solvers.

Given a set of problems P and a set of solvers S, let $t_{p,s}$ denote time necessary to solve a problem $p \in P$ with a solver $s \in S$. A *performance ratio* $r_{p,s}$ is then defined as

$$r_{p,s} = \frac{t_{p,s}}{\min\{t_{p,s} : s \in S\}}. \tag{4.1}$$

If a solver s cannot solve a problem p, $r_{p,s}$ takes the value of r_M, where $r_M \geq r_{p,s}$ for all p, s. A performance profile is a function of a real factor $\tau \in [1, \infty)$

$$\rho_s(\tau) = \frac{1}{|P|} |\{p \in P : r_{p,s} \leq \tau\}|, \tag{4.2}$$

to a fraction of problems that are solved with a performance ratio less than or equal to the given factor. For example, $\rho_s(\tau) = 0.1$ means that the solver s solves a subset of 10% of all problems with at most τ times resources relative to the most efficient solver of each problem in that subset.

4.1.3 Problem randomization

To obtain representative data for a performance analysis, an optimization method is tested with several different problem variants. There are at least two reasons for doing so. First, performance of non-deterministic optimization methods can vary significantly depending on the seed provided to a random number generator. Second, some of the commonly used optimization problems that are used to assess performance of optimization methods have characteristics that can be exploited by certain methods [7]. The one most self-evident of such characteristics is a prior knowledge of a decision vector that yields the optimal solution, which in case of several well-known problems, is the origin of a coordinate system.

Others include solutions located at points with all coordinates equal to the same value and symmetries that drive solvers toward a solution faster than expected. These characteristics can introduce a bias during analysis.

For non-deterministic methods, using different seeds over a course of multiple runs offers a way to change the set of evaluated decision vectors. This discrepancy results in divergent information about the problem that can lead to dissimilar solutions. However, this technique cannot be applied to deterministic methods, which do not use random numbers, and in result it does not make a test representative. To gain a similar confidence in performance of stochastic and deterministic solvers and to mitigate potential weaknesses of benchmark tests, we use surjective transformations on the search space to vary the initial conditions of problems.

The regularities and exploitable features of benchmark problems can be mitigated by applying a series of linear transformations to a candidate decision vector and evaluating an objective function for the resulting vector. The transformations must not change the solution to a problem. For example, to eliminate a bias toward the origin of a coordinate system, we apply a random offset to an argument of the objective function before evaluation. This is known as the *center offset* method [11].

Problem randomization can apply translation, scaling, and rotation to an input vector of a objective function. These transformations are represented by matrices in a homogeneous coordinate system that allows for representing any combination of them in one matrix. A two-dimensional translation matrix $\boldsymbol{T}$, a scaling matrix $\boldsymbol{S}$, and a rotation matrix $\boldsymbol{R}$ are defined as follows:

$$\boldsymbol{T} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ T_x & T_y & 1 \end{bmatrix}, \quad \boldsymbol{S} = \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix}, \quad \boldsymbol{R} = \begin{bmatrix} \cos(\phi) & \sin(\phi) & 0 \\ -\sin(\phi) & \cos\phi) & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

A product of these matrices results in matrix $\boldsymbol{M}$, which is reused for all input vectors of a benchmark function. An input vector is extended with an additional coordinate equal to one and multiplied by $\boldsymbol{M}$. For example, a translation of a two-dimensional vector $\boldsymbol{x} = [x, y]$

follows:

$$\boldsymbol{x}' = \begin{bmatrix} x + T_x & y + T_y & 1 \end{bmatrix} = \begin{bmatrix} x & y & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ T_x & T_y & 1 \end{bmatrix}. \tag{4.3}$$

After removing the last element of $\boldsymbol{x}'$, the vector can be used as an input to an objective function.

Translation and scaling easily generalize to higher dimensions, but rotation requires a special procedure for generating the $\boldsymbol{R}$ matrix. In two-dimensional space, there exists only one rotation; in three-dimensional space, there exist three independent rotations. A rotation matrix for a d-dimensional vector is an orthogonal $d \times d$ matrix. `pythOPT` uses Salomon's algorithm [7][40] to generate a rotation matrix from a set of randomly generated angles. The algorithm generates two matrices by multiplying d matrices of size $d \times d$ each of which contains four random components. The final rotation matrix is the product of those two matrices.

4.1.4 Methodology

The studied cases (except for the Rational Material Design) all use the same methodology for generating data. Each solver is tested with the same set of initial conditions. This set is a Cartesian product of two settings for the maximal number of function evaluations (100,000 and 500,000) and two settings for tolerances (10^{-5} and 10^{-8}). Moreover, each set of initial conditions is used on a set of problems with problem randomization. The data obtained from these experiments are used to generate WDL metrics and performance profiles metrics.

The tested optimization methods are Direct, GWO, and the PSO-based methods: GBPSO, SPSO, DWPSO, TVACPSO, GCPSO, DAPSO, and CDIWPSO. Each PSO-based method comes in two variants: one that uses the mesh communication topology, and the other that uses the ring topology (denoted as `-ring` in tables and figures). All of these methods are presented in Chapter 2.

Settings for implementations of the optimization methods used in the experiments are given in Table 4.1 and Table 4.2. Whenever possible, settings are taken from a paper that introduces a given method. In some cases, implementation requires additional settings that

are also presented in the tables.

Table 4.1: Settings of optimization methods used in experiments I

Setting	GBPSO	DWPSO	TVACPSO	SPSO
`nparticles`	50	50	50	50
`inertia`	1	(0.9, 0.4)	(0.9, 0.4)	0.73
`c1`	2	2	(0.5, 2.5)	1.496
`c2`	2	2	(2.5, 0.5)	1.496
`seed`	1	1	1	1

Table 4.2: Settings of optimization methods used in experiments II

Setting	CDIWPSO	DAPSO	GCPSO	Direct	GWO
`nparticles`	50	50	50	–	16
`inertia`	(0.9, 0.4)	(0.9, 0.4)	0.73	–	-
`c1`	2	2	1.496	–	-
`c2`	2	2	1.496	–	-
`seed`	1	1	1	–	1
ρ	-	-	1	–	-
`succthr`	-	-	15	–	-
`failthr`	-	-	5	–	-
`n_bests`	-	-	-	–	3
`min_dia`	-	-	-	10^{-5}	-

All tested solvers have a maximal number of function evaluations set to 100,000 or 500,000. Any of the solvers can exit early if it solves the problem before reaching the limit on the number of function evaluations. In the experiments, a problem is considered solved if the decision vector $\boldsymbol{x}$ satisfies a condition on the relative error of the form

$$\begin{cases} f(\boldsymbol{x}) - (1+\epsilon)f^* \leq 0, & \text{if } f^* \geq 0, \\ f(\boldsymbol{x}) - (1-\epsilon)f^* \leq 0, & \text{otherwise,} \end{cases}$$

where ϵ is the tested tolerance and f^* is the known optimum. Tolerances used with the WDL metric denote the precision that was used in comparing the minima.

4.2 Benchmark Problems

In the Benchmark Problems (BP) we compare sixteen solver configurations on a common set of problems and present metrics of the results. We use well-known benchmark problems

that, despite having objective functions that are inexpensive to evaluate, are hard to solve due to characteristics like being multi-modal, high-dimensional, or non-linear.

4.2.1 Task setup

The test set consists of twenty-three benchmark problems. Definitions of these problems, their bounds, dimensionality, and known solutions are presented in Appendix A. We use the methodology presented in Section 4.1 for generating experimental data.

4.2.2 Results

The sample data generated during experiments are analyzed with the two metrics. First, we present WDL metrics for non-randomized and randomized test. Next, we present performance profiles metrics for the same set of results.

Table 4.3: BP, original formulations, $N = 100{,}000$

(a) tolerance: 10^{-5}

Solver	W	D	L
CDIWPSO-mesh	1	6	16
CDIWPSO-ring	1	7	15
DAPSO-mesh	0	4	19
DAPSO-ring	0	3	20
DWPSO-mesh	1	6	16
DWPSO-ring	1	5	17
Direct	8	6	9
GBPSO-mesh	0	1	22
GBPSO-ring	0	2	21
GCPSO-mesh	0	10	13
GCPSO-ring	0	6	17
GWO	0	4	19
SPSO-mesh	0	10	13
SPSO-ring	0	6	17
TVACPSO-mesh	0	3	20
TVACPSO-ring	0	4	19

(b) tolerance: 10^{-8}

Solver	W	D	L
CDIWPSO-mesh	0	6	17
CDIWPSO-ring	1	6	16
DAPSO-mesh	0	3	20
DAPSO-ring	0	4	19
DWPSO-mesh	1	7	15
DWPSO-ring	0	5	18
Direct	10	6	6
GBPSO-mesh	0	2	21
GBPSO-ring	0	1	22
GCPSO-mesh	0	7	16
GCPSO-ring	1	6	15
GWO	0	5	18
SPSO-mesh	0	7	16
SPSO-ring	0	6	17
TVACPSO-mesh	1	2	20
TVACPSO-ring	0	2	20

Table 4.4: BP, original formulations, $N = 500{,}000$

(a) tolerance: 10^{-5}

Solver	W	D	L
CDIWPSO-mesh	0	7	16
CDIWPSO-ring	0	6	17
DAPSO-mesh	0	5	18
DAPSO-ring	0	5	18
DWPSO-mesh	0	7	16
DWPSO-ring	1	8	14
Direct	8	6	9
GBPSO-mesh	0	3	20
GBPSO-ring	0	3	20
GCPSO-mesh	0	11	12
GCPSO-ring	0	7	16
GWO	0	6	17
SPSO-mesh	0	10	13
SPSO-ring	0	8	15
TVACPSO-mesh	0	5	18
TVACPSO-ring	0	6	17

(b) tolerance: 10^{-8}

Solver	W	D	L
CDIWPSO-mesh	0	5	18
CDIWPSO-ring	0	5	18
DAPSO-mesh	0	2	21
DAPSO-ring	0	3	20
DWPSO-mesh	0	6	17
DWPSO-ring	1	8	14
Direct	10	6	7
GBPSO-mesh	0	2	21
GBPSO-ring	0	1	22
GCPSO-mesh	0	7	16
GCPSO-ring	0	7	16
GWO	0	3	20
SPSO-mesh	0	7	16
SPSO-ring	0	7	16
TVACPSO-mesh	1	3	19
TVACPSO-ring	0	3	20

Table 4.5: BP, 5 problem variants, $N = 100{,}000$

(a) tolerance: 10^{-5}

Solver	W	D	L
CDIWPSO-mesh	2	8	13
CDIWPSO-ring	1	10	12
DAPSO-mesh	0	8	15
DAPSO-ring	2	7	14
DWPSO-mesh	0	6	17
DWPSO-ring	1	9	13
Direct	0	6	17
GBPSO-mesh	2	3	18
GBPSO-ring	0	4	19
GCPSO-mesh	0	7	16
GCPSO-ring	1	11	11
GWO	0	4	19
SPSO-mesh	0	7	16
SPSO-ring	0	12	11
TVACPSO-mesh	0	6	17
TVACPSO-ring	1	6	16

(b) tolerance: 10^{-8}

Solver	W	D	L
CDIWPSO-mesh	2	7	14
CDIWPSO-ring	1	8	14
DAPSO-mesh	0	5	18
DAPSO-ring	2	4	17
DWPSO-mesh	0	7	16
DWPSO-ring	1	8	14
Direct	0	5	18
GBPSO-mesh	2	2	19
GBPSO-ring	1	2	20
GCPSO-mesh	0	8	15
GCPSO-ring	1	9	13
GWO	0	3	20
SPSO-mesh	0	8	15
SPSO-ring	1	9	13
TVACPSO-mesh	0	3	20
TVACPSO-ring	1	3	19

Table 4.6: BP, 5 problem variants, $N = 500{,}000$

(a) tolerance: 10^{-5}

Solver	W	D	L
CDIWPSO-mesh	0	8	15
CDIWPSO-ring	0	11	12
DAPSO-mesh	0	9	14
DAPSO-ring	2	7	14
DWPSO-mesh	0	7	16
DWPSO-ring	0	11	12
Direct	0	7	16
GBPSO-mesh	1	5	17
GBPSO-ring	0	5	18
GCPSO-mesh	0	7	15
GCPSO-ring	1	12	10
GWO	0	5	18
SPSO-mesh	0	8	15
SPSO-ring	1	13	9
TVACPSO-mesh	3	7	13
TVACPSO-ring	1	6	16

(b) tolerance: 10^{-8}

Solver	W	D	L
CDIWPSO-mesh	0	7	16
CDIWPSO-ring	0	12	11
DAPSO-mesh	0	5	18
DAPSO-ring	1	6	16
DWPSO-mesh	0	8	15
DWPSO-ring	0	11	12
Direct	1	5	17
GBPSO-mesh	1	2	19
GBPSO-ring	2	2	19
GCPSO-mesh	0	8	15
GCPSO-ring	0	9	14
GWO	0	5	18
SPSO-mesh	0	8	15
SPSO-ring	1	12	10
TVACPSO-mesh	3	5	15
TVACPSO-ring	0	5	18

