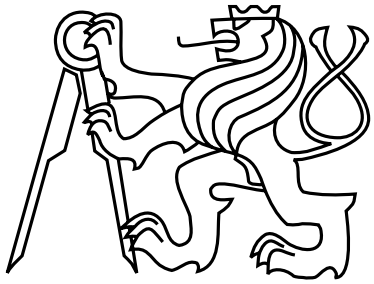
Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science and Engineering



Master's Thesis

Continuous optimization algorithms

Bc. Vladimír Bičík

Supervisor: Ing. Pavel Kordík, Ph.D.

Study Programme: Electrical Engineering and Information Technology

Field of Study: Computer Science and Engineering

May 2010

Acknowledgements

First and foremost I would like to thank my thesis supervisor Ing. Pavel Kordík, Ph.D, for leading me in a friendly way and creating motivating working environment.

I would also like to thank to Bc. Ondřej Skalička who supplied computational resources for testing and shared his comments and to Bc. Jakub Řezníček for helping me with formal aspects of the thesis and his helpful ideas.

Finally my greatest thanks belong to my whole family and Anastasia especially who supported me all the way.

Declaration

I hereby declare that I have completed this thesis independently and that I have listed all the literature and publications used.

I have no objection to usage of this work in compliance with the act § 60 under Act No. 121/2000, Coll. (copyright law), and with the rights connected with the copyright act including the changes in the act.

In Prague on May 14th, 2010 .....

Abstract

This thesis describes methods suitable for unconstrained optimization of real-valued functions. Listed are both numerical methods and methods inspired by nature. We explain ideas behind these methods as well as their intended application. Implementation of these methods in the JCool environment, which is briefly introduced, is explained and an evaluation of their convergence for different optimization tasks is presented. Tasks used for convergence evaluation were selected in consideration of objective comparison of methods and thus we introduced some widely used test functions to the JCool environment. We observed and discussed parameter dependend behaviour of each method for these tasks and as a result a set of recommended parameter values is proposed together with detailed comparison of benchmarked methods.

Abstrakt

Tato práce popisuje metody použitelné pro obecnou optimalizaci v prostoru reálných parametrů. Uvedeny jsou jak numerické metody, tak i metody inspirované přírodou. U všech metod je vysvětlen základní princip fungování a předpokládané použití. V práci je vysvětlena jejich implementace v prostředí JCool, které je také krátce představeno, a dále je hodnocena jejich konvergence nad různými problémy. Problémy byly vybírány s ohledem na objektivní porovnání metod a proto byly do prostředí JCool převedeny standardně používané testovací funkce. Pozornost je věnována zvláště nastavení parametrů metod pro daný problém, doporučení jejich optimálních hodnot a nakonec i porovnání metod mezi sebou.

Contents

List of Figures **xv**

List of Tables **xix**

List of Algorithms **xxi**

1 Introduction **1**
1.1 Environment . . . 1
1.2 Goals of the thesis . . . 1
1.3 JCool project . . . 2
1.3.1 Purpose . . . 2
1.3.2 Implementation of optimization methods . . . 2
1.3.3 Implementation of benchmark functions . . . 2
1.3.4 Changes to the JCool source code . . . 3
1.4 Organization of the thesis . . . 3

2 Continuos optimization algorithms **5**
2.1 Common background theory . . . 5
2.2 Numerical methods . . . 7
2.2.1 Gradient methods . . . 7
2.2.1.1 Steepest Descent method (SD) . . . 7
2.2.1.2 Conjugate Gradient method (CG) . . . 8
2.2.1.3 Levenberg–Marquardt method (LM) . . . 9
2.2.1.4 quasi-Newton method (QN) . . . 10
2.2.2 Orthogonal search (OS) . . . 13
2.2.3 Powell's method (PM) . . . 13
2.2.4 Covariance Matrix Adaptation Evolution Strategy (CMA-ES) . . . 15
2.3 Nature inspired methods . . . 16
2.3.1 Ant algorithms . . . 17
2.3.1.1 Continuous Ant Colony Optimization (CACO) . . . 18
2.3.1.2 *Pachycondyla apicalis* method (API) . . . 18
2.3.1.3 Adaptive Ant Colony Algorithm (AACA) . . . 20
2.3.1.4 Extended Ant Colony Optimization (ACO*) . . . 20
2.3.1.5 Direct Ant Colony Optimization (DACO) . . . 21
2.3.2 Particle Swarm Optimization (PSO) . . . 22
2.3.3 Genetic algorithms . . . 23
2.3.3.1 Differential Evolution (DE) . . . 24
2.3.3.2 Simplified Atavistic Differential Evolution (SADE) . . . 24
2.3.3.3 Population-Based Incremental Learning (PBIL) . . . 25
2.3.4 Hybrid of the GA and the PSO (HGAPSO) . . . 25

3 Benchmarking of optimization methods **27**
3.1 Goals of benchmarking . . . 27
3.2 Selected test functions . . . 27
3.2.1 Unimodal functions . . . 28
3.2.1.1 Beale's function (BE) . . . 28
3.2.1.2 Booth's function (BO) . . . 28
3.2.1.3 De Jong's (sphere) function (DJ) . . . 28
3.2.1.4 Easom's function (EA) . . . 28
3.2.1.5 Matyas' function (MA) . . . 28
3.2.1.6 Rosenbrock's valley (RO) . . . 28
3.2.1.7 Trid function (TR) . . . 29
3.2.1.8 Zakharov's function (ZA) . . . 29
3.2.2 Multimodal functions . . . 29
3.2.2.1 Ackley's path (AC) . . . 29
3.2.2.2 Branin's function (BR) . . . 29
3.2.2.3 Goldstein–Price function (GP) . . . 30
3.2.2.4 Griewangk's function (GR) . . . 30
3.2.2.5 Himmelblau function (HI) . . . 30
3.2.2.6 Langerman's function (LA) . . . 30
3.2.2.7 Levy function no. 3 (L3) . . . 30
3.2.2.8 Levy function no. 5 (L5) . . . 30
3.2.2.9 Michalewicz's function (MI) . . . 31
3.2.2.10 Rana's function (RN) . . . 31
3.2.2.11 Rastrigin's function (RA) . . . 31
3.2.2.12 Shekel's foxholes (SH) . . . 31
3.2.2.13 Shubert's function (SB) . . . 31
3.2.2.14 Schwefel's function (SW) . . . 32
3.2.2.15 Whitley's function (WH) . . . 32
3.3 Benchmark configuration . . . 32
3.3.1 Parameter range and step . . . 32
3.3.2 Convergence evaluation . . . 33

4 Results **35**
4.1 Observed convergence . . . 35

5 Recommended parameter values **65**
5.11 Comparison of the default and the recommended setting . . . 70

6 Comparison of methods **73**
6.1 Numerical methods . . . 73
6.2 Nature inspired methods . . . 75
6.3 Overall comparison . . . 76

7 Summary and conclusions **81**
7.1 Future work . . . 81

References **83**

A Changes to the JCool source code **87**

B Default method parameters **91**

C Test functions detailed **95**

D Additional results of experiments **113**

E Contents of the attached CD **121**

List of Abbreviations **123**

Index **127**

List of Figures

2.1 Typical algorithm run heading to Stiefel's cage. 8
2.2 Visualisation of pheromone levels assigned to search vectors in CACO. 19
2.3 API algorithm. 19
2.4 Visualisation of three different pheromone levels. 20
2.5 Visualisation of pheromone level. 21

4.1 Steepest descent applied to Booth function. 36
4.2 Average number of iterations taken by CG. 38
4.3 Convergence of the LM algorithm on the Rosenbrock's valley. 39
4.4 Average number of iterations taken by QN. 40
4.5 CMA-ES results for $\sigma = 0.1, ..., 10.0$, step = 0.1. 42
4.6 API results for moving the nest to the best point every 5, ..., 100 iterations, step = 5. 44
4.7 API results for hunting sites count = 2, ..., 20, step = 1. 44
4.8 API results for starvation = 1, ..., 20 iterations, step = 1. 45
4.9 AACA: occurence of bit swap. 46
4.10 AACA results for encoding length $= 8, ..., 128$, step = 8. 47
4.11 AACA results for evaporation factor $= 0.0, ..., 1.0$, step = 0.05. 47
4.12 ACO* results for population size $= 5, ..., 100$, step = 5. 49
4.13 ACO* results for number of replaced ants $= 0, ..., 60$, step = 5. 49
4.14 ACO* results for deviation parameter $\omega = 0.05, ..., 1.00$, step = 0.05. 49
4.15 ACO* results for convergence parameter $\sigma = 0.00, ..., 1.00$, step = 0.05. 50
4.16 DACO results for population size = 5, ..., 100, step = 5. 51
4.17 DACO results for evaporation factor $\lambda = 0.00$, ..., 1.00, step = 0.05. 51
4.18 PSO results for population size = 5, ..., 100, step = 5. Original update formula used. 53
4.19 PSO results for $\phi_1 = 0.00$, ..., 1.00, step = 0.05. Original update formula used. 53
4.20 PSO results for $\phi_2 = 0.00$, ..., 1.00, step = 0.05. Original update formula used. 53
4.21 PSO: particles inside a valley. 54
4.22 PSO results for $\phi_1 = 0.05$, ..., 1.00, step = 0.05. FI update formula used. . . . 54
4.23 PSO results for neighbourhood distance $m = 1$, ..., 20, step = 1. FI update formula used. 55
4.24 PSO results for population size = 5, ..., 100, step = 5. C update formula used. 55
4.25 PSO results for $\kappa = 0.00$, ..., 1.00, step = 0.05. C update formula used. 56
4.26 DE results for population size = 5, ..., 100, step = 5. 57
4.27 DE results for crossover rating = 0.00, ..., 1.00, step = 0.05. 57
4.28 DE results for mutation constant = 0.00, ..., 1.00, step = 0.05. 57
4.29 SADE results for radiation = 0.00, ..., 1.00, step = 0.05. 59
4.30 SADE results for local radiation = 0.00, ..., 1.00, step = 0.05. 59
4.31 SADE results for mutation rate = 0.00, ..., 1.00, step = 0.05. 60
4.32 SADE results for mutagen rate = 50, ..., 1000, step = 50. 60

4.33 SADE results for crossover rate = 0.00, ..., 1.00, step = 0.05. 60
4.34 PBIL results for learning rate = 0.05, ..., 1.00, step = 0.05. 61
4.35 PBIL results for negative learning rate = 0.00, ..., 1.00, step = 0.05. 62
4.36 PBIL results for mutation probability = 0.00, ..., 1.00, step = 0.05. 62
4.37 PBIL results for mutation shift = 0.00, ..., 1.00, step = 0.05. 62
4.38 HGAPSO results for population size = 5, ..., 100, step = 5. 63
4.39 HGAPSO results for elite ratio = 0.05, ..., 1.00, step = 0.05. 64
4.40 HGAPSO results for mutation probability = 0.00, ..., 1.00, step = 0.05. 64
4.41 HGAPSO results for $\chi = 0.05$, ..., 1.00, step = 0.05. 64

5.1 CG with different update formulas. 65
5.2 Comparison of the average success rate for default parameter setting and for recommended setting. 70

6.1 Results of numerical optimization. 73
6.2 Results of optimization by nature inspired methods. 75
6.3 Results of optimization by implemented methods. 77
6.4 Results of optimizing unimodal functions. 77
6.5 Results of optimizing multimodal functions. 78
6.6 Solution for Booth's function found by DACO. 78
6.7 Solution for Booth's function found by SD. 79
6.8 Detail of the solution found by DACO. 79

A.1 Final code of `CentralDifferenceHessian.hessianAt` routine. 88
A.2 Corrected `ValuePoint.compareTo`. 89
A.3 Extended class `Point`. 89
A.4 Added code of Gauss–Jordan elimination for square matrices. 90

C.1 Ackley's path for $d = 2$. 95
C.2 Beale function. 96
C.3 Bohachevsky's function. 96
C.4 Booth's function. 97
C.5 Branin's function. 97
C.6 Dixon–Price function. 98
C.7 Easom's function. 99
C.8 Goldstein–Price function. 99
C.9 Griewangk's function for $d = 2$. 100
C.10 Himmelblau function. 101
C.11 Hump's function. 102
C.12 Langerman's function. 102
C.13 Levy's function. 103
C.14 Levy function no. 3. 103
C.15 Levy function no. 5. 104
C.16 Matyas function. 104
C.17 Michalewitcz's function for $d = 2$. 105
C.18 Perm's functions for $d = 2$, $\beta = 0.5$. 106
C.19 Rana's function for $d = 2$. 107
C.20 Rastrigin's function for $d = 2$. 107
C.21 Rosenbrock's valley for $d = 2$. 108
C.22 Shekel's foxholes for $d = 2$, $m = 30$. 109
C.23 Shubert's function. 110

C.24 Schwefel's function. 110
C.25 Trid function. 111
C.26 Whitley's function for $d = 2$. 112
C.27 Zakharov's function for $d = 2$. 112

D.1 CACO results for population size = 5, ..., 100, step = 5. 113
D.2 CACO results for search radius = 5, ..., 100, step = 5. 113
D.3 CACO results for radius multiplier = 0.0, ..., 1.0, step = 0.05. 114
D.4 CACO results for generations before radius decrease = 5, ..., 100, step = 5. . . 114
D.5 CACO results for evaporation factor = 0.0, ..., 1.0, step = 0.05. 114
D.6 CACO results for initial pheromone level = 0.0, ..., 1.0, step = 0.05. 114
D.7 CACO results for pheromone increase = 0.0, ..., 1.0, step = 0.05. 115
D.8 CACO results for minimal pheromone level = 0.0, ..., 1.0, step = 0.05. 115
D.9 API results for population size = 5, ..., 100, step = 5. 115
D.10 AACA results for population size = $5, ..., 100$, step = 5. 116
D.11 AACA results for pheromone index = $0.0, ..., 1.0$, step = 0.05. 116
D.12 AACA results for cost index = $0.0, ..., 1.0$, step = 0.05. 116
D.13 ACO* results for neighborhood size for force diversity = $0.0, ..., 1.0$, step = 0.05, force diversity = *true* 117
D.14 PSO results for population size = 5, ..., 100, step = 5. FI update formula used. 117
D.15 PSO results for $\phi_1 = 0.05$, ..., 1.00, step = 0.05. C update formula used. 118
D.16 PSO results for $\phi_2 = 0.05$, ..., 1.00, step = 0.05. C update formula used. 118
D.17 SADE results for population size = 5, ..., 100, step = 5. 118
D.18 PBIL results for population size = 5, ..., 100, step = 5. 119
D.19 PBIL results for encoding length = 8, ..., 128, step = 8. 119
D.20 HGAPSO results for $\phi_1 = 0.05$, ..., 1.00, step = 0.05. 119
D.21 HGAPSO results for $\phi_2 = 0.05$, ..., 1.00, step = 0.05. 120

List of Tables

4.1 Average success rate for SD. . . . 36
4.2 Average number of iterations taken by SD. . . . 37
4.3 Average success rate for CG. . . . 37
4.4 Average success rate for LM. . . . 38
4.5 Average number of iterations taken by LM. . . . 38
4.6 Average success rate for QN. . . . 40
4.7 Average success rate for OS. . . . 40
4.8 Average number of iterations taken by OS. . . . 41
4.9 Average success rate for PM. . . . 41
4.10 Average number of iterations taken by PM. . . . 41
4.11 Comparison of average success rate of CACO method and Random search. . . . 43
4.12 Average success rate for ACO* when using standard deviation or average. . . . 50
4.13 Average number of iterations taken by ACO* when using standard deviation or average. . . . 50

5.1 Recommended parameter values for API. . . . 66
5.2 Recommended parameter values for AACA. . . . 66
5.3 Recommended parameter values for ACO*. . . . 67
5.4 Recommended parameter values for DACO. . . . 67
5.5 Recommended parameter values for original PSO. . . . 67
5.6 Recommended parameter values for PSO–FI. . . . 68
5.7 Recommended parameter values for PSO–C. . . . 68
5.8 Recommended parameter values for DE. . . . 68
5.9 Recommended parameter values for SADE. . . . 69
5.10 Recommended parameter values for PBIL. . . . 69
5.11 Recommended parameter values for HGAPSO. . . . 70

6.1 Summary of function, gradient and hessian evaluations taken by numerical methods. . . . 75
6.2 Summary of function evaluations taken by methods inspired by nature. . . . 76

E.1 Contents of the attached CD. . . . 121

List of Algorithms

2.1 Conjugate Gradient method. 9
2.2 Newton's method. 10
2.3 Levenberg–Margardt method. 11
2.4 quasi-Newton method. 13
2.5 (Stochastic) Orthogonal search. 14
2.6 Powell's method. 15
2.7 Covariance Matrix Adaptation Evolution Strategy. 17
2.8 Particle Swarm Optimization algorithm. 23
2.9 Genetic algorithm. 24
2.10 Population-Based Incremental Learning algorithm. 26

Chapter 1

Introduction

Optimization problems are very common and it has been so since long ago. Wide variety of technical fields offers new challenges every day. Researchers have proposed many different methods for finding a solution to such a task. Among these we can find well known nature inspired algorithms as well as strictly numerical ones or a combination of both. Some are easy to understand, others are based on tough mathematical theories and principles. This thesis aims to present consistent description of some of the most used methods to the present day, discussing their rate of convergence on different typical tasks and finally concluding with their comparison and recommendation for such tasks. We hope that this work will help future researchers to get fast and solid insight into the world of function optimization.

1.1 Environment

This thesis is reassuming the JCool project [1]. As such we have chosen to continue with this framework and to use it to implement optimization algorithms in question. This tool is intended exactly for such use and since it has been created at our faculty we decided to include implemented methods and test tasks into the future distributions of the framework.

We also make use of another faculty project FAKE GAME [2]. Specifically we used code of optimization methods implemented in the FAKE GAME platform as a guideline to implementing these methods in the JCool project. Some methods were rewritten from scratch since the code in FAKE GAME was only adopted from public implementations and adapted to the FAKE GAME needs, thus although working, it was not providing easy survey of the code; others were refactored and improved if needed. Each method implementation is well documented and states the amount of the new code produced to make sure future users will be able to refer to appropriate sources.

1.2 Goals of the thesis

The goals of this thesis are these:

- Offer unified description of methods used for continuous optimization.
- Present implementation of these methods in JCool benchmarking environment.
- Describe standard test functions for method benchmarking.

- Evaluate these methods on different tasks with respect to the rate of convergence.
- Propose recommended parameter values for these methods based on completed tests.
- Compare these methods using widely accepted metrics.

1.3 JCool project

JCool stands for *Java Continuous Optimization Library*. It was proposed as an open-source Java library for continuous optimization by Martin Hvizdoš as his Master's Thesis at CTU FEE in 2009 [1].

It is now evolving project at the Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague. It is basically an easy to understand framework for implementing own methods for continuous optimization, specifying test functions and statistical benchmarking. Very useful is simple and yet effective visualisation of the algorithm state.

JCool in this thesis was used as a building site for various methods either converted from FAKE GAME software [2] or implemented from scratch according to the original authors.

1.3.1 Purpose

JCool was designed to split implementation of the algorithms and of the test functions. Its architecture incorporates very elegant ways to allow the use of every implemented algorithm to solve every implemented test function within the same environment and conditions. This is fundamental requirement for algorithm benchmarking since we need to compare algorithms independently of the test environment.

JCool also offers statistical data collecting wich helps to inspect the algorithm behaviour and allows basic comparison.

1.3.2 Implementation of optimization methods

In the JCool framework, optimization methods are defined by interface `OptimizationMethod`. This interface specifies three main methods of optimization process: initialization, which is called before the optimization begins and forwards information on the optimized function in the form of reference to implementation of `ObjectiveFunction` interface; stop conditions, which inform the caller whether the optimization process is to be terminated or should continue with next iteration; optimization step, which implements one iteration of the optimization process and is called repeatedly.

This interface is parametrized to select between optimizing one point at the time (usually the numerical methods) and optimization of a whole set of points (e.g. population used in the nature inspired algorithms). Each implementation of this interface is also a producer of telemetry, which consists basically of the actual state and results of the process. These are collected and used for statistical evaluation and for visualization of the algorithm run.

1.3.3 Implementation of benchmark functions

There are four interfaces for defining testing functions in JCool. These are `Function`, `FunctionBounds`, `FunctionGradient` and `FunctionHessian`. The only one mandatory is the in-

terface `Function`, which provides basic information about the implemented function: function value at a specified point and function dimension (i.e. number of variables to optimize).

Rest of the interfaces are implemented only if the function is bounded (`FunctionBounds`), has specified an analytic gradient (`FunctionGradient`) respectively analytic Hessian matrix (`FunctionHessian`).

Moreover the JCool framework supplies routines for computing numerical gradients and hessians by central difference method. These are used whenever a function is asked for gradient or hessian and does not provide an analytic formula for their evaluation. Similar approach is used when a function is asked for minimal or maximal bound of its variables although it is unbounded. JCool framework returns the smallest possible, resp. the largest possible number instead. This set of interfaces ensures that any arbitrary mathematical function can be implemented in the JCool environment.

All this is achieved through interface `ObjectiveFunction` which is an extension of the `Function` interface and incorporates design patterns *Decorator*, *Strategy* and *NullObject* [1]. Optimization method then uses this interface to get information about the optimized function, which ensures the possibility to call any arbitrary method on any arbitrary function.

1.3.4 Changes to the JCool source code

Although this thesis' goal was not to change or upgrade source code of the framework, some changes were inevitable. Some minor bugs were found that affected functionality of the framework as the whole and thus needed to be corrected. Commented summary of these changes can be found in Appendix A.

1.4 Organization of the thesis

Next chapter will introduce the algorithms used in this thesis, their implementation in JCool and provide the background needed to enter the world of unconstrained optimization. Third chapter presents the benchmark set of testing functions used to evaluate the performance of individual algorithms and describes characteristics of the selected tasks. Next, fourth chapter gives a summary of experimental results with commentary on the observed convergence. Fifth chapter directly continues with recommendation of parameter settings for individual algorithms with regard to the optimized function. Sixth chapter follows with a comparison of the optimization methods based on defined indicators a concludes with recommendation of the most able algorithm when facing a task of specific characteristics. Last chapter concludes the thesis and summarizes obtained results. An encouragement for future researchers in the field of optimization methods is also presented.

Chapter 2

Continuos optimization algorithms

This chapter describes algorithms implemented into the JCool framework. These are both numerical and nature inspired and all are referenced to the original authors. Documentation is accessible both in JavaDoc format and online in the project Wiki at `http://fakegame.sourceforge.net/doku.php?id=developer_documentation`.

First we present a simple division of the optimization algorithms with description of both different approaches and then we continue with detailed documentation of implemented algorithms.

2.1 Common background theory

All of the following optimization algorithms have common goal, to find a minimum or a maximum of a real–valued multidimensional function. The JCool framework works exclusively with algorithms searching for minimum and thus we will describe optimization algorithms as algorithms for minimization of an objective function. This doesn't affect the generality since minimization and maximization tasks are convertible.

Each presented algorithm seeks the function minimum either by optimizing one vector of function variables values (we will reference to this vector as a *point*) or by optimizing some set of these vectors and selecting the best at the end. There is a limited amount of information available to these algorithms: thanks to the numerical gradient and hessian we can work with function value, function gradient and function hessian (i.e. with first and second derivatives).

Optimization of such a function can be expressed as

$$\min f(\vec{x}), f : \mathbb{R}^n \mapsto \mathbb{R} \tag{2.1}$$

where $\vec{x}$ is a vector defined in $\mathbb{R}^n$. Methods optimizing such a function can be divided into two basic classes: numerical methods and nature inspired methods. This thesis focuses mainly on the numerical methods but we present many of the nature inspired methods as well.