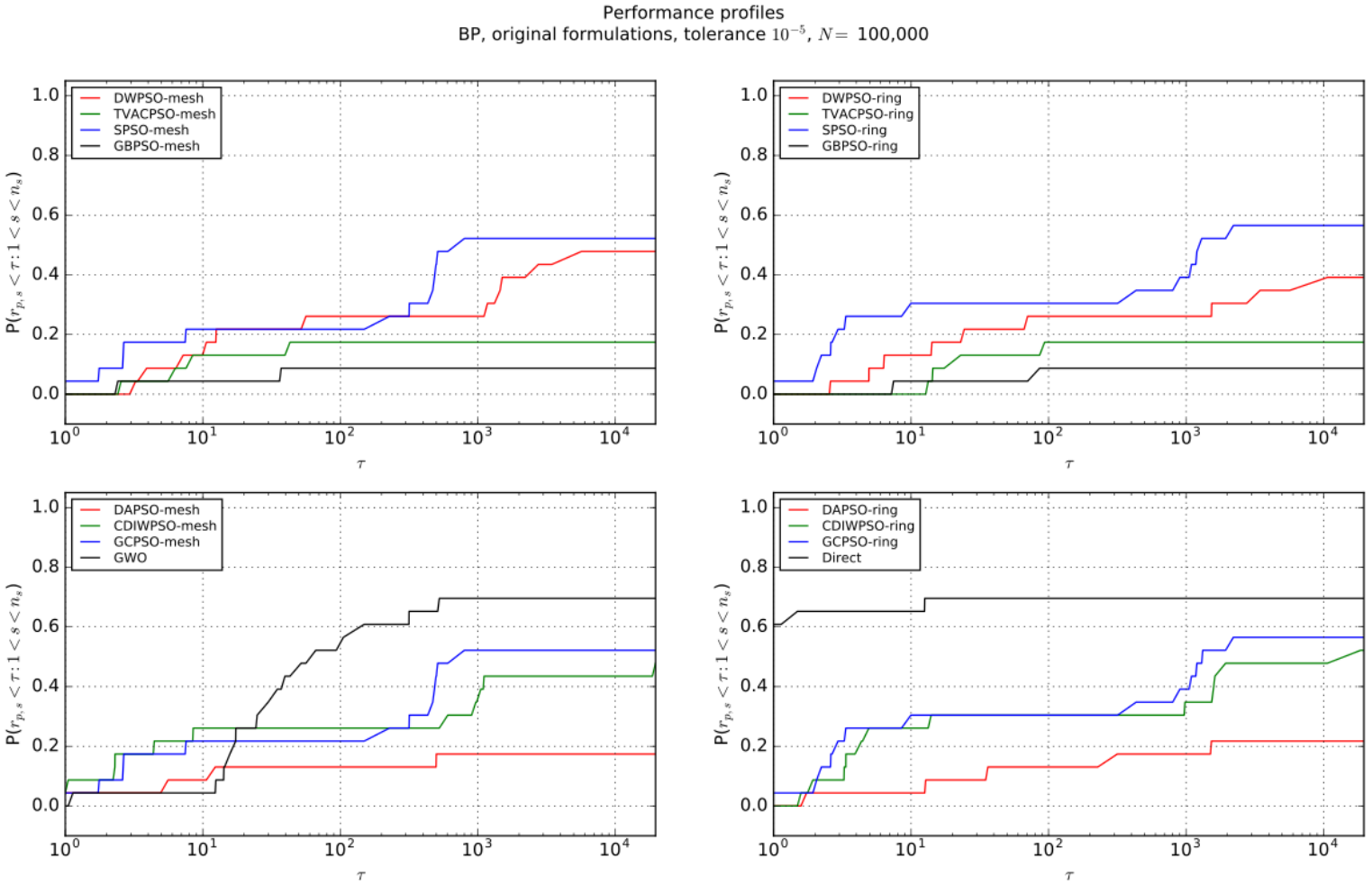

Figure 4.1: BP, original formulations, tolerance 10^{-5}, $N = 100{,}000$

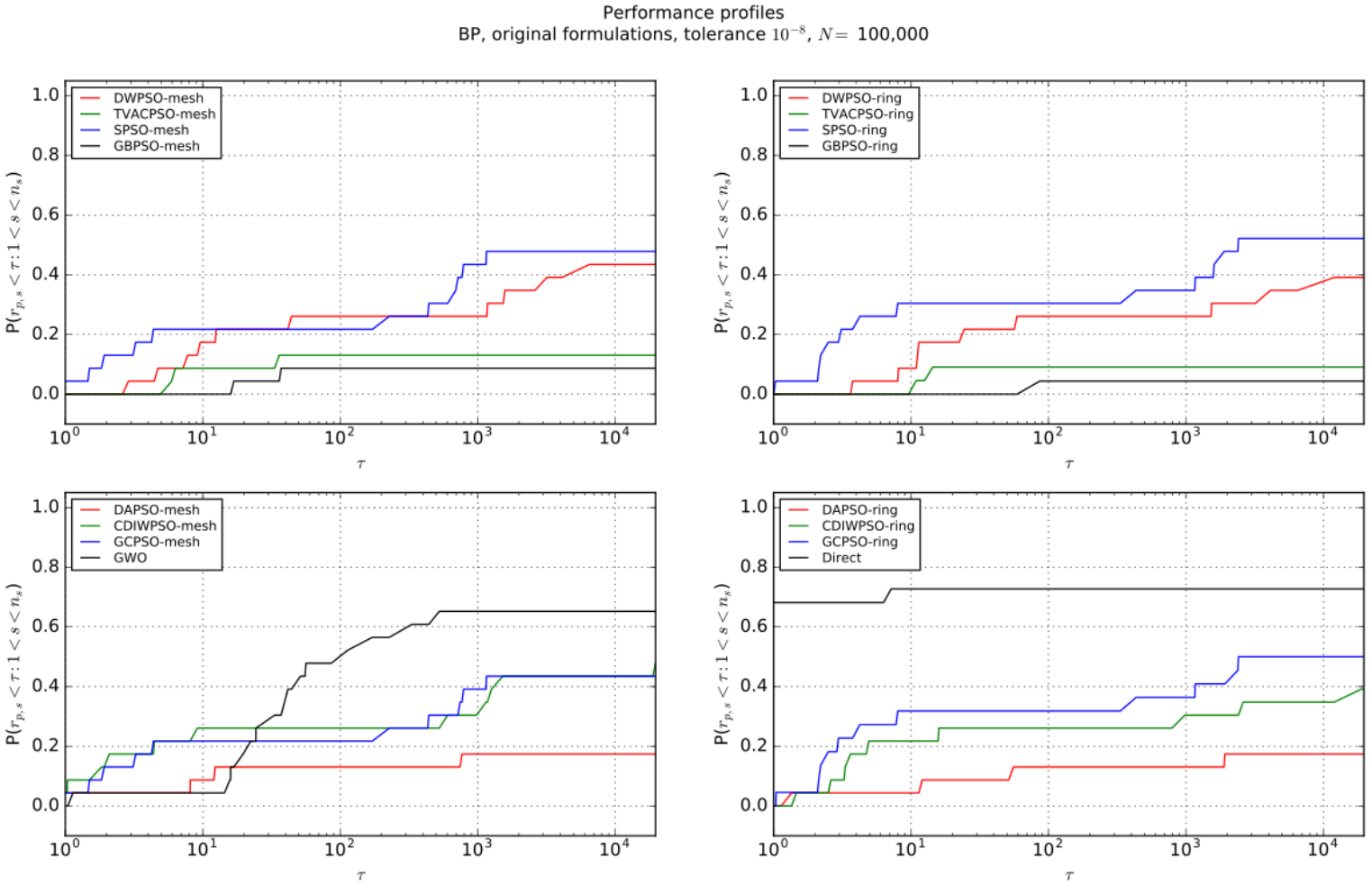

Figure 4.2: BP, original formulations, tolerance 10^{-8}, $N = 100{,}000$

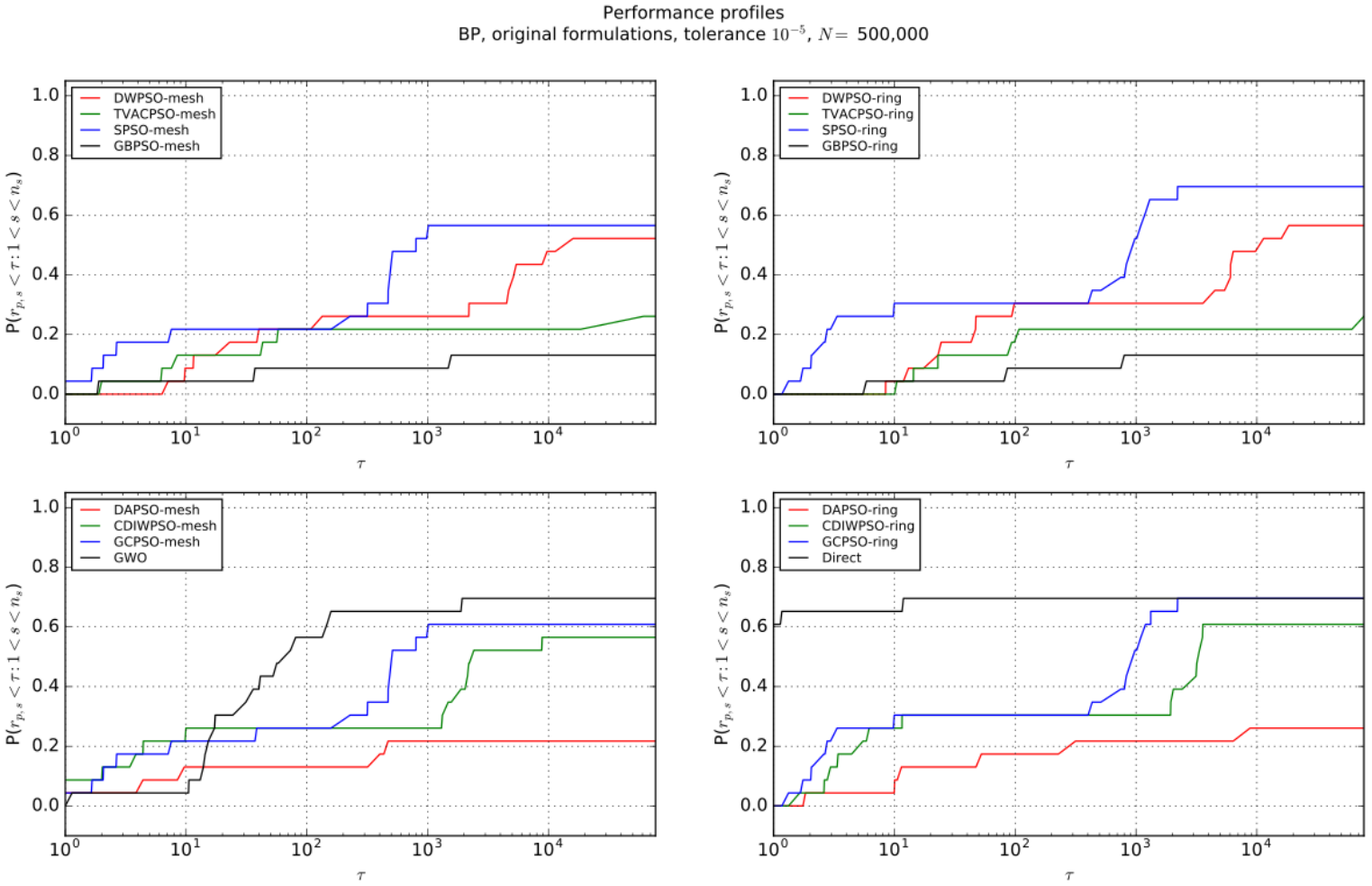

Figure 4.3: BP, original formulations, tolerance 10^{-5}, $N = 500{,}000$

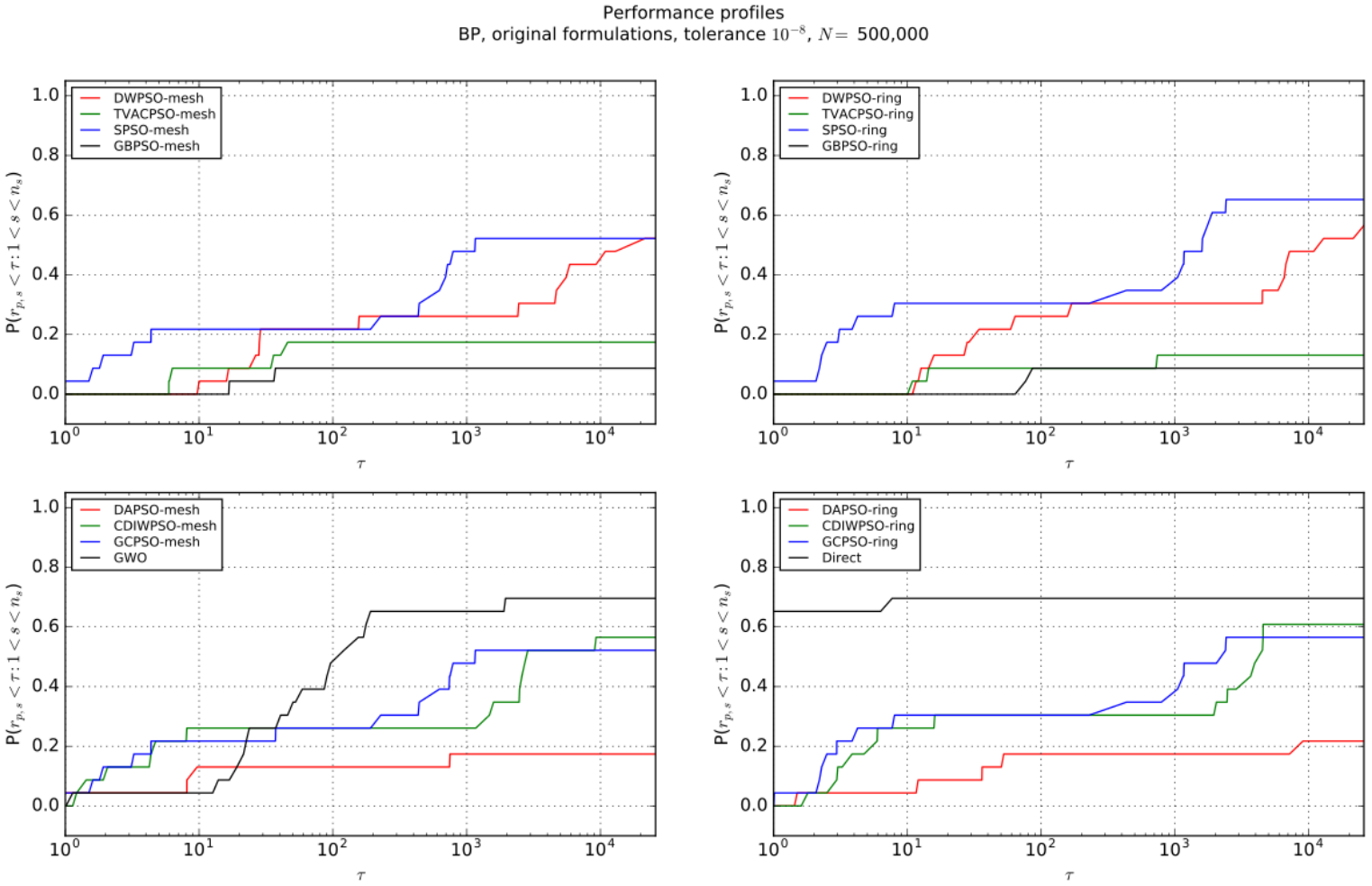

Figure 4.4: BP, original formulations, tolerance 10^{-8}, $N = 500{,}000$

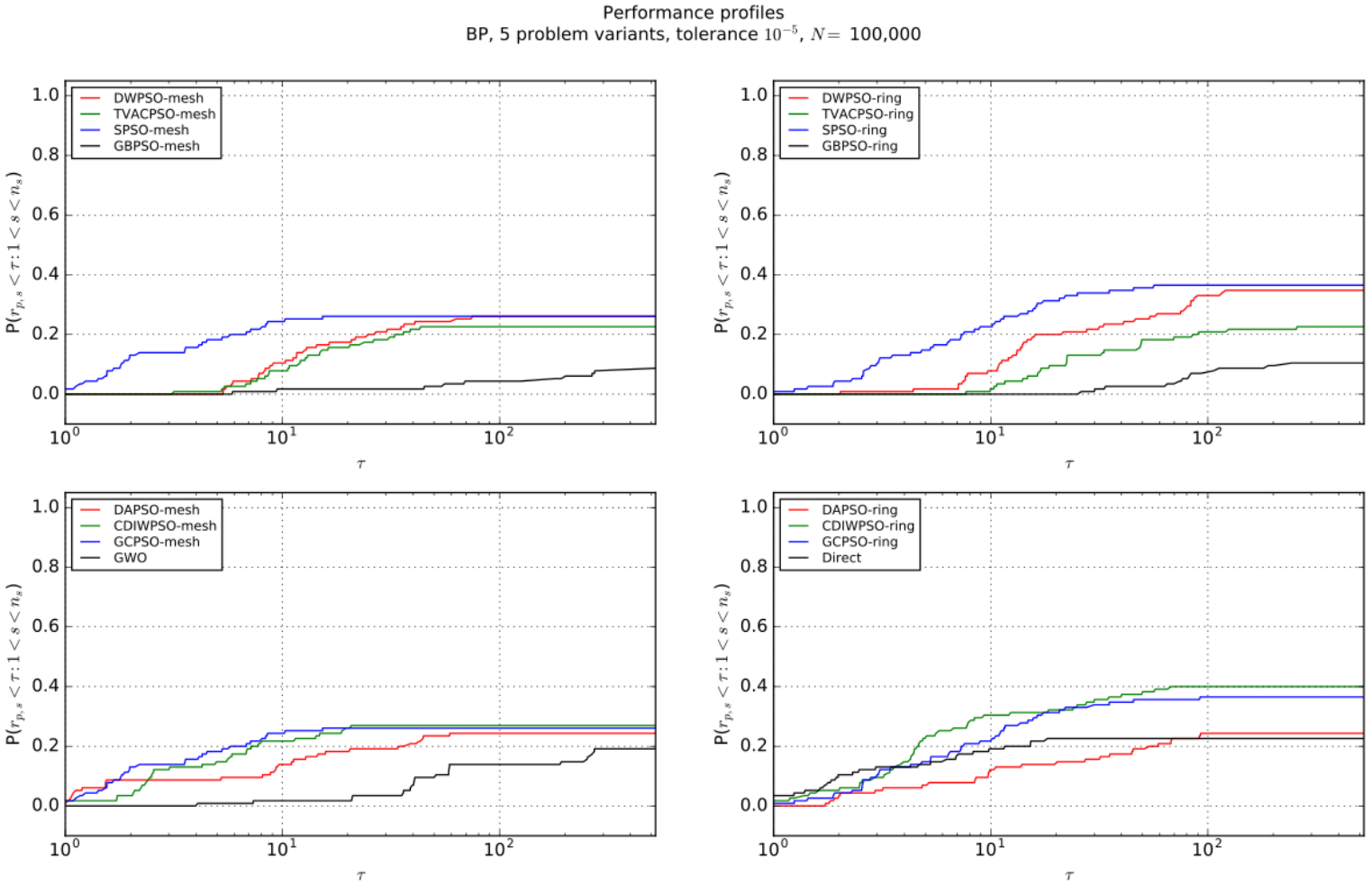

Figure 4.5: BP, 5 problem variants, tolerance 10^{-5}, $N = 100{,}000$

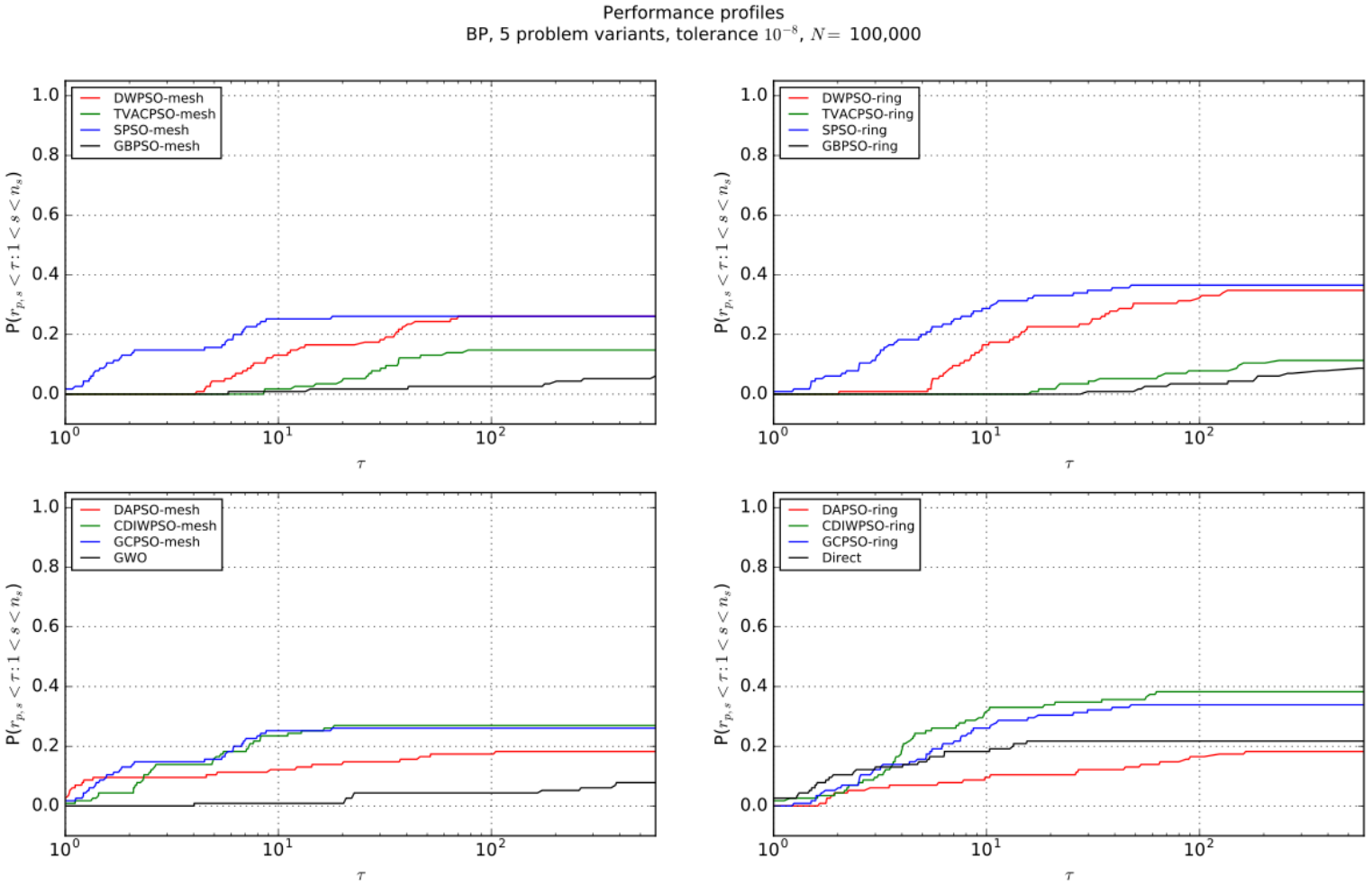

Figure 4.6: BP, 5 problem variants, tolerance 10^{-8}, $N = 100{,}000$

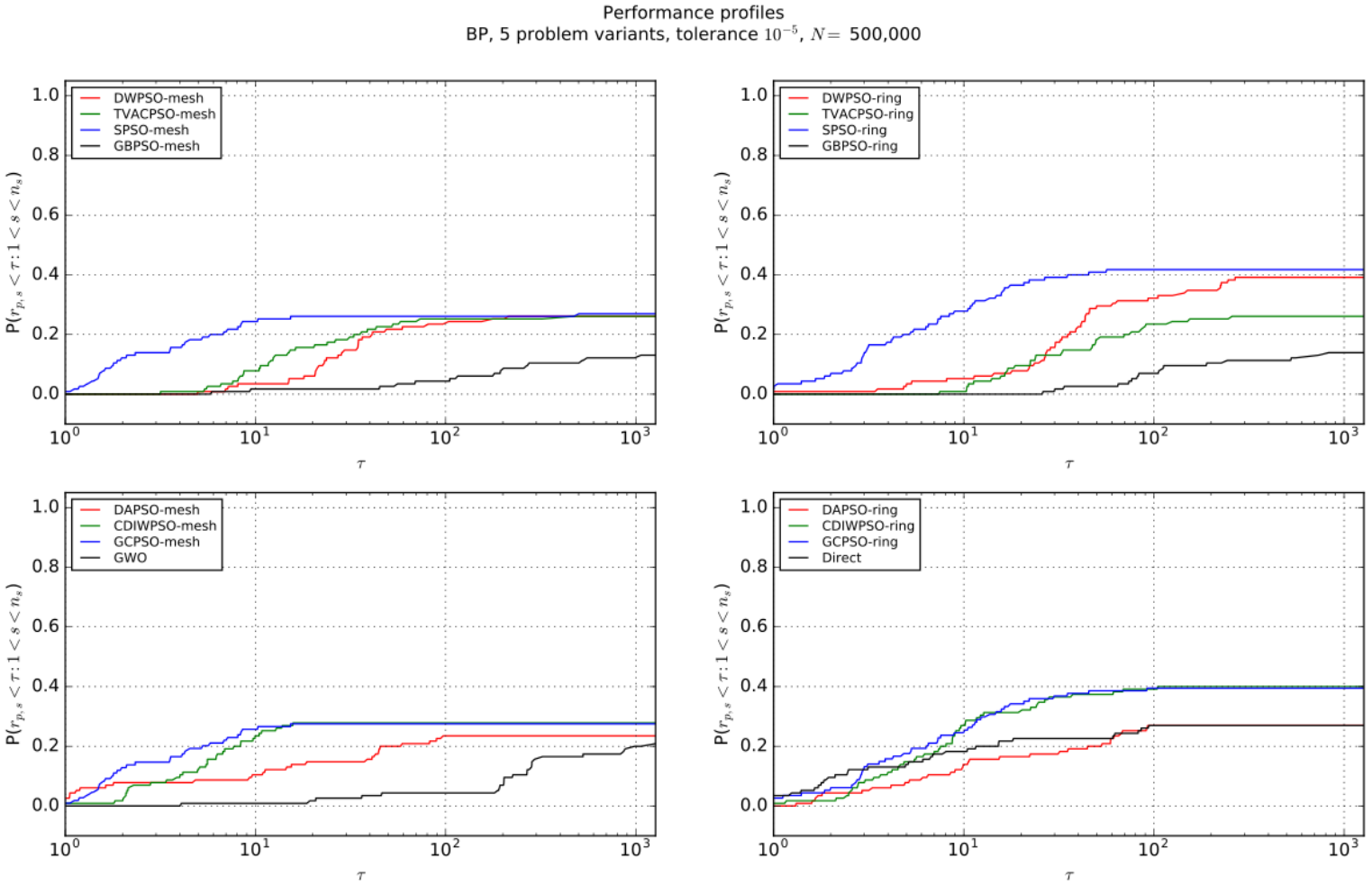

Figure 4.7: BP, 5 problem variants, tolerance 10^{-5}, $N = 500{,}000$

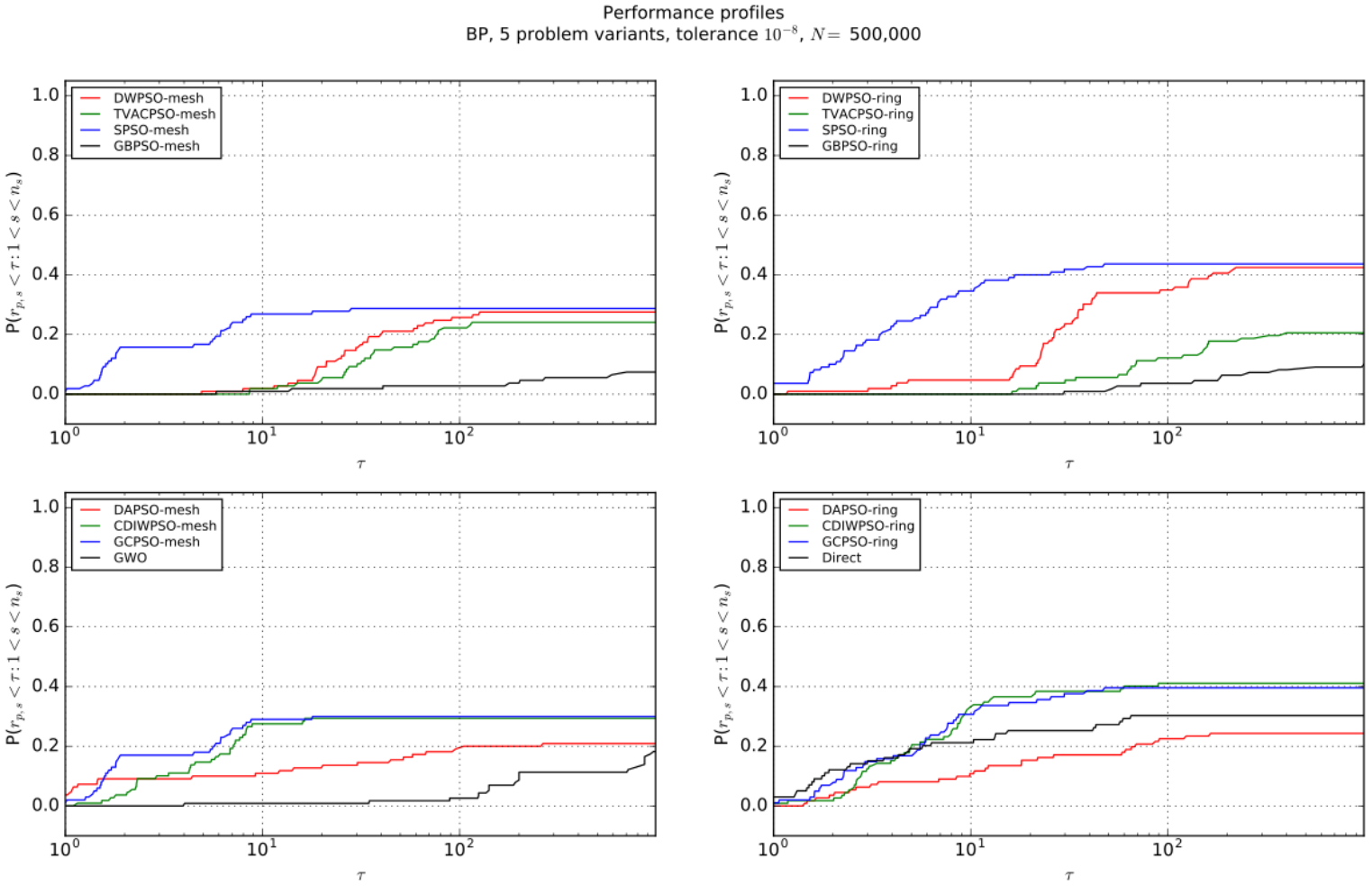

Figure 4.8: BP, 5 problem variants, tolerance 10^{-8}, $N = 500{,}000$

4.2.3 Observations

We first look at the results of tests that were performed on the set of benchmark problems in their original formulations. The first part presents performance of optimization methods that evaluate benchmark problems only in their original form, i.e., how they are defined in the original papers. The second part presents the cumulative performance of each method on five randomized variants of the benchmark problems.

Two non-PSO-based methods, Direct and GWO, perform best on the original benchmark problems; however, their performance is nearly the worst on the randomized variants of these problems. Among PSO-based methods, SPSO performs best of all other methods, and its performance is relatively close to that of Direct and GWO. Surprisingly, all tested optimization methods fail to identify solutions to four of the benchmark problems: Deceptive3, Dropwave, Rosenbrock, and Shubert (see Appendix A). We now look at these results in more detail.

Original formulations The WDL metric of the gathered data indicates outstanding performance of the Direct algorithm. It reaches the best results on the original formulation of the benchmark problems for all combinations of evaluation limits and tolerances presented in Tables 4.3 and 4.4. Also, its metrics are nearly identical for both evaluation limits. With a tolerance of 10^{-5}, it attains fourteen of the best solutions (wins and ties); with a tolerance of 10^{-8}, it attains sixteen of the best solutions.

GCPSO-mesh and SPSO-mesh get second best result of ten and seven solutions for the high and the low tolerance, respectively. Variants of these solvers based on the ring topology perform worse with the high tolerance but with the low tolerance both topologies acquire the same score. CDIWPSO and DWPSO place third with approximately seven solutions. On the test with 500,000 evaluations, DWPSO-ring has the best result after Direct. TVACPSO, DAPSO, and GBPSO attain the worst results with both topologies and both tolerances. These solvers attain the best results for approximately three problems.

The performance profiles metric, presented in Figures 4.1 through 4.4, indicate exceptionally good and consistent performance of Direct and GWO. Direct identifies solutions to

60% of the test set with fewest number evaluations for all of these problems. GWO requires two orders of magnitude more evaluations to reach similar number of solutions. These results are consistent between tests that vary the number of evaluations and tolerance.

SPSO and GCPSO consistently show nearly identical performance. They find solutions to 50% to 60% of the problems. However, both solvers need three orders of magnitude more evaluations than the lowest number of evaluations needed by solvers that were most effective on particular problems. That places their convergence to best solutions on the third place behind Direct and GWO.

CDIWPSO and DWPSO are the last two solvers that reach over 40% of all solutions. The number of solutions identified by these solvers is close to SPSO-mesh and GCPSO-mesh; however, their convergence is worse. TVACPSO, DAPSO, and GBPSO find solutions to a maximum of 20% of all problems. That places them last in our tests.

The topology that is used by a PSO-based solver influences performance. Performance profiles show a small increase in the number of identified solutions in comparison to the solvers that use the mesh topology, in tests using a maximum of 100,000 objective function evaluations. That difference is mostly apparent for SPSO in tests performed with a limit on the number function evaluations of 500,000 (see Figure 4.3). In that case, SPSO solves almost 10% more problems with the ring topology.

Randomized problems The second group of tests performed on the set of benchmark problems considers five randomized variants of each problem. We find these tests more representative because they target characteristics of a problem and not a specific problem variant. We now present the WDL metric and performance profiles metric on the set of randomized problems.

The WDL metric that is presented in Table 4.5 and Table 4.6 exhibits a drastic decrease in the performance of Direct in comparison to its performance on the set of original, non-randomized problems. GWO finds four solutions, which is a similar result to the one attained without randomization. SPSO, GCPSO, and CDIWPSO present the best performance. They reach the best solutions for approximately ten out of twenty-three problems. In contrast, Direct reaches six solutions. The three PSO-based solvers reach the winning results only

with the ring topology. Where SPSO and GCPSO show the superiority of ring topology for all tested cases, CDIWPSO is not affected by change in topology for 100,000 evaluations. With 500,000 evaluations, the difference becomes apparent. The CDIWPSO version that uses mesh topology identifies seven solutions, whereas the version based on ring topology identifies twelve solutions.

The performance profiles metric performed on the randomized problems (Figures 4.5 through 4.8) confirm the decrease in performance of Direct and GWO in comparison to the performance profile metric of data generated on non-randomized tests. Direct identifies nearly 70% of solutions on non-randomized problems but only 20% of randomized problems. GWO, similarly to Direct, identifies nearly 70% of solutions on non-randomized problems but only 20% of randomized problems. Also, on randomized problems, GWO has one of the two slowest convergence rates (GBPSO is the other).

SPSO, GCPSO, and CDIWPSO have highly similar performance profiles across number of evaluations, topology, and tolerance. All three solvers identify the highest number of solutions (between 25% and 42%). They also show the best convergence rate. DWPSO requires an order of magnitude more function evaluations, but it identifies a comparable number of solutions. TVACPSO-mesh and DWPSO-mesh have close performance profiles; however, the performance profiles of DWPSO-ring differ significantly. TVACPSO-ring identifies just as many solutions as TVACPSO-mesh, but DWPSO-ring identifies almost twice as many solutions as TVACPSO or DWPSO-mesh. In general, PSO-based solvers show the same or better performance if they use the ring topology in comparison to the mesh topology. Lastly, GBPSO and GWO attain the fewest solutions and have the slowest convergence rates.

Summary Our tests compare the obtained minima of twenty-three well-known benchmark problems. All the optimization methods being tested fail to identify solutions to four of the benchmark problems: Deceptive3, Dropwave, Rosenbrock, and Shubert (see Appendix A). The reference solution values are presented along the problem definitions.

We first perform tests on the original formulations of these problems. Direct and GWO have exceptionally good results, reaching 70% of solutions. GWO takes two orders of magnitude more evaluations to reach this result. SPSO and GCPSO present good and consistent