Before describing both similarities and differences of both classes, frequently used terms should be made clear:

- **direction** is a vector specifying the direction that should be examined in the optimization process. In the terms of implementation it is essentially the same as a point except for without the starting point the direction itself has no meaning. Vector acting as a direction can be scaled any way and still would be considered the same, implying that the absolute size is of no exact meaning.

- **line search** is a process of looking for an optimal function value, in our case the minimum, from a given starting point along a given direction. In this way the optimized multivariate function is optimized as a univariate one. JCool implements two basic line search methods: Brent's line search method with and without the use of derivates (marked as B and BWD respectively). This takes the burden of implementing these routines from the researchers as well as it enables objective comparison of the algorithms by using the same routine. Details of line search methods are out of the scope of this thesis and readers are to see [3] for more information. For implementation details please refer to [4].

- **population** is generally a set of points over which the optimization algorithm runs. Member of a population can store more information than just actual point, e.g. the best value so far (together with the point of this value) or an actual search direction for this point, which is used in the next optimization step. This approach is usual for the nature inspired algorithms.

- **local best** point is a point for which the objective function yielded best value so far for a member of a population.

- **global best** point is a point for which the objective function yielded the best value so far for all members of a population. Naturally this global best is a local best for one or more members of the population.

- **global minimum** is a point for which the function yields the minimal function value. This is the goal we seek through function minimization. This point does not need to be unique, i.e. the objective function might have several (or infinitely many) global minima and in this case it is usually enough to find one such a minimum.

- **local minimum** is a point which acts as a global minimum for certain bounded area of function values. Objective functions can also have many of these local minima and methods for finding global minima of such functions have to be very sophisticated. Based on the ability of finding global minimum in these cases we recognize methods for global optimization and methods for local optimization.

  $\vec{x}$ is a local minimum of an objective function f when

  $$f(\vec{x}) \leqq f(\vec{x}'), \parallel \vec{x} - \vec{x}' \parallel \leqq d \tag{2.2}$$

  and

  $$f'(\vec{x}) = \mathbf{0} \tag{2.3}$$

  where $d > 0$ defines the explored area of the function in question. Points with $f'(\vec{x}) = \mathbf{0}$ are called stationary points and the local minima are among these. Among these are also local maxima and so called *saddle points*. To distinguish between these and the local minima we make use of the final definition: assuming $\vec{x}$ is a stationary point and $f''(\vec{x})$ is positive definite, then $\vec{x}$ is a local minimizer (and vice versa). This definition (along with the previous one) is derived from the Taylor expansion of the objective function in point neighbouring $\vec{x}$.

- **Hessian** matrix is a square matrix of second-order partial derivatives of a function in a given point. In the following text we will use this term to mark $f''(\vec{x})$ and we will denote it as $\mathbf{H}$.

A little bit more should be said about the difference between the local and the global minimum. When method finds a local minimum we usually do not know whether it is one of the local ones or a global one.

2.2 Numerical methods

Numerical methods are based on mathematical principles which they exploit in order to iterate through the state space of the objective function towards the local minimum. To understand these methods one must understand the mathematical theory used. All following numerical methods except for the *Covariance Matrix Adaptation Evolution Strategy (CMA-ES)* method work with just one point and try to move it closer to the neighbourhood of the minimum every step they take. Some of these make use of gradient and / or of hessian to get enough information for determining where the next step should be taken, i.e. they are computing the direction of the step and its length. The length is usually determined by the line search while the direction depends on the internal processes and its computation forms the core of the method.

These methods share the idea of the optimization based on iteration from the starting point $\vec{x}_0$ in the form

$$\vec{x}_{k+1} = \vec{x}_k + \alpha_k \vec{d}_k, k = 1, 2, ... \tag{2.4}$$

where α is the step length and $\vec{d}$ the step direction. $\vec{x}_k, k \to \infty$ should converge to some local minimizer of the objective function. We also want to ensure the descending property of such an iteration:

$$f(\vec{x}_{k+1}) < f(\vec{x}_k), \tag{2.5}$$

that our steps do not converge to some stationary point which is not the local minimum.

We need to find search direction $\vec{d}$ such that it is a descent direction. Only in this manner we can reach to a point with lower function value if we select appropriate step length. From the Taylor expansion of the function value in an neighbourhood of $\vec{x}$ in the direction $\vec{d}$

$$f(\vec{x} + \alpha\vec{d}) \approx f(\vec{x}) + \alpha\vec{d}^T f'(\vec{x}) \tag{2.6}$$

we see that to satisfy Equation 2.5 we have to choose $\vec{d}$ such that

$$\vec{d}^T f'(\vec{x}) < \mathbf{0}. \tag{2.7}$$

The direction $\vec{d}$ is then called a **descent direction**.

2.2.1 Gradient methods

As the name supposes, gradient methods use information obtained by computing function gradient at some arbitrary point, usually in the fashion of Equation 2.7. Gradient is a vector which points in the direction of the greatest rate of increase of the scalar field and whose magnitude is of the greatest rate of change. Thus taking the negative gradient yields the direction along which the objective function decreases the most.

2.2.1.1 Steepest Descent method (SD)

SD is the most straightforward of the gradient methods. Each iteration it simply chooses the optimization step to be taken in the direction of negative gradient direction:

$$\vec{d}_k = -\nabla f(\vec{x}_k). \tag{2.8}$$

This method is bound to get stuck in a local minimum if it does not start near the global one close enough, since the direction is always downhill. Furthermore there is a major drawback:

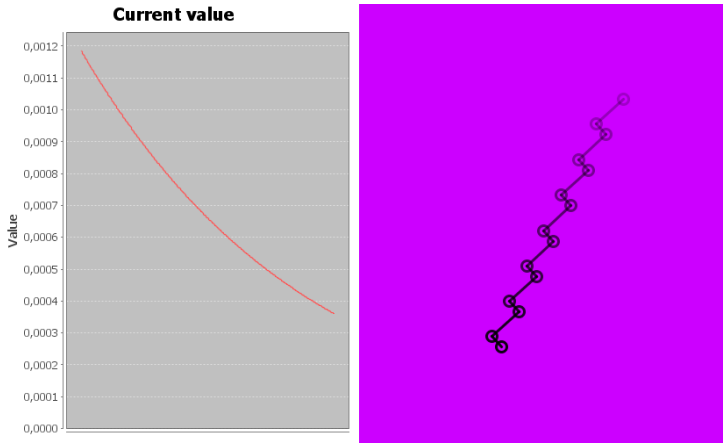


Figure 2.1: Typical algorithm run heading to Stiefel's cage.

even if it starts near the minimum, it tends to be slow as it gets closer because of the *zig–zag* style movement through the state space. The zig–zagging is also very common for objective functions with narrow valleys (e.g. Rosenbrock function, see Section 3.2.1.6). It is caused by taking successive steps in directions that are by definition orthogonal [5]. The α found using line search is such that it minimizes f along the search direction: $\frac{d}{d\alpha} f(\vec{x}_{k+1}) = \nabla f(\vec{x}_k)^T \vec{d}_k$. Setting this to zero means α is chosen so that $\nabla f(\vec{x}_k)$ and $\vec{d}_k$ are orthogonal.

In other words the zig–zaging causes that final convergence will be linear and usually very slow or even worse, that the iteration will get caught in so called *Stiefel's cage*, meaning the iteration might stop even when still far away from the minimum since the steps taken will be too small to affect the function value when computed with a limited number of digits as illustrates Figure 2.1. On the left are recorded successive function values, on the right we see visualisation of successive point coordinates. The steps taken and function value decrease will be smaller and smaller until it will decrease under the machine accuracy and the algorithm will stop.

2.2.1.2 Conjugate Gradient method (CG)

The zig–zag movement – taking steps in the same direction as earlier ones – is quite unnecessary. One would say that we should take exactly one step in each direction with correct length towards the minimum and thus find the minimum after n steps. This leads to idea of taking new steps in the direction that is a linear combination of the previous search direction and the current steepest descent direction:

$$\vec{d}_{k+1} = -\nabla f(\vec{x}_k) + \beta \vec{d}_k, \tag{2.9}$$

where β is computed using several formulas described later. Equation 2.9 does not guarantee that the new search direction will lead downhill. For these cases CG switches to the formula of SD (Equation 2.8), which is also used for the first step. Three of the most known formulas for computing β follow:

- **Fletcher–Reeves** [6]:

$$\beta = \frac{\nabla f(\vec{x}_k)^T \nabla f(\vec{x}_k)}{\nabla f(\vec{x}_{k-1})^T \nabla f(\vec{x}_{k-1})} \tag{2.10}$$

- **Polak—Ribière** [7]:

$$\beta = \frac{\nabla f(\vec{x}_k)^T (\nabla f(\vec{x}_k) - \nabla f(\vec{x}_{k-1}))}{\nabla f(\vec{x}_{k-1})^T \nabla f(\vec{x}_{k-1})} \tag{2.11}$$

- **Beale–Sorenson–Hestenes–Stiefel** [8], [9], [10]:

$$\beta = \frac{\nabla f(\vec{x}_k)^T(\nabla f(\vec{x}_k) - \nabla f(\vec{x}_{k-1}))}{\vec{d}_{k-1}^T(\nabla f(\vec{x}_k) - \nabla f(\vec{x}_{k-1}))} \tag{2.12}$$

The CG method can be of course used to solve a positive definite linear system of

$$\mathbf{A}\vec{x} = \vec{b} \tag{2.13}$$

where $\mathbf{A}$ is $n \times n$ symmetric positive definite matrix and $b \in \mathbb{R}^n$. Pseudocode of the CG algorithm is given in Listing 2.1.

Algorithm 2.1 Conjugate Gradient method.

```
x := x_0; β := 0; d := 0
repeat
  d_old := d
  d := −∇f(x) + β d_old
  if ∇f(x)^T d ≧ 0 then
    d := −∇f(x)
  end if
  α := lineSearch(x, d)
  x := x + α d
  β := ...
until found
```

2.2.1.3 Levenberg–Marquardt method (LM)

The LM method together with the quasi-Newton methods are so called *Newton–type methods*, LM particularly belongs to Damped Newton methods class. To explain these two methods we need to briefly describe the **Newton's method** so the basic idea behind these algorithms is clear.

Newton's method is presently used to solve systems of non-linear equations, but had also been used for unconstrained optimization. We should start describing Newton's method by repeating the Taylor expansion of the objective function in the neighbourhood of some point $\vec{x}$:

$$f(\vec{x} + \Delta\vec{x}) \approx f(\vec{x}) + \Delta\vec{x}^T\nabla(\vec{x}) + \frac{1}{2}\Delta\vec{x}^T\mathbf{H}\Delta\vec{x}. \tag{2.14}$$

This gives us a quadratic model of f. Now we need to minimize the model at the current $\vec{x}$. If $\mathbf{H}$ is positive definite we know where the minimum is:

$$\nabla(\vec{x}) + \mathbf{H}\Delta\vec{x} = \mathbf{0}. \tag{2.15}$$

Now we can draft a pseudocode of the Newton's method (Listing 2.2).

This algorithm works well as long as $\mathbf{H}$ is non-singular. For $\mathbf{H}$ positive definite, each step leads downhill. Under these conditions Newton's method converges quadratically towards minimum provided the starting point is close enough. This would be very fine if there were no drawbacks. The biggest one is the lack of global convergence, which means the method does not guarantee that it will find minimum in all cases. Another serious situation may arise when starting the method's iterations far from the minimum. In the areas not close enough

Algorithm 2.2 Newton's method.

$\vec{x} := \vec{x}_0$
repeat
  solve $\mathbf{H}\Delta\vec{x} = -\nabla(\vec{x})$
  $\vec{x} := \vec{x} + \Delta\vec{x}$
until found

to the solution we may expect that the Hessian matrix will not be positive definite and then we could find ourselves stepping toward a stationary point (e.g. saddle point or even local maximum). That is because without positive definite Hessian matrix the algorithm reduces to solving non-linear system of equations $\nabla(\vec{x}) = \mathbf{0}$. Other difficulties with the Hessian matrix include ill-conditioning or singular matrix, both lead to errors in the computed step. There are techniques for detecting these situations, but even if we know this situation arised it takes additional effort to handle it. Perhaps the last (and yet the most practical) drawback is the need for analytic second derivatives of the objective function. Even if they are known to exist it might prove difficult to obtain them and further the calculation of the Hessian matrix may be costly.

These disadvantages led the researchers to the idea of somehow combining the original Newton's method with the method of steepest descent. We would simply like to use the global convergence properties of the SD if the Newton's method seems to be unusable and as soon as we get close to the solution we would switch to the Newton's method and enjoy its quadratic convergence.

Obvious modification is to replace the actual step with a line search with a direction $\vec{d} = -\mathbf{H}^{-1}\nabla(\vec{x})$, which could work well as long as $\mathbf{H}$ is positive definite, because then $\vec{d}$ is a descent direction. This still does not solve the possibility of not having the Hessian matrix positive definite. For this, we will describe the class of Damped Newton methods and later give an example of the Levenberg–Marquardt method.

The step in the direction of a stationary point of the quadratic approximation of f (Equation 2.14) is replaced by a step in the direction of a stationary point of modified approximation:

$$f(\vec{x} + \Delta\vec{x}) \approx f(\vec{x}) + \Delta\vec{x}^T\nabla(\vec{x}) + \frac{1}{2}\Delta\vec{x}^T(\mathbf{H} + \lambda\mathbf{I})\Delta\vec{x}, \tag{2.16}$$

where $\mathbf{I}$ is the identity matrix and λ is an algorithm parameter. This is contribution of K. Levenberg and it has been shown that for sufficiently large λ the matrix $\mathbf{H} + \lambda\mathbf{I}$ is positive definite [11] and thus the step taken leads to a minimum of the modified quadratic model. The algorithm is simple and for large λ we get SD-like behaviour which is desirable in early iterations and as we get closer the λ is modified (decreased) and we are taking Newton steps. Marquardt had shown that these corrections of the damping parameter can be considered as a sort of trust region approach [12]. Again we need to have defined second derivatives of the objective function but other difficulties had effectively vanished. The only risk is that the (modified) quadratic approximation will not be a good one.

Final form of the Levenberg–Margardt was born after Marqardt's suggestion of replacing the identity matrix with a diagonal matrix of the Hessian matrix (see Listing 2.3).

2.2.1.4 quasi-Newton method (QN)

In the previous section we defined one of the Damped Newton methods, the LM method, which solved all the difficulties of using the original Newton's method except for the last one

Algorithm 2.3 Levenberg–Margardt method.

$\vec{x} := \vec{x}_0; \lambda := \lambda_0$
repeat
  solve $\frac{1}{2}(\mathbf{H} + \lambda \mathrm{diag}(\mathbf{H}))\Delta\vec{x} = -\frac{1}{2}\nabla(\vec{x})$
  if $f(\vec{x} + \Delta\vec{x}) < f(\vec{x})$ **then**
    $\vec{x} := \vec{x} + \Delta\vec{x}$
    $\lambda := \lambda \cdot 0.1$
  else
    $\lambda := \lambda \cdot 10.0$
  end if
until found

concerning the need of defined analytic second derivatives (or supplied routines for numerical computing of these values). Now we shall present the well known quasi-Newton method that overcomes even this disadvantage. The idea behind the QN method is not to use a *real* Hessian matrix nor its inverse but rather their approximations which are iteratively modified according to the information gained from function evaluation and its gradient.

We may notice that using numerical computing of the Hessian matrix (e.g. when the analytic formula is not available) in the original Newton's method is actually a quasi-Newton method since it is just an approximation of the real values. But computing numerical Hessian matrix is usually costly (depending on the implementation it is very costly or just costly enough not to use it regularly) and we can not guarantee that this matrix will be positive (semi-)definite.

Modern QN methods use sophisticated formulas for updating the approximation of the Hessian matrix (or its inverse) so that it converges to the $\mathbf{H}$ or $\mathbf{H}^{-1}$ respectively for the minimum. In the following text we will prefer to use an approximation of the inverse Hessian matrix as it is usually done – there are three main reasons to do so:

- The search direction is obtained by multiplying the approximation of the inverse with the negative gradient, that requires $\mathcal{O}(d^2)$ operations whereas when using approximation of the Hessian matrix we would have to solve the linear system with given matrix, that is $\mathcal{O}(d^3)$ operations, d is the number of dimensions.

- It is possible to use approximations to the Cholesky factor of the Hessian matrix which reduces the computational burden of solving this linear system to $\mathcal{O}(d^2)$ but this requires additional techniques and the code of the algorithm becomes less explicit. For details consult [13] and [4].

- Otherwise the number of computations needed to update either the approximation of the Hessian matrix or of its inverse is the same.

The updating formula must comply with the so called *quasi-Newton condition*, also known as the *secant condition*. The formula's task is to compute a new approximation to the inverse of the Hessian matrix using a correction of the current one. This correction must somehow contain (even only approximate) information about the second derivatives of the objective function. The way how to get this information is to use gradients of the function at two different points. Remembering the Taylor expansion Equation 2.14 we can now work with Taylor expansion of f' in the neighbourhood of $\vec{x}$:

$$f'(\vec{x}) \approx f'(\vec{x} + \Delta\vec{x}) - f''(\vec{x} + \Delta\vec{x})\Delta\vec{x}. \tag{2.17}$$

Higher order terms are assumed to be negligible. Lets mark $\vec{y} = \nabla f(\vec{x} + \Delta\vec{x}) - \nabla f(\vec{x})$, rewriting Equation 2.17 to

$$\vec{y} \approx f''(\vec{x} + \Delta\vec{x})\Delta\vec{x}. \tag{2.18}$$

It is now obvious that the updated approximation of the inverse Hessian matrix must satisfy

$$\mathbf{H}^{-1}_{\text{approx}}\vec{y} = \Delta\vec{x}. \tag{2.19}$$

The approximated matrix is being usually updated every iteration, thus we need to be able to compute the update fast and effectively. The most used update is therefore in the form of recursive relation

$$\mathbf{H}^{-1}_{\text{approx,k+1}} = \mathbf{H}^{-1}_{\text{approx,k}} + \vec{a}\vec{b}^T + \vec{c}\vec{d}^T \tag{2.20}$$

where vectors $\vec{a}, \vec{b}, \vec{c}, \vec{d} \in \mathbb{R}^n$ and usually $\vec{a} = \vec{b}, \vec{c} = \vec{d}$ to achieve symmetric updates. These are classified as a *rank-one update* if the last term is omitted and as a *rank-two update* if not.

Now we have to mention why it is sometimes more advantageous to use an approximation instead of a real inversion of Hessian matrix. As already stated, there is no guarantee whatsoever that the Hessian matrix is positive definite. Now if we initialize the approximated matrix with the identity matrix $\mathbf{I}$ and ignore the updating formula, we once again obtain the SD method. Its global convergence properties were already examined so we know that if we use this approach while far from the minimum, we will get good results. As the point closes to the solution, the quadratic approximation of the objective function becomes valid and the Newton's method could be used with profit. This was the case for the LM method as well as it is for the quasi-Newton method: use the identity matrix for the first step and adjust it iteratively to approximate the inverse of the Hessian matrix, going from the SD method to something close to the Newton's method.

There are the three most know update formulas up to date and a combination of these (we use $\mathbf{D}$ to denote $\mathbf{H}^{-1}_{\text{approx}}$):

- **Broyden's** formula [14]:

$$\mathbf{D}_{k+1} = \mathbf{D}_k + \frac{(\Delta\vec{x}_k - \mathbf{D}_k\vec{y})\Delta\vec{x}_k^T\mathbf{D}_k}{\vec{y}_k^T\mathbf{D}_k\Delta\vec{x}_k} \tag{2.21}$$

- **Davidon—Fletcher—Powell (DFP)** formula [15], [16]:

$$\mathbf{D}_{k+1} = \mathbf{D}_k + \frac{\Delta\vec{x}_k\Delta\vec{x}_k^T}{\vec{y}_k^T\Delta\vec{x}_k} - \frac{\mathbf{D}_k\vec{y}_k\vec{y}_k^T\mathbf{D}_k^T}{\vec{y}_k^T\mathbf{D}_k\vec{y}_k} \tag{2.22}$$

- **Broyden—Fletcher—Goldfarb—Shanno (BFGS)** formula [17], [18], [19], [20]:

$$\mathbf{D}_{k+1} = \mathbf{D}_k + \frac{(\Delta\vec{x}_k^T\vec{y}_k + \vec{y}_k^T\mathbf{D}_k\vec{y}_k)(\Delta\vec{x}_k\Delta\vec{x}_k^T)}{(\Delta\vec{x}_k^T\vec{y}_k)^2} - \frac{\mathbf{D}_k\vec{y}_k\Delta\vec{x}_k^T + \Delta\vec{x}_k\vec{y}_k^T\mathbf{D}_k}{\Delta\vec{x}_k^T\vec{y}_k} \tag{2.23}$$

- so called **Broyden family** formula:

$$\mathbf{D}_{k+1} = (1-\phi)\mathbf{D}^{\text{BFGS}}_{k+1} + \phi\mathbf{D}^{\text{DFP}}_{k+1}, \phi \in < 0, 1 > \tag{2.24}$$

where $\vec{y}_k = \nabla f(\vec{x}_{k+1}) - \nabla f(\vec{x}_k)$ is the difference of the gradients in two succesive points and $\Delta\vec{x}_k$ is the step taken in the k^{th} iteration. The parameter ϕ of the Broyden's family formula can

be adjusted during the algorithm run, but setting $\phi = 0$ to obtain BFGS updating is usually the best [21].

The update of the approximation matrix is intended to take place every iteration but sometimes it is wise to skip the update. This is usual when the update would result in a non positive definite matrix [13]. Pseudocode of the QN method is listed in Listing 2.4 with this condition added.

Algorithm 2.4 quasi-Newton method.

```
x := x_0; D_0 := I
repeat
  d := -D∇f(x)
  α := lineSearch(x, d)
  Δx := αd
  y := ∇f(x + Δx) - ∇f(x)
  x := x + Δx
  if Δx^T y > √ε · ||Δx||_2 · ||y||_2 then
    D_{k+1} = ...
  end if
until found
```

2.2.2 Orthogonal search (OS)

When introducing the line search method, we mentioned that it can be considered a minimization algorithm for unidimensional functions. The idea of searching for a minimum along a given direction from an arbitrary point in one dimension can be thus easily extended to multiple dimensions just by altering the dimensions in which we search. This is the basis of a very simple and yet sometimes effective algorithm called *orthogonal search*.

We simply set each of d search directions parallel to the axis and thus obtain orthogonal search directions in d-dimensional space (these directions are unit vectors). Then we apply a line search method to each direction to find a minimum for a given dimension while the others are held fixed and repeat it until the method converge to some local minimum. Thus the whole algorithm is just a sequence of successive line minimizations. This method can again of course get caught in the Stiefel's cage (see Section 2.2.1.1) by taking many little steps. This method is common to use a random order of the search directions (search dimensions) each iteration, then it is usually called Stochastic orthogonal search.

Readers should notice we left the class of gradient methods as this algorithm does not make any use of the gradient information. On the other hand, it is usually a good idea to make use of it during the line search and then it would be too expensive not to use them in the method itself as well. The pseudocode of this method can be found in Listing 2.5. Note that $\mathbf{A}_k$ denotes a k^{th} row vector of the matrix $\mathbf{A}$, $\vec{o}_j$ denotes j^{th} element of the vector $\vec{o}$ and d is dimension of the objective function f. $\vec{o}$ is the ordering of search dimensions.