performance in both metrics. They identify between 40% and 60% of the solutions. The performance of PSO-based solvers, with exception of GBPSO, TVACPSO, and DAPSO, responds to change in topology in favour of using the ring topology for communication between particles. GBPSO and DAPSO identify the lowest number of solutions, and most of the time, they need more objective function evaluations than the other solvers.

Tests performed on the set of randomized benchmark problems confirm good and consistent performance of SPSO and GCPSO. Performance of these two solvers and CDIWPSO is close on both metrics. Direct identifies 20% of solutions to randomized problems. GWO also identifies 20% of solutions to these problems, but its converge rate is lower. Consistently with non-randomized tests, PSO-based solvers respond to change in topology. The ring topology demonstrates better performance in comparison to the mesh topology for SPSO, GCPSO, DWPSO, and CDIWPSO.

Overall, the presented metrics show significant variability in performance between randomized and non-randomized problems. The difference is especially visible for the Direct and GWO algorithms. Direct performs the first objective function evaluation at the origin of the given search space, and the origin is in several cases the location of the minimum of a benchmark problem. As a result, Direct exhibits an outstanding performance on the original formulations of those problems. The reason for the notable difference in performance of GWO is unclear. We find that performance of some PSO-based solvers responds to changes in the underlying communication topology. The ring topology frequently achieves better results. On this class of problems, SPSO and GCPSO exhibit the highest number of identified solutions over all tests, and they need fewer function evaluations than other solvers. The best results, which are consistent for all randomized problems, are reached by SPSO-ring and GCPSO-ring.

4.3 Models

The optimization methods from Chapter 2 are tested on four classes of models. These classes contain a list of search spaces and a list of optimization problems. Benchmark problems, such as those presented in the previous section, are an example of a class with multiple

models.

This section presents the remaining three classes of models. The experiment methodology is similar to the one that was used to generate data for benchmark problems. That methodology is described in Section 4.1. The following subsections present how the data are generated, the metrics generated based on that data, and observations based on these metrics.

The first two classes, the Narrow Escape Problem and Gold Particle Freezing, are tested on randomized versions of the problems. Each solver attempts to solve five variants of each problem based on the underlying model. The third project, Rational Material Design, is used for testing four selected optimization methods on a non-randomized problem.

4.3.1 Narrow Escape Problem

The Narrow Escape Problem (NEP) is a problem of calculating the mean first passage time that a Brownian particle requires on average to escape a domain that is confined by a reflecting boundary. The boundary contains W circular windows through which a particle can escape, as illustrated in Figure 4.9. The problem arises in biology, where it may be used to model diffusing ions and the time they need to find an open channel in a cell membrane [12].

The windows are characterized by a radius and an electrostatic capacitance. The difference in radii divides the windows into two groups. The capacitance introduces a repulsive force between the windows within each group and between the groups. The pairwise interaction force between two windows is expressed by

$$h(x_i; x_j) = \frac{1}{|x_i - x_j|} - \frac{1}{2}\log|x_i - x_j| - \frac{1}{2}\log(2 + |x_i - x_j|),$$

where x_i and x_j denote the position of a window. The total interaction energy between all windows is defined as

$$\tilde{H}(x_1, \ldots, x_{2W}) = \sum_{i=1}^{W}\sum_{j=i+1}^{W} h(x_i; x_j) + \alpha\sum_{i=1}^{W}\sum_{j=W+1}^{2W} h(x_i; x_j) + \alpha^2\sum_{i=W+1}^{2W}\sum_{j=i+1}^{2W} h(x_i; x_j),$$

where α is the ratio between the radii of the larger and the smaller window sizes.

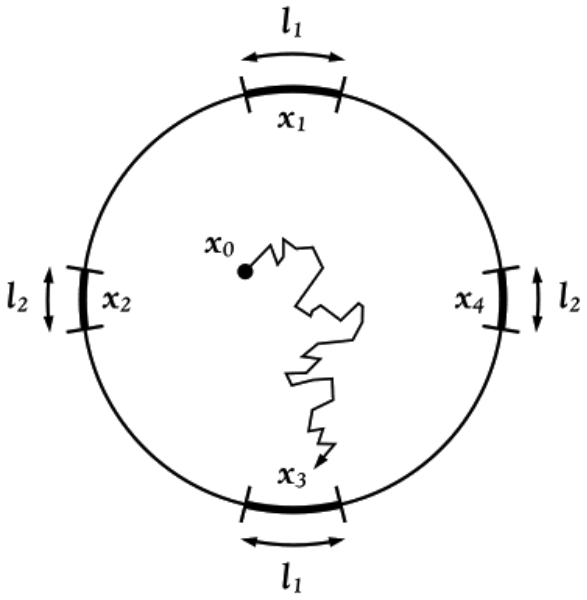

Figure 4.9: Random path of a particle in 2D variant of NEP

This model is used for designing a unit sphere with $2W$ windows on its surface in a way that minimizes escape time for a randomly moving particle that is trapped in a unit sphere. The decision vector in this problem represents the spherical coordinates of the window locations on the sphere. The objective function defines the total interaction energy $\tilde{H}$ using the model.

4.3.1.1 Settings

The presented experiments follow the scheme described in Section 4.1.4. We study several variants of NEP with two kinds of traps. These variants differ in the number of windows on a sphere; we solve cases for $2W \in 2, 4, 6, \ldots, 30$. W of these windows have a radius of r, and the other W have a radius of $10r$. The results are presented in the next subsection.

Reference solutions for $2W = 8$ and $2W = 10$ were published in [12]. For $2W = 8$, the global minimum is $\tilde{H} = -163.61503$; for $2W = 10$ the global minimum is $\tilde{H} = -198.80759$. These solutions are reproduced in our experiments. The reference solutions for other cases

are unknown, however, we can reproduce the two known solutions. For our tests, we assume that the best solution to a problem that is attained by any of the tested methods is the reference solution to that problem. The reference solutions that we use in tests are presented in Table 4.7.

Table 4.7: Best solutions identified in the NEP

Variant	Best solution
nep:2W=2	−5.39720771
nep:2W=4	−58.76585512
nep:2W=6	−112.90225971
nep:2W=8	−163.61502677
nep:2W=10	−198.80758824
nep:2W=12	−224.95410626
nep:2W=14	−230.99133965
nep:2W=16	−220.67723388
nep:2W=18	−191.03440964
nep:2W=20	−139.80802861
nep:2W=22	−66.37038352
nep:2W=24	26.54884125
nep:2W=26	145.45618027
nep:2W=28	285.99700101
nep:2W=30	451.49693061

4.3.1.2 Results

This subsection presents the data collected from experiments in the form of the WDL metric and the performance profiles metric. We use a modified version of the experiment methodology presented in Section 4.1. In these tests, only randomized sets of problems are used; tests performed on benchmark problems also use non-randomized problems. Table 4.7 presents the best solutions to each variant of NEP.

Table 4.8: NEP, 5 problem variants, $N = 100{,}000$

(a) tolerance: 10^{-5}

Solver	W	D	L
CDIWPSO-mesh	2	3	10
CDIWPSO-ring	0	3	12
DAPSO-mesh	1	3	11
DAPSO-ring	0	1	14
DWPSO-mesh	2	3	10
DWPSO-ring	0	3	12
Direct	0	2	13
GBPSO-mesh	1	3	11
GBPSO-ring	0	1	14
GCPSO-mesh	0	6	9
GCPSO-ring	0	3	12
GWO	0	1	14
SPSO-mesh	0	6	9
SPSO-ring	0	3	12
TVACPSO-mesh	3	3	9
TVACPSO-ring	0	2	13

(b) tolerance: 10^{-8}

Solver	W	D	L
CDIWPSO-mesh	2	3	10
CDIWPSO-ring	1	3	11
DAPSO-mesh	2	3	10
DAPSO-ring	0	1	14
DWPSO-mesh	2	3	10
DWPSO-ring	0	2	13
Direct	0	2	13
GBPSO-mesh	0	2	13
GBPSO-ring	0	1	14
GCPSO-mesh	0	6	9
GCPSO-ring	0	3	12
GWO	0	1	14
SPSO-mesh	0	6	9
SPSO-ring	0	3	12
TVACPSO-mesh	2	3	10
TVACPSO-ring	0	2	13

Table 4.9: NEP, 5 problem variants, $N = 500{,}000$

(a) tolerance: 10^{-5}

Solver	W	D	L
CDIWPSO-mesh	2	4	9
CDIWPSO-ring	3	3	9
DAPSO-mesh	1	3	11
DAPSO-ring	0	2	13
DWPSO-mesh	0	4	11
DWPSO-ring	0	3	12
Direct	0	2	13
GBPSO-mesh	0	3	12
GBPSO-ring	0	2	13
GCPSO-mesh	0	4	11
GCPSO-ring	0	4	11
GWO	0	2	13
SPSO-mesh	0	4	11
SPSO-ring	2	4	9
TVACPSO-mesh	1	3	11
TVACPSO-ring	1	3	11

(b) tolerance: 10^{-8}

Solver	W	D	L
CDIWPSO-mesh	3	5	7
CDIWPSO-ring	2	4	9
DAPSO-mesh	1	3	11
DAPSO-ring	0	2	13
DWPSO-mesh	0	4	11
DWPSO-ring	0	4	11
Direct	0	2	13
GBPSO-mesh	0	3	12
GBPSO-ring	0	2	13
GCPSO-mesh	0	4	11
GCPSO-ring	0	5	10
GWO	0	1	14
SPSO-mesh	0	4	11
SPSO-ring	2	5	8
TVACPSO-mesh	1	3	11
TVACPSO-ring	0	2	13