2.2.3 Powell's method (PM)

The use of search directions introduced in the previous section is very simple and very commonly leads to a slow rate of convergence. Powell [22] realized that we need a better set of directions, perhaps a bit different with every iteration. This is not very different from the idea of the CG method, which is connected to the conjugated directions.

Algorithm 2.5 (Stochastic) Orthogonal search.

$\vec{x} := \vec{x}_0; \mathbf{A} := \mathbf{I}; \vec{o} = (1, ..., d)$
repeat
  if use stochastic **then**
    shuffle($\vec{o}$)
  end if
  for $i = 1$ to d **do**
    α :=lineSearch($\vec{x}, \mathbf{A}_{\vec{o}_i}$)
    $\vec{x} := \vec{x} + \alpha \mathbf{A}_{\vec{o}_i}$
  end for
until found

Powell's method is again very simple and produces d mutually conjugate directions. We have the same set of search directions as before, i.e. we initialize them as unit vectors. Then we search one by one which results in a new position. Vector marking displacement of this point to the one at the beginning of the iteration is added to the set of directions and the first one is discarded. Powell proved that for a quadratic form k iterations of this algorithm produces modified set of directions whose last k elements are mutually conjugate and thus after d iterations (yielding in $d(d-1)$ line searches) we have the minimum of a quadratic form.

The problem is that this method tends to produce a set of directions that become linearly dependent and thus it is able to find a minimum only in a subspace of the d-dimensional objective function, i.e. it can not find the correct solution.

The correction is not to discard the first search direction but instead to discard the one that was the most successful, i.e. that contributed the most to the decrease of the function value. Although paradoxical at the first glance, the explanation is simple. This direction indeed was the best one but this implies that it will be most likely the major component of the new direction, which is the displacement of the original point after searching through the whole set of search directions. Thus if we discard this one, we get the biggest chance to avoid a folding up of the search directions.

There are two exceptions to updating the set of directions. We define $f_0 \equiv f(x_0)$ where x_0 is the point at the beginning of iteration, i.e. before commencing the succesive line searches; next we define $f_d \equiv f(x_d)$ where x_d is analogically the point at the end of iteration, i.e. after optimizing it in the all search directions; next $f_e \equiv f(2\vec{x_d} - \vec{x_0})$, this is the function value further along the new direction; and finally Δf is a value of the biggest decrease in the function value among one direction. Conditions under which the set is left unchanged (no new direction is added and no old one is discarded) are these:

- $f_e \geqq f_0$ – this says that the new direction would not bring much use anyway and we might find the would be discarded direction still useful,
- $2(f_0 - 2f_d + f_e)((f_0 - f_d) - \Delta f)^2 \geqq (f_0 - f_e)^2 \Delta f$ – it would not help us to discard the *best* direction since the decrease in the function value was not due to any single direction in particular or there is a substantial second derivative along the average direction and we seem to be near the bottom of its minimum [4].

With these conditions included the algorithm is defined by Listing 2.6 where the orthogonal search notation is used and $\vec{u}_i$ is a unit vector in the i^{th} dimension.

Algorithm 2.6 Powell's method.

$\vec{x} := \vec{x}_0$
for $i = 1$ to d **do**
  $\vec{d_i} := \vec{u}_i$
end for
repeat
  $\vec{x}_{\text{start}} := \vec{x}$
  $f_0 := f(\vec{x}_{\text{start}})$
  $\Delta f := 0$
  for $i = 1$ to d **do**
    $\alpha :=$lineSearch$(\vec{x}, \vec{d_i})$
    if $f(\vec{x}) - f(\vec{x} + \alpha\vec{d_i}) \geqq \Delta f$ **then**
      $\Delta f := f(\vec{x}) - f(\vec{x} + \alpha\vec{d_i})$
      $dir := i$
    end if
    $\vec{x} := \vec{x} + \alpha\vec{d_i}$
  end for
  $f_d := f(\vec{x})$
  $f_e := f(2\vec{x} - \vec{x}_{\text{start}})$
  if $f_e < f_0$ && $2(f_0 - 2f_d + f_e)((f_0 - f_d) - \Delta f)^2 < (f_0 - f_e)^2\Delta f$ **then**
    $\vec{d}_{dir} := \vec{x} - \vec{x}_{\text{start}}$
    $\alpha :=$lineSearch$(\vec{x}, \vec{d}_{dir})$
    $\vec{x} := \vec{x} + \alpha\vec{d}_{dir}$
  end if
until found

2.2.4 Covariance Matrix Adaptation Evolution Strategy (CMA-ES)

The CMA-ES is a stochastic optimization method for optimizing non-linear and non-convex functions [23], [24]. This method is both numerical method and a population based, thus sometimes referred to as a hybrid between numerical method and a nature inspired one.

The whole algorithm is well defined and described by authors and as such we recommend to the reader to refer to an excellent algorithm derivation in [25], which also presents a MATLAB source code for the algorithm. The code we implemented is thus merely an adapted version of this one and here we will only briefly describe the main characteristics of the algorithm itself.

The CMA-ES method works with a population which is generated by sampling a multivariate normal distribution:

$$\vec{x}_k^{(g+1)} = \vec{m}^{(g)} + \sigma^{(g)}\mathcal{N}(0, \mathbf{C}^{(g)}) \tag{2.25}$$

where the upper indexes denote the generation number (i.e. the iteration number), $\mathcal{N}(0, \mathbf{C})$ is a multivariate normal distribution with zero mean and a covariance matrix $\mathbf{C}$, $\vec{x}_k$ is a k^{th} member of the current population, $\vec{m}$ is the mean value of the search distribution, σ is the standard deviation used as a step size and $\mathbf{C}$ is the covariance matrix of the search distribution up to the scalar factor of σ^2 [25]. Now all we need to do is to compute the mean value, the standard deviation and the covariance matrix.

The current mean value is computed as a weighted average of μ selected points which act as parents to the new population:

$$\vec{m}^{(g+1)} = \sum_{i=1}^{\mu} w_i \vec{x}_{i:\lambda}^{(g+1)} \tag{2.26}$$

where $w_i \in \mathbb{R}_+$ is the weight coefficient for a recombination (sum of weights equals one), λ is the population size and $\vec{x}_{i:\lambda}$ is a member of the population with i^{th} lowest function value (i.e. for $i = 1$ it is the *best* individual). The algorithm also introduces measure called *variance effective selection mass* defined as

$$\mu_{\text{eff}} = \left(\sum_{i=1}^{\mu} w_i^2 \right)^{-1} \tag{2.27}$$

which is used e.g. to indicate reasonable setting of the recombination weights.

Next the covariance matrix is estimated. The algorithm uses combination of a rank one update (for a limit case when only a single point is used to update the covariance matrix) and of a rank μ update (when we use all the parents available). This is used because when estimating the covariance matrix from a small population the estimate is not reliable [23]. Equation 2.28 gives the final form of covariance matrix update.

$$\mathbf{C}^{(g+1)} = (1 - c_1 - c_\mu)\mathbf{C}^{(g)} + c_1 \vec{p}_c^{\,(g+1)} \vec{p}_c^{\,(g+1)^T} + c_\mu \sum_{i=1}^{\mu} w_i \vec{y}_{i:\lambda}^{\,(g+1)} \vec{y}_{i:\lambda}^{\,(g+1)^T} \tag{2.28}$$

where c_1, c_μ are learning rate for the rank one update and learning rate for updating the covariance matrix respectively, $\vec{p}_c$ is so called *evolution path* [23] and $\vec{y}_{i:\lambda} = \frac{\vec{x}_{i:\lambda} - \vec{m}}{\sigma}$. The second term is the rank one update, the last term is the rank μ update. Rank μ update uses the information content of one generation (thus of one current population), the rank one update interprets the correlations between generations by using the evolution path.

The last thing to determine is the value of the standard deviation used as a step size, the value of σ. Similar idea to computing evolution path for tracking the correlations between generations is incorporated and CMA-ES introduces a *conjugate evolution path* to track successive steps [23]:

$$\sigma^{(g+1)} = \sigma^{(g)} e^{\left(\frac{c_\sigma}{d_\sigma} \left(\frac{\|\vec{p}_\sigma^{\,(g+1)}\|}{\mathrm{E}\|\mathcal{N}(\mathbf{0},\mathbf{I})\|} - 1 \right) \right)} \tag{2.29}$$

where $\vec{p}_\sigma$ is the conjugate evolution path for σ, c_μ is the learning rate for the step size control and d_σ is a damping parameter for scaling the magnitude of change of σ and $\mathrm{E}\|...\|$ is an expectation of the Euclidean norm of a random vector under given distribution.

Listing 2.7 gives a basic pseudocode of the complete algorithm. For additional details please refer to [25] where several improvements can be found. Default recommended parameter values can be found in Appendix B.

2.3 Nature inspired methods

There exist many optimization methods utilizing or mimicking processes commonly found in the nature. These then usually work with a set of individuals with specific properties and iteratively adapt this set to optimally converge to the solution. A population of such individuals is thus the basis of all of the following algorithms, but each algorithm differs in what exactly these individuals represent and how they take part in the construction of the next iteration step.

Important characteristic of the following optimization methods inspired by nature is that these do not use nor need the availability of analytic gradient and/or hessian, thus using the objective function only as a black box that gives us information on our solution candidates. The lack of gradient/hessian computation is advantageous but this is counterbalanced by populous function evaluations as these methods work with a set of points instead of just one.

Algorithm 2.7 Covariance Matrix Adaptation Evolution Strategy.

Initialize parameters λ, μ, $w_{i=1\ldots\mu}$, c_σ, d_σ, c_c, c_1, c_μ
Set $\vec{m}$ and σ problem dependent
$\vec{p}_\sigma = \vec{p}_c = \mathbf{0}$
$\mathbf{C} = \mathbf{I}$
$g = 0$
repeat
  $g := g + 1$
  for $k = 1$ to λ **do**
    $\vec{z}_k \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$
    $\vec{y}_k = \mathbf{B}\mathbf{D}\vec{z}_k$
    $\vec{x}_k = \vec{m} + \sigma\vec{y}_k$
  end for
  $\langle\vec{y}_\mathrm{w}\rangle = \sum_{i=1}^{\mu} w_i \vec{y}_{i:\lambda}$
  $\vec{m} = \vec{m} + \sigma\langle\vec{y}_\mathrm{w}\rangle$
  $\vec{p}_\sigma = (1 - c_\sigma)\vec{p}_\sigma + \sqrt{c_\sigma(2 - c_\sigma)\mu_\mathrm{eff}}\mathbf{C}^{-\frac{1}{2}}\langle\vec{y}_\mathrm{w}\rangle$
  $\sigma = \sigma e^{\left(\frac{c_\sigma}{d_\sigma}\left(\frac{\|\vec{p}_\sigma\|}{\mathrm{E}\|\mathcal{N}(\mathbf{0},\mathbf{I})\|} - 1\right)\right)}$
  $\vec{p}_c = (1 - c_c)\vec{p}_c + \sqrt{c_c(2 - c_c)\mu_\mathrm{eff}}\langle\vec{y}_\mathrm{w}\rangle$
  $\mathbf{C} = (1 - c_1 - c_\mu)\mathbf{C} + c_1\vec{p}_c\vec{p}_c^{\,T} + c_\mu \sum_{i=1}^{\mu} w_i \vec{y}_{i:\lambda}\vec{y}_{i:\lambda}^{\,T}$
until found

2.3.1 Ant algorithms

A well known and wide class of nature inspire optimization algorithms are the Ant inspired algorithms. These algorithms focus on the way the ants communicate and how they proceed in the search of food sources, resulting particularly in the discovery of shortest path from their hive to the food source. Common for all these algorithms is the matter called *pheromone*, which is a chemical substance layed by ants on the path they travel. The more pheromone on a particular patch, the higher probability is that an ant encountering this path will follow it rather then continuing some other way. This substance eventually evaporates ensuring that only frequently used paths are marked in a such fashion.

Ant algorithms were first proposed as an alternative methods of discrete optimization. Over the time they proved effective in solving multiple different problems. First implementation of an ant algorithm presented the Ant Colony Optimization [26]. However it did not took long and this idea was applied to continuous optimization as well. To do so there are generally three possibilities:

1. To simulate ant behaviour directly (Continuous Ant Colony Optimization, *Pachycondyla apicalis* method).

   Ants' behaviour is naturally continuous and the idea is to mimic the movement and ways the ants communicate. The search locations are thus represented by vectors in the objective function search space and the pheromone is then added to these vectors indicating the function value for each particular vector, simulating amount of the food found. When an ant sets off from the nest in search of food a probabilistic choice is made based on the pheromone amount. The ant then explores the area *around* the selected vector simulating a path leading to a food source.

2. To extend the existing Ant Colony Optimization by discretization (Adaptive Ant Colony Algorithm).

   This approach is a result of simple fact that we can convert continuous optimization problem to a discrete one by encoding the real–valued parameters as e.g. a binary string with limited length (and thus limiting the precision of the original parameters). The algorithm then can assign different pheromone levels to each digit place of the resulting encoded string and ants explore the space by traversing from one end of the string (e.g. lowest bit) to the other (highest bit) and choose current digit value from the possible options. Obviously the ants' behaviour is there mapped to the number itself rather then to an area of the search space.

3. To extend the existing Ant Colony Optimization by probabilistic sampling (Extended Ant Colony Optimization, Direct Ant Colony Algorithm).

   Previous techniques represented the amount of pheromone only in connection with some discrete position although the pheromone levels layed by ants are continuous. To achieve this one can represent position of the ant probabilistically instead of by discrete vectors. Of course the probability of the ant position is again outspread to a certain area of the search space.

Algorithms described in this chapter represent all three approaches. Implementations of these in the JCool environment are based on the previous work on FAKE GAME software by Oleg Kovářík [27] and on corresponding papers.

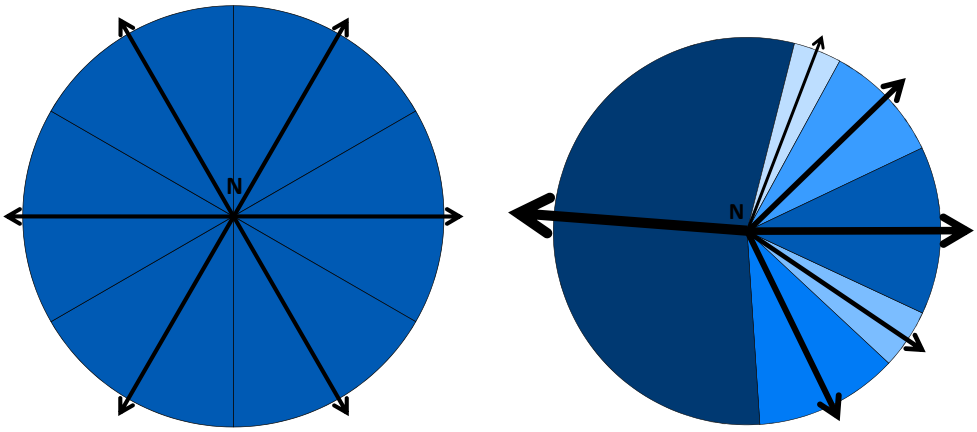
2.3.1.1 Continuous Ant Colony Optimization (CACO)

Continuous Ant Colony Optimization (CACO) is considered to be the first application of ant algorithm in the domain of continuous optimization [28]. It works with the concept of a hive and search vectors representing the ants' paths to food sources, yielding in a local search method. Authors of the algorithm suggested to search for optimal hive position by some external global search method.

The number of vectors is constant and they are initialized uniformly within the search radius with initial pheromone amount (see Figure 2.2a, N marks the nest, usually set to zero coordinates). Ant chooses its path by probabilistic selection based on the pheromone amount attached to the vectors, travels to the place where the vector points and performs a local random search within its local search radius. The local search is actually done by shifting a point coordinates by vector using d-dimensional spherical coordinates [27]. This local search radius might change as the algorithm iterates to stress more detailed search. Afterwards we compute the function value representing amount of food each ant found and the amount of pheromone attached to each vector is corrected accordingly. Furthermore the vector itself is rearranged to point to the best location found within according local search radius (Figure 2.2b). For implementation details please refer to [27] and Appendix B.

2.3.1.2 *Pachycondyla apicalis* method (API)

Method called API is named after one particular ant specie, *Pachycondyla apicalis* [29]. Again the algorithm uses a nest from which the ants start their search. Each ant has assigned its own search area in the vicinity of the nest and in this area it has several so called *hunting sites*. These hunting sites represent areas that are actually explored by this particular ant. Hunting sites are added in the same fashion as CACO method explores the space, by counting



(a) Initial state (b) After several iterations

Figure 2.2: Visualisation of pheromone levels assigned to search vectors in CACO.

a shift vector using d-dimensional spherical coordinates with respect to the nest position. The exploration of the area around the hunting site is again based on this approach.

Figure 2.3 shows how does this algorithm work: N marks the nest coordinates (i.e. center), $r1$ is the (global) search radius, $Hi, i = 1, 2, 3$ marks individual hunting sites, $r2$ is the local search radius (i.e. how far from the hunting site is the ant allowed to explore) and e marks individual search spots belonging to a given hunting site.

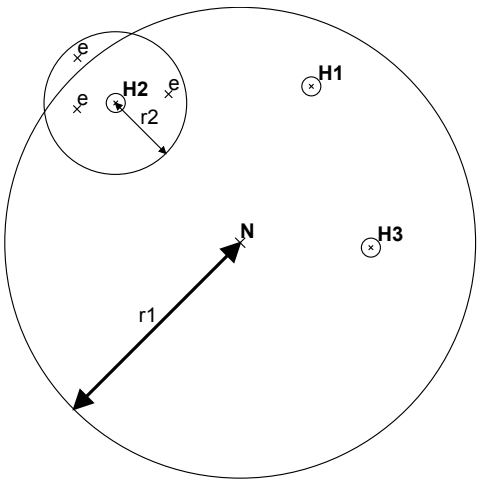


Figure 2.3: API algorithm.

Hunting site is explored repeatedly if the last visit was successful (i.e. led to a lower function value) but is abandoned if the search was unsuccessful for some predefined number of trials (this parameter is called *starvation* limit). The nest is periodically moved to the best location (hunting site) found so far and the whole process begins again.

Together with so called *tandem run*, a process of selecting two random ants and substituting the best hunting site of the one with worse solution by the best hunting site of the other ant leading in directing more stress to the promising areas, it is simple and yet effective algorithm.

2.3.1.3 Adaptive Ant Colony Algorithm (AACA)

This method was first proposed as Basic Ant Colony Algorithm (BACA), successively modified by authors and renamed to Adaptive Ant Colony Algorithm [30]. It is basically an ant algorithm applied to an oriented graph of zeros ane ones representing digits of a binary representation of a real-value number. Process of encoding and decoding real values as e.g. binary strings implies that we need to select minimal and maximal value of the optimized parameter and thus forces us to limit the search space.

Once the number is encoded as n-bit string, the optimization proceeds as follows: Ant's first "trip" is from a start position to the first node (in this case the MSB) and afterwards step by step with probabilistic choice between the edges ahead to the LSB. At the end the encoded number is decoded and evaluated for a function value. The amount of pheromone is finally modified along the visited edges accordingly. Three unrelated pheromone levels for $n = 8$ bits are shown on Figure 2.4. The thickness of the arrows connecting individual bit states (either 0 or 1) represents amount of the pheromone currently assigned to this edge. The **S** state represents starting position. Each ant then represents only a sample of the probabilistic definition of a *path* between individual bit positions.

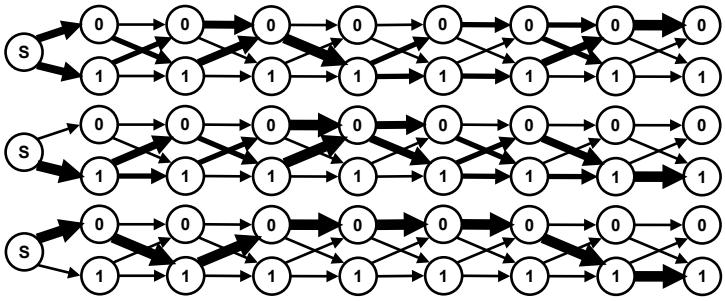


Figure 2.4: Visualisation of three different pheromone levels.

The AACA was designed to overcome a disadvantage of BACA consisting of exploring the search space rather nonsystematically. This was caused by frequent changes in more significant parts of the number. For this authors proposed to lower the probability of change in these parts of number as solution quality gets higher by increasing the rate of pheromone layed on according edges. The pheromone increment for the edge (i, j) is given by Listing 2.30; k is the bit position this edge corresponds to, f is a function value for the decoded point, $f_{\min}$ is the best function value found so far and $\beta > 0$, $\delta > 0$ are parameters of the algorithm refered to as *ant like behaviour coefficient* and *heuristic behaviour coefficient* respectively.

$$\Delta\phi_{i,j} = \frac{1}{1 + e^{\beta k f (f - (f_{\min} + \delta))}} \tag{2.30}$$

2.3.1.4 Extended Ant Colony Optimization (ACO*)

The Extended ACO method is based on the idea of using a probability distribution to choose the value of a function variable [31], [32]. The most commonly used probability distribution is the normal distribution. The probability distribution transforms to a probability density function in the continuous domain:

$$p(\vec{x}, \vec{\mu}, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(\vec{x} - \vec{\mu})^2}{2\sigma^2}}, \tag{2.31}$$

where $\vec{\mu}$ is a mean value of the distribution and σ is a standard deviation. As stated before, this one function can describe only one promising area of the search space and thus the ACO* method uses weighted sum of k such functions:

$$p(\vec{x}, \mathbf{M}, \vec{\sigma}, \vec{w}) = \sum_{i=1}^{k} w_i p(\vec{x}, \mathbf{M}_i = \vec{\mu}^{(i)}, \sigma_i), \tag{2.32}$$

where $\mathbf{M}$ is a vector of mean values, $\vec{\sigma}$ is a vector of standard deviations and $\vec{w}$ is a vector of weights associated to each particular probability distribution function. The resulting distribution describes the whole search space as needed, see Figure 2.5. The Equation 2.32 is represented by purple color ($w_i = 1, i = 1, ..., 4$).

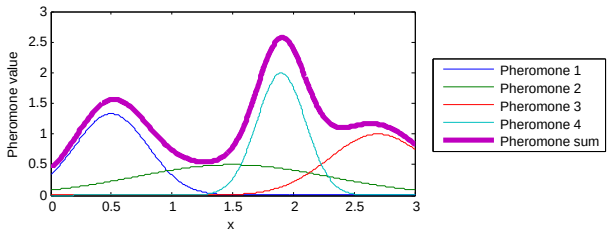


Figure 2.5: Visualisation of pheromone level.

To sample such weighted sum would be difficult and thus it is advisable to choose one of these probability distribution functions with probability given by its corresponding proportional weight.

The weight corresponding to a given probability density function is based on its rank among others which is based on the function value. It is then transformed by a Gaussian function:

$$w_i = \frac{1}{\sqrt{2\pi\omega^2 k^2}} e^{-\frac{(r-1)^2}{2\omega^2 k^2}} \tag{2.33}$$

where ω is an algorithm parameter and k is the population size (i.e. number of ants used or in other words the number of probability density functions used).

Several methods of updating the pheromone amount are suggested by [31]:

- we can add new or remove old probability density functions (i.e. ants) to / from the population when appropriate (adding new when a promising area has been found, removing old that proved misleading)
- modifying weights assigned to individual ants (i.e. adding pheromone of evaporate it) so that the weight changes proportionally
- increasing deviations assigned to individual ants – this is then called *dissolving* instead of evaporation as it causes numbers generated by given probability density function to be spread over a larger area of the search space

These approaches can be of course combined.