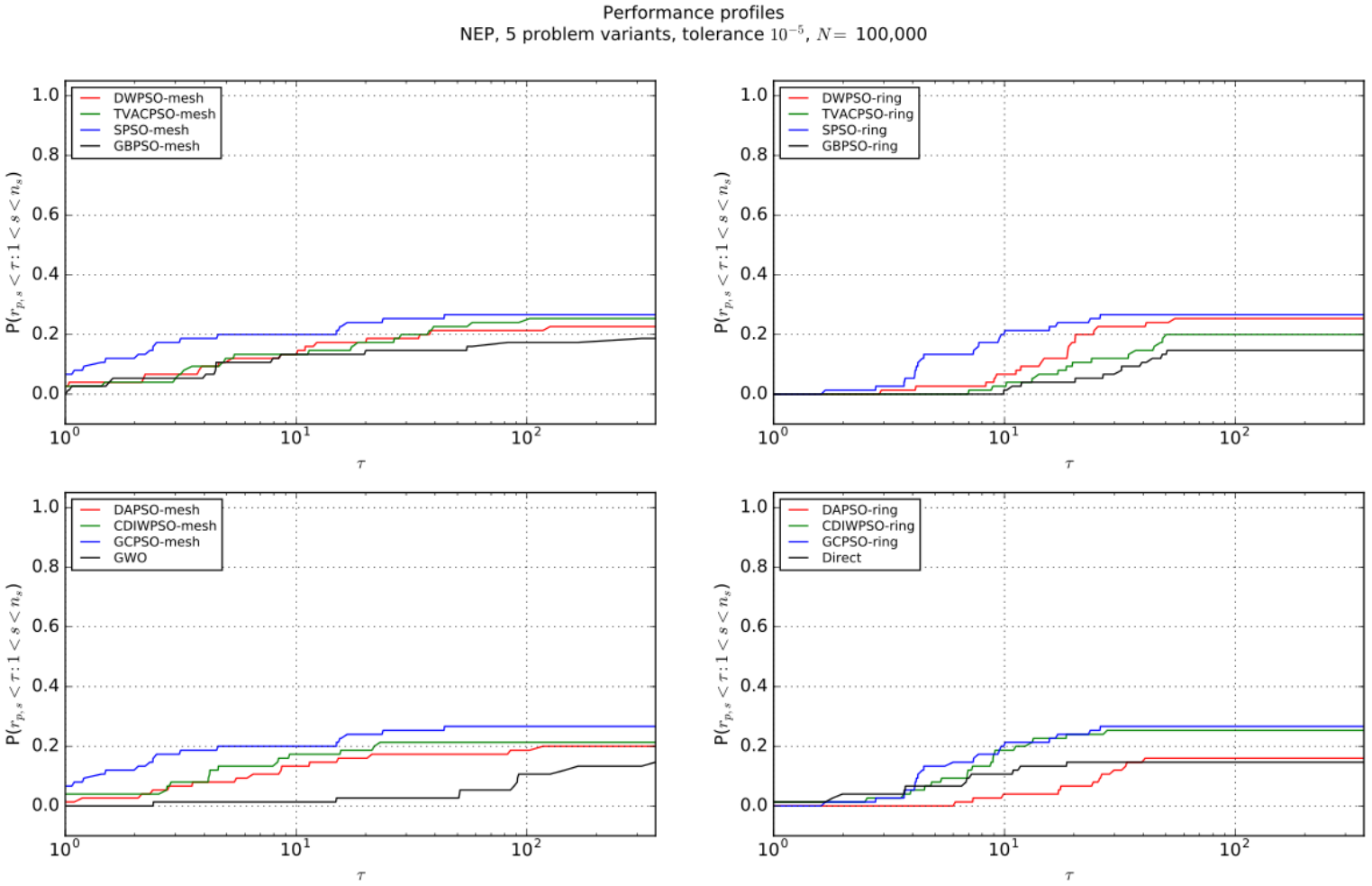

Figure 4.10: NEP, 5 problem variants, tolerance 10^{-5}, $N = 100{,}000$

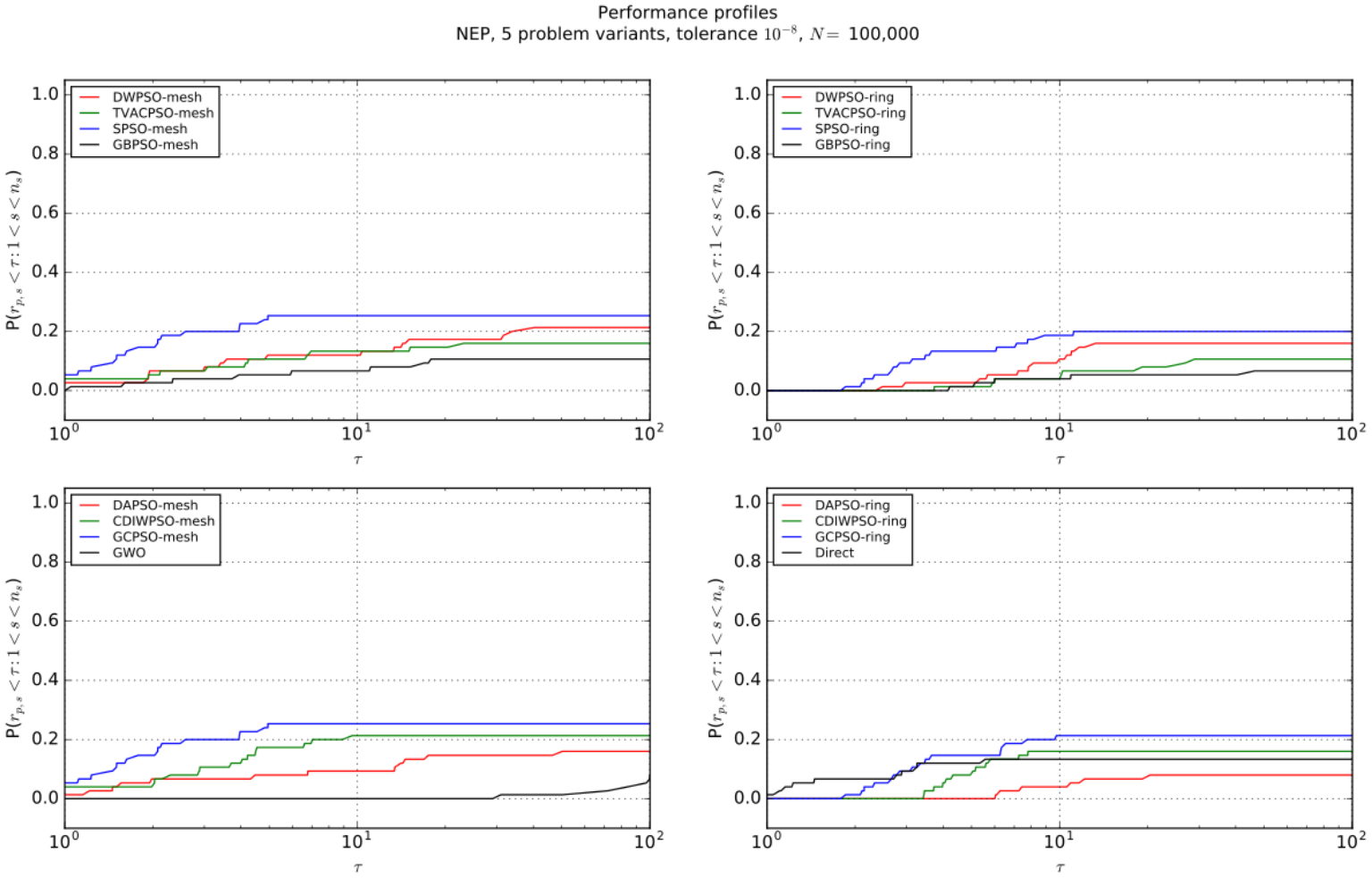

Figure 4.11: NEP, 5 problem variants, tolerance 10^{-8}, $N = 100{,}000$

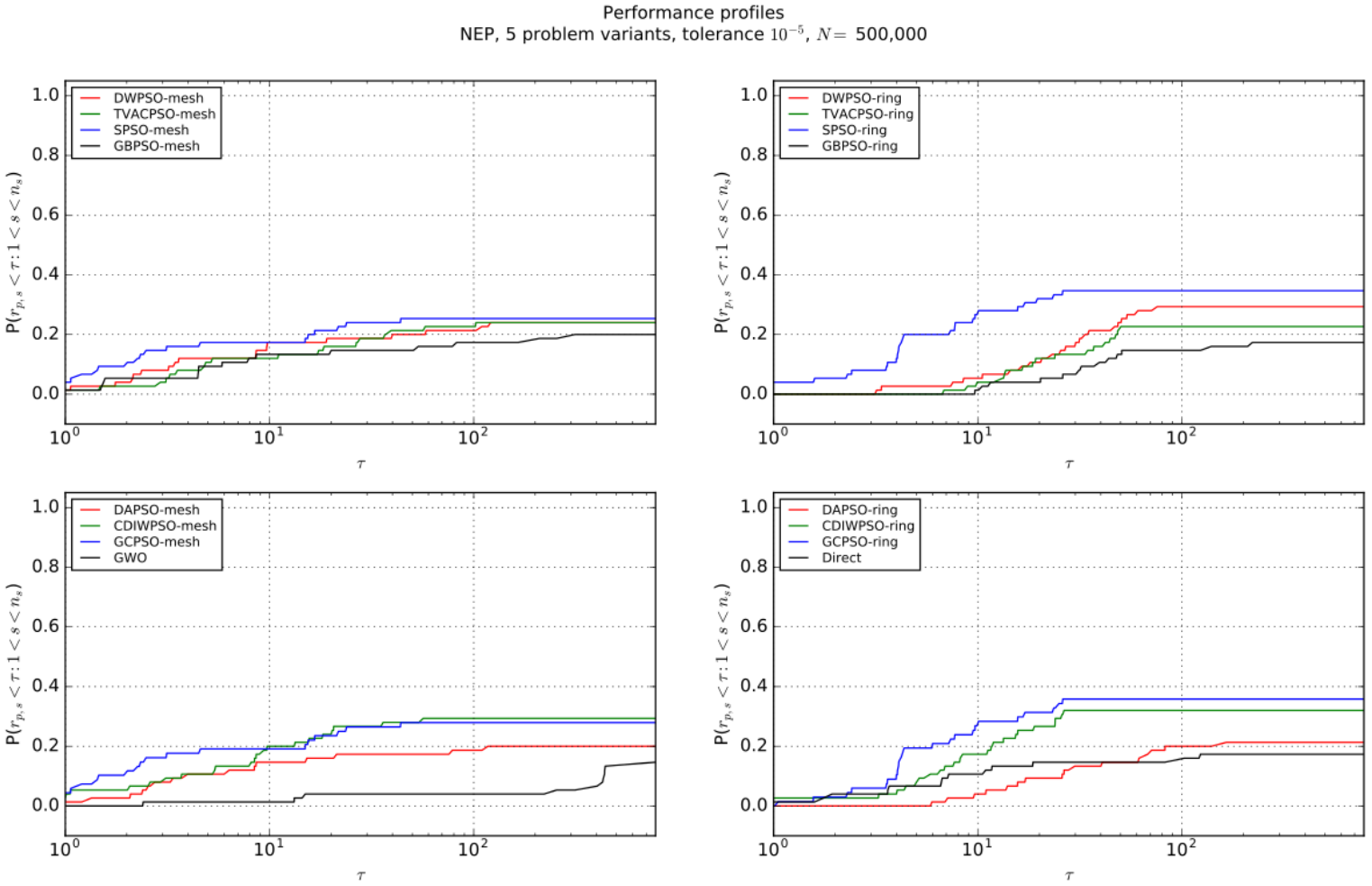

Figure 4.12: NEP, 5 problem variants, tolerance 10^{-5}, $N = 500{,}000$

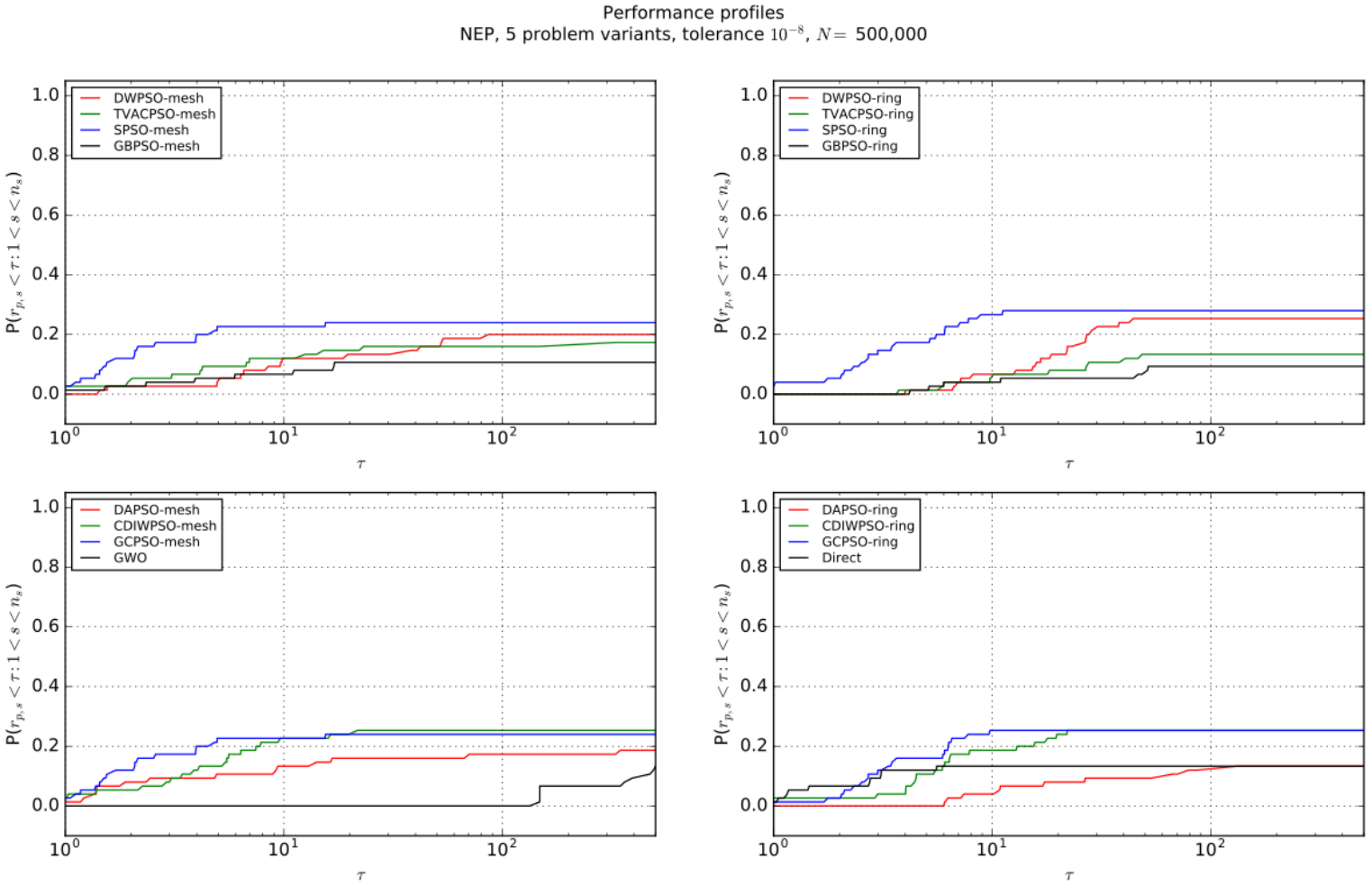

Figure 4.13: NEP, 5 problem variants, tolerance 10^{-8}, $N = 500{,}000$

4.3.1.3 Observations

Analysis of our results identifies a few solvers that consistently perform better than others. The SPSO performance profile displays outstanding performance of this method. SPSO and GCPSO are similar algorithms, and their performance profiles are close. However, the data show significant difference in their WDL and performance profile metrics. CDIWPSO attains top results but only when using the mesh topology. Additionally, the performance of the ring and the mesh topology depends on the number of available evaluations. A higher number of evaluations increases the relative performance of the ring topology. For the fewer evaluations, the mesh topology attains better results. This analysis is presented in more detail in the following paragraphs.

We begin with an analysis of the WDL metric. Tests with 100,000 evaluations reveal a group of solvers that achieve superior performance on the collection of randomized NEP variants. The collected data show that the mesh topology used in PSO-based methods achieves better results than the ring topology. SPSO-mesh and GCPSO-mesh identify 6 solutions, representing the highest WDL score for these test conditions. The rest of the PSO-based methods that use mesh topology, except for GBPSO, find four to five solutions, depending on the used tolerance. GBPSO-mesh method identifies four solutions with the tolerance of 10^{-5} and two solutions with the tolerance of 10^{-8}. The two non-PSO-based methods have the lowest scores: Direct identifies two solutions and GWO identifies one solution.

The analysis of performance profiles shows similar results. With 100,000 evaluations SPSO and GCPSO exhibit superior performance. SPSO-mesh and GCPSO-mesh find solutions faster, and they find more of them than any other solver. In general, PSO-based methods that use the mesh topology start to find solutions faster than the same methods based on the ring topology. Ignoring topology, the individual performance of each of the PSO-based methods is similar. Direct and GWO achieve the worst results in these tests.

In tests that use 500,000 evaluations, the WDL metric shows that both topologies used in the PSO-based solvers show similar performance. CDIWPSO-mesh, CDIWPSO-ring, and SPSO-ring identify at least six solutions regardless of the tolerance used. This is the highest

WDL score for these test settings. With a lower limit of function evaluations, the use of the mesh topology finds more solutions than the ring topology. It also finds the solutions in less time. Performance profiles confirm that result.

In summary, the SPSO method achieves the best results on the Narrow Escape problem. SPSO and GCPSO are similar algorithms, and both present almost identical performance in all tests. CDIWPSO identifies as many solutions as GCPSO and SPSO; however, it converges more slowly than the other two solvers. Regardless of the topology used, Direct, GWO, and GBPSO find similar numbers of solutions. DWPSO attains one of the highest numbers of identified best solutions. However, it takes more time to achieve these results compared to the best results. TVACPSO shows a similar to DWPSO-mesh; however, DWPSO-ring has better convergence and the ability to find more results than TVACPSO-mesh or TVACPSO-ring. GWO identifies a similar number of solutions to these of GBPSO and Direct, but it shows a significantly slower convergence than GBPSO and Direct. In general, GWO performs worst out of all solvers, regardless of the number of evaluations and tolerance. Performance profiles of GWO are similar regardless of the limit of evaluations. However, using the tolerance of 10^{-5}, the number of identified solutions increases in comparison to the lower tolerance.

With the mesh topology and high tolerance, SPSO, DWPSO, TVACPSO, and GBPSO perform similarly regardless of number of evaluations. The difference in performance becomes apparent in favour of SPSO when the ring topology is used. SPSO is able to find most solutions, and it does so with significantly fewer evaluations. The effect is magnified by the increase in the required number of function evaluations.

4.3.2 Gold Particle Freezing

The Gold Particle Freezing (GPF) model is a parametrized model of a nucleus formation process. Accurate modelling of this process is vital to major methods of producing materials made of nanoparticle structures. The optimization adjusts the model so that it closely matches reference data. The objective function defines discrepancy between reference data and the data that are predicted by the model that we use. The solution represents the model that best matches reference data.

The reference data consist of a set of pairs:

$$\{(k, \Delta G_k), k = 1, 2, \ldots, m\},$$

where k is the nucleus size, ΔG_k is the Gibbs free energy value for the nucleus size k, and m is the number of data points. The Gibbs free energy value represents the maximal rate at which a thermodynamic system can release energy. A model that best matches that data set is defined as a model minimizing the following expression:

$$f(\boldsymbol{x}) = \sum_{k=1}^{m} \left(\Delta G_k - \Delta G(\boldsymbol{x}; k)\right)^2,$$

where $\Delta G(\boldsymbol{x}; k)$ represents a model generated from a decision vector $\boldsymbol{x}$ with components that lie within boundaries of a given search space.

The GPF objective function is built upon a modified spherical cap model with variable contact angles that is based on models presented by Asuquo [6]. The model is parametrized by the decision vector $\boldsymbol{x}$ consisting of eight parameters described in Table 4.10. It estimates Gibbs free energy as a function of nucleus size k.

Table 4.10: Decision variables in the GPF problem

Name	Description
$\Delta\mu$	change in chemical potential
σ_{sv}	surface tensions on the solid-vapor interface
σ_{lv}	surface tensions on the liquid-vapor interface
σ_{sl}	surface tensions on the solid-liquid interface
τ	line tension
δ_{sv}	Tolman length on the solid-vapor interface
δ_{lv}	Tolman length on the liquid-vapor interface
δ_{sl}	Tolman length on the solid-liquid interface

Decision vectors must also satisfy the following inequality constraint:

$$\sigma_{sv} - \sigma_{lv} - \sigma_{sl} \leq 0.$$

It asserts favourable conditions for nucleation. This constraint is implemented with an exterior point method.

The contact angle α is expressed as:

$$\alpha = (\sigma_{lv} - \sigma_{sv})/\sigma_{sl}.$$

The radius of a particle cluster is expressed as:

$$R = \sqrt[3]{\frac{3PV_p(k_BT)^{3/2}}{4\pi}},$$

where P is the number of particles in the cluster, V_p is the volume of a single gold particle, k_B is the Boltzmann constant, and T is a given temperature.

The contact angle θ represents the size of a spherical cone in a sphere. We use it to calculate the spherical cap volume. For each nucleus size k between one and eighty particles, we determine θ that produces a spherical cap with a volume of 0. That value of volume occurs when there is no interface between solid and vapour, and, as a result, a solid is immersed in a liquid. The volume of a spherical cap is expressed by the function:

$$\begin{aligned} V_{\text{lens}}(\theta(k)) = \frac{\pi}{3}\Big(& R^3\left(\cos^3(\theta(k)) - 3\cos(\theta(k)) + 2\right) \\ & + R_p^3(k)\left(\cos^3(\gamma(k)) - 3\cos(\gamma(k)) + 2\right)\Big) \\ & - \frac{kV_p}{(k_BT)^{3/2}}, \quad \text{for} \quad \theta \in (0, \pi), \end{aligned}$$

where

$$\gamma(k) = \alpha - \theta(k)$$

and

$$R_p(k) = \frac{R\sin(\theta(k))}{\sin(\gamma(k))}.$$

The Gibbs free energy $\Delta G(\boldsymbol{x}; k)$, which is the value that we minimize, is defined for each nucleus size k as:

$$\begin{aligned} \Delta G(\boldsymbol{x}; k) = \Delta_\mu k & + \tau R_\lambda(k) \\ & + A_{sv}(k)\sigma_{sv}(1 - 2\delta_{sv}/R) \\ & - A_{sv}(k)\sigma_{lv}(1 - 2\delta_{lv}/R) \\ & + A_{sl}(k)\sigma_{sl}(1 - 2H(\gamma(k))\delta_{sl}/R_p(k)), \end{aligned}$$

where

$$R_\lambda(k) = 2\pi R\sin(\theta(k)),$$

$$A_{sv}(k) = 2\pi R^2(1 - \cos(\theta(k))),$$

$$A_{sl}(k) = 2\pi R_p^2(k)(1 - \cos(\gamma(k))),$$

τ is the line tension, and $H(\cdot)$ is the Heaviside step function.

4.3.2.1 Settings

We consider sixty-four instances of the GPF problem: a product of eight sets of reference data and a set of eight search spaces. The model characterizes a particle cluster that consists of $P = 456$ gold atoms. Reference data are organized by the initial temperature; the data represent the difference in the Gibbs free energy as a function of nucleus size. The reference data are given in the Appendix B. The search spaces are presented in Table 4.11.

Table 4.11: Search spaces in the GPF problem

Space	$\Delta\mu$	σ_{sv}	σ_{lv}	σ_{sl}	τ	δ_{sv}	δ_{lv}	δ_{sl}
`case0`	$(-5, 0)$	$(0, 2)$	$(0, 2)$	$(0, 2)$	$(-5, 5)$	$(0, 10)$	$(0, 10)$	$0, 10)$
`case1`	$(-5, 0)$	0.9	0.74	$(0, 2)$	0	0	0	0
`case2`	$(-5, 0)$	0.9	0.74	$(0, 2)$	0	$(0, 10)$	$(0, 10)$	$0, 10)$
`case3`	$(-5, 0)$	0.9	0.74	$(0, 2)$	$(-5, 5)$	0	0	0
`case4`	$(-5, 0)$	0.9	0.74	$(0, 2)$	$(-5, 5)$	$(0, 10)$	$(0, 10)$	$0, 10)$
`case5`	$(-5, 5)$	0.9	0.74	$(0, 2)$	$(-5, 5)$	$(0, 10)$	$(0, 10)$	$0, 10)$
`case6`	$(-5, 0)$	0.9	0.74	$(-2, 2)$	$(-5, 5)$	$(0, 10)$	$(0, 10)$	$0, 10)$
`case7`	$(-5, 0)$	$(0, 2)$	$(0, 2)$	$(0, 2)$	0	$(0, 10)$	$(0, 10)$	$0, 10)$

4.3.2.2 Results

This subsection presents the data collected from experiments on the GPF model. The best solution to a problem that is attained by any of the tested methods is assumed to be the reference solution to that problem. We use the same methodology that was used for analysing the NEP model, which is described in Section 4.3.1.2. Table 4.12 presents best solutions to GPF set of problems on all of the tested search spaces.

Table 4.12: Minima identified in the GPF problem

Space	$T = 650$	$T = 660$	$T = 670$	$T = 680$	$T = 690$	$T = 710$	$T = 730$	$T = 750$
`case0`	0.146571	0.652929	0.888788	1.130149	1.126301	1.148068	0.992876	3.055726
`case1`	1225.851	1229.452	1188.912	1208.501	1212.812	1206.178	1171.054	1069.477
`case2`	29.67588	33.70974	38.52012	41.67996	35.02530	35.67017	33.24278	36.35668
`case3`	146.2763	144.1392	164.6903	172.0467	165.4296	165.6344	142.0223	165.9332
`case4`	27.70958	31.59569	36.34548	39.41294	32.95949	33.57501	31.18707	34.40790
`case5`	27.70958	31.59569	36.34548	39.41294	32.95949	33.57501	31.18707	34.40790
`case6`	27.70958	31.59569	36.34548	39.41294	32.95949	33.57501	31.18707	34.40790
`case7`	0.514900	1.366752	4.985581	1.943307	1.677687	1.655758	3.229082	6.999192

Table 4.13: GPF, 5 problem variants, $N = 100{,}000$

(a) tolerance: 10^{-5}

Solver	W	D	L
CDIWPSO-mesh	12	16	36
CDIWPSO-ring	0	10	54
DAPSO-mesh	8	0	56
DAPSO-ring	2	0	62
DWPSO-mesh	9	16	39
DWPSO-ring	3	9	52
Direct	0	0	32
GBPSO-mesh	0	0	64
GBPSO-ring	2	0	62
GCPSO-mesh	0	24	40
GCPSO-ring	0	10	54
GWO	0	0	64
SPSO-mesh	0	24	40
SPSO-ring	0	11	53
TVACPSO-mesh	2	0	62
TVACPSO-ring	2	0	62

(b) tolerance: 10^{-8}

Solver	W	D	L
CDIWPSO-mesh	12	16	36
CDIWPSO-ring	0	8	56
DAPSO-mesh	8	0	56
DAPSO-ring	2	0	62
DWPSO-mesh	9	16	39
DWPSO-ring	3	8	53
Direct	0	0	32
GBPSO-mesh	0	0	64
GBPSO-ring	2	0	62
GCPSO-mesh	0	24	40
GCPSO-ring	0	8	56
GWO	0	0	64
SPSO-mesh	0	24	40
SPSO-ring	0	8	56
TVACPSO-mesh	2	0	62
TVACPSO-ring	2	0	62

Table 4.14: GPF, 5 problem variants, $N = 500{,}000$

(a) tolerance: 10^{-5}

Solver	W	D	L
CDIWPSO-mesh	7	17	40
CDIWPSO-ring	1	17	46
DAPSO-mesh	1	0	63
DAPSO-ring	1	0	63
DWPSO-mesh	11	20	33
DWPSO-ring	6	16	42
Direct	0	0	27
GBPSO-mesh	0	0	64
GBPSO-ring	0	0	64
GCPSO-mesh	0	22	42
GCPSO-ring	3	16	45
GWO	0	0	64
SPSO-mesh	2	22	40
SPSO-ring	3	20	41
TVACPSO-mesh	2	0	62
TVACPSO-ring	2	0	62

(b) tolerance: 10^{-8}

Solver	W	D	L
CDIWPSO-mesh	8	17	39
CDIWPSO-ring	1	16	45
DAPSO-mesh	2	0	61
DAPSO-ring	1	0	62
DWPSO-mesh	11	18	34
DWPSO-ring	6	16	41
Direct	0	0	23
GBPSO-mesh	0	0	63
GBPSO-ring	0	0	63
GCPSO-mesh	0	20	43
GCPSO-ring	3	9	52
GWO	0	0	64
SPSO-mesh	3	21	40
SPSO-ring	3	18	42
TVACPSO-mesh	2	0	61
TVACPSO-ring	2	0	62

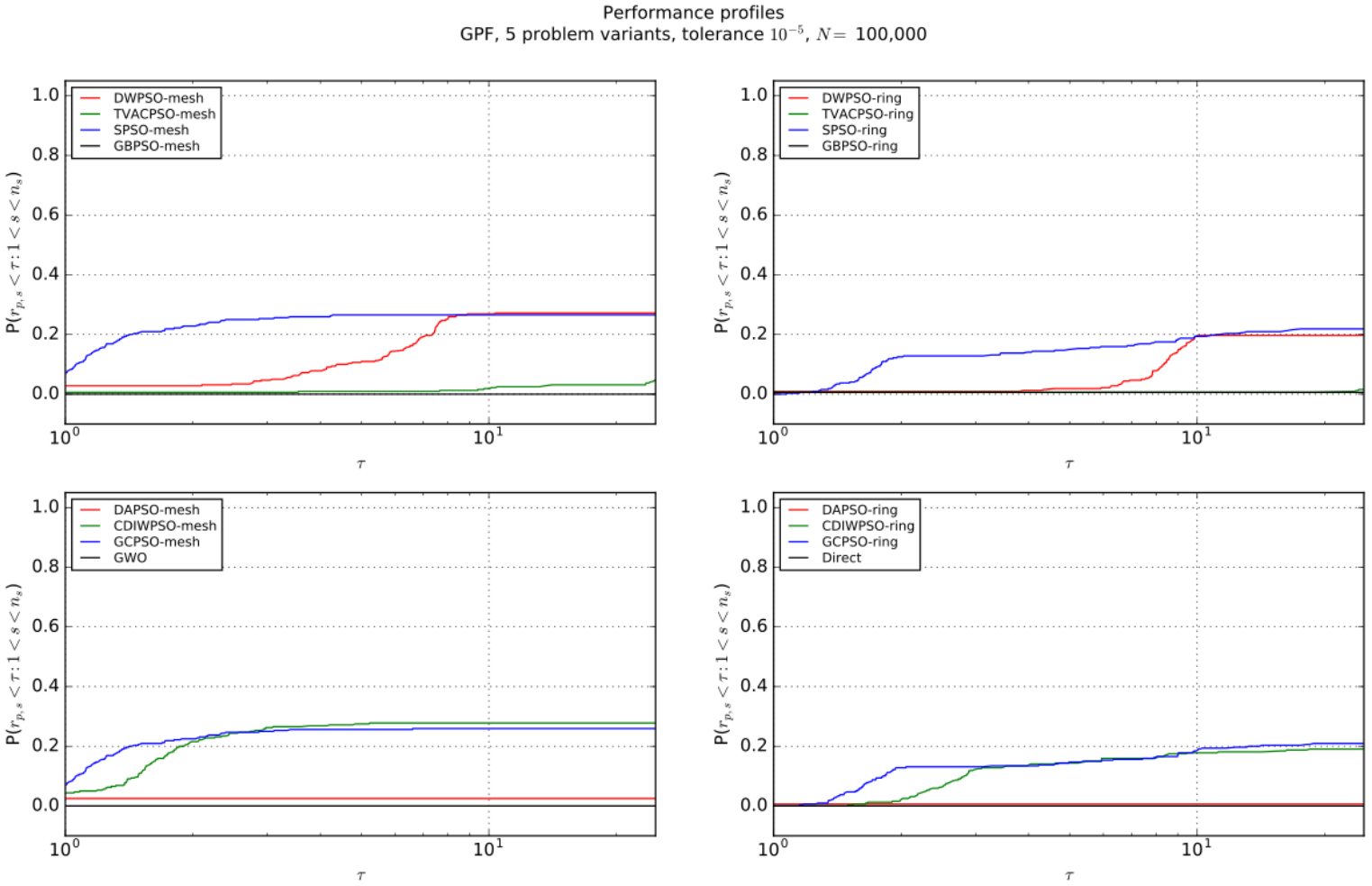

Figure 4.14: GPF, 5 problem variants, tolerance 10^{-5}, $N = 100{,}000$

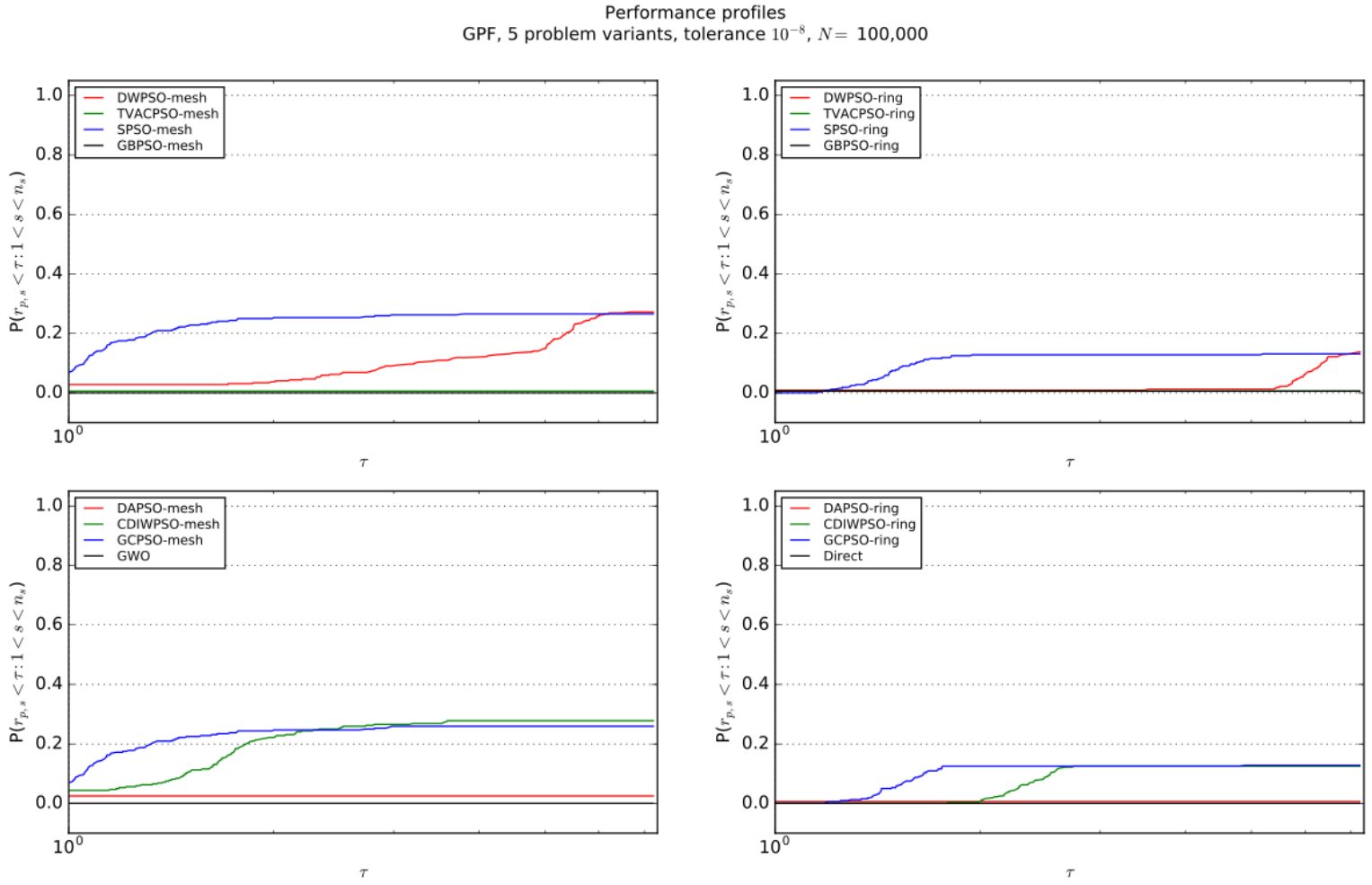

Figure 4.15: GPF, 5 problem variants, tolerance 10^{-8}, $N = 100{,}000$

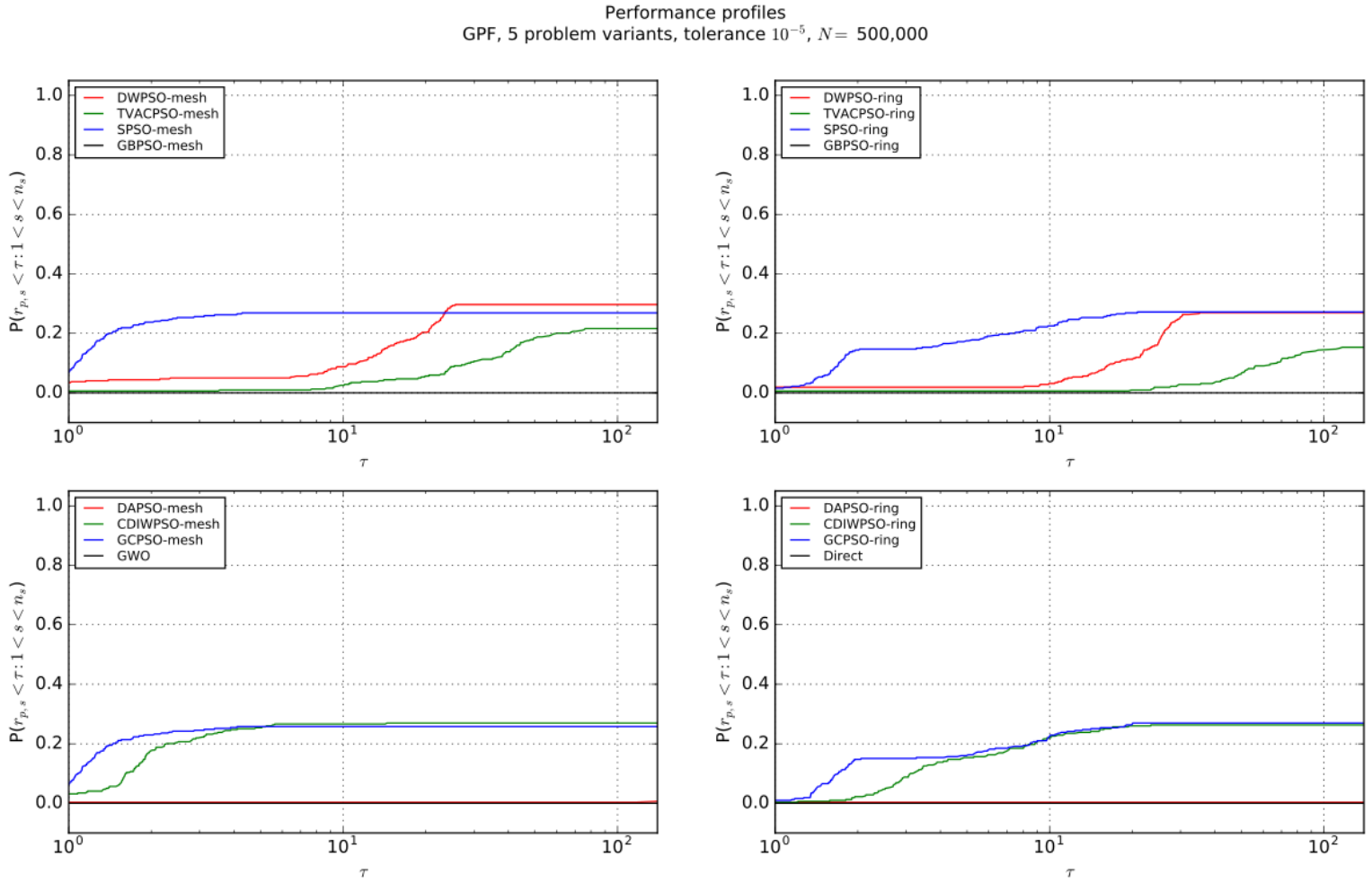

Figure 4.16: GPF, 5 problem variants, tolerance 10^{-5}, $N = 500{,}000$

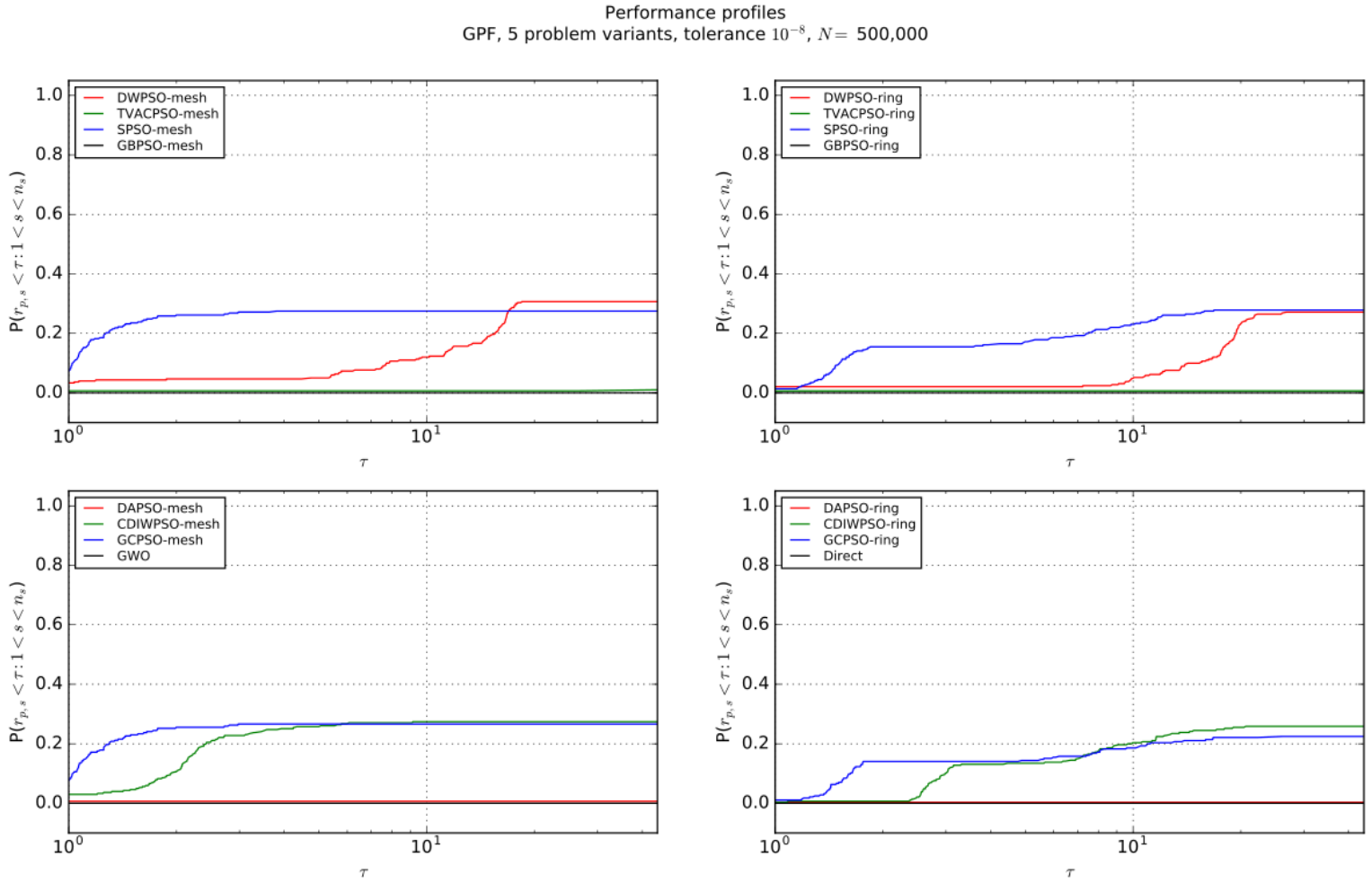

Figure 4.17: GPF, 5 problem variants, tolerance 10^{-8}, $N = 500{,}000$

4.3.2.3 Observations

The WDL metric finds four solvers that show good performance. With 100,000 evaluations, CDIWPSO-mesh achieves the highest number of wins and draws (twelve and sixteen). Regardless of the tolerances used, DWPSO-mesh, GCPSO-mesh, and SPSO-mesh have around twenty-four draws. However, the same solvers reach only half as many solutions if they use the ring topology. With 500,000 evaluations DWPSO-mesh finds the most solutions of all the solvers: thirty-one and twenty-nine, depending on the tolerance. Both results are the best in their categories. SPSO, GCPSO, and CDIWPSO get similar result (twenty-four solutions) regardless of the used topology. This is the second best group of solvers in this metric.