2.3.1.5 Direct Ant Colony Optimization (DACO)

The DACO algorithm also uses the probability distribution but in a different fashion. For each of the variables of the minimized multivariate function there is one probability distribution

with specific mean value and standard deviation as associated parameters [33]. These then act as a measure of pheromone amount. New points are generated by sampling these n probability distributions and the fitness of the best individual ant (i.e. the minimal function value found so far) is then used to update the pheromone quantity. The update is composed of evaporation

$$\vec{\mu}_{\mathrm{p}}^{(k+1)} = (1-\rho)\vec{\mu}_{\mathrm{p}}^{(k)} \tag{2.34}$$

$$\vec{\sigma}_{\mathrm{p}}^{(k+1)} = (1-\rho)\vec{\sigma}_{\mathrm{p}}^{(k)} \tag{2.35}$$

and of pheromone intensification

$$\vec{\mu}_{\mathrm{p}}^{(k+1)} = \vec{\mu}_{\mathrm{p}}^{(k+1)} + \rho\vec{x}_{\mathrm{best}} \tag{2.36}$$

$$\vec{\sigma}_{\mathrm{p}}^{(k+1)} = \vec{\sigma}_{\mathrm{p}}^{(k+1)} + \rho|\vec{x}_{\mathrm{best}} - \vec{\mu}_{\mathrm{p}}^{(k)}|, \tag{2.37}$$

where $\vec{\mu}_{\mathrm{p}}^{(k)}$ is a vector storing the amount of *mean value*-pheromone for each dimension in iteration k, $\vec{\sigma}_{\mathrm{p}}^{(k)}$ is a vector storing the amount of *standard deviation*-pheromone for each dimension and ρ is a pheromone evaporation factor. $\vec{x}_{\mathrm{best}}$ is the best solution found by the algorithm so far.

Authors also suggest several improvements to prevent premature convergence, these can be found in [33].

2.3.2 Particle Swarm Optimization (PSO)

Particle Swarm Optimization is another well known optimization method. It is attributed to [34] and [35]. PSO maintains a population of particles (common metaphor is a swarm of birds or fish) which acts as a swarm moving through a terrain, searching for food and communicating actual information.

Each particle has its own speed, local best solution found so far and a knowledge about the location of the global best solution found so far by the swarm as a whole. This data are enough to compute particle's position at the next iteration implementing the idea that the particle should be attracted to both its local best solution and to the swarm's global best solution as these represent points in the search space with higher likelihood of having promising neighbourhood. There are several velocity update formulas:

- **Original** update formula:

$$\vec{v} = w\vec{v} + r_{\mathrm{local}}\phi_{\mathrm{local}}(\vec{x}_{\mathrm{local\ best}} - \vec{x}) + r_{\mathrm{global}}\phi_{\mathrm{global}}(\vec{x}_{\mathrm{global\ best}} - \vec{x}) \tag{2.38}$$

- **Fully Informed** [36]:

$$\vec{v} = w\vec{v} + r_{\mathrm{local}}\phi_{\mathrm{local}}\frac{\sum_{i=1}^{m}(\vec{x}_{i_{\mathrm{local\ best}}} - \vec{x})}{m} \tag{2.39}$$

- **Canonical** [37]:

$$\vec{v} = \chi(\vec{v} + r_{\mathrm{local}}\phi_{\mathrm{local}}(\vec{x}_{\mathrm{local\ best}} - \vec{x}) + r_{\mathrm{global}}\phi_{\mathrm{global}}(\vec{x}_{\mathrm{global\ best}} - \vec{x})) \tag{2.40}$$

$$\chi = \begin{cases} \frac{2\kappa}{2-(\phi_{\mathrm{local}}+\phi_{\mathrm{global}})-\sqrt{(\phi_{\mathrm{local}}+\phi_{\mathrm{global}})((\phi_{\mathrm{local}}+\phi_{\mathrm{global}})-4)}} & \text{if } \phi_1+\phi_2 > 4 \\ \kappa & \text{otherwise} \end{cases} \tag{2.41}$$

where w is an inertia weight of the particle, usually randomized every iteration, $r \in \langle 0, 1\rangle$ is a random scalar, ϕ_{local} is *cognitive acceleration coefficient*, ϕ_{global} is *social acceleration coefficient*, $\vec{x}$, $\vec{x}_{\text{local best}}$ and $\vec{x}_{\text{global best}}$ are particle's position, particle's local best solution and swarm's global best solution respectively. m is an algorithm parameter defining number of neighbours to be used for particle's velocity update. Parameter χ is called *constriction coefficient* and $\kappa \in \langle 0, 1\rangle$ is an algorithm parameter.

Note that while the original version of PSO used only one particle from the swarm to update the velocity of the whole swarm (the one with the global best solution as its local best), the fully informed version uses m other particles to modify speed of each of the particles (and thus when m is equal to the population size, the particle would be fully informed of the search state). Pseudocode of PSO with k particles is given in Listing 2.8.

Algorithm 2.8 Particle Swarm Optimization algorithm.

$min := +\infty$
for $i = 1$ to k **do**
  $\vec{x}_i :=$ rand()
  $\vec{x}_{i_{\text{local best}}} := \vec{x}_i$
  $\vec{v}_i :=$ rand()
  if $f(\vec{x}_i) < min$ **then**
    $\vec{x}_{\text{global best}} := \vec{x}_i$
    $min := f(\vec{x}_i)$
  end if
end for
repeat
  for $i = 1$ to k **do**
    $\vec{v}_i := ...$
    $\vec{x}_i := \vec{x}_i + \vec{v}_i$
    if $f(\vec{x}_{i_{\text{local best}}}) > f(\vec{x}_i)$ **then**
      $\vec{x}_{i_{\text{local best}}} := \vec{x}_i$
      if $f(\vec{x}_{\text{global best}}) > f(\vec{x}_i)$ **then**
        $\vec{x}_{\text{global best}} := \vec{x}_i$
      end if
    end if
  end for
until found

2.3.3 Genetic algorithms

Genetic algorithms (GA) are perhaps the most famous optimization methods inspired by nature. This particular class of optimization methods is inspired by evolutionary biology and as such it mimics the natural evolution of species through time by implementing mutation, crossover, selection and inheritance over members of a population [38]. The idea is to apply these operators on the population iteratively and after certain amount of steps (i.e. generations) the final population will represent the best solution candidates. This is ensured by so called *fitness function* which tells us how good a particular solution is. During the selection of parent individuals for reproduction (crossover and mutation) the fitter a candidate solution is the more likely it will be selected (analogy to the biology: only the strongest individuals survive). In our case the fitness function is the function value of the objective function we minimize, thus we seek lower values of the fitness function.

When using genetic algorithms one usually needs to decide on how to represent an individual so it could be evaluated by the fitness function, mutated and crossed over with another individual yielding a new *offspring* individual. This is usually done by encoding all the information about the individual into a binary string, creating phenotype suitable for mentioned operations. For detail please refer to [39].

Basic framework defining genetic algorithm is drafted in Listing 2.9. First we need to select two parents from the population based on their fitness. Then we cross them over (usually with some random element introduced) and get two *children*. These children are then with predefined probability mutated (e.g. switching one bit of their phenotype) and finally we decide whether we transfer these new solution candidates to the new generation or whether we keep the parents.

Algorithm 2.9 Genetic algorithm.

```
Initialize population of k individuals
repeat
  for i = 1 to k/2 do
    [p1, p2] := Selection()
    [c1, c2] := Crossover(p1, p2)
    [c1', c2'] := Mutate(c1, c2)
    if fitness(c1') > fitness(p1) then
      Replace p1 with c1'
    end if
    if fitness(c2') > fitness(p2) then
      Replace p2 with c2'
    end if
  end for
until found OR max # of generations reached
```

2.3.3.1 Differential Evolution (DE)

The Differential evolution is a special variation of the genetic algorithm which uses different formula for reproduction (incorporating the crossover and mutation in one place) [40], [41]. The basic formula follows:

$$\vec{x}_{i_{\text{new}}} = \vec{x}_1 + c_m(\vec{x}_2 - \vec{x}_3), i \in 1, ..., k \tag{2.42}$$

where k is the population size $c_m \in \langle 0, 1\rangle$ is a parameter controlling the amplitude of the differential and vectors $\vec{x}_1$, $\vec{x}_2$ and $\vec{x}_3$ are three different previously randomly selected parents. The only use of the fitness function is when deciding whether we keep $\vec{x}_{i_{\text{old}}}$ or whether it gets replaced by $\vec{x}_{i_{\text{new}}}$.

To increase a diversity of the population, the crossover scheme is defined as using Equation 2.42 by dimensions and under a condition that $r \leqq c_{cr}$, where r is a random scalar and c_{cr} is defined as a cross over coefficient (or probability). Using this condition we ensure that some dimensions remain unchanged from the $\vec{x}_{i_{\text{old}}}$ and some will mutate according to Equation 2.42.

2.3.3.2 Simplified Atavistic Differential Evolution (SADE)

This algorithm was proposed as an extension of the DE algorithm and combines features of the DE with those of the traditional genetic algorithm. It uses similar differential operator to the

one used in DE (Equation 2.42) and adds several improvements to avoid premature convergence [42]. The operators used are

- **mutation** – instead of three randomly selected parents there is only one parent $\vec{x}_{i_{\text{old}}}$ and a randomly generated vector $\vec{x}_{i_{\text{random}}}$:

$$\vec{x}_{i_{\text{new}}} = \vec{x}_{i_{\text{old}}} + mr(\vec{x}_{\text{random}} - \vec{x}_{i_{\text{old}}}), i \in 1, ..., k \tag{2.43}$$

  mr is called *mutation rate*.

- **local mutation** – all coordinates of a given point are altered by a random value from a given (usually very small) range.

- **crossover** – see Equation 2.42

- **selection** – authors proposed a tournament selection, but other selection methods can be used as well.

Another innovation this algorithm uses is called *radiation fields*. Whenever a local minimum is encountered new radiation field is created in this area. These fields then excite higher mutation probability resulting in heavier exploration of such an area. The size of the field gradually decreases.

2.3.3.3 Population-Based Incremental Learning (PBIL)

Population-Based incremental learning is a method combining the mechanisms of a generational genetic algorithm with simple competitive learning [43]. Individuals are encoded as a binary strings (see Section 2.3.1.3) and new members of population are generated using a vector of probabilities for individual bits. This vector makes basis of the method and it is this vector that is iteratively modified according to minimal and maximal fitness values in generated population. Initially the probability vector elements are set to 0.5 meaning random initial sampling. The genetic algorithm operations are thus applied on the probability vector instead on the population itself.

Although it might seem so, the probability vector does not represent the entire population but tries to represent a point with the highest fitness value (lowest function value) instead. In that way the generated population is bound to explore surroundings of such a point. Several extensions were proposed by author including updating the probability vector based upon not the best one but rather upon several best individuals, which would prove useful when working with large population since we would not ignore the work spent on generating other individuals. Another improvement implemented is to move the probability vector away from the worst individual in addition to moving it towards the (several) best one. Empirical analysis of the two suggested extension can be found in [43] and a different approach in using the PBIL in continuous domain without encoding the variables is presented in [44].

Pseudocode of the PBIL used in our implementation is listed in Listing 2.10. $\vec{p}$ is the probability vector, k is the population size, e is encoding length (number of bits), c_{learn} is learning coefficient, p_{mutate} is the mutation probability and c_{mutate} is the mutation coefficient.

2.3.4 Hybrid of the GA and the PSO (HGAPSO)

We conclude this chapter with an algorithm created as a combination of two others already presented. The Hybrid of the GA and the PSO algorithm is (as the name suggests) a mixture

Algorithm 2.10 Population-Based Incremental Learning algorithm.

$\vec{p} = (0.5, ..., 0.5)$
repeat
  $min := +\infty$
  for $i = 1$ to k **do**
    for $bit = 1$ to e **do**
      $\vec{x}_i^{(bit)} := (random() < \vec{p}^{(bit)} \ ? \ 1 \ : \ 0)$
    end for
    if $f(\vec{x}_i) < min$ **then**
      $\vec{x}_{\min} := \vec{x}_i$
    end if
  end for
  for $bit = 1$ to e **do**
    $\vec{p}^{(bit)} := \vec{p}^{(bit)}(1 - c_{\text{learn}}) + \vec{x}_{\min}^{(bit)} c_{\text{learn}}$
    if $random() < p_{\text{mutate}}$ **then**
      $\vec{p}^{(bit)} := \vec{p}^{(bit)}(1 - c_{\text{mutate}}) + random(0 \ or \ 1) c_{\text{mutate}}$
    end if
  end for
until found OR max # of generations reached

of the genetic algorithm and the particle swarm optimization method. In HGAPSO, GA and PSO both work with the same population and the individuals are regarded as chromosomes in terms of GA, and as particles in terms of PSO [45]. The difference between GA and PSO is how they treat population of individuals. In GA, we iterate from generation to generation creating new populations and forgetting the old, which results in evolving populations. On the other hand, PSO tries to improve one population of individuals (thus of the same generation). This might be regarded as ageing or getting experienced. To combine these, HGAPSO creates a population in the way the GA does and then lets it *age* in the way the PSO does. Repeat this and the HGAPSO is created. Basically it is switching between GA and PSO.

HGAPSO defines three operations:

- **enhancement** – when using elitism in GA, the elites are usually transferred directly to the next generation. HGAPSO uses elitism differently: individuals marked as elite are first enhanced and only after that they are transferred to the next generation. The process of enhancing is handled by the PSO (see Listing 2.8) for predefined number of iterations and then the new generation is completed by individuals created by following operations;

- **crossover** – parents for crossover operation (see Section 2.3.3) are selected from the already enhanced elites by a tournament selection (again we can use other methods for selection).

- **mutation** – the operation from the classical GA when one of the genes of the newly created offspring is mutated under predefined probability.

By using this form of elitism new generation is expected to perform significantly better than the old one.

Chapter 3

Benchmarking of optimization methods

In this chapter we describe selected benchmark functions we used while observing algorithms' behaviour. These functions were selected in order to represent a wide variety of multivariate optimization tasks. Many of these are standard benchmark functions used in testing and comparison of methods for continuous optimization. We hope that besides experimental results this set of benchmark functions will help future researchers in the goal of developing advanced algorithms as these functions were implemented into the JCool environment (for all implemented benchmark functions please refer to Appendix C).

3.1 Goals of benchmarking

During the experiments with implemented optimization methods we aimed to describe the methods in terms of

- rate of convergence on different multivariate functions,
- global and local convergence capabilities,
- number of algorithm iterations taken and
- response to different parameter settings.

For these tasks we selected benchmark functions representing classes of both unimodal and multimodal functions, low and high dimensional functions and functions with scarce and high number of local extremes. As said before, all the benchmark functions are presented as minimization tasks.

3.2 Selected test functions

Brief description of the selected test functions follows – type of the function together with its formula and a visualisation of its function values if appropriate, bounds on input variables if needed and a global minimum with its coordinates if known are stated in Appendix C.

3.2.1 Unimodal functions

Unimodal functions are class of convex functions containing nice functions that are easily optimized as well as difficult cases forcing the optimization method to drop into a slow convergence region (see Section 2.2.1.1). Nevertheless it is these functions that allow design of efficient optimization algorithms.

3.2.1.1 Beale's function (BE)

Beale's function is a unimodal convex test function with global minimum found at a narrow valley [46]. This usually causes trouble to simple optimization methods. Figure C.2a shows a large scale view of the function values which gives the impression of easily optimized function. When examining the area around the global minimum (Figure C.2b), we see that it is hidden in a curved valley. This valley is not easy to find opposite to the Rosenbrock's valley.

3.2.1.2 Booth's function (BO)

Booth's function is a well known unimodal convex test function. The minimum can be found in a wide valley with steep slopes.

3.2.1.3 De Jong's (sphere) function (DJ)

Considered the simplest test function. It is continuous, convex and unimodal [47]. No optimization method should fail on this task.

3.2.1.4 Easom's function (EA)

Easom's function is a unimodal test function with the global minimum taking a very small area compared to the search space [48]. This makes it difficult to optimize for methods using gradient or depending on just small function value region. It is used to show that population based algorithms are much more suitable for such a search.

Figure C.7a shows how small the minimum is compared to the usual start bounds of optimization methods. Figure C.7b illustrates that the area around minimum is convex and well approximable by quadratic forms, therefore when this area is reached, it should not be difficult to find the solution. Finding this area is why the Easom's function is classified as a difficult task.

3.2.1.5 Matyas' function (MA)

This function is deceptive as it seems to be a simple task and yet it causes problems to many optimization methods as it is very flat and stretched near the equiline $x_1 = x_2$. Figure C.16 shows rotated function surface around this equiline.

3.2.1.6 Rosenbrock's valley (RO)

Also called the Banana function, this classic optimization task is unimodal for dimension $d < 3$ and its global minimum is hidden in a long, narrow parabolic shaped valley [49], [50]. This valley is relatively easy to find, but the core of the problem lies in the steps taken then. This is

why it is used very often as a benchmark task – convergence is usually very slow in this valley and some optimization methods might get caught in the Stiefel's cage (see Section 2.2.1.1).

Figure C.21a depicts general overview of the function surface, Figure C.21b gives us the idea of the valley hiding the minimum.

3.2.1.7 Trid function (TR)

Trid function is convex, quadratic and unimodal with a unique global minimizer and a tridiagonal Hessian [51]. For increasing dimension we get a benchmark function testing the efficiency of finding global minimum in the terms of number of function evaluations because most local methods only need $\mathcal{O}(d^2)$ function evaluations or only $\mathcal{O}(d)$ function and gradient evaluations. Important characteristic of this function is a strong coupling between the variables.

3.2.1.8 Zakharov's function (ZA)

Multidimensional Zakharov's function is a unimodal test function with the solution at a corner of a wide plain. Figure C.27a and Figure C.27b show large scale overview of the function surface and the area around the minimum respectively, both for $d = 2$ dimensions. The Zakharov's function can be easily modelled by quadratic forms and thus should not represent any significant difficulties for optimization algorithms even for higher dimensions.

3.2.2 Multimodal functions

Multimodal functions might have any amount of local extrema as well as a large amount of global ones. This makes them ultimate benchmark tasks for virtually any optimization method since to find such a global minimum the method is required to be resistant to premature convergence into a local minimum. Real world problems are usually of this class with high number of dimensions.

3.2.2.1 Ackley's path (AC)

Ackley's path is a multimodal test function with numerous local minima [52]. It is used to distinguish between optimization methods using only local information about the function, which will most likely converge into a local minimum, and these using other, more global information. These are expected to find the global minimum without too much trouble.

Figure C.1a looks similar to the Easom's function as the global minimum takes relatively small area compared to the function surface in large scale. The difference is that this function is highly multimodal and the minimum can be hardly reached when using local search. The close up on the global minimum is shown by Figure C.1b. Both figures are for $d = 2$ dimensions.

3.2.2.2 Branin's function (BR)

Branin's function is multimodal convex test function with three global optima [53]. All of these are located near the function bounds (Figure C.5).

3.2.2.3 Goldstein–Price function (GP)

Well defined with several local minima and one global minimum [54]. The function surface is shown by Figure C.8. We can see that the global minimum is located at a wide plateau together with the local minima. This usually leads to taking a wrong direction near the global minimum.

3.2.2.4 Griewangk's function (GR)

Griewangk's function has many regularly placed local minima and thus is highly multimodal [55]. It forces interdependence among the variables with a goal of failure of the methods optimizing each variable independently. It is similar to the Rastrigin's function.

This function is very interesting as we can see from Figure C.9a – this seems to be convex and smooth function, in other words very easy to find global minimum. However as we take a closer look (Figure C.9b) we notice that the function surface is somehow noisy. Final close up of the global minimum (Figure C.9c) reveals the real characteristic of this function – highly multimodal function surface with large number of local minima all around the global one. Local searches are bound to get stuck and are not likely to reach the solution.

3.2.2.5 Himmelblau function (HI)

This function is interesting by having four global optima and no local ones (see Figure C.10).

3.2.2.6 Langerman's function (LA)

A highly multimodal but non symmetrical function [56]. Local optima are randomly distributed implying the optimization method cannot depend on any symmetric characteristic of the optimized function. Figure C.12a shows the area around the global minimum containing three subareas with local minima. Figure C.12b then focuses on the global minimum as it is surrounded by numerous local minima with steep slopes.

3.2.2.7 Levy function no. 3 (L3)

This highly multimodal function has a large number of local minima together with 18 global minima. It is very hard optimization task for numerical methods.

The function is unbounded but usually is optimized only with given small bounds. This area is depicted on Figure C.14a, one of the global minima is detailed on Figure C.14b. For generalized Levy function with n dimensions please see Appendix C.

3.2.2.8 Levy function no. 5 (L5)

Similar to the previous function, highly multimodal with has large number of local minima. Difference is in the unique global minimum surrounded by these local ones, see Figure C.15a and Figure C.15b.

3.2.2.9 Michalewicz's function (MI)

Michalewicz's function is another multimodal benchmark function with $!n$ local minima [57]. Parameter m is used to alter the falling angle of the valleys and thus acts as a factor to the search complexity. For large m the search becomes very difficult as the plateaus takes majority of the function value space and these do not provide too much information on the function behaviour.

Overall characteristic of the function's behaviour can be seen in Figure C.17a. The steepness of the valleys depending on the parameter m is illustrated by Figure C.17b ($m = 10$) and Figure C.17c ($m = 500$). It is obvious that for large values of m the valleys are deeper and steeper, posing serious obstacles for optimization methods. For our benchmarking we used $m = 100$.

3.2.2.10 Rana's function (RN)

This is another very hard multimodal test function with optimum close to the function bounds [58]. Optimization methods are prone to stop at one of many local minima or get trapped in Stiefel's cage while traversing the narrow valleys, which act as a hard-to-get-over obstacles for many optimization methods as they are surrounded by high and steep slopes.

Function surface for $d = 2$ is visualized by Figure C.19a. We can see many sharp edges along numerous valleys and symmetrical and yet randomly distributed local minima. Figure C.19b is a close up on the global minimum near the function bounds. It is hidden in a valley within another valley.

3.2.2.11 Rastrigin's function (RA)

This function is an extension of the function presented in Section 3.2.1.3. It is modulated by added cosine, which results in many evenly distributed local minima [59]. This function is thus highly multimodal.

Rastrigin's function behaves very similarly to Griewangk's function as it seems to be convex and smooth in large scale (Figure C.20a) but taking a closer look we see that this is far from true (Figure C.20b). The global minimum is surrounded by many local ones (Figure C.20c).

3.2.2.12 Shekel's foxholes (SH)

So called Shekel's foxholes is a benchmark function with m local minima close to the unique global minimum, m being a function parameter [56]. As the name suggests, there are several areas containing local minima and in one of these areas there is a global minimum hidden (see Figure C.22a). For optimization algorithm it should not be a problem to find one of these *foxholes* but problematic is either to find the correct one and to avoid a local minima this foxhole contains (Figure C.22b).

3.2.2.13 Shubert's function (SB)

Shubert's function is a multimodal test function with 18 global minima surrounded by several local ones and several local and global maxima. These are equally spaced. Function behaviour is similar to the Levy function no. 3 (see Figure C.23a for full scale and Figure C.23b for detail of one of the global minima).

3.2.2.14 Schwefel's function (SW)

Schwefel's function has a global minimum relatively far away from the best local minimum [60]. Furthermore the global minimum is close to the function bounds (see Figure C.24). This causes optimization methods converge into a wrong function extremes.