In tests that use 100,000 evaluations, SPSO and GCPSO have the best performance profiles with both topologies. CDIWPSO and DWPSO need more evaluations to find a similar number of solutions as these two solvers also with both topologies. DWPSO needs five to seven times more evaluations than the best result to find just as many solutions as SPSO. CDIWPSO reaches a similar number of solutions, but with only twice as many function evaluations as the best results. PSO-based solvers that use the ring topology need more evaluations to start finding solutions, and the number of identified solutions is lower in comparison to the mesh topology. Direct, GWO, DAPSO, TVACPSO, and GBPSO find nearly no solutions at all.

Increasing the limit of evaluations to 500,000 does not change the performance profiles of SPSO-mesh and GCPSO-mesh. With that limit, DWPSO-mesh takes eleven times more evaluations than these two solvers to solve the same number of problems; however, it has the highest number of identified solutions of all solvers. TVACPSO with 100,000 evaluations hardly finds any of the best solutions, giving it low scores in both metrics. With the limit of 500,000 evaluations TVACPSO finds 20% of solutions but only with the tolerance of 10^{-5}. With the tolerance of 10^{-8}, the method finds nearly no solutions. That could happen if TVACPSO solutions were within the range of $(10^{-7}, 10^{-5})$ of the best solutions.

In summary, in our tests at most 30% of all solutions are identified by each of the tested methods. The solutions are assumed to be the lowest values that any of these methods finds for the corresponding problems. SPSO and GCPSO are two solvers that find most solutions

of all solvers in nearly every test case, and they find these solutions quickly. CDIWPSO finds a similar number of solutions, but it takes more evaluations than SPSO and GCPSO. DWPSO takes an order of magnitude more evaluations to identify just as many solutions as the previous three solvers; however, after reaching that number of solutions, it finds a few more, and as a result it finds the highest number of solutions of all solvers. TVACPSO identifies 15% to 20% of all solutions but only with a limit of 500,000 evaluations and the tolerance of 10^{-5}. If the tolerance is low (10^{-8}), then TVACPSO does not identify any results that would be close to any of the best results. Table 4.12 contains the list of the best solutions for each problem in each of the search spaces presented in Table 4.11. Making some parameters constant does not change the minima attained in few cases. That suggests that the model may be over-fitting.

4.3.3 Rational Material Design

Rational Material Design (RMD) is a crystal structure prediction project in which we are given a chemical composition and we try to identify potential crystal structures based on that composition. An ability to predict feasible crystal structures and their properties based on a chemical composition can help in discovery of new crystal structures with valuable combinations of physical properties, e.g., with high hardness and high thermal or electrical conductivity. We compare the performance of four optimization methods in terms of their efficiency and their ability to identify potentially existing structures. Our results help to improve currently used methods of crystal structure prediction.

The RMD project attempts to reproduce a structure of the Si_2 crystal. The unit cell of this crystal is presented in Figure 4.18. We use GBPSO, SPSO-ring, GWO, and Direct optimization algorithms to generate candidate crystal structures. Our objective function uses the Vienna Ab-initio Simulation Package (VASP) [28, 29] to find the enthalpy of these structures; it uses constraints to eliminate the infeasible structures before their evaluation. A crystal structure with a minimal enthalpy has a good chance of being a stable structure and to exist in nature.

The procedure used in the RMD project has been proven successful in the past. CALYPSO is a software package that uses that procedure, and it has successfully identified

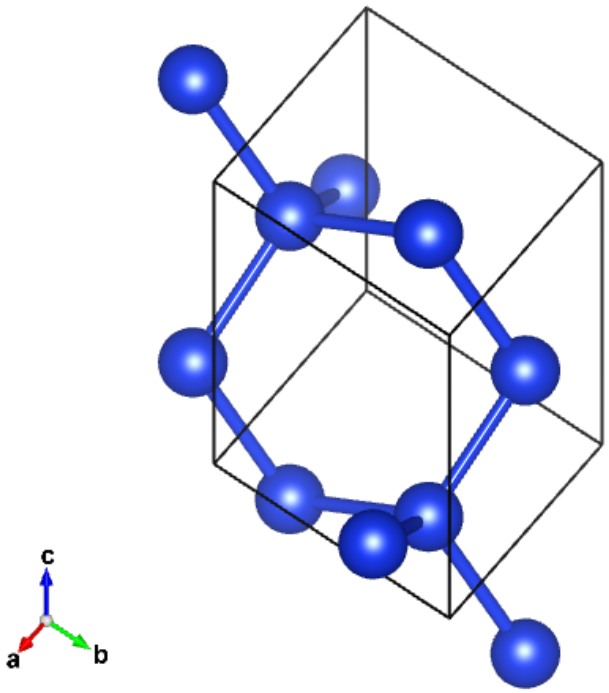

Figure 4.18: Si_2 crystal structure

new crystal structures [48]. It implements some advanced techniques that help in narrowing down the search space, and in turn they speed up the process. However, the method used for solving the underlying optimization problem (DWPSO), which is the fundamental part of this approach, is shown to perform sub-optimally on a set of benchmark problems (see Section 4.2). Our study compares the efficiency of four optimization methods applied to a simplified procedure that is used in CALYPSO.

The optimization problem is stated as finding a crystal structure, of a given chemical composition, that is characterized by the minimal value of the free energy. The free energy at the temperature of 0K, which we assume in simulations, is equivalent to the enthalpy of a system.

4.3.3.1 Settings

This subsection presents the objective function and the search space in more detail. It also briefly presents the configuration of the VASP simulator and the meaning of its input files

as described in the VASP documentation [30]. The minimal VASP configuration consists of three files: INCAR, POTCAR, and POSCAR. INCAR is the main configuration file for VASP. It contains parameters that control execution of the simulator, e.g., accuracy of simulations, algorithms used for interpolation, and a selection of libraries to link, to name a few. It is also the main source of errors and false results due to the number of available parameters; however, most of the default values can remain unchanged, and the simulator still provides correct results. POTCAR contains pseudo potentials of ions that comprise the simulated crystal structure. POSCAR defines the lattice geometry and the ionic positions within a unit cell.

An optional KPOINTS file defines the mesh size for the automatic generation of k-mesh, or, alternatively, the coordinates and weights of k-points that constitute the reciprocal lattice. The resulting mesh is used for sampling the physical properties of a unit cell. Our configuration uses the Monkhorst–Pack scheme to generate the complete k-mesh, and we only provide the number of subdivisions along the reciprocal lattice vectors. That number is derived from the lattice geometry that is provided in the POSCAR file. The reciprocal lattice and k-points exists in the K-space, i.e., the momentum space. The K-space is a Fourier transform of the Bravais lattice, which in turn exists in the real space, and it represents the real positions of atoms or ions [28, 29].

The objective function is defined over a search space that represents lattice geometry and ionic positions. The parameters that characterize the lattice include side lengths and angles between the sides as presented in Fig. 4.19. The bounds of that space are provided in Table 4.15. A decision vector is used to generate contents of POSCAR and KPOINTS files. Files that are used to generate the best of the identified solutions, with the exception of POTCAR, are provided in Appendix C.

The search space is adjusted to the characteristics of the Si_2 crystal. We additionally impose a constraint on the volume of a unit cell to eliminate simulations of infeasible structures. A single simulation takes approximately thirty minutes, and using the constrained search space provided in Table 4.15 significantly reduces the overall number of simulations and effectively the duration of the experiment. From Figure 4.19, the volume of a unit cell

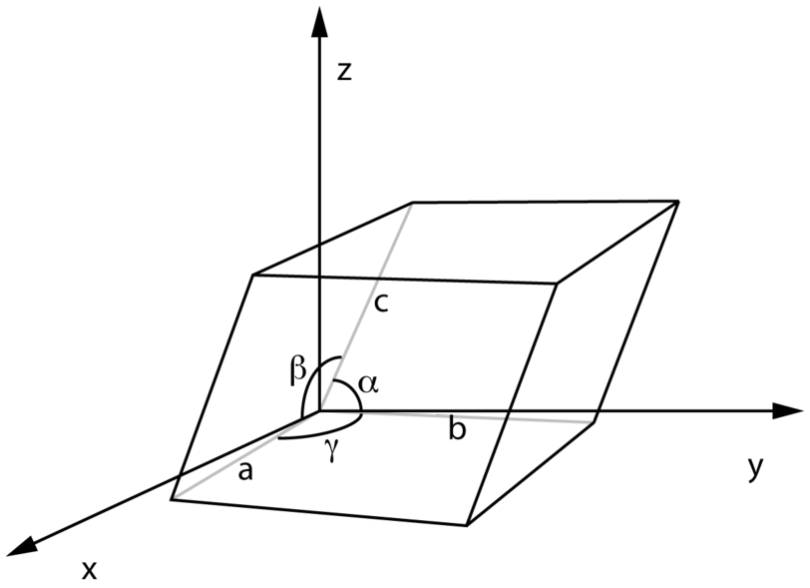

Figure 4.19: Lattice structure

is given by

$$V = abc\sqrt{1 + 2\cos\alpha\cos\beta\cos\gamma - \cos^2\alpha - \cos^2\beta - \cos^2\gamma},$$

and it is constrained to values that are feasible for the Si_2 crystal. The constraint is implemented with an exterior penalty method that adds a relatively high value to the violation of the allowed unit cell volume. This way solvers can differentiate between appropriate candidate structures that violate the constraint, and it helps in selecting candidates that are more likely to be feasible.

Two termination criteria are used in the experiment. The first criterion imposes the maximum number of successful simulations to one hundred. We define a successful evaluation as one that does not violate constraints and returns a result calculated with VASP. Effectively, it allows each algorithm to use the same number of simulations, and it limits the time of the experiment. The second criterion imposes a total number of function evaluations to 50,000. This number is set high to provide a margin for a number of decision vectors that violate the unit cell volume constraint. This limit is imposed to allow the use of GWO, which uses the number of remaining function evaluation to change its internal state.

Table 4.15: Search space in the RMD problem

	Variable	Range
Unit cell	α	(80°, 130°)
	β	(80°, 130°)
	γ	(80°, 130°)
	a	(2Å, 4Å)
	b	(2Å, 4Å)
	c	(2Å, 4Å)
Atom_1	x_1	(0, 1)
	y_1	(0, 1)
	z_1	(0, 1)
Atom_2	x_2	(0, 1)
	y_2	(0, 1)
	z_2	(0, 1)
Constraint	V	(35Å^3, 45Å^3)

4.3.3.2 Results

The methodology used for the RMD project differs from the methodology that is used in the other three experiments. The previous experiments randomize problems and use a set of a few solver configurations. The RMD experiment does not randomize the problem, and there is no variation in solver parameters, e.g., the number of available function evaluations or tolerance. The previous analyses present the WDL metric and the performance profile metric. This analysis presents the decrease in the current global minimum as a function of a number of the attempted objective function evaluations.

We test GBPSO, SPSO-ring, GWO, and Direct on two variants of the problem: first, with the KPOINTS file provided by a user, and second, without the KPOINTS file. Each solver is used exactly once with each version of the problem; we do not randomize the objective function in this experiment.

The best solutions obtained by the tested solvers are presented in Table 4.16.

Figures 4.20 and 4.21 represent the decrease in value of an objective function in relation to number of performed objective function evaluations. Infeasible solutions were removed from plots to emphasize difference between quality of feasible solutions and to report the number of evaluations it takes to find the first feasible solution.

Table 4.16: Minima identified in the RMD problem

Solver	With KPOINTS	Without KPOINTS
GBPSO	−10.9089437	−10.8468598
SPSO	−10.91030851	−10.84909535
GWO	−10.91051732	−10.84932729
Direct	−10.65299864	−10.25169214

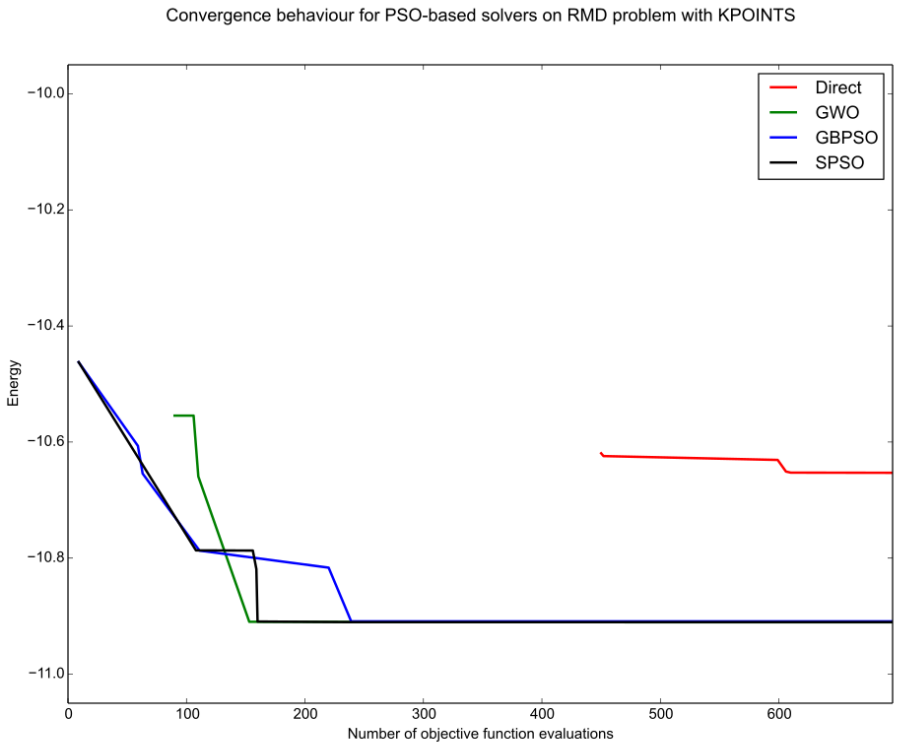

Figure 4.20: Convergence for PSO-based solvers on RMD problem with KPOINTS

4.3.3.3 Observations

Our experiment considers only the ability to identify feasible solutions, the convergence rate of optimization methods, and the minima that these methods can reach. However, a solution can represent a stable or a meta-stable structure. Another process is necessary to determine which of these two kinds of structures the solution represents. In our case, we reproduce an already existing crystal structure, and we have a priori knowledge that allows us to determine its kind. Based on that, we observe that GWO finds a meta-stable structure that is characterized by the lowest enthalpy found. SPSO yields a marginally worse minimum;

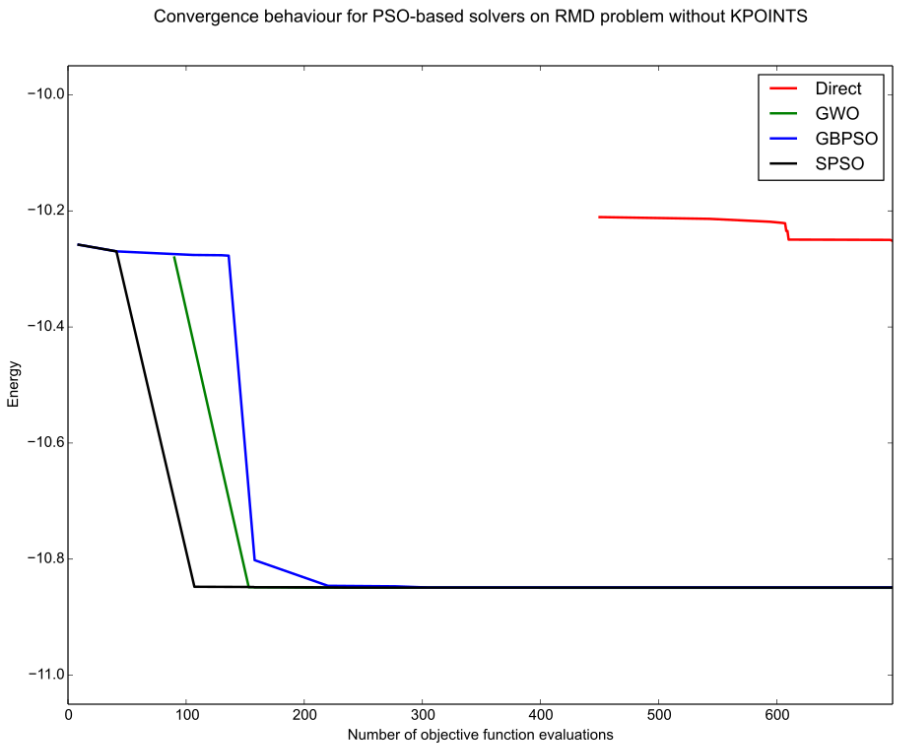

Figure 4.21: Convergence for PSO-based solvers on RMD problem without KPOINTS

however, it represents a stable crystal structure.

With KPOINTS file SPSO and GBPSO identify feasible solutions after only ten evaluations. GWO and Direct find feasible solutions after one hundred and four hundred fifty evaluations, respectively. Potential solutions, i.e., solutions that nearly reach the global minimum and thus can represent existing crystal structures, are first identified by SPSO and GWO that both need around one hundred fifty evaluations to find them. GBPSO needs roughly two hundred fifty evaluations to obtain its first of such solutions. Direct does not identify any of these.

SPSO and GBPSO improve on their solutions at an almost constant rate for the first one hundred evaluations. SPSO subsequently stagnates for another fifty evaluations, and then it reaches potential solutions after one hundred fifty evaluations in total. GBPSO has a similar convergence rate, but it needs two hundred fifty evaluations to become competitive. GWO initially stagnates for the first ten evaluations after it identifies its first feasible solution, but

then its solution rapidly decreases, and in result it reaches a competitive result before any other solver. Direct does not improve its feasible solutions significantly, and it does not reach a potential solution.

In summary, GWO attains a potential solution first, and it is marginally faster than SPSO. GBPSO needs 60% evaluations more than these solvers before it identifies a potential solution. Direct finds feasible solutions, but it does not identify any potential solutions.

Without KPOINTS file SPSO, GWO, and GBPSO find potential solutions after one hundred, one hundred fifty, and two hundred ten evaluations, respectively. Direct does not find any solution that are potential crystal structures. The convergence rates of SPSO and GWO are similar; once either of them finds a feasible solution, the subsequent solutions quickly decrease in value to potential solutions. GBPSO is the first solver to find a feasible solution but the last to find a solution that can represent a potential crystal structure. Direct marginally improves on its initial solution, but it still is not competitive with other methods.

Summary GWO finds the best solutions in both tests. After identifying the first feasible solution, the method quickly improves on it and locates some of the most promising decision vectors. The results of GWO and SPSO are highly similar both in value and convergence rate. The minima identified by SPSO and GWO are the same to five digits of precision. GBPSO takes twice as many evaluations as these two solvers, and its result get worse after the third significant digit. Direct identifies feasible solutions, but their values are relatively high in comparison to the minima identified by other solvers. It also requires most evaluations to find the first feasible solution. Overall, SPSO and GWO are the two solvers that identify possible crystal structures, and it takes them the fewest number of objective function evaluations of all the solvers considered.