3.2.2.15 Whitley's function (WH)

Whitley's function is a combination of steep slope and a multimodal region close to the optimum [58]. Therefore it is hard for an optimization method not to take wrong step into a local minimum. The area of the global minimum itself is located in a wide plateau and is easy to find (see Figure C.26a). It consists of many local minima, some of which are deep and inside narrow alleys as can be seen in Figure C.26b. The global minimum is surrounded by these and that makes this function valuable benchmark candidate.

3.3 Benchmark configuration

The methodology of benchmarking in this thesis is based on the capabilities of the JCool environment. For all the tests we used constant environment parameters, e.g. number of test runs (repeats), random initialization within the same boundaries (or within the boundaries given by test function definition) and a full range of optimization method parameters with constant step increase. Furthermore in order to be able to compare the population based algorithms we aimed to experiment with the same population sizes.

All results in the following chapter are an average of 100 test runs for each particular algorithm parameter setting with a limitation of 2000 algorithm iterations. This number of successive runs should compensate the random initialization to some extent. All relevant implemented parameters were tested in their full recommended range. We recorded number of objective function evaluations, number of (analytic) gradient evaluations, number (analytic) hessian matrix evaluations, average number of iterations needed to reach the optima and percentage of succesful algorithm runs (i.e. how many times the algorithm had found the optimal solution to the given task). Given these results we will be able to observe changes in the convergence depending on the trend of change of the algorithm parameters and to recommend an optimal setting for different typical optimization problems. Finally we will compare the algorithms based on common indexes such as number of function evaluations or iterations needed to reach the optimum.

3.3.1 Parameter range and step

The parameter range for algorithm benchmark was the whole scale of the definition range of particular parameter if constrained. For unconstrained parameters (e.g. population size) we choose to use upper limit of a value generally considered as high, meaning that further increasing of such parameter is not expected to result in any significant change in the rate of convergence or that the effect of the different parameter values shall project within this given range and there would be no need for higher values.

Parameters were left set to their default settings recommended by authors and one by one were changed while the others were kept constant. This is a simplification against testing all combinations of parameter settings which would be very time consuming. Such a task could be considered a continuous optimization itself (although constrained) as we would be

searching the parameter settings space for optimal parameter setting yielding best results for given algorithm on the given function. This is then usually called a meta-optimization. We are aiming for similar goal but optimizing one variable at a time.

The step (increase) of the parameter value was usually a $\frac{1}{20}$ of the whole range. This gave us a solid insight into the parameter dependent behaviour of examined algorithms. For default algorithm parameter values please refer to Appendix B. For the test functions' parameters we decided to set all to their default values.

As for the stopping conditions used to determine when to terminate the algorithm run we used the already implemented `SimpleStopCondition` from JCool. The condition watches the function value for decrease and sets a limit on number of iterations without any significant improvement.

$$2|f(\vec{x}_k) - f(\vec{x}_{k-1})| \leqq t(|f(\vec{x}_k)| + |f(\vec{x}_{k-1})| + \varepsilon) \tag{3.1}$$

If Equation 3.1 holds true, the *stalled* iteration counter is increased as there was no significant improvement in the function value. $\vec{x}_k$ is the point in the k^{th} iteration (or the best point found so far in the case the algorithm works with a set of points). ε is a *machine epsilon*, number representing the difference between 1 and the smallest exactly representable number greater than one. t then denotes a tolerance on the function value change and we used a value of $\sqrt{\varepsilon}$. For values of these constants please see Appendix D. Limit on number of iterations without improvement was set to 100 iterations. Therefore the algorithm might end after 100 iterations if the point would not move at all but these iterations were not taken into account for the average number of iterations indicator (for these are not real algorithm iterations).

3.3.2 Convergence evaluation

We have to decide how to distinguish a *good* convergence from a *poor* one. This will allow us both to evaluate an optimization algorithm's performance on different benchmark functions and to compare individual algorithms with each another. We decided to express the rate of convergence using both average success rate and average iteration count. These indicators allows to compare performance on various test functions as well as to compare individual algorithms among others.

The average success rate states how many times out of all algorithm runs the solution was found. We can not use this measure in exact digits since this indicator would probably yield 0% success rate for an algorithm which would achieve a solution of lets say 10^{-30} instead of 0.0. Therefore a small enough constant $\varepsilon = 10^{-4}$ was chosen as a tolerance to the final solution of an algorithm run as no tested function has the local minima far from the global minimum with function value within this range. The algorithm run is the considered successful when the result $r \leqq$ real solution$+\varepsilon$. This will ensure that we will not discard quite successful algorithms just for the minor numerical imprecision. However we have to be aware that this value will handicap most of the nature inspired methods as these are not as precise as the numerical ones.

The average number of iterations taken is not that straightforward indicator as it might seem. This is caused by the hidden number of function evaluations, gradient and hessian evaluations as these usually differ for each particular algorithm. Therefore this indicator will be used only to provide us with generall idea of how the algorithm performs and then we will have to examine closer the other statistics.

Except for the average number of successful algorithm runs all these values are direct output from the JCool environment. We also guaranteed by using the same line search routines that every algorithm has equal conditions which is a basis to objective mutual comparison.

Chapter 4

Results

This chapter presents experimental results obtained from the JCool environment. The algorithms described in Chapter 2 were benchmarked on performance using the test functions from Chapter 3 under the same conditions. We focused on the relationship of algorithm parameter setting to the rate of convergence on different test functions and on common parameters among different algorithms to compare their effectiveness.

All results were obtained as an average values for 100 algorithm runs with constant parameter setting and each run was limited to 2000 iterations. In this chapter we present only the most significant parameter dependent behaviour of tested algorithms on the benchmark set. The rest of the experiment results is gathered in Appendix D.

For the stopping condition adds its limit to the iteration counter, we subtracted this number from the observed data if the number of iterations was lower than the overall limit of iterations taken (2000). By doing this we get better idea about the speed of convergence for tested algorithms as the number used in stopping criterion is very high for numerical methods but proved to be useful for the methods inspired by nature.

4.1 Observed convergence

Following data illustrate the observed convergence of individual algorithms applied on different benchmark tasks with variable parameter settings. If any algorithm was unable to find the minimum within the given limit of 2000 iterations we will try to discuss the reasons leading to such a situation, focusing on whether it was due to a weak global convergence properties of the algorithm or whether it was due to a mistake in implementation. We will also consider the possibility of roundoff errors and numerical instability.

Optimization algorithms using line search had two variants of Brent's line search method available. One was the standard Brent's method without the use of gradient (derivatives), the other was an extension of Brent's method to use gradient information. The main difference in their performance is in the number of function evaluations. The latter needs considerably less function evaluations, but the former makes no calls to the gradient routines. Furthermore the gradient information is obtained either when the function has defined analytic gradient (and thus there is a computation burden of computing this value) as well as when it has no analytic gradient specified. Then the gradient information is obtained via numerical central difference method, which makes successive calls to the function value. One has to be aware of these limitations of the second method when choosing which one to use.

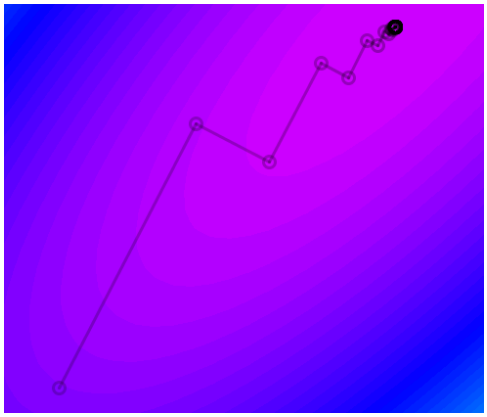


Figure 4.1: Steepest descent applied to Booth function.

	Unimodal							
	BE	BO	DJ	EA	MA	RO	TR	ZA
B	64%	100%	100%	10%	100%	29%	100%	100%
BWD	54%	100%	100%	10%	100%	28%	100%	100%

	Multimodal														
	AC	BR	GP	GR	HI	LA	L3	L5	MI	RN	RA	SH	SB	SW	WH
B	50%	100%	54%	2%	100%	2%	41%	19%	3%	0%	18%	8%	46%	1%	17%
BWD	41%	100%	45%	3%	100%	1%	4%	3%	4%	0%	1%	5%	29%	0%	13%

Table 4.1: Average success rate for SD.

In general when examining behaviour of methods inspired by nature and using populations we will expect that when using higher numbers of points in population better success rate will be observed than when fewer points are used. This is based on the idea of better coverage of the search space, of its better sampling and thus having higher chance for correct interpretation of function behaviour. Therefore the results for changing the population size are collected in Appendix D as well if they do not show any unexpected behaviour.

4.1.1 SD

Convergence properties of the SD method were already discussed: it has no global convergence at all and even if we start to iterate near some local optimum we have no guarantee that the algorithm will find it because the orthogonal movement of the optimized point can cause the algorithm to stop. Otherwise we know that even for simple unimodal functions the method of steepest descent might perform poorly because even when it does not get caught in the Stiefel's cage the convergence is slow due to repeated small steps the algorithm makes. This is illustrated by Figure 4.1 which depictures successful SD run on the Booth function (Section 3.2.1.2). Although the algorithm had found the solution, in the end it needed 53 iterations to do so. As we can see, the closer to the solution the slower convergence was observed. The Stiefel's cage or the excessively small steps are also the reason why would SD ever cross the limit of 2000 iterations. The average number of iterations taken on a given tasks is summarized by Table 4.2 where we can see that e.g. for the famous Rosenbrock's valley the algorithm was converging too slow to get to the optimum.

In generall, SD method is suitable only for unimodal convex functions as can be seen from Table 4.1. These results also suggest that using original Brent's method (without the use of derivatives) as a line search is a good idea. This might be caused by this method being *too* exact and thus taking more precise and smaller steps.

	Unimodal							
	BE	BO	DJ	EA	MA	RO	TR	ZA
B	876	36	1	4	51	1894	9	12
BWD	1157	45	1	3	193	1878	9	11

	Multimodal														
	AC	BR	GP	GR	HI	LA	L3	L5	MI	RN	RA	SH	SB	SW	WH
B	5	32	51	7	23	102	9	9	46	4	4	5	5	3	130
BWD	5	31	30	7	24	59	4	8	13	4	4	5	5	3	137

Table 4.2: Average number of iterations taken by SD.

		Unimodal							
		BE	BO	DJ	EA	MA	RO	TR	ZA
FR	B	66%	100%	76%	5%	96%	98%	100%	78%
PR	B	58%	100%	100%	10%	91%	100%	100%	95%
BSHS	B	65%	99%	100%	4%	100%	100%	100%	99%
FR	BWD	57%	100%	100%	5%	57%	100%	100%	95%
PR	BWD	47%	100%	100%	9%	100%	100%	100%	100%
BSHS	BWD	49%	100%	100%	10%	100%	100%	100%	100%

		Multimodal													
		AC	BR	GP	GR	HI	LA	L3	L5	MI	RN	RA	SH	SB	SW	WH
FR	B	0%	100%	71%	4%	99%	1%	66%	16%	18%	0%	30%	2%	78%	1%	42%
PR	B	53%	100%	70%	6%	100%	1%	52%	10%	6%	0%	8%	6%	51%	0%	10%
BSHS	B	45%	100%	81%	10%	100%	0%	40%	10%	8%	0%	16%	11%	49%	0%	14%
FR	BWD	2%	100%	46%	7%	99%	1%	33%	10%	10%	2%	5%	5%	75%	0%	5%
PR	BWD	42%	100%	56%	10%	100%	1%	5%	3%	9%	0%	1%	7%	38%	1%	0%
BSHS	BWD	65%	98%	38%	6%	98%	0%	49%	4%	6%	9%	3%	4%	45%	3%	3%

Table 4.3: Average success rate for CG.

4.1.2 CG

This method was designed to solve the disadvantages of the SD method, mainly the succesive steps in the same directions. As we can see in Table 4.3 there is a noticable improvement to the overall effectiveness of this algorithm against the SD method. This method attains the *almost*-quadratic convergence especially on unimodal and convex functions as promised which can be seen from the number of iterations taken depicted by Figure 4.2. The only exception for unimodal functions is the Easom's function since even with the gradient information the area around the global minimum is just too flat and the CG takes too small steps. The Easom's function was designed exactly for this purpose, to show that using only first derivatives might lead to excessively slow convergence in some cases.

Multimodal functions still represent a serious problem if their surface changes too rapidly since this method still generally lacks any global convergence properties. It is very effective in solving mentioned tasks though.

One very important fact is to be noted – although generally better on the hard multivariate functions, the Fletcher–Reeves update formula needs except for the simplest functions from 2 to 145 times more iterations to either find the minimum or to finish the algorithm in a local minimum. Due to this the formula might prove unusable for functions with costly evaluation. Additionally there are fluctuations in the formula's success rate: for some functions it seems to be superior to the other two (e.g. Levy no. 3, Rastrigin, Shubert), for other it is inferior (e.g. Ackley).

4.1.3 LM

The LM method aimed for the same goal as the CG did – to combine both SD method with the Newton's method in an effective way so that we enjoy fast convergence and avoid the drawbacks of the latter method. The idea proved correct many times and our case is no different. Looking

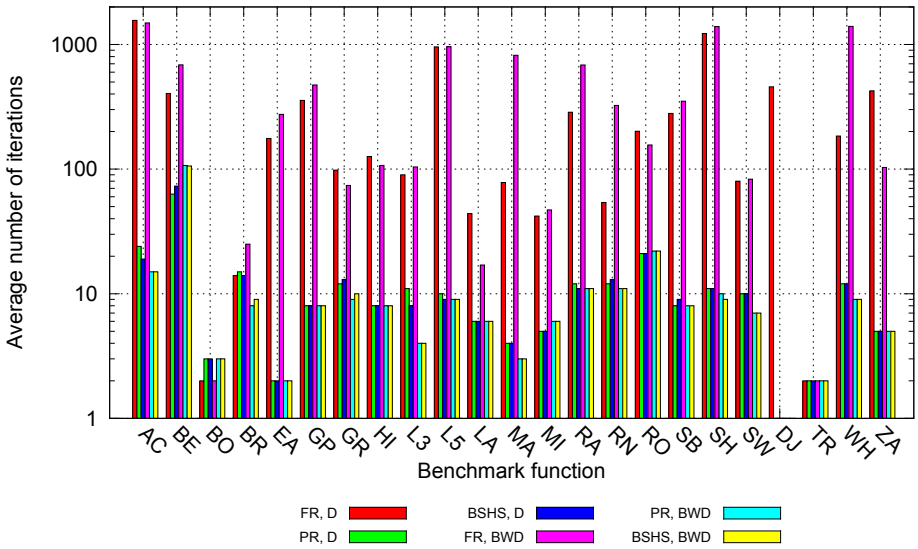


Figure 4.2: Average number of iterations taken by CG.

	Unimodal														
	BE	BO	DJ	EA	MA	RO	TR	ZA							
B	59%	100%	100%	12%	100%	66%	100%	100%							
BWD	63%	100%	100%	14%	100%	83%	100%	100%							
	Multimodal														
	AC	BR	GP	GR	HI	LA	L3	L5	MI	RN	RA	SH	SB	SW	WH
B	6%	100%	49%	7%	100%	0%	3%	2%	2%	0%	0%	4%	5%	0%	2%
BWD	4%	100%	57%	9%	100%	0%	6%	0%	4%	0%	0%	5%	8%	1%	0%

Table 4.4: Average success rate for LM.

at the results (Table 4.4) we see very similar succes rate to the previous method, the CG. The LM method gives us good convergence for unimodal functions with less algorithm iterations (Table 4.5). This comes for the price of one gradient and one hessian evaluation per iteration (in the case no derivatives were used in the line search method). There is no significant difference when we use Brent's method for line search with or without derivatives in the means of success rate or the number of algorithm iterations.

Interesting property of the LM method is the speed of convergence in time – as we can see from Figure 4.3, the algorithm starts with large steps at the beginning and as approaches the solution it returns to the small steps in order not to "step over" the solution. On the left we can see the relative step size taken each iteration and the according function value on the right.

	Unimodal														
	BE	BO	DJ	EA	MA	RO	TR	ZA							
B	1025	103	100	113	144	153	109	105							
BWD	842	103	100	110	132	165	110	105							
	Multimodal														
	AC	BR	GP	GR	HI	LA	L3	L5	MI	RN	RA	SH	SB	SW	WH
B	104	103	103	107	105	131	104	103	106	104	103	104	103	103	104
BWD	252	103	103	108	105	186	381	103	104	107	103	104	103	103	104

Table 4.5: Average number of iterations taken by LM.

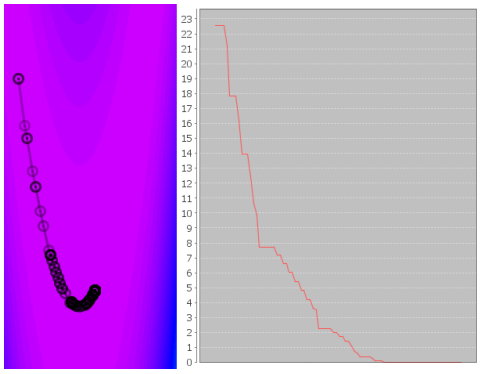


Figure 4.3: Convergence of the LM algorithm on the Rosenbrock's valley.

4.1.4 QN

The QN method is very similar to the CG method; its goal is to accumulate enough information from succesive line searches to have after n such searches in n dimensional quadratic form an exact minimum. Both these methods need the availability of (analytic) gradient routine for arbitrary points. As we can see, performance of QN is therefore quite similar. Again we see that for general smooth functions it performs very well (Table 4.6, for clarity only the values for Broyden family update formula with $\phi = 0.5$ were included; for detailed results please see Appendix D). The number of algorithm iterations (each requiring at least one gradient evaluation) is also low – for nice unimodal convex functions the convergence is near quadratic (Figure 4.4).

The results show another interesting fact, the effectiveness of Broyden's update formula is unstable as was the Fletcher–Reeves formula in CG method. For some functions it seems to perform much better than the others while used in combination with Brent's method using derivatives (Ackley, Goldstein–Price, Levy no. 3, Shubert), for others it does very poorly no matter which line search method we use (mainly the Rosenbrock). Moreover for most of the tested functions it requires the algorithm to take many more steps compared to other update formulas (up to 22 times more in case of Michalewitcz's function). This is known to be the issue with Broyden's method as it is a rank one update formula and it terminates in no less than $2n$ steps where n is the dimension of the quadratic form [61].

Additionally we observed that the line search with the use of derivatives leads to worse results then without it. The price for using standard line search method is higher function evaluations, but opposite to the former method no gradient evaluations. This might be of lesser importance.

4.1.5 OS

The OS method is intended as a very simple optimization method with several drawbacks. One is obvious from the definition of the method, the number of line searches at each iteration is n which makes it very costly in the terms of function evaluations and gradient evaluations in case derivatives were used in the line search method.

The main problem of OS method is illustrated by Table 4.7 where we can see that the narrow valleys cause problems (Beale, Rosenbrock). Corresponding to this we notice that the number of iterations for these functions reached the limit for all of these test functions no matter what variant we used (either normal of stochastic search, see Table 4.8). For multimodal functions this methods performs poorly, although it stops after just a few steps as it has nowhere else to

		Unimodal							
		BE	BO	DJ	EA	MA	RO	TR	ZA
BRO	B	64%	100%	90%	13%	24%	70%	100%	10%
DFP	B	59%	100%	97%	6%	93%	100%	100%	82%
BFGS	B	60%	100%	93%	7%	96%	100%	100%	81%
BF	B	66%	100%	94%	5%	97%	100%	100%	81%
BRO	BWD	48%	100%	100%	6%	20%	71%	100%	45%
DFP	BWD	60%	100%	100%	3%	98%	100%	100%	90%
BFGS	BWD	43%	100%	100%	6%	99%	100%	100%	88%
BF	BWD	53%	100%	100%	12%	98%	100%	100%	94%

		Multimodal													
		AC	BR	GP	GR	HI	LA	L3	L5	MI	RN	RA	SH	SB	SW	WH
BRO	B	36%	100%	56%	8%	100%	1%	42%	7%	7%	8%	17%	5%	49%	2%	7%
DFP	B	21%	100%	59%	5%	100%	0%	29%	9%	4%	5%	14%	6%	43%	0%	5%
BFGS	B	30%	100%	61%	5%	100%	1%	30%	8%	8%	1%	5%	10%	40%	2%	6%
BF	B	30%	100%	53%	9%	100%	1%	24%	12%	5%	3%	9%	7%	41%	0%	3%
BRO	BWD	29%	100%	46%	1%	100%	2%	13%	2%	4%	2%	0%	3%	31%	6%	1%
DFP	BWD	15%	100%	43%	1%	100%	2%	8%	1%	6%	2%	0%	2%	29%	0%	1%
BFGS	BWD	29%	100%	47%	0%	100%	3%	8%	0%	5%	0%	0%	10%	31%	2%	4%
BF	BWD	33%	100%	39%	1%	100%	0%	7%	1%	5%	2%	0%	4%	35%	2%	0%

Table 4.6: Average success rate for QN.

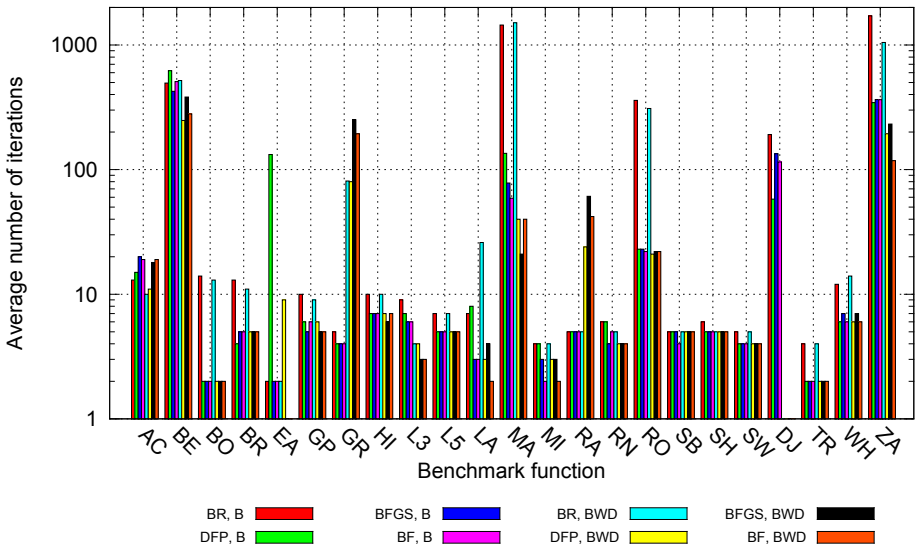


Figure 4.4: Average number of iterations taken by QN.

go.

In general the use of stochastic search (i.e. random order of dimensions to search) leads to higher number of algorithm iterations. This is caused by the fact that previous searches are not connected in any way to the present one, which projected also in the average success rate.

		Unimodal							
		BE	BO	DJ	EA	MA	RO	TR	ZA
	B	38%	100%	100%	3%	100%	73%	100%	100%
Stochastic	B	44%	100%	100%	3%	100%	32%	100%	100%
	BWD	36%	100%	100%	6%	100%	69%	100%	100%
Stochastic	BWD	45%	100%	100%	4%	100%	39%	100%	100%

		Multimodal													
		AC	BR	GP	GR	HI	LA	L3	L5	MI	RN	RA	SH	SB	SW	WH
	B	96%	100%	33%	7%	100%	5%	9%	7%	34%	0%	30%	8%	47%	1%	40%
Stochastic	B	93%	100%	42%	13%	100%	3%	17%	5%	29%	0%	36%	4%	42%	0%	36%
	BWD	94%	100%	36%	9%	100%	3%	0%	5%	19%	0%	28%	9%	39%	1%	19%
Stochastic	BWD	93%	100%	39%	4%	100%	1%	4%	5%	30%	1%	32%	4%	45%	0%	24%

Table 4.7: Average success rate for OS.

		Unimodal							
		BE	BO	DJ	EA	MA	RO	TR	ZA
	B	1253	55	1	1	147	2000	7	13
Stochastic	B	1174	74	1	1	197	2000	9	16
	BWD	1284	56	1	1	147	2000	8	13
Stochastic	BWD	1151	74	1	1	196	2000	10	16

		Multimodal														
		AC	BR	GP	GR	HI	LA	L3	L5	MI	RN	RA	SH	SB	SW	WH
	B	3	8	98	2	8	23	1	2	1	1	3	2	2	1	44
Stochastic	B	3	12	103	2	11	39	1	2	1	1	3	3	2	1	53
	BWD	3	8	84	2	8	97	1	2	1	1	4	2	2	1	61
Stochastic	BWD	4	12	107	2	11	31	2	2	1	1	4	3	2	1	72

Table 4.8: Average number of iterations taken by OS.

	Unimodal							
	BE	BO	DJ	EA	MA	RO	TR	ZA
B	40%	100%	100%	24%	100%	100%	100%	100%
BWD	35%	100%	100%	23%	100%	100%	100%	100%

	Multimodal														
	AC	BR	GP	GR	HI	LA	L3	L5	MI	RN	RA	SH	SB	SW	WH
B	80%	100%	37%	9%	100%	4%	17%	4%	25%	1%	19%	6%	42%	0%	24%
BWD	80%	100%	42%	6%	100%	2%	4%	2%	24%	0%	22%	10%	41%	2%	13%

Table 4.9: Average success rate for PM.

Particularly good example is the Rosenbrock's valley where the stochastic search was unable to find the solution whereas the normal search did so in about 60% of algorithm runs. The random search order actually lead to the algorithm step being caught in the Stiefel's cage as we observed during the experiments since the steps were absolutely unrelated to the previous ones and were smaller and smaller.

4.1.6 PM

The Powell's method was expected to cure some of the problems the OS methods was unable to deal with. The only improvement we observed was on the Rosenbrock's valley benchmark. The PM was able to find the solution every time (Table 4.9) and in general the algorithm took much less iterations to terminate (Table 4.10). Otherwise the results are very similar to the OS method. The line search without derivatives led to better results which we explain by this search being just too precise to be usable in these methods as it leads to smaller steps.

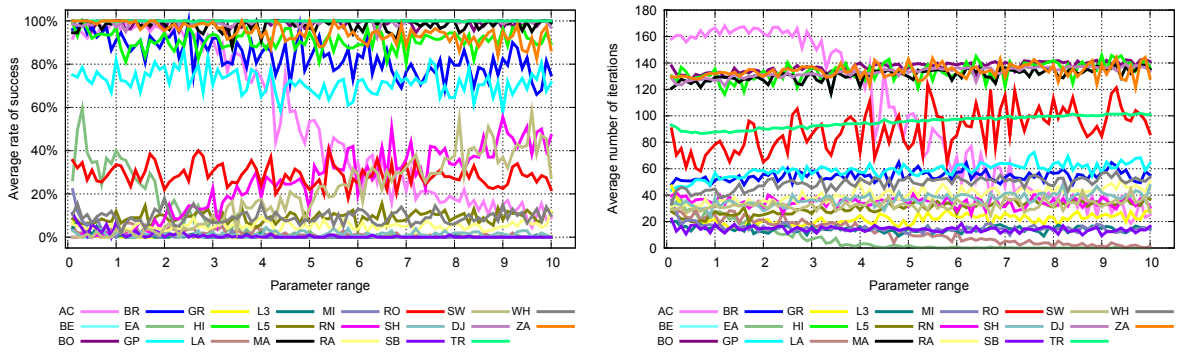
4.1.7 CMA-ES

CME-ES method is the last one considered numerical although it also works with the term of population. Figure 4.5a shows average success rate of CMA-ES when applied to different test functions for different initial setting of the parameter σ.

	Unimodal							
	BE	BO	DJ	EA	MA	RO	TR	ZA
B	1022	3	1	1	3	24	2	5
BWD	989	3	1	2	3	25	2	5

	Multimodal														
	AC	BR	GP	GR	HI	LA	L3	L5	MI	RN	RA	SH	SB	SW	WH
B	3	3	4	2	4	6	1	2	1	1	2	2	2	1	5
BWD	4	3	4	2	4	6	2	2	1	1	3	3	2	1	5

Table 4.10: Average number of iterations taken by PM.



(a) Average success rate for CMA-ES.

(b) Average number of iterations taken by CMA-ES.

Figure 4.5: CMA-ES results for $\sigma = 0.1, ..., 10.0$, step $= 0.1$.

This revealed some very interesting relations between this parameter, optimized function and achieved success rate. In general for convex, unimodal or otherwise *simple* functions we did not recognize any significant influence of this parameter to the results. However when testing special cases, most notably the Easom's function and the Ackley's path, we observed rapid deterioration of the rate of convergence. This is due to the fact that initial value of σ tells the algorithm how far away – how much scattered – the initial population should be. When applied to these functions, the population did not represent or actually did not cover the function value space well enough to allow the algorithm take the right step, in this case to adjust the mean and the standard deviation (σ) correctly since the population from which these updates come and which is actually a sampling of this function value space missed the important features of the function behaviour because these are too much locally oriented.

On the other side, exactly opposite effect of the initial value of σ was observed on the extremely hard test functions which proved to be very resistant to any optimization attempt so far: the Rana's function and the Schwefel's function. The average success rate rapidly increased as the parameter σ did. This is due to exactly the same reason as with the prior functions but reversed. These functions have impassable features mainly of sharp slopes and deep local minima when examined in small scale, but has rather smooth characteristics when examined in large scale. This makes it possible for a scattered sampled representation of the function value space to span the right direction for optimization resulting in correct update of the mean and standard deviation.

The number of iterations the algorithm needed to terminate also seems to be related to the initial setting of parameter σ (Figure 4.5b). Except for the first mentioned special cases (Easom, Ackley) and not very successful Langreman's function the number of iterations steadily increased. However this was in units and does not seem to need further examination.

4.1.8 CACO

Parameter values used for the following ant algorithms are these recommended by Oleg Kovářík in his thesis [27]. These were tuned to yield good results on several benchmark data in the FAKE GAME software and are expected to yield good results this time as well. Values used might therefore differ from the original ones recommended by the authors.

CACO algorithm is unfortunately also the first and only algorithm that we tested unsuccessfully. No matter how we set the parameters the success rate did not change significantly even when we increased the iteration limit to 10 000 iterations and did not use the stopping

	Unimodal							
	BE	BO	DJ	EA	MA	RO	TR	ZA
CACO	0%	1%	1%	1%	1%	0%	1%	1%
Random	3%	1%	1%	1%	9%	0%	6%	2%

	Multimodal														
	AC	BR	GP	GR	HI	LA	L3	L5	MI	RN	RA	SH	SB	SW	WH
CACO	1%	1%	1%	1%	1%	1%	1%	0%	3%	0%	1%	2%	2%	0%	1%
Random	0%	0%	0%	4%	0%	0%	37%	1%	49%	0%	0%	6%	33%	0%	1%

Table 4.11: Comparison of average success rate of CACO method and Random search.

conditions. The only exception was the evaporation factor which for higher values gave better results, although in general the CACO algorithm was very unsuccessful compared to other algorithms. Uncommented results are included in section D.1 for convenience only.

Table 4.11 compares the CACO algorithm with a completely random search results using same conditions (10 000 iterations limit, no stopping criterion). Random search is used as a reference to the worst possible scenario as it uses no information about the optimized function. Unfortunately it proved to be true that CACO algorithm gives worse results than this method and thus two options emerge: either the default parameter values are very far from the optimum range and our benchmarking was not conducted on values close enough to these ranges or there is an error in implementation. The first suspect was the second option because the current implementation places the nest to zero coordinates (axis center). Many tested functions have optimum exactly at that point, however the search vectors are scattered all around and it takes longer to move them close enough. Even then the accuracy would be poor and thus the success rate would reflect that with very low percentage. However, poor results of CACO were observed even for unimodal convex functions with the minimum displaced to other point and thus it seems that if the implementation is incorrect this is not the reason. We were unable to find any other difference between the papers and current JCool implementation.

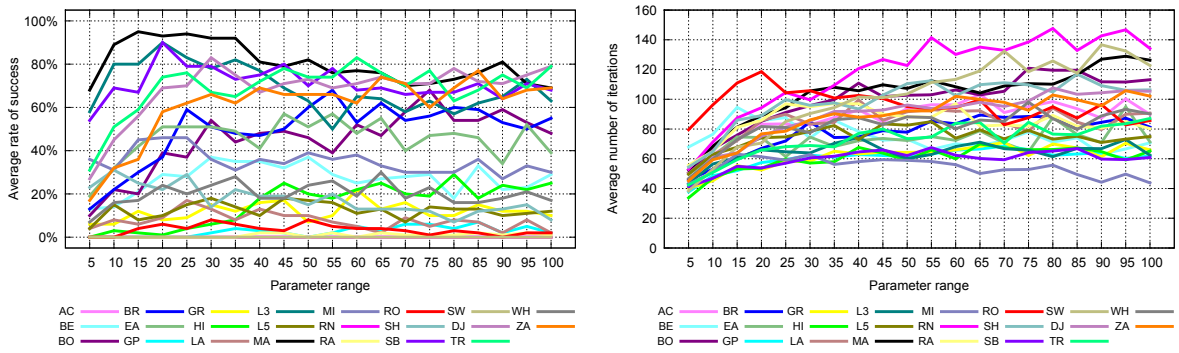
4.1.9 API

API algorithm uses four parameters set at the initialization stage. We conducted test on all of them one by one, each time keeping all but one constant. Recorded observations follow:

- **population size** – both Figure D.9a and Figure D.9b conform to the expectation that higher number of ants used is better. Number of iterations needed to terminate is slowly decreasing for all functions. This is due to more detailed search in the vicinity of the nest and thus discovering promising areas sooner.

- **nest moving** – this parameter seems to be optimal in the default setting (Figure 4.6a). For most of the functions the API yields worse success rate for lower values. After the value of 30 iterations success rate stabilizes and remains constant. Lower values lead to faster convergence but might end in a local minimum. On the other side, if the nest is moved after a large number of iterations the algorithm might end on stopping criterion by being unable to reach the solution fast enough.

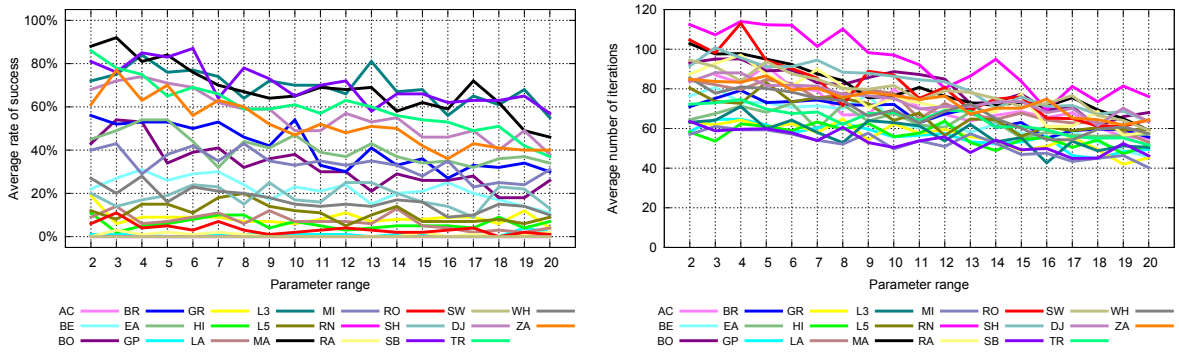
- **number of hunting sites** – This time the default setting seems to give optimal results also as can be seen in Figure 4.7a. This is due to actually searching the space with less stress on the good areas because when we use fewer hunting sites, we will replace them faster in case of unsuccessful searches. However when having a lot of these we will simply move to another (already existing) and try this one instead. There is no guarantee that this will be a better one and if it proves not to be, we continue to a next one. In this way



(a) Average success rate for API.

(b) Average number of iterations taken by API.

Figure 4.6: API results for moving the nest to the best point every 5, ..., 100 iterations, step = 5.



(a) Average success rate for API.

(b) Average number of iterations taken by API.

Figure 4.7: API results for hunting sites count = 2, ..., 20, step = 1.

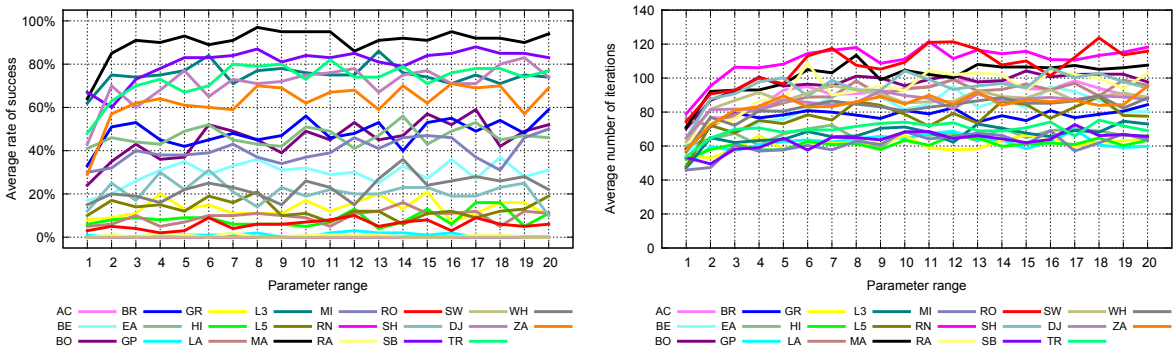
it takes longer to replace unpromising hunting sites and therefore to discover a promising area.

Iteration count (Figure 4.7b) decreases also slowly, unfortunately in the same way so that for the value of 3 hunting sites per ant the iteration steps taken by algorithm represent second highest value for every function tested.

- **starvation** – In order to reflect the success rate of API when applied on multimodal functions (Figure 4.8a) we would increase the number of allowed unsuccessful iterations before removing a hunting site to no less then 6 iterations for with this value API gives on most of the functions the best results with only few exceptions having slightly better success rate with even higher values. This is a result of more detailed search of the area assigned to each ant and therefore we have higher chance of discovering the best value within these bounds. The reason why we might want to avoid higher values is shown on Figure 4.8b where we can see slightly uphill trend in the number of iterations taken by algorithm.

4.1.10 AACA

Five parameters were examined:



(a) Average success rate for API.

(b) Average number of iterations taken by API.

Figure 4.8: API results for starvation = 1, ..., 20 iterations, step = 1.

- **population size** – Figure D.10a and Figure D.10b summarize results observed for variable population size. It is not unexpected to see the average rate of success increase as the size of population increases since the more ants we use the better we cover the function value surface and thus we are more likely to come across the solution.

  However for the initial experiment setting there was one interesting exception, the Zakharov's function. The success rate seemed to decrease steadily as the population size was increased. Closer examination revealed that this particular function causes the AACA algorithm to perform a *bit swap* due to the functions bounds because these are used for bit string decoding. Whenever the algorithm found the minimum $\vec{x}_{\text{MIN}} = (0, 0)$ for $d = 2$ dimensions the decoding process together with pheromone assignment strategy caused certain higher bits to be swapped and thus shifting all points relatively far away from the solution. Figure 4.9 illustrates this situation. The cross marks the global optimum and as we can see in the upper left part, the global optimum has already been found. However, in this exact moment the bit swap occurred and the whole population of solution candidates was shifted (these are marked by circles). The right part of the figure shows historical best value in the population in the given iteration. We can clearly see the jump to worse values right when the value reached global optimum, that is a value of 0. Following this we modified the algorithm to include the best value-point found so far into the telemetry sent through the JCool framework to ensure that this behaviour would not affect the results and so we can consider even this jump to worse value as a success in the end since the algorithm clearly did found the solution.

  The reason why smaller population size was not affected that much is easily explained by the fact that with fewer ants the algorithm was unable to find the minimum exactly (for convenience lets say that the point coordinates were $\pm\varepsilon$ from 0) and thus the bit swap did not occur. This might present a challenge for future researchers as the AACA method was developed to overcome exactly such a situation by employing a pheromone update formula taking into account bit positions. Our solution is of rather temporary character.

  The iteration count the AACA needed to terminate seemed to be decreasing in general with one exception, the Schwefel's function. This function is tricky in leading algorithms in the wrong directions, away from the solution as it is geometrically distant from surrounding local minima and furthermore it is close to the function bounds.

- **encoding length** – except for a significant change in both success rate and iteration count (Figure 4.10a and Figure 4.10b) when increased from 8 bits to 16 bits there seemed to be no other influence of this parameter. This is partly against expectation as it is natural

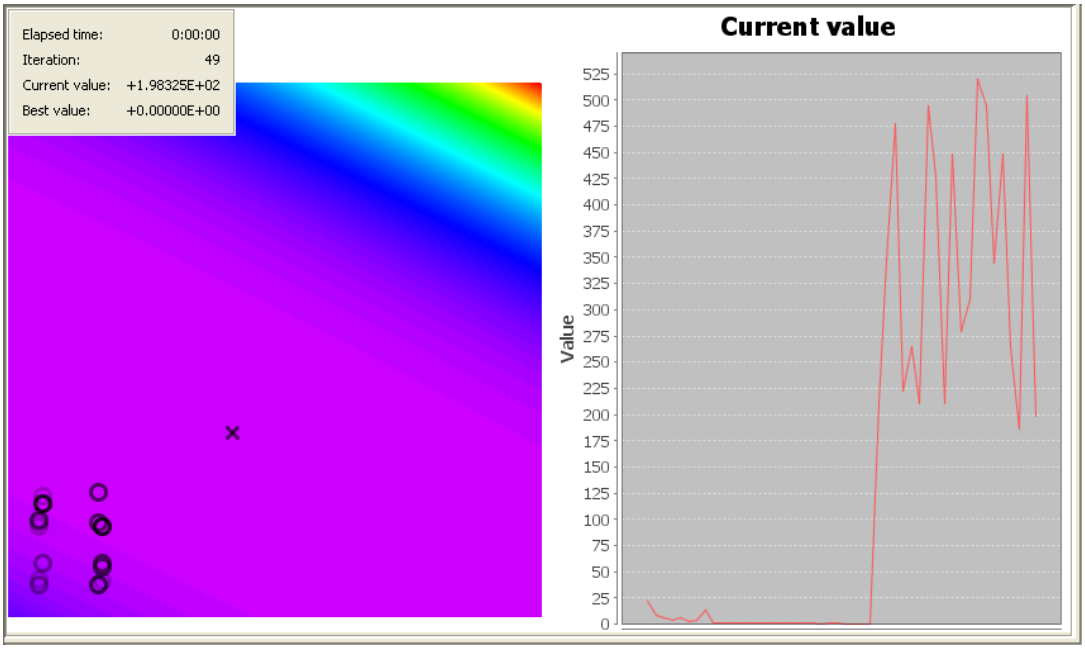


Figure 4.9: AACA: occurence of bit swap.

that with more bits available to encode a fractional number we get better precision and thus more algorithm runs should finish within the tolerance for a solution.

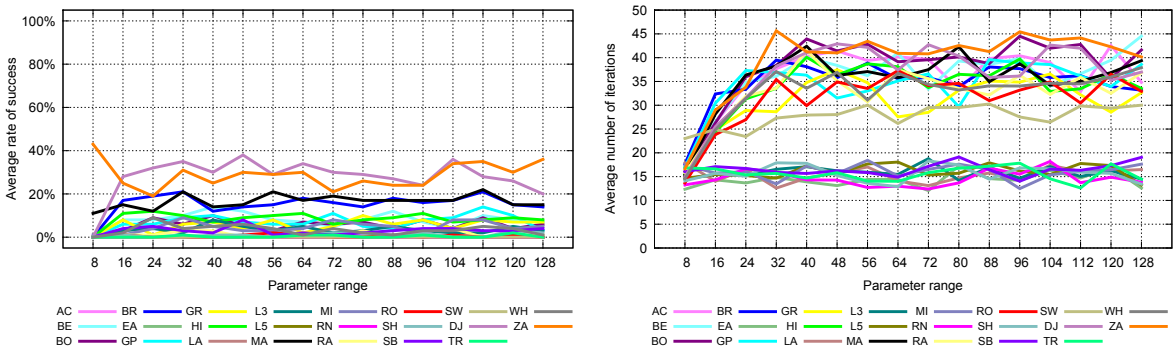
- **pheromone index** – According to Figure D.11a and Figure D.11b this parameter seems to have very little impact when examined under the default settings as both the average succes rate and iteration count remained constant within reasonable tolerance.

- **cost index** – this parameter has similar impact as the previous one. Both Figure D.12a and Figure D.12b give no clue on whether there is any significant parameter dependent behaviour of the AACA algorithm. The reason behind this and behind results for previous parameter might be that the default setting of other parameters does not give any room for improvement or change at all.

- **evaporation factor** – interesting behaviour can be observed when examining this last parameter. The success rate (Figure 4.11a) was increasing for most of the tested functions faster and faster up to the value of 0.9. Then followed an abrupt decrease as the parameter value approached its upper bound. This decrease is logical as for the maximum value there is no pheromone information used since it gets all evaporated. Unfortunately similar trends are recorded for the iteration count (Figure 4.11b). Up to the point of 0.95 it steadily increases and for the upper bound there is a decrease. We can assume that the decrease occurs close to the extreme of the definition range of this parameter. Although it increases up to almost 7 times for some functions, the iteration count is still quite low (in this case a maximum of 39 iterations in average was reached). This is due to the fact that the algorithm is encountering more candidate solutions and therefore it spends more time to terminate yielding in better results.

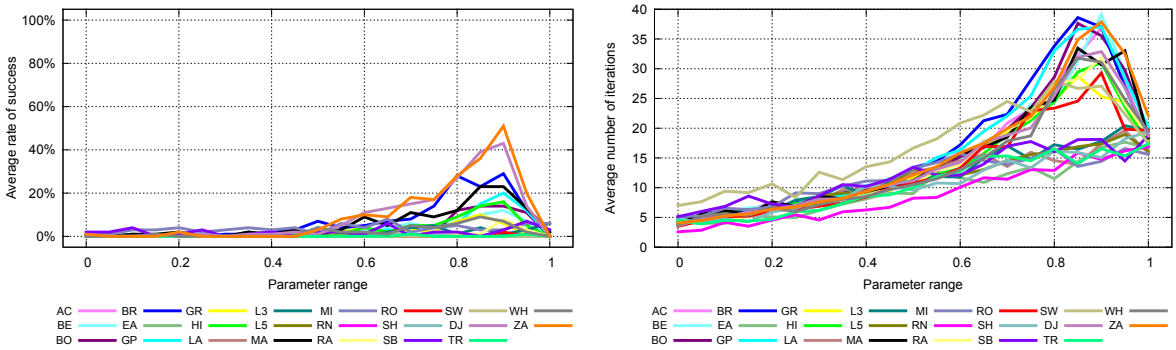
4.1.11 ACO*

ACO* was tested for all parameters:



(a) Average success rate for AACA. (b) Average number of iterations taken by AACA.

Figure 4.10: AACA results for encoding length $= 8, ..., 128$, step $= 8$.



(a) Average success rate for AACA. (b) Average number of iterations taken by AACA.

Figure 4.11: AACA results for evaporation factor $= 0.0, ..., 1.0$, step $= 0.05$.