4.4 General Observations

This section contains the overall summary of the observed performance of a few of the tested optimization methods. It also presents a few observations made during implementation of solvers, execution of test-cases, and analysis of the gathered data.

The most efficient solvers SPSO and GCPSO achieve the best results with the best performance most of the time. Most of their parameters are the same, and their execution proceeds almost identically. CDIWPSO achieves similar results, but it uses more function evaluations. This method uses different parameter values than SPSO and GCPSO. In that respect, CDIWPSO is similar to DWPSO; the only difference is the added chaotic number factor z that is used in the calculation of inertia in CDIWPSO. Aside from that, all other settings are the same for both solvers.

Termination criteria The only termination criterion imposed on solvers is the total number of function evaluations (as opposed to time constraints). Some solvers produce a significant overhead due to their complexity, e.g., Direct calculates a convex hull and thus uses more computation to handle its own logic than to perform evaluations of an inexpensive objective function. We constrain Direct only to a maximal number of function evaluations because benchmark problems are inexpensive to evaluate. That makes tests dependent only on the difficulty of objective functions. Additionally, timing test-cases in a distributed system can generate incorrect timings. Consequently, no test-cases use termination criterion based on execution time, and no metrics use timing information.

A number of solvers reach nearly the same solution with a low tolerance We test two tolerances for comparing solution values with a reference solution. A higher tolerance compares solutions with a lower precision, and, as a result, more solvers are considered successful in identifying them. It is most noticeable with the WDL metric. With a lower tolerance of 10^{-8}, solvers need to reach nearly identical results. A high number of solvers that converge to nearly the same solution (with a low tolerance) increases confidence in that a solution is the global solution. Of course, it does not imply that a solution is the global optimum.

Metrics and the number of the solved problems The randomized variant of the performance profile metric can misrepresent the number of distinct problems that are solved by a solver. Although one solver may be able to solve just one problem in a set but solve it in five random variants, another solver on the same set of problems may find solutions to

five separate problems but each in only one variant. The score is identical in both cases.

Profiling linearly decreasing PSO variants A data set that is used to generate a performance profile is ideally generated with solvers that have unlimited resources. The results of using algorithms that linearly decrease an internal parameter, i.e., DWPSO, TVACPSO, and GWO, depend on a limit of function evaluations imposed on a method, and it is necessary to provide this limit. For that reason, all tests have a limit on the number of evaluations.

Randomizing optimization problems We increase the confidence of our performance tests by randomizing behaviour of the objective function, but that randomization can change the value of the global minimum. The process is transparent to solvers; an objective function randomizes its argument and imposes search space bounds on the transformed argument. If the bounds are imposed with a simple clamping of the infeasible decision vector components, then the global minimum may not be in the range of the values that the randomized function returns. Additionally, if two solvers use different transformations on the same problem, then one of them may be able to identify solutions that are much better than the solutions of the other solver. For that reason, the transformations that randomize optimization problems must be surjective. The procedure that we use in our tests employs a transformation based on the sawtooth wave to arbitrarily extend a problem domain. As a result, all values that the original optimization functions returns (including the global minimum) the transformed function can also return.

Round-off errors increase the performance of Direct The Direct algorithm divides intervals in each dimension by three to generate positions of the new candidate solutions. When using floating-point arithmetic, such division leads to round-off errors in a ratio between the size of a rectangle and the objective function value at its center. As a result, the convex hull algorithm, that is used for selecting rectangles for evaluation, returns more rectangles of a similar size. In contrast, if we compare these ratios with a tolerance, then fewer rectangles are selected and evaluated in the subsequent iterations. We observe that using comparison with a tolerance decreases the performance of Direct.

Chapter 5

Conclusions

Optimization problems are common in science and engineering. They can be difficult to solve and time consuming, and frequently we only approximate solutions. Methods for solving optimization problems exploit properties of these problems to improve on execution time and quality of solutions. For example, a convex problem can be solved efficiently with a local optimization method. However, not all problems offer properties that allow such improvements. Selecting an appropriate method in these circumstances can significantly increase quality of solutions and reduce time necessary to find them.

The methodology described in Chapter 4 generates relative performance statistics for a set of optimization methods. Random samples are generated by randomizing the search spaces of a set of given optimization problems. Analysis of results shows that SPSO outperforms all tested methods on all tested classes of optimization problems. Using our methodology, we can quickly asses performance of optimization algorithms and identify those that perform best on a given class of problems.

The data for this study are generated and analyzed with the `pythOPT` problem-solving environment. This environment automates performing similar analyses and allows for their verification. The set of standard benchmark optimization problems as well as the tested optimization methods are part of `pythOPT`. All the methods are controlled through a set of parameters to facilitate creating and assessing new methods. The data from our experiments are processed and visualized with `pythOPT`, and the results of that process are presented in Chapter 4.

5.1 Overview

The numerical experiments presented in Chapter 4 are based on four sets of problems. We present performance metrics of sixteen solver configurations that we test with problems from three of these sets, and we analyse these metrics. Metrics for the fourth set of problems are simplified due to long running time of the objective function. We conclude that:

- `pythOPT` provides insightful performance metrics,
- SPSO demonstrates superior performance on all of our problem sets,
- problem randomization produces visible differences in algorithms performance,
- the win-draw-loss metric can be misleading.

In our experiments, each of the sixteen optimization method is tested with one hundred two optimization problems, each of which is solved in five randomized variants. This procedure is repeated with four different sets of termination criteria, giving a total of thirty-two thousand six hundred forty optimization cases. The `pythOPT` PSE distributes solving those cases on a computer cluster to improve on execution time; our experiments are embarrassingly parallel. We use a cluster of eighty workstations with quad-core `Intel(R) Core(TM) i7-6700 3.40GHz CPU`, and five servers with twenty-four core `Intel(R) Xeon(R) E5-2690 v3 2.60GHz CPU`. Our calculations take approximately four thousand CPU-hours. Data from all experiments are stored in a database, and all experiments can be reproduced and verified. The procedure we developed for our numerical experiments can be applied to any class of optimization problems. This combination of functionalities aids designing new optimization methods that are efficient and tailored for a given set (a class) of problems.

Measuring the performance of optimization methods is often based on a popular set of benchmark problems. We use twenty-three such problems in the first experiment set of our research. We find that the performance of these methods that is measured with the original versions of these problems is significantly different from the performance that is based on the randomized variants of these problems. Experiments with other two sets of problems

use only randomized problems. The last problem set (Section 4.3.3) is infeasible for such analysis, and it uses a simplified procedure.

Our analysis shows that the relative mutual performance between optimization methods depends on the set of optimization problems, the resources available to a solver, desired precision, and, in the case of PSO-based solvers, the topology of the communication network used by particles. However, two methods, SPSO and GCPSO, continue to outperform other algorithms in nearly all test cases. These methods are similar and our results show that the differences between their performance is negligible.

The GBPSO algorithm was introduced in [25]. SPSO [11] is a GBPSO version with improvements that were developed for more than a decade, e.g., parameter values, communication topology, number of particles. SPSO has been proposed as a reference PSO method. Our results show that SPSO achieves the best results out of all PSO variants implemented in `pythOPT`.

The No Free Lunch theorem for optimization states that there is no one optimization method better than all other methods for solving all optimization problems [49]. However, we can evaluate and compare performance of a set of methods on a set of problems. Selection of the input problems that can provide reliable performance metrics is an important element of this process. We assume that a higher variation in problems from a given set offers a better base for evaluating performance metrics and for comparing them. We observe that input problem randomization significantly influences our metrics. With the randomized problem set we identify methods that perform better than others and are independent of particular problem variants.

Our software, `pythOPT` offers a convenient way to assess optimization methods. It evaluates their performance on a set of problems. Problem randomization and performance metrics help to identify optimization methods that are optimal for that set. That potentially improve quality of solutions as well as lowers the amount or required resources. The `pythOPT` PSE is a robust, distributed, and reliable environment for testing optimization methods and solving optimization problems. Given an experiment design as an input parameter `pythOPT` manages its execution, data acquisition, and data visualization. These features provide means for rapid development of optimization methods with fast and comprehensive testing.

5.2 Future Work

The research and software presented in this work can be extended and built upon. Optimization method libraries, e.g., the Open Optimization Library (OOL), should be integrated with `pythOPT`. Every additional method provides additional samples for mutual performance tests. Our PSE provides examples of integrating `FORTRAN` and Python optimization solvers.

In our experiments, we use only two tolerances and two limits on objective function evaluations. With PSO-based methods, we additionally vary two network topologies of communication between particles. This choice is dictated by exponential increase in computation resources requirements. A better coverage of this parameter space and increasing the number of random samples would provide more accurate information for analyses. The source code of `pythOPT` contains the design of our experiments that can be adjusted to generate that coverage.

Currently, we provide implementations of nine algorithms. In our tests, we vary the topology of seven of them, and we treat these variants as separate solvers; this adds up to sixteen configurations that are tested. Two other optimization solvers, VTDirect95 and LGO, are also supported. However, these solvers are not fully compatible with our system. For example, the control over their termination is limited, and currently they can be used only with objective functions provided as `FORTRAN` source code. As a result, we exclude them from our experiments. The PSE needs to be extended to support these and other solvers in a consistent manner. This will allow for choosing more appropriate solvers, and it will provide a better base for estimating the relative performance of new methods.

The win-draw-loss and performance profiles are used to analyze our experimental data. Implementations of other metrics will provide new insight into the performance of available optimization methods. Also, a more comprehensive interface to solver states and an ability to change these states facilitates development of new metrics. Finally, `pythOPT` needs an ability to confirm its analyses statistically.

Bibliography

[1] Nonlinear Systems Modeling and Optimization. http://pinterconsulting.com/. Accessed: 2016-12-27.

[2] VTDIRECT95. http://vtopt.github.io/vtdirect95/. Accessed: 2016-12-27.

[3] IEEE Standard for Information Technology–Systems Design–Software Design Descriptions. *IEEE STD 1016-2009*, pages 1–35, July 2009.

[4] G. Andreatta, F. Mason, and P. Serafini. *Advanced School on Stochastics in Combinatorial Optimization: CISM, Udine, Italy, September 22-25, 1986*. World Scientific, 1987.

[5] M. A. Arasomwan and A. O. Adewumi. *The Scientific World Journal*, volume 2013, chapter On the Performance of Linear Decreasing Inertia Weight Particle Swarm Optimization for Global Optimization. 2013.

[6] C. Asuquo. Phenomenological and semi-phenomenological models of nano-particle freezing. Master's thesis, University of Saskatchewan, 2009.

[7] J. Barrera and C. A. Coello Coello. *Test Function Generators for Assessing the Performance of PSO Algorithms in Multimodal Optimization*, pages 89–117. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[8] M. Bhattacharya. Reduced computation for evolutionary optimization in noisy environment. In *Proceedings of the 10th Annual Conference Companion on Genetic and Evolutionary Computation*, GECCO '08, pages 2117–2122, New York, NY, USA, 2008. ACM.

[9] A. J. Booker, J. E. Dennis, P. D. Frank, D. B. Serafini, and V. Torczon. *Optimization Using Surrogate Objectives on a Helicopter Test Example*, pages 49–58. Birkhäuser Boston, Boston, MA, 1998. ISBN 978-1-4612-1780-0.

[10] A. J. Booker, J. E. Dennis, P. D. Frank, D. B. Serafini, V. Torczon, and M. W. Trosset. A rigorous framework for optimization of expensive functions by surrogates. *Structural and Multidisciplinary Optimization*, 17:1–13, 1999.

[11] D. Bratton and J. Kennedy. Defining a Standard for Particle Swarm Optimization. In J. Kennedy, editor, *Proc. IEEE Swarm Intelligence Symposium SIS 2007*, pages 120–127, 2007.

[12] A. F. Cheviakov, A. S. Reimer, and M. J. Ward. Mathematical modeling and numerical computation of narrow escape problems. *Physical Review E - Statistical, Nonlinear, and Soft Matter Physics*, 85(2), 2012.

[13] M. Clerc and J. Kennedy. The particle swarm - explosion, stability, and convergence in a multidimensional complex space. *Evolutionary Computation, IEEE Transactions on*, 6(1):58–73, Feb 2002.

[14] C. A. C. Coello. Theoretical and numerical constraint-handling techniques used with evolutionary algorithms: a survey of the state of the art. *Computer Methods in Applied Mechanics and Engineering*, 191(11.12):1245 – 1287, 2002.

[15] A. R. Conn, K. Scheinberg, and L. N. Vicente. *Introduction to Derivative-Free Optimization*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2009.

[16] E. D. Dolan and J. J. Moré. Benchmarking optimization software with performance profiles. *Mathematical Programming*, 91:201–213, 2002.

[17] A. P. Engelbrecht. *Computational Intelligence: An Introduction*. Wiley, 2nd edition, 2007.

[18] Y. Feng, G. F. Teng, A. X. Wang, and Y. M. Yao. Chaotic inertia weight in particle swarm optimization. In *Proceedings of the Second International Conference on Innovative Computing, Information and Control*, ICICIC '07, pages 475–, Washington, DC, USA, Sept 2007. IEEE Computer Society.

[19] J. Fernández-Martínez and E. García-Gonzalo. What makes particle swarm optimization a very interesting and powerful algorithm? In B. Panigrahi, Y. Shi, and M.-H. Lim, editors, *Handbook of Swarm Intelligence*, volume 8 of *Adaptation, Learning, and Optimization*, pages 37–65. Springer Berlin Heidelberg, 2011.

[20] P. Gill, W. Murray, and M. Wright. *Practical optimization*. Academic Press, London, 1981.

[21] A. Homaifar, C. X. Qi, and S. H. Lai. Constrained optimization via genetic algorithms. *SIMULATION*, 62(4):242–253, 1994.

[22] R. Hooke and T. A. Jeeves. "Direct Search" Solution of Numerical and Statistical Problems. *Journal of The ACM*, 8:212–229, 1961.

[23] J. Joines and C. Houck. On the use of non-stationary penalty functions to solve nonlinear constrained optimization problems with GA's. In *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on*, pages 579–584 vol.2, Jun 1994.

[24] D. R. Jones, C. D. Perttunen, and B. E. Stuckman. Lipschitzian optimization without the Lipschitz constant. *Journal of Optimization Theory and Applications*, 79:157–181, 1993.

[25] J. Kennedy and R. Eberhart. Particle swarm optimization. *Proceedings, IEEE International Conference on Neural Networks*, 4:1942–1948, 1995.

[26] T. G. Kolda, R. M. Lewis, and V. Torczon. Optimization by direct search: New perspectives on some classical and modern methods. *SIAM Review*, 45:385–482, 2003.

[27] O. Kramer, D. E. Ciaurri, and S. Koziel. Derivative-free optimization. In S. Koziel and X.-S. Yang, editors, *Computational Optimization, Methods and Algorithms*, volume 356 of *Studies in Computational Intelligence*, pages 61–83. Springer, 2011.

[28] G. Kresse and J. Furthmüller. Efficient iterative schemes for ab initio total-energy calculations using a plane-wave basis set. *Phys. Rev. B*, 54:11169–11186, Oct 1996.

[29] G. Kresse and J. Furthmüller. Efficiency of ab-initio total energy calculations for metals and semiconductors using a plane-wave basis set. *Computational Materials Science*, 6 (1):15 – 50, 1996.

[30] G. Kresse, M. Marsman, and J. Furthmüller. Vienna Ab-initio Simulation Package. https://cms.mpi.univie.ac.at/vasp/vasp.pdf. Accessed: 2017-01-29.

[31] Q. Liu and W. Cheng. A modified DIRECT algorithm with bilevel partition. *Journal of Global Optimization*, 60(3):483–499, 2014.

[32] J. C. Meza and M. L. Martinez. On the use of direct search methods for the molecular conformation problem. *Journal of Computational Chemistry*, 1994.

[33] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer, 1996.

[34] M. M. Millonas. Swarms, Phase Transitions, and Collective Intelligence. 1993.

[35] S. Mirjalili, S. M. Mirjalili, and A. Lewis. Grey wolf optimizer. *Adv. Eng. Softw.*, 69: 46–61, Mar. 2014. ISSN 0965-9978.

[36] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer, New York, 2nd edition, 2006.

[37] B. K. Panigrahi, Y. Shi, and M.-H. Lim. *Handbook of Swarm Intelligence: Concepts, Principles and Applications*, volume 8 of *Adaptation, Learning, and Optimization*. Springer Publishing Company, Incorporated, 1st edition, 2011.

[38] A. Ratnaweera, S. K. Halgamuge, and H. C. Watson. Self-organizing hierarchical particle swarm optimizer with time-varying acceleration coefficients. 8:240–255, 2004.

[39] J. R. Rice and R. F. Boisvert. From scientific software libraries to problem-solving environments. *IEEE Computational Science and Engineering*, 3:44–53, 1996.

[40] R. Salomon. Re-evaluating Genetic Algorithm Performance under Coordinate Rotation of Benchmark Functions. *BioSystems*, 39:263–278, 1996.

[41] D. Sedighizadeh and E. Masehian. Particle swarm optimization methods, taxonomy and applications. *International Journal of Computer Theory and Engineering*, 1(5):486–502, 2009.

[42] Y. Shi and R. Eberhart. A modified particle swarm optimizer. In *1998 IEEE International Conference on Evolutionary Computation Proceedings. IEEE World Congress on Computational Intelligence (Cat. No.98TH8360)*, pages 69–73, May 1998.

[43] B. O. Shubert. A Sequential Method Seeking the Global Maximum of a Function. *Siam Journal on Numerical Analysis*, 9, 1972.

[44] D. Simon. *Evolutionary Optimization Algorithms*. Wiley, 2013.

[45] V. Torczon. Pattern search methods for nonlinear optimization. *SIAG/OPT Views and News*, 6:7–11, 1995.

[46] I. C. Trelea. The particle swarm optimization algorithm: convergence analysis and parameter selection. *Information Processing Letters*, 85(6):317 – 325, 2003.

[47] F. van den Bergh and A. P. Engelbrecht. A new locally convergent particle swarm optimiser. In *IEEE International Conference on Systems, Man, and Cybernetics*, volume 3, 2002.

[48] Y. Wang, J. Lv, L. Zhu, and Y. Ma. Calypso: A method for crystal structure prediction. *Computer Physics Communications*, 183(10):2063 – 2070, 2012. ISSN 0010-4655.

[49] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *Trans. Evol. Comp*, 1(1):67–82, Apr. 1997. ISSN 1089-778X.

[50] J. Xin, G. Chen, and Y. Hai. A particle swarm optimizer with multi-stage linearly-decreasing inertia weight. In *2009 International Joint Conference on Computational Sciences and Optimization*, volume 1, pages 505–508, April 2009.

Appendix A

Benchmark problems

Definitions of benchmark problems that were used to evaluate algorithms. D represents the problem dimensionality. The search space is defined next. Any problem specific parameters are provided after.

Ackley $D = 30$, $\boldsymbol{x} \in [-30, 30]^D$, $a = 20$, $b = 0.2$, $c = 2\pi$

$$f(\boldsymbol{x}) = a + e - a \exp\left(- b \sqrt{\frac{1}{D} \sum_{d=1}^{D} x_d^2} \right) - \exp\left(\frac{1}{D} \sum_{d=1}^{D} \cos(c x_d) \right)$$

Global optimum: $f(\boldsymbol{x}^*) = 0$ for $\boldsymbol{x}^* = \mathbf{0}$.

Alpine $D = 10$, $\boldsymbol{x} \in [-10, 10]^D$

$$f(\boldsymbol{x}) = \sum_{d=1}^{D} |x_d \sin(x_d) + 0.1 x_d|$$

Global optimum: $f(\boldsymbol{x}^*) = 0$ for $\boldsymbol{x}^* = \mathbf{0}$.

Six-hump camel back $D = 2$, $\boldsymbol{x} \in [-2, 2]^D$

$$f(x_1, x_2) = \left(4 - 2.1 x_1^2 + \frac{x_1^4}{3}\right) x_1^2 + x_1 x_2 + (-4 + 4 x_2^2) x_2^2$$

Global optimum: $f(\boldsymbol{x}^*) = -1.0316$ at $\boldsymbol{x}^* = (0.0898, -0.7126)$, and $\boldsymbol{x}^* = (-0.0898, 0.7126)$.