- **population size** – according to Figure 4.12a the optimal value would be around 15 ants for the most of the functions as there are two groups of functions, one group is optimized with great success rate with large number of ants while the second is giving worse results with increasing size of population. The first group consists of all the unimodal functions and three multimodal functions (Ackley, Branin and Goldstein–Price). These are relatively simple since the global characteristics of the function point to the solution.

  The second group embodying worsening with more ants in the population are exclusively multimodal functions. For these lower values of the parameter should be used. The cause lies in the fact that too many ants (i.e. too many probability distributions) lead to covering most of the function value surface and thus giving paradoxically less information on the global characteristic of the function behaviour then when sampling the surface by fewer distributions. To avoid such a situation we might use a more complex weight distribution to distinguish more clearly between promising areas and those not worth exploring.

  The number of iterations increases for value of 10 ants, decreases next and then steadily rises for the rest of the tested parameter values. The only exception is the Rosenbrock's valley for which the iterations taken create a hill with peak between 35 and 50 ants. This range also gives the best success rate for this function; the success rate actually copies the hill shape of the iteration count (Figure 4.12b).

- **# of ants replaced each iteration** – from Figure 4.13a we can clearly see that the

more ants we replace each iteration the better results ACO* gives. As for iterations these are increasing as the success rate increases and start to decrease slowly after the success rate reaches its peak (Figure 4.13b). For unimodal functions this peak comes even when using lower values such as 10 or 15 ants.

- **deviation parameter** ω – According to Figure 4.14a the optimal value for this parameter would be between 0.05 and 0.10 because for the most of the functions ACO* gives best results within this range. In general for multimodal functions the success rate starts to decrease after this point. Number of iterations is very slowly increasing except for the same functions as before: Rosenbrock's valley and Beale's function (Figure 4.14b). We assume that this confirms ACO* having troubles when it comes to optimizing functions with global minimum in long narrow valleys (for both these functions have them) since it is difficult to cover such a valley with probability distribution – we need to set the mean value to the middle of the valley and small deviation to reflect the valley's width. However with small deviation we will take only a small step away from the mean and thus improving only a little. This leads to a higher number of iterations needed to terminate the algorithm.

- **convergence parameter** σ – this parameter is tricky. For many functions value between 0.15 and 0.20 is optimal because in this range the succes rate steeply increases and hits its maximum (Figure 4.15a). However for multimodal functions it is exactly opposite as the success rate starts to decrease towards zero, although with slower rate and lower values should be used. Two exceptions were observed: Rana's and Schwefel's functions. ACO* starts to be successful from $\sigma > 0.45$ on. Last two functions have global minimum close to the function bounds and having higher sigma yields in exploring wider area. This helps to locate even these minima as the algorithm avoids premature convergence into a local minimum. Algorithm iterations taken are shown by Figure 4.15b. Results for all functions correspond to the success rate in the trend of the iteration count – it increases rapidly up to the point $\sigma = 0.15$ and then decreases as the success rate changes slowly. For the above mentioned functions yawing from this scheme the number of iterations follows the trend of the success rate – it rises steadily.

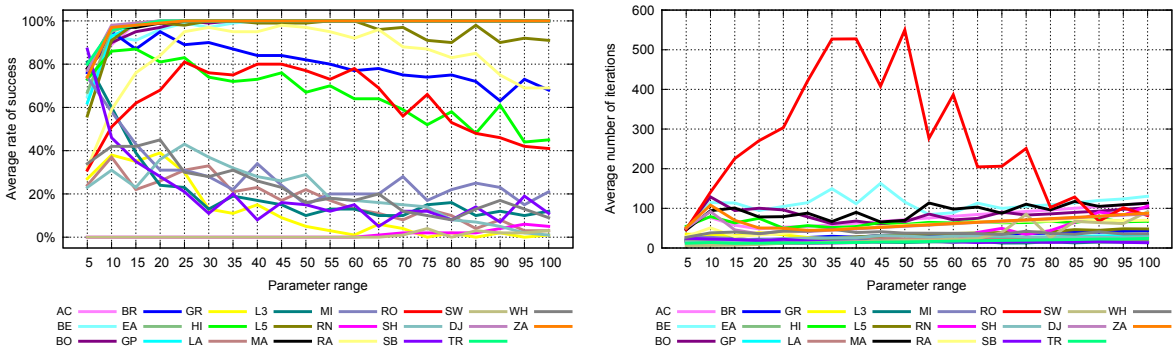
- **force diversity and its neighbourhood size** – except for Rastrigin's function change in this parameter did not projected into the rate of success (Figure D.13a). For the mentioned function the success rate increased slowly with the parameter value. Iteration count was constant, see Figure D.13b.

- **using standard deviation or average** – Table 4.12 summarizes results for ACO* when using standard deviation (marked "S") or an average (marked "AVG"). Using average proved to yield much better results for multimodal functions and comparable results for unimodal functions. This also projected into the iteration count where using average led to slightly longer algorithm runs (Table 4.13).

4.1.12 DACO

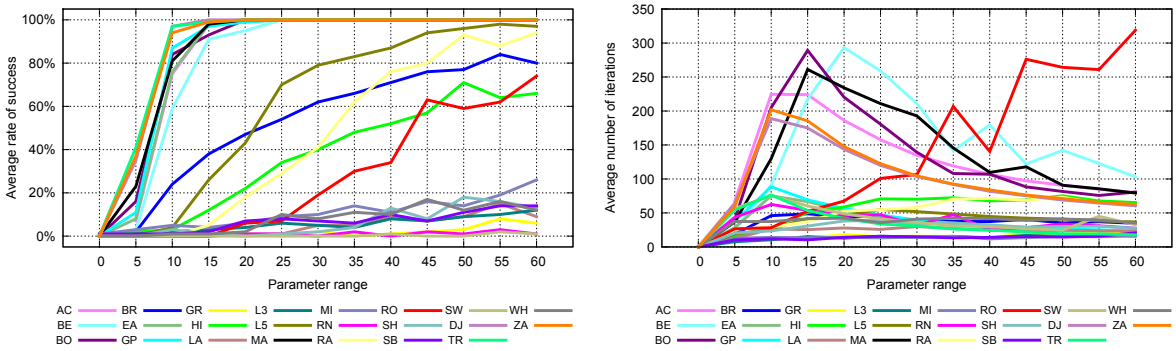
DACO is the last of the ant algorithms we implemented and tested in JCool environment. It uses only two input parameters:

- **population size** – As usually the more ants we use the better results we get (Figure 4.16a). There is no exception. As for iterations taken by DACO, refer to Figure 4.16b. Values increased as the success rate was improving and consequently started to decrease



(a) Average success rate for ACO*. (b) Average number of iterations taken by ACO*.

Figure 4.12: ACO* results for population size $= 5, ..., 100$, step $= 5$.

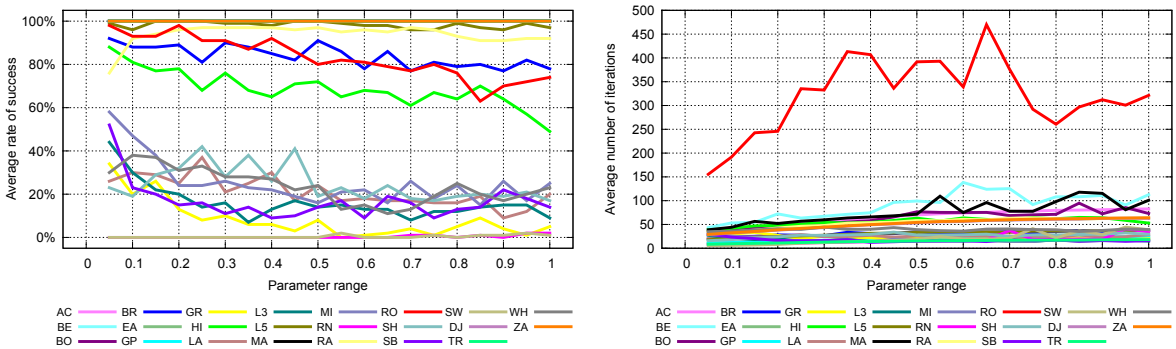


(a) Average success rate for ACO*. (b) Average number of iterations taken by ACO*.

Figure 4.13: ACO* results for number of replaced ants $= 0, ..., 60$, step $= 5$.

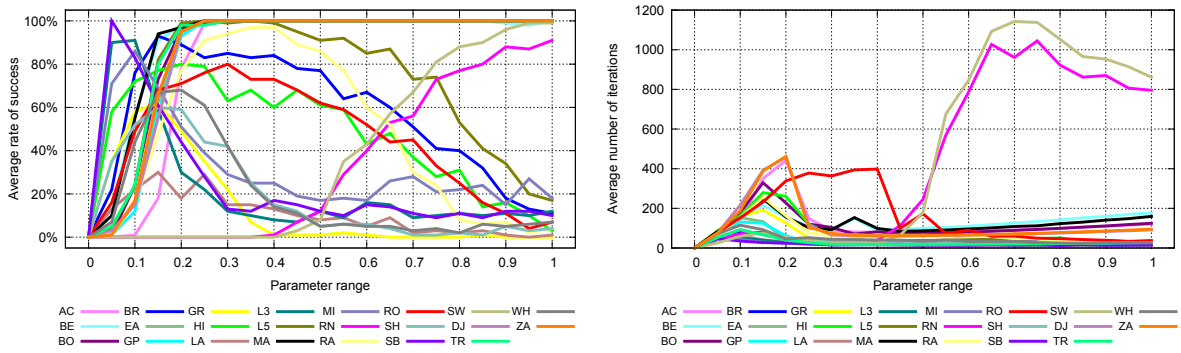
as the success rate reached its maximum. The values observed were low except for Rosenbrock's valley and Beale's function, both involving narrow valleys. For these the DACO method needed more iterations.

- **evaporation factor** λ – the default value is set to $\lambda = 0.5$. However results of our experiments suggest lowering this value according to Figure 4.17a, where it is obvious



(a) Average success rate for ACO*. (b) Average number of iterations taken by ACO*.

Figure 4.14: ACO* results for deviation parameter $\omega = 0.05, ..., 1.00$, step $= 0.05$.



(a) Average success rate for ACO*.

(b) Average number of iterations taken by ACO*.

Figure 4.15: ACO* results for convergence parameter $\sigma = 0.00, ..., 1.00$, step $= 0.05$.

	Unimodal							
	BE	BO	DJ	EA	MA	RO	TR	ZA
SD	100%	100%	100%	100%	100%	74%	100%	100%
AVG	99%	100%	100%	100%	100%	63%	100%	100%

	Multimodal														
	AC	BR	GP	GR	HI	LA	L3	L5	MI	RN	RA	SH	SB	SW	WH
SD	100%	85%	100%	4%	57%	16%	11%	99%	23%	1%	96%	22%	11%	0%	16%
AVG	100%	95%	100%	29%	97%	44%	19%	100%	35%	0%	96%	26%	19%	0%	16%

Table 4.12: Average success rate for ACO* when using standard deviation or average.

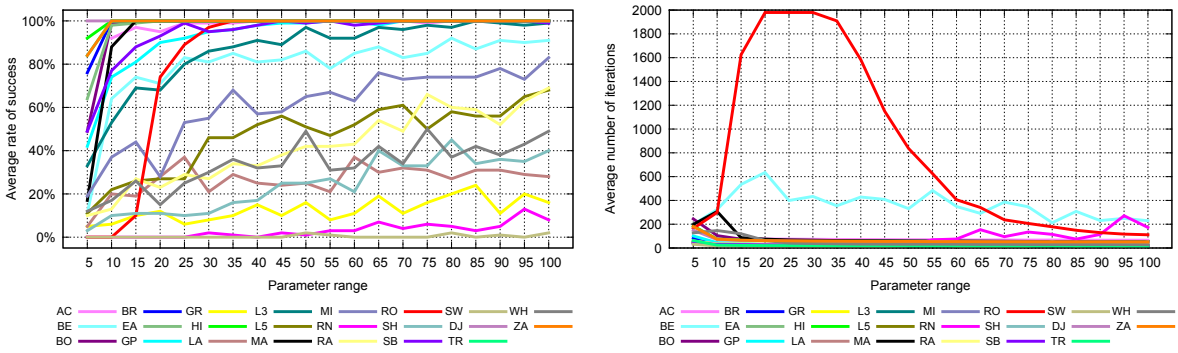
that values of $(0.15, 0.25)$ yield best success rate with just few exceptions. These are Rana's function and Schwefel's function, both having optimum close to the upper limit of parameter value, $\lambda > 0.95$). These have global optimum close to the function bounds. Perhaps having low pheromone evaporation (i.e. high evaporation factor) allows the ants to explore further areas of search space lying close to the limits of individual variables.

These results show another interesting fact – that for extreme values of this parameter the DACO method gives very poor results. This is caused for low values by the fact that pheromone never lasts very long (evaporates almost instantly) and for high values by pheromone staying virtually forever as soon as it is placed. This of course resigns the advantages that we hoped to gain out of the usage of pheromone. Iterations taken by DACO follow the pattern of increasing until the success rate reaches its maximum and then decreasing slowly (Figure 4.17b).

	Unimodal							
	BE	BO	DJ	EA	MA	RO	TR	ZA
SD	110	85	62	23	85	309	18	63
AVG	129	94	66	24	112	338	19	66

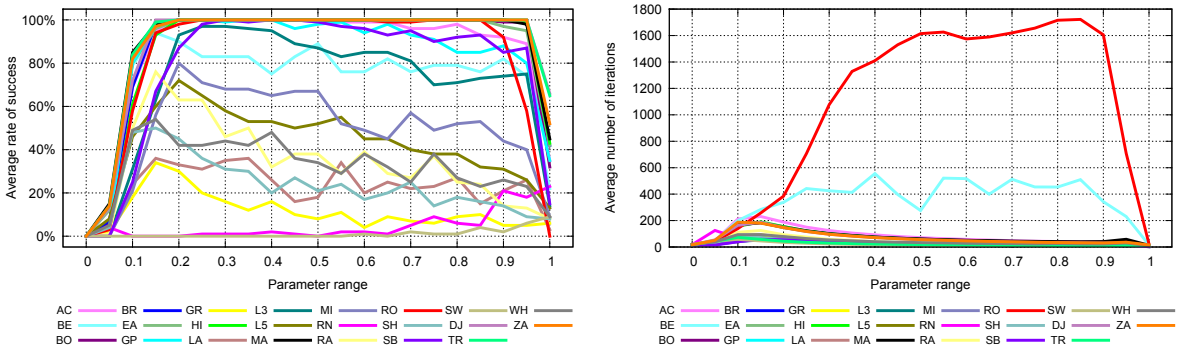
	Multimodal														
	AC	BR	GP	GR	HI	LA	L3	L5	MI	RN	RA	SH	SB	SW	WH
SD	80	37	24	19	58	25	15	37	16	24	64	31	17	23	38
AVG	85	29	27	36	77	31	19	31	22	21	60	29	21	23	44

Table 4.13: Average number of iterations taken by ACO* when using standard deviation or average.



(a) Average success rate for DACO. (b) Average number of iterations taken by DACO.

Figure 4.16: DACO results for population size = 5, ..., 100, step = 5.



(a) Average success rate for DACO. (b) Average number of iterations taken by DACO.

Figure 4.17: DACO results for evaporation factor $\lambda = 0.00, ..., 1.00$, step = 0.05.

4.1.13 PSO

PSO method was examined for all three update methods – the **original**, the **fully informed** and the **canonical** formula. The original formula uses four parameters:

- **population size** – increasing this parameter leads automatically to better results (see Figure 4.18a) and it is well within expectation as the more particles we use the better function value surface coverage we get. This increases the probability of a particle *flying over* a good function value and thus reaching areas with better solution candidates.

  As for the iteration count there were four exceptions from the overall slow decrease after the success rate reaches it top value, until that the iteration count increases rapidly (Figure 4.18b). Two most noticable functions are both offering a global minimum hidden in a valley: Rosenbrock's and Beales's functions. There is a connection between the success rate and iteration number, both were rising together, although their iteration count much more faster. This is again caused by more particles resulting in better searching through the value space and thus taking more iteration before giving up. We can see that the peaks are found at parameter values for which the success rate still rises, although not that fast anymore. This most likely mean that higher values lead both to better solution and to a faster convergence to the solution, which is conformable with our findings about the connection between success rate and the population size.

- **cognitive acceleration coefficient** ϕ_1 – this parameter is deceptive as it is in a product term with a random scalar. This means that effect of this parameter is never the same. Accordingly we have to think of following results only as of guidelines on how to set this parameter, although these results are obtained as an average from 100 runs, which could reduce the effect of the random element to some extent.

  Figure 4.19a shows that even so there are only three functions reflecting change of this parameter on the average succes rate (Figure 4.19b). These are Rosenbrock's valley, Michalewicz's function and Rastrigin's function. PSO was increasingly successful in finding their minimum value as the ϕ_1 value was larger. First two functions provide difficult valleys and better performance of PSO with higher ϕ_1 might be credited to the fact the this parameter *drags* particles to their private best solution found so far and thus keeps them in the valleys, where the solution is hidden.

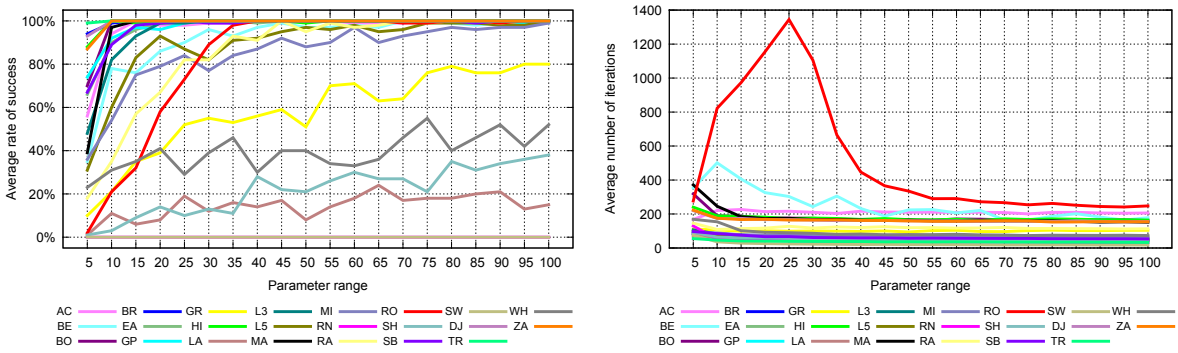
  The last function is highly multimodal and the PSO's performance can be explained in a way that particles keep their private best solution and it is usually some local one. By being attracted to this point rather then to a common global one (which is most of the time just another local one) the particles are not easily dragged into a deep local minima. However, when a new global solution candidate deep enough is found, the particles will be moved closer to it to another local minimum. The difference between the private and the global solution candidate decreases and particles are again more likely to avoid a common local minimum. Other functions seem to have constant success rate.

- **social acceleration coefficient** ϕ_2 – second parameter affected by a random scalar each time it is utilized. However this time the effect of change in the method's success rate is obvious – the higher this parameter was set, the better results we observed (Figure 4.20a).

  We also recorded two exceptions, the Rosenbrock's valley and the Rastrigin's function. For the Rosenbrock's valley the success rate was decreasing up to the value of $\phi_2 = 0.5$ and then followed the rising trend. This might be caused by having $\phi_2 < \phi_1$ means the particles are staying in the valley disregarding whether it is the correct side; the valley is curved. But as the parameter starts to outweight the local attraction rate, particles are more easily shifted to the lower areas of the valley and keep "sliding" towards the global optimum as the particle with the lowest value leads the way. The latter function is highly multimodal and setting the *global attraction* to a high value leads to a premature convergence into a shared global candidate which is unfortunately one of the many local minima. This is exactly the opposite case than with the previous parameter as the decrease is clearly visible for values when $\phi_2 > \phi_1$, meaning that this parameter outweighs the *local attraction* parameter. Therefore the particles are more easily dragged into a local minimum.

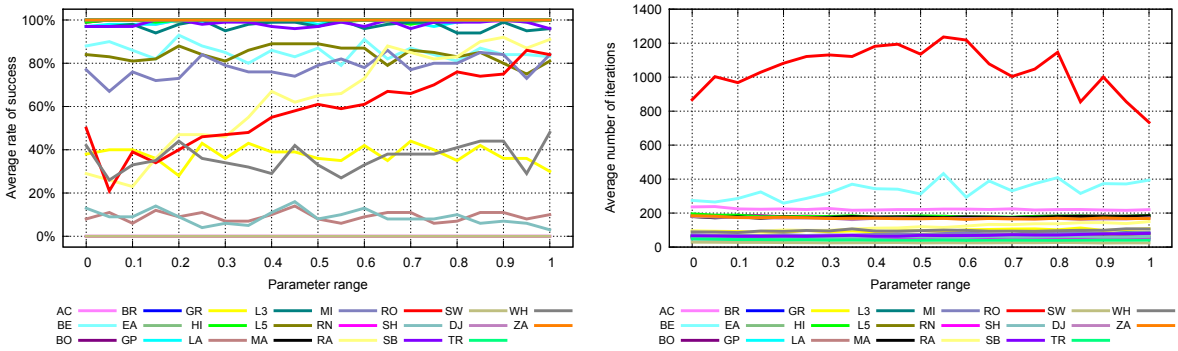
  Considering the iteration count we can notice that again the "valley" functions reflect change of ϕ_2 significantly: the Beale's function and the Rosenbrock's valley (Figure 4.20b). For these the iteration count taken by PSO increases to a certain point and then decreases, both times with the same rate. Global optimum of Beale's function is hidden in a hard to find valley, that might be the cause of te particles missing it when strongly attracted to the global optimum. However the Rosenbrock's valley is easy to find and the iterations taken by PSO on this function correspond to the succes rate: as soon as the success rate increases the iteration count decreases. This follows the idea of particles forming a kind of a snake or a queue with head of this formation having the lowest value and speeding it towards the solution as illustrated by Figure 4.21. Particles are marked by circles, the solution by a cross.

For the fully informed (FI) version of PSO three parameters were tested:



(a) Average success rate for PSO–original. (b) Average number of iterations taken by PSO–original.

Figure 4.18: PSO results for population size = 5, ..., 100, step = 5. Original update formula used.

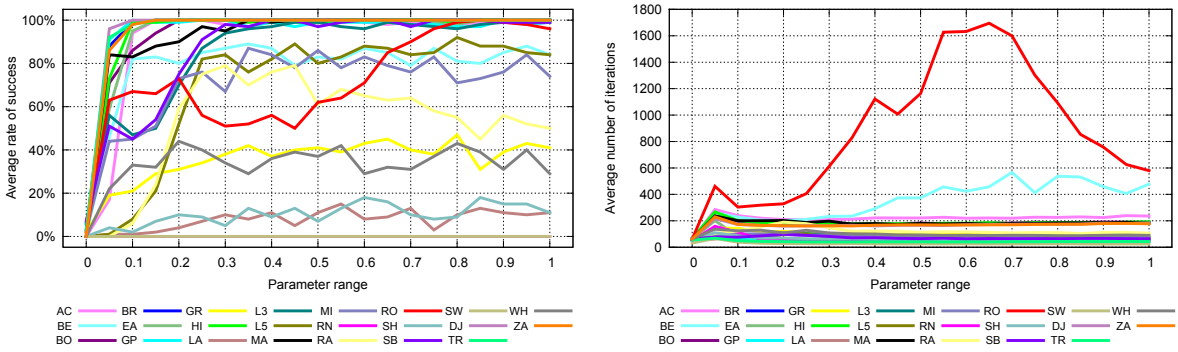


(a) Average success rate for PSO–original. (b) Average number of iterations taken by PSO–original.