De Jong 5 $D = 2$, $\boldsymbol{x} \in [-65.536, 65.536]^D$

$$f(x_1, x_2) = \left(0.002 + \sum_{d=1}^{25} \left[d + (x_1 - A_{1,d})^6 + (x_2 - A_{2,d})^6 \right]^{-1} \right)^{-1}$$

$$\boldsymbol{A} = \begin{bmatrix} -32 & -16 & 0 & 16 & 32 & -32 & \dots & 0 & 16 & 32 \\ -32 & -32 & -32 & -32 & -32 & -16 & \dots & 32 & 32 & 32 \end{bmatrix}.$$

Deceptive Type 3 $D = 30$, $\boldsymbol{x} \in [0,1]^D$, $\beta = 2.5$, $\check{\boldsymbol{c}} \in [0,1]^D$

$$
\begin{aligned}
f(\boldsymbol{x}) &= -\left[\frac{1}{n}\sum_{d=1}^{30} g_1(x_d)\right]^{\beta}, \\
\text{where } g_1(x_d) &= \begin{cases} -\frac{x_d}{\check{c}_d} + \frac{4}{5}, & \text{if } 0 \le x_d \le \frac{4}{5}\check{c}_d, \\ \frac{5x_d}{\check{c}_d} - 4, & \text{if } \frac{4}{5}\check{c}_d < x_d \le \check{c}_d, \\ \frac{5(x_d - \check{c}_d)}{\check{c}_d - 1} + 1, & \text{if } \check{c}_d < x_d \le \frac{1+4\check{c}_d}{5}, \\ \frac{x_d - 1}{1 - \check{c}_d} + \frac{4}{5}, & \text{if } \frac{1+4\check{c}_d}{5} < x_d \le 1, \end{cases}
\end{aligned}
$$

Global optimum: $f(\boldsymbol{x}^*) = -1$ for $\boldsymbol{x}^* = \check{\boldsymbol{c}}$.

Drop Wave $D = 2$, $\boldsymbol{x} \in [-5.12, 5.12]^D$

$$
f(x_1, x_2) = -\frac{1 + \cos\left(12 + \sqrt{x_1^2 + x_2^2}\right)}{\frac{1}{2}(x_1^2 + x_2^2) + 2}
$$

Global optimum: $f(\boldsymbol{x}^*) = -1$ for $\boldsymbol{x}^* = \boldsymbol{0}$.

Easom $D = 2$, $\boldsymbol{x} \in [-100, 100]^D$

$$
f(x_1, x_2) = -\cos(x_1)\cos(x_2)\exp(-(x_1 - \pi)^2 - (x_2 - \pi)^2)
$$

Global optimum: $f(\boldsymbol{x}^*) = -1$ for $\boldsymbol{x}^* = (\pi, \pi)$.

Penalty 1 $D = 30$, $\boldsymbol{x} \in [-50, 50]^D$

$$
\begin{aligned}
f(\boldsymbol{x}) &= \frac{\pi}{D}\Bigg[10\sin^2(\pi y_1) + (y_d - 1)^2 \\
&\qquad + \sum_{d=1}^{D-1}(y_d - 1)^2(1 + 10\sin^2(\pi y_{d+1}))\Bigg] \\
&\qquad + \sum_{d=1}^{D} u(x_d, 10, 100, 4), \\
\text{where } y_i &= 1 + \frac{1}{4}(x_i + 1), \text{and} \\
u(x_i, a, k, m) &= \begin{cases} k(x_i - a)^m, & \text{if } x_i > a, \\ 0, & \text{if } -a \le x_i \le a, \\ k(-x_i - a)^m, & \text{if } x_i < -a \end{cases}
\end{aligned}
$$

Global optimum: $f(\boldsymbol{x}^*) = 0$ for $x_i = -1$ for $i = 1, 2, \ldots, D$.

Griewank $D = 30$, $\boldsymbol{x} \in [-600, 600]^D$

$$f(\boldsymbol{x}) = \sum_{d=1}^{D} \frac{x_d^2}{4000} - \prod_{d=1}^{D} \cos\left(\frac{x_d}{\sqrt{d}}\right) + 1$$

Global optimum: $f(\boldsymbol{x}^*) = 0$ for $\boldsymbol{x}^* = \mathbf{0}$.

Goldstein–Price $D = 2$, $\boldsymbol{x} \in [-2, 2]^D$

$$\begin{aligned} f(x_1, x_2) = & \Big[1 + (x_1 + x_2 + 1)^2 (19 - 14x_1 + 3x_1^2 - 14x_2 + 6x_1x_2 + 3x_2^2)\Big] \\ & \cdot \Big[30 + (2x_1 - 3x_2)^2 (18 - 32x_1 + 12x_1^2 + 48x_2 - 36x_1x_2 + 27x_2^2)\Big] \end{aligned}$$

Global optimum: $f(\boldsymbol{x}^*) = 3$ for $\boldsymbol{x}^* = (0, -1)$.

Axis parallel hyper-ellipsoid $D = 100$, $\boldsymbol{x} \in [-5.12, 5.12]^D$

$$f(\boldsymbol{x}) = \sum_{d=1}^{D} (dx_d^2)$$

Global optimum: $f(\boldsymbol{x}^*) = 0$ for $\boldsymbol{x}^* = \mathbf{0}$.

Michalewicz $D = 10$, $\boldsymbol{x} \in [0, \pi]^D$, $m = 10$

$$f(\boldsymbol{x}) = -\sum_{d=1}^{D} \sin(x_d) \sin^{2m}\left(\frac{dx_d^2}{\pi}\right)$$

Global optimum: $f(\boldsymbol{x}^*) = -9.66015$ for $\boldsymbol{x}^* = \mathbf{0}$.

Non-continuous Rastrigin $D = 30$, $\boldsymbol{x} \in [-5.12, 5.12]^D$

$$\begin{aligned} f(\boldsymbol{x}) &= 10D + \sum_{d=1}^{D} \big(y_d^2 - 10\cos(2\pi y_d)\big), \\ \text{where } y_i &= \begin{cases} x_i, & \text{if } |x_i| < 0.5, \\ \frac{\text{round}(2x_i)}{2}, & \text{if } |x_i| \geq 0.5 \end{cases} \end{aligned}$$

Global optimum: $f(\boldsymbol{x}^*) = 0$ for $\boldsymbol{x}^* = \mathbf{0}$.

Parabola $D = 200$, $\boldsymbol{x} \in [-20, 20]^D$

$$f(\boldsymbol{x}) = \sum_{d=1}^{D} x_d^2$$

Global optimum: $f(\boldsymbol{x}^*) = 0$ for $\boldsymbol{x}^* = \mathbf{0}$.

Rastrigin $D = 30$, $\boldsymbol{x} \in [-5.12, 5.12]^D$

$$f(\boldsymbol{x}) \;=\; 10D + \sum_{d=1}^{D} \left(x_d^2 - 10\cos(2\pi x_d)\right)$$

Global optimum: $f(\boldsymbol{x}^*) = 0$ for $\boldsymbol{x}^* = \mathbf{0}$.

Rosenbrock $D = 30$, $\boldsymbol{x} \in [-10, 10]^D$

$$f(\boldsymbol{x}) \;=\; \sum_{d=1}^{D-1} \left(100(x_{d+1} - x_d^2)^2 + (x_d - 1)^2\right)$$

Global optimum: $f(\boldsymbol{x}^*) = 0$ for $x_i = 1$ for $i = 1, 2, \ldots, D$.

Schaffer's F6 $D = 2$, $\boldsymbol{x} \in [-100, 100]^D$

$$f(x_1, x_2) \;=\; 0.5 + \frac{\sin^2\left(\sqrt{x_1^2 + x_2^2}\right) - 0.5}{\left(1 + 0.001(x_1^2 + x_2^2)\right)^2}$$

Global optimum: $f(\boldsymbol{x}^*) = 0$ for $\boldsymbol{x}^* = \mathbf{0}$.

Schwefel 22 $D = 30$, $\boldsymbol{x} \in [-10, 10]^D$

$$f(\boldsymbol{x}) \;=\; \sum_{d=1}^{D} |x_d| + \prod_{d=1}^{D} |x_d|$$

Global optimum: $f(\boldsymbol{x}^*) = 0$ for $\boldsymbol{x}^* = \mathbf{0}$.

Shubert $D = 2$, $\boldsymbol{x} \in [-10, 10]^D$

$$f(x_1, x_2) \;=\; \left(\sum_{d=1}^{5} d\cos((d+1)x_1 + d)\right)\left(\sum_{d=1}^{5} d\cos((d+1)x_2 + d)\right)$$

Global optimum: $f(\boldsymbol{x}^*) = -186.7309$. There exist 18 global optima.

Sphere $D = 100$, $\boldsymbol{x} \in [-5.12, 5.12]^D$

$$f(\boldsymbol{x}) \;=\; \sum_{d=1}^{D} x_d^2$$

Global optimum: $f(\boldsymbol{x}^*) = 0$ for $\boldsymbol{x}^* = \mathbf{0}$.

Step $D = 30$, $\boldsymbol{x} \in [-100, 100]^D$

$$f(\boldsymbol{x}) \;=\; \sum_{d=1}^{D} (\lfloor x_d \rfloor + 0.5)^2$$

Global optimum: $f(\boldsymbol{x}^*) = 0$ for $x_i = 0.5$ for $i = 1, 2, \ldots, D$.

Tripod $D = 2$, $\boldsymbol{x} \in [-100, 100]^D$

$$\begin{aligned} f(x_1, x_2) &= y_2(1 + y_1) + |x_1 + 50y_2(1 - 2y_1)| \\ &\quad + |x_2 + 50(1 - 2y_2)|, \\ \text{where } y_i &= \begin{cases} 1, & \text{if } x_i \geq 0, \\ 0, & \text{if } x_i < 0 \end{cases} \end{aligned}$$

Global optimum: $f(\boldsymbol{x}^*) = 0$.

Trefethen 4 $D = 2$, $\boldsymbol{x} \in [-1, 1]^D$

$$\begin{aligned} f(x_1, x_2) &= 0.25x_1^2 + 0.25x_2^2 + e^{\sin(50x_1)} - \sin(10x_1 + 10x_2) \\ &\quad + \sin(60e^{x_2}) + \sin\left(70\sin(x_1)\right) + \sin\left(80\sin(x_2)\right) \end{aligned}$$

Global optimum: $f(\boldsymbol{x}^*) = -3.3068686474$
for $\boldsymbol{x} = (-0.02440307923, 0.2106124261)$.

Appendix B

Gold Particle Freezing

Reference data for temperature 650

[4.5893962, 5.9460132, 6.5608444, 7.1814044, 7.4699254,
7.974138, 8.3179732, 8.5897085, 8.815866, 8.9838444,
9.097024, 9.2101201, 9.3256198, 9.4424618, 9.5572252,
9.666489, 9.7483201, 9.7968224, 9.8326308, 9.8763797,
9.935423, 9.9969102, 10.05117, 10.01815, 9.9550623,
9.9571281, 9.9696313, 9.9638934, 9.9203968, 9.8755837,
9.8613757, 9.8639473, 9.8699385, 9.8659895, 9.8387404,
9.7882525, 9.7256209, 9.6540369, 9.5738475, 9.5063682,
9.4460584, 9.3905118, 9.3373218, 9.2840821, 9.2283862,
9.1678276, 9.1, 9.0263127, 8.9495042, 8.8691619,
8.7848731, 8.6962252, 8.6028054, 8.5042013, 8.4,
8.2916156, 8.1810115, 8.0685997, 7.954792, 7.84,
7.7246247, 7.6082502, 7.4900633, 7.369251, 7.245,
7.1076322, 6.9545862, 6.7957241, 6.640908, 6.5,
6.3714076, 6.2456229, 6.1206343, 5.9944305, 5.865,
5.7333029, 5.6013657, 5.4687771, 5.3351257, 5.2]

Reference data for temperature 660

[4.3996028, 6.0174587, 6.7295721, 7.2003355, 7.4824306,
7.9871213, 8.1203103, 8.4343445, 8.6344533, 8.8455355,
8.9173731, 9.1223343, 9.2068644, 9.3299543, 9.4419146,
9.5388557, 9.6357968, 9.7191349, 9.7942107, 9.8615448,
9.878231, 9.8949172, 9.9116034, 9.9587025, 10.005802,
10.052901, 10.1, 10.123, 10.145999, 10.168999,
10.155199, 10.141399, 10.1276, 10.1138, 10.1,
10.066, 10.032, 9.998, 9.964, 9.93,
9.888, 9.846, 9.804, 9.762, 9.72,
9.6629025, 9.605805, 9.5487076, 9.4916101, 9.4345126,
9.3576498, 9.2807869, 9.2039241, 9.1270612, 9.0501984,
8.9601587, 8.870119, 8.7800794, 8.6900397, 8.6,
8.482, 8.364, 8.246, 8.128, 8.01,
7.898, 7.786, 7.674, 7.562, 7.45,
7.32, 7.19, 7.06, 6.93, 6.8,
6.6859178, 6.5718355, 6.4577533, 6.343671, 6.2295888]

Reference data for temperature 670

[5.3157398, 6.2737999, 6.870868, 7.2955571, 7.7108124,
8.048609, 8.3263213, 8.5928178, 8.827315, 9.009029,
9.1535442, 9.2884749, 9.4110826, 9.5186286, 9.6083744,
9.6875519, 9.764476, 9.8380331, 9.904312, 9.970591,
10.033347, 10.096103, 10.155074, 10.214044, 10.256221,
10.298399, 10.315039, 10.331679, 10.340992, 10.350306,
10.357405, 10.364504, 10.371603, 10.378703, 10.385802,
10.392901, 10.4, 10.356, 10.312, 10.268,
10.224, 10.18, 10.13875, 10.0975, 10.05625,
10.015, 9.97375, 9.9325, 9.89125, 9.85,
9.788, 9.726, 9.664, 9.602, 9.54,
9.474, 9.408, 9.342, 9.276, 9.21,
9.126, 9.042, 8.958, 8.874, 8.79,
8.692, 8.594, 8.496, 8.398, 8.3,
8.2, 8.1, 8.0, 7.9, 7.8,
7.6830703, 7.5661406, 7.4492109, 7.3322812, 7.2153515]

Reference data for temperature 680

[5.0387548, 6.1393987, 6.8826814, 7.4081738, 7.762137,
8.0706371, 8.3897402, 8.6255124, 8.7695327, 8.979078,
9.1287614, 9.2631959, 9.3769948, 9.464771, 9.5384554,
9.6128907, 9.6870579, 9.7599381, 9.8305122, 9.8977612,
9.9606662, 10.018208, 10.069368, 10.113127, 10.151962,
10.188887, 10.223687, 10.256147, 10.286054, 10.313191,
10.338808, 10.364013, 10.388382, 10.41149, 10.432912,
10.452224, 10.469, 10.482818, 10.493251, 10.499876,
10.502267, 10.5, 10.466667, 10.433333, 10.4,
10.368, 10.336, 10.304, 10.272, 10.24,
10.192, 10.144, 10.096, 10.048, 10.0,
9.95, 9.9, 9.85, 9.8, 9.75,
9.69, 9.63, 9.57, 9.51, 9.45,
9.37, 9.29, 9.21, 9.13, 9.05,
8.948, 8.846, 8.744, 8.642, 8.54,
8.452, 8.364, 8.276, 8.188, 8.1]

Reference data for temperature 690

[4.3224852, 5.9052153, 6.6808315, 7.2, 7.5955837,
7.9534101, 8.25, 8.4986109, 8.7224137, 8.9128,
9.080354, 9.2356357, 9.37, 9.4899497, 9.6038421,
9.7058134, 9.79, 9.8621462, 9.932066, 10.0,
10.066782, 10.13, 10.186678, 10.237434, 10.283023,
10.324205, 10.361736, 10.396374, 10.428876, 10.46,
10.490091, 10.518913, 10.54635, 10.572288, 10.596612,
10.619207, 10.639958, 10.658751, 10.67547, 10.69,
10.701998, 10.711398, 10.718502, 10.723611, 10.72703,
10.729059, 10.73, 10.72, 10.71, 10.7,
10.674881, 10.649762, 10.624644, 10.599525, 10.574406,
10.541093, 10.50778, 10.474468, 10.441155, 10.407842,
10.363171, 10.318501, 10.27383, 10.22916, 10.184489,
10.118108, 10.051728, 9.9853473, 9.9189668, 9.8525862,
9.7862057, 9.7276796, 9.6691535, 9.6018876, 9.5288543,
9.4507056, 9.3680938, 9.2816709, 9.1920889, 9.0]

Reference data for temperature 710

[4.3224852, 5.9052153, 6.6, 7.15, 7.47,
7.79, 8.1, 8.3, 8.53, 8.76,
8.93, 9.0688215, 9.2054209, 9.34, 9.4732978,
9.6041714, 9.73, 9.846866, 9.9534632, 10.051088,
10.141037, 10.224606, 10.303092, 10.377791, 10.45,
10.520472, 10.588763, 10.654373, 10.716802, 10.775549,
10.830115, 10.88, 10.926613, 10.971608, 11.014849,
11.056201, 11.095527, 11.132693, 11.167562, 11.2,
11.230355, 11.259096, 11.286265, 11.311904, 11.336054,
11.358759, 11.38006, 11.4, 11.418786, 11.436402,
11.452556, 11.466953, 11.479299, 11.489302, 11.496667,
11.501101, 11.50231, 11.5, 11.493628, 11.483353,
11.469933, 11.454129, 11.436699, 11.418403, 11.4,
11.381138, 11.36085, 11.338972, 11.31534, 11.28979,
11.262159, 11.232284, 11.2, 11.160123, 11.108601,
11.046967, 10.976756, 10.899502, 10.816738, 10.73]

Reference data for temperature 730

[4.6003716, 5.6372814, 6.2320109, 6.6848021, 7.0688849,
7.457493, 7.783821, 8.022273, 8.215727, 8.554573,
8.733774, 8.961843, 9.146133, 9.29699, 9.58037,
9.719982, 9.752077, 9.959827, 10.149854, 10.183937,
10.371969, 10.56, 10.689558, 10.884046, 10.917312,
11.027185, 11.007682, 11.204644, 11.335406, 11.410607,
11.518301, 11.586643, 11.659563, 11.732482, 11.805402,
11.878321, 11.951241, 12.024161, 12.09708, 12.17,
12.224, 12.278, 12.332, 12.386, 12.44,
12.496083, 12.552167, 12.60825, 12.664334, 12.720417,
12.753636, 12.786855, 12.820075, 12.853294, 12.876279,
12.905828, 12.935376, 12.959389, 12.981036, 13.002394,
13.020549, 13.038704, 13.052415, 13.066515, 13.080614,
13.091237, 13.100186, 13.098544, 13.096902, 13.088878,
13.080853, 13.072829, 13.064805, 13.05678, 13.048756,
13.040731, 13.032707, 13.02365, 13.014594, 13.005537]

Reference data for temperature 750

[5.1, 6.030066, 6.6560599, 6.9769892, 7.3871978,
7.7377843, 8.1772508, 8.4329, 8.6241575, 8.9891032,
9.0793028, 9.3886886, 9.543061, 9.71716, 9.93,
10.118153, 10.321051, 10.426488, 10.544195, 10.696143,
10.81569, 11.0, 11.118768, 11.233269, 11.376162,
11.44766, 11.396192, 11.617703, 11.729285, 11.70494,
11.897901, 12.024276, 12.054163, 12.186007, 12.223845,
12.405973, 12.438927, 12.650781, 12.736549, 12.78,
12.758496, 13.001, 13.086668, 13.123, 13.233447,
13.294811, 13.395103, 13.410942, 13.469362, 13.567139,
13.694237, 13.70461, 13.819995, 13.867694, 13.894542,
13.904653, 13.99396, 13.946765, 14.079558, 14.017336,
14.054136, 14.191757, 14.117265, 14.222733, 14.249632,
14.284093, 14.335174, 14.381866, 14.399504, 14.436916,
14.49668, 14.529593, 14.548851, 14.625626, 14.603901,
14.625627, 14.682413, 14.697029, 14.721349, 14.806606]

Appendix C

Rational Material Design Files

INCAR

```
SYSTEM = local optimisation
PREC = Normal
ENCUT = 520.0
EDIFF = 1e-6
IBRION = 2
ISIF = 3
NSW = 100
ISMEAR = 1 ; SIGMA = 0.2
POTIM = 0.10
#Wavefunction and charge
LWAVE = FALSE
LCHARG = FALSE
#Target Pressure
PSTRESS = 0.01
#Finer optimization
EDIFFG =-0.5e-5
SYMPREC = 1e-4
LPLANE = .TRUE.
NCORE   = 1
LSCALU = .FALSE.
NSIM   = 4
NPAR   = 12
KSPACING = 0.2
KGAMMA = TRUE
```

POSCAR

```
CIF file
   1.0
        3.6993421813         0.0000000000         0.0000000000
        -0.7898257294        3.2660363033         0.0000000000
        -1.1533711099        -1.7360421208        3.0039081302
   Si
    2
Direct
     0.655783409         0.477032320         0.905704760
     0.419699845         0.571584069         0.842177991
```

KPOINTS

```
Automatic mesh
 0
Monkhorst-Pack
        4.000000000          4.000000000          4.000000000

  0.00  0.00 0.00
```