Figure 4.19: PSO results for $\phi_1 = 0.00, ..., 1.00$, step = 0.05. Original update formula used.

- **population size** – Figure D.14a and Figure D.14b show that the more particles we use in PSO–FI, the better results we get. Success rate increases steadily up to the upper limit and iteration count is constant.

- **cognitive acceleration coefficient** ϕ_1 – in the FI case the effects of changing $\phi 1$ are



(a) Average success rate for PSO–original. (b) Average number of iterations taken by PSO–original.

Figure 4.20: PSO results for $\phi_2 = 0.00, ..., 1.00$, step = 0.05. Original update formula used.

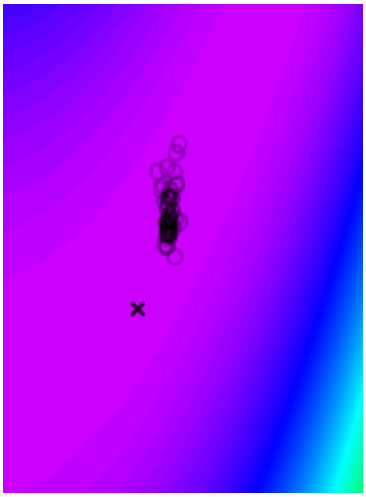
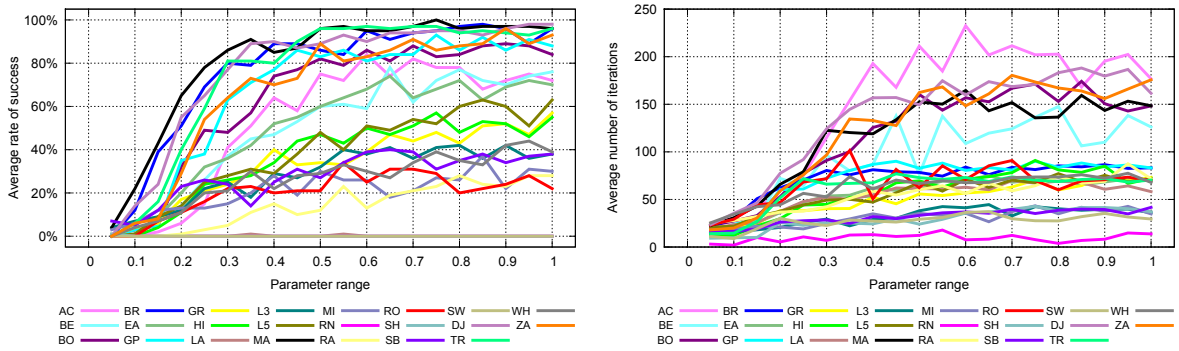


Figure 4.21: PSO: particles inside a valley.



(a) Average success rate for PSO–FI.

(b) Average number of iterations taken by PSO–FI.

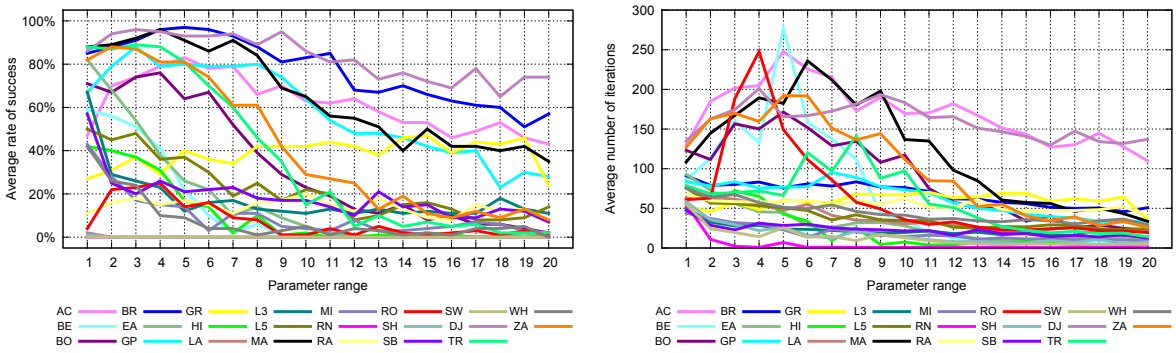
Figure 4.22: PSO results for $\phi_1 = 0.05, ..., 1.00$, step $= 0.05$. FI update formula used.

different from the original version of PSO. Figure 4.22a gives us good overview of the method's performance – the higher parameter value the better. Iterations count merely copies the trend of the success rate as it is growing steadily.

- **neighbourhood distance** m – this parameter is the fundamental part of the FI variation as it says how many other particles are to be asked for the shared global best solution candidate. The idea was that the more we ask the better approximation of function behaviour we get, however it seems that for $m = 1$ majority of tested functions are optimized with greater success than for higher values. For $m = 1$ the FI does not behave in the same way as the standard PSO since the individual used for global information is different for each particle. For other functions values between 3 and 5 yielded best results. Figure 4.23a shows that the rate improved until a point of $m = 6$ and then started to decrease. This means that using global information from more particles then of about $\frac{1}{3}$ does not lead to any further improvement. Iterations also copied the trend of the success rate as the number of iterations needed to terminate raised to a peak and then decreased again, see Figure 4.23b.

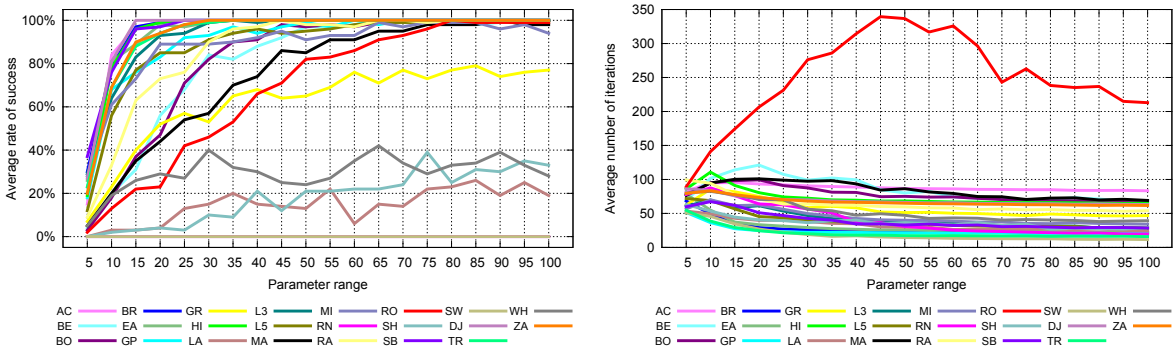
The canonical (C) version of PSO employs four parameters:

- **population size** – this time the population size affects both the success rate and the



(a) Average success rate for PSO–FI. (b) Average number of iterations taken by PSO–FI.

Figure 4.23: PSO results for neighbourhood distance $m = 1, ..., 20$, step $= 1$. FI update formula used.



(a) Average success rate for PSO–C. (b) Average number of iterations taken by PSO–C.

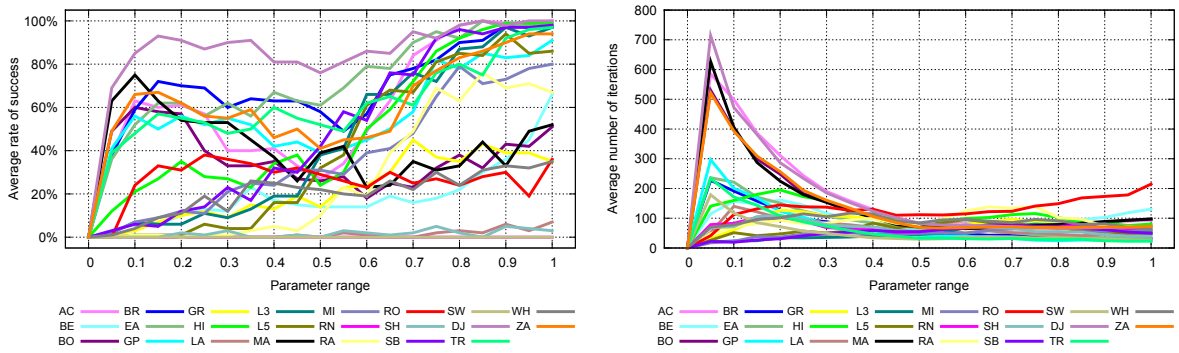
Figure 4.24: PSO results for population size $= 5, ..., 100$, step $= 5$. C update formula used.

iteration count in the expected way. Both Figure 4.24a and Figure 4.24b show that the more particles we use for searching the function variable space the better results we obtain.

- **cognitive acceleration coefficient** ϕ_1 – effects are very similar to both previous variations of PSO, for the success rate is improving with higher values of this parameter (Figure D.15a) and the number of iterations needed to terminate is near constant or rising slowly (Figure D.15b), although this time the improvements in success rate are rather small.

- **social acceleration coefficient** ϕ_2 – exactly the same situation as with ϕ_1. See Figure D.16a and Figure D.16b – there is a decrease in iteration count at the lower values of parameter.

- χ **parameter** κ – from both Figure 4.25a and Figure 4.25b one can see that higher value of this parameter should be used since the success rate was improving as the parameter value was increased. Iterations count was decreasing up to the point where $\kappa = 0.5$ from where number of iterations needed to terminate was constant in general. This behaviour means that higher constraints should be used on the new velocity values because low velocity will lead to only local convergence.



(a) Average success rate for PSO–C.

(b) Average number of iterations taken by PSO–C.

Figure 4.25: PSO results for $\kappa = 0.00, ..., 1.00$, step $= 0.05$. C update formula used.

4.1.14 DE

Differential evolution needs three parameters to work properly. We tested them all within their full range (except for the population size) with constant step:

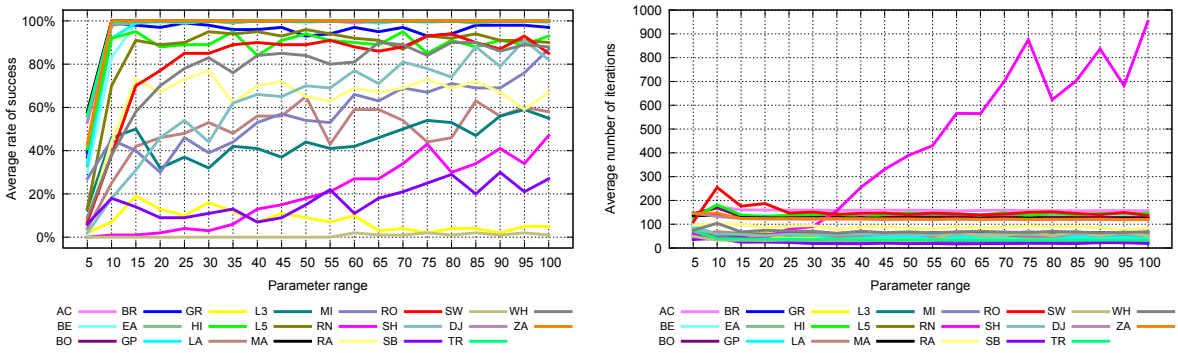
- **population size** – for majority of the test functions the rule of larger population being better holds (Figure 4.26a). There is one exception to note, the Griewangk's function. For this one the success rate increases up to a certain point and then it starts to decrease slowly. For unimodal functions and functions with no local minima and few global ones it is sufficient to use 15 ants.

  Iteration count behaves predictably – it steeply rises until the success rate reaches its peak and then it falls quickly back. See Figure 4.26b and notice the iteration count spent on Rana's function. It is ascending steadily since the success rate for this range of tested parameter did not reach its peak.

- **crossover rating** – this parameter divides the tested functions into three classes: success rate steadily improving, steadily worsening and forming an up–down formation (Figure 4.27a). First class contains *nice* functions, e.g. Booth, Matyas. Second group includes hard multimodal functions: Levy no. 3 and no. 5, Rastrigins's function and Michalewicz's function. The last one is formed by Rosenbrock, Shekel, Langerman and Griewangk.

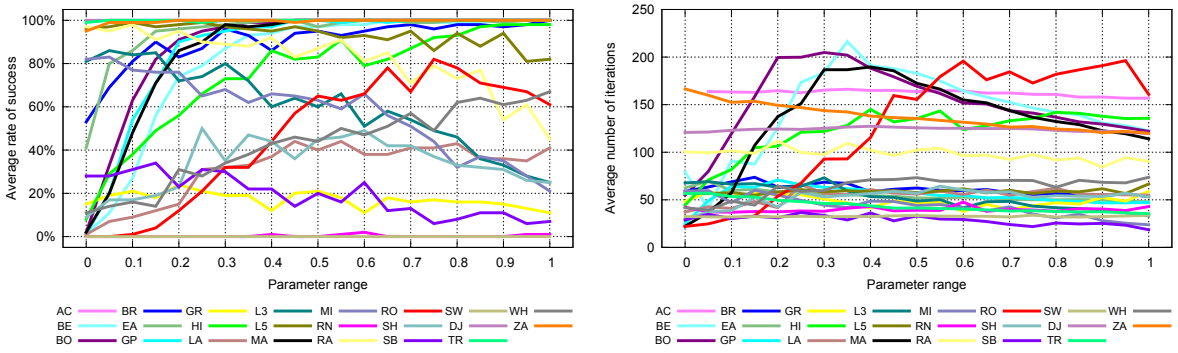
  The first group is easy to describe as these functions have none or just a few local minima and thus most of the crossovers yields better results as the individual points are moving toward the solution. The second is the exact opposite; higher crossover rate leads to a premature convergence which is confirmed by decreasing number of iterations taken with higher values of this parameter for functions in the second class (Figure 4.27b). The last one contains both functions with randomly positioned local minima close to the global one.

- **mutation constant** – DE with mutation constant > 0.35 gives us the best results for the most of the test functions (Figure 4.28a). For this parameter setting there are exceptions too. These are the highly multimodal functions (L3, Shubert, Michalewicz). For these we would recommend either lower value of circa 0.15 or a highest possible value. This is because using lower values avoids fast convergence into a local minimum as the differential part of newly created offspring does not affect it too much. On the other hand, highest value seems to make new offspring to take larger *steps* between generations and to move over local minima, thus avoiding them as well.



(a) Average success rate for DE. (b) Average number of iterations taken by DE.

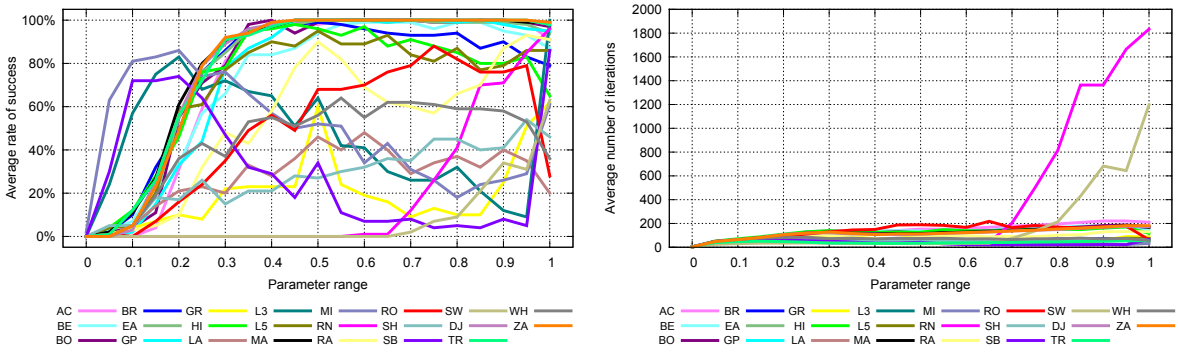
Figure 4.26: DE results for population size = 5, ..., 100, step = 5.



(a) Average success rate for DE. (b) Average number of iterations taken by DE.

Figure 4.27: DE results for crossover rating = 0.00, ..., 1.00, step = 0.05.

Number of iterations is increasing slowly except for the Rana and for the Schwefel. For these the iteration count starts increasing quickly as their success rate starts to improve. See Figure 4.28b.



(a) Average success rate for DE. (b) Average number of iterations taken by DE.

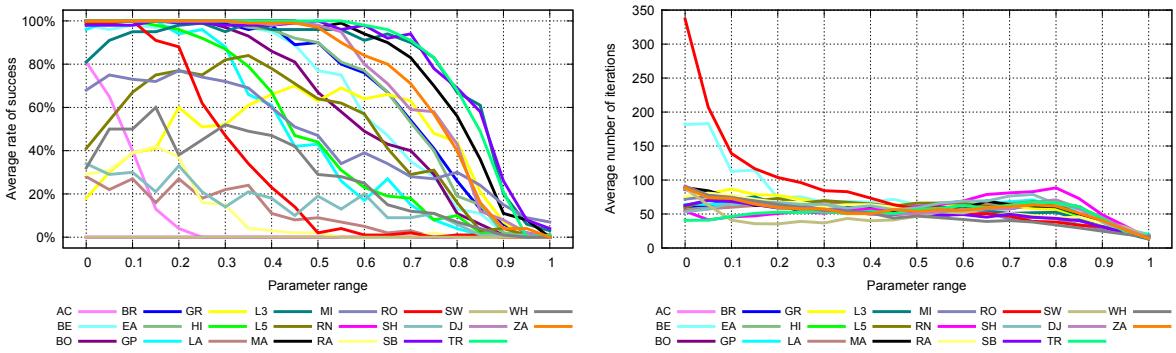
Figure 4.28: DE results for mutation constant = 0.00, ..., 1.00, step = 0.05.

4.1.15 SADE

Experiments with setting of the six parameters used as input to the Simplified Atavistic Differential Evolution follow. These are similar to parameters used in the DE:

- **population size** – as was expected, the more individuals are used the better results are observed. Both Figure D.17a and Figure D.17b are in favor of using higher values for this parameter.
- **radioactivity** – optimal value for unimodal functions seems to be 0.1, see Figure 4.29a. However for multimodal functions (especially highly multimodal ones: Levy 5, Michalewicz, Rastrigin, Griewangk) the value used should be higher – in range from 0.15 to 0.45 because for these values the SADE method gives the best results. This is expectable since a low radioactivity yields only few changes in the newly generated individuals which are placed near their parents and thus are prone to achieve only local convergence. However with higher radioactivity the changes in population are more significant and wider area of function arguments is explored, avoiding local minima when optimizing multimodal functions. Iterations taken by SADE are in general constant or decreasing with falling trend for values higher then 0.8 (Figure 4.29b).
- **local radioactivity** – this parameter should be different for multimodal functions and unimodal functions as well. Unimodal functions give good results for lower values of this parameter (0.3 seems to be the optimum for most of these), multimodal functions reach their optimum for values very near the lower bound, around value of 0.05 (Figure 4.30a). This behaviour has the very same reasons as for the previous parameter had. The iterations count is shown on Figure 4.30b. For the most of the functions there is slow increase in the iterations needed to terminate until the success rate raches its peak. However we can clearly see two exceptions, both of the functions with narrow valleys. The Rosenbrock's valley and the Beale's function seem to cause trouble to the SADE method as the iteration count was increasing for the whole parameter range we used.
- **mutation rate** – optimal value for unimodal functions seems to be located from 0.45 to upper limit of the parameter (Figure 4.31a) with the exception of the Rosenbrock's valley for which we would recommend setting this parameter to zero. For multimodal functions with symmetric local minima the value should be 0.1 (for Rastrigin and Whitley) and for these with randomly placed global and local minima the SADE method gives best results while using the highest parameter value. This is logical since for the unimodal functions high mutation is desired to achieve fast and global convergence, however for multimodal functions we need either low values not to end up in a local minima or higher values in order to cover asymmetric parameter space of functions with randomized function value extremes.

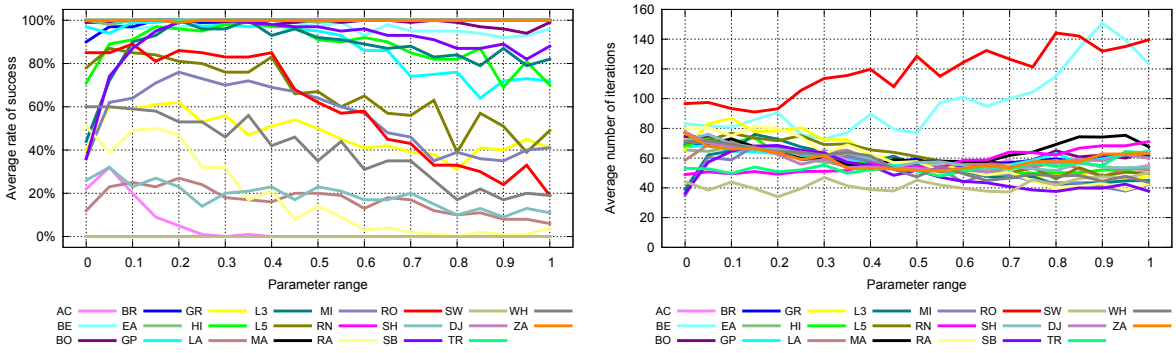
  The number of iterations taken by algorithm decreases steeply and stabilizes from parameter value of 0.1 on, see Figure 4.31b. The Rosenbrock's valley however exhibits slow increase in the iteration count.
- **mutagen rate** – Figure 4.32a shows behaviour of the SADE method when changing this parameter. For values > 300 the success rate is either constant and at its maximum or still increasing slightly (this is the case for the most of the multimodal functions with randomly distributed local minima). Iteration count (Figure 4.32b) is increasing slowly as well but this time for all the functions.
- **crossover rate** – low values should be used since for the majority of the functions we tested values between 0.05 and 0.20 resulted in highest success rate. The only exception



(a) Average success rate for SADE.

(b) Average number of iterations taken by SADE.

Figure 4.29: SADE results for radiation = 0.00, ..., 1.00, step = 0.05.



(a) Average success rate for SADE.

(b) Average number of iterations taken by SADE.

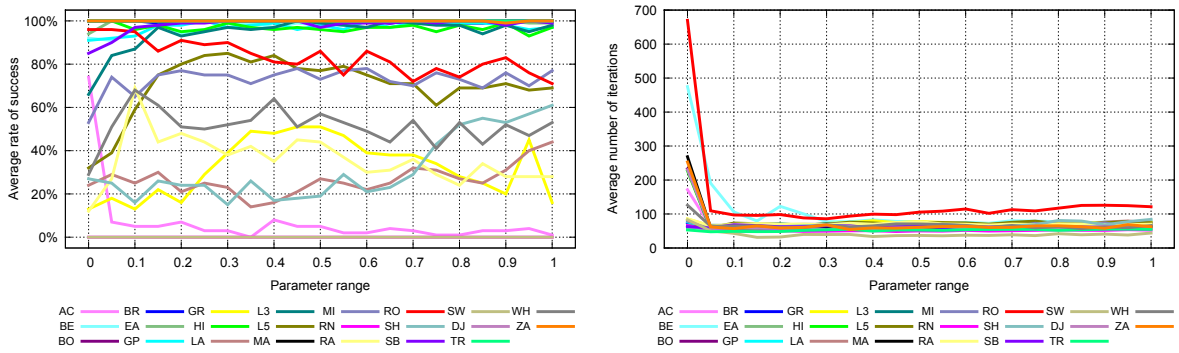
Figure 4.30: SADE results for local radiation = 0.00, ..., 1.00, step = 0.05.

were Rana's function and Schwefel's because these reached best results only for the highest possible value of this parameter (Figure 4.33a). Iteration count is also highest for lower values as it is decreasing slowly for larger values. However for the two mentioned functions the number of iterations taken is increasing quickly (Figure 4.33b). Both functions in question have global minimum close to the function bounds and this seem to be the cause of need for higher values since the algorithm explores wider area with such a setting.

4.1.16 PBIL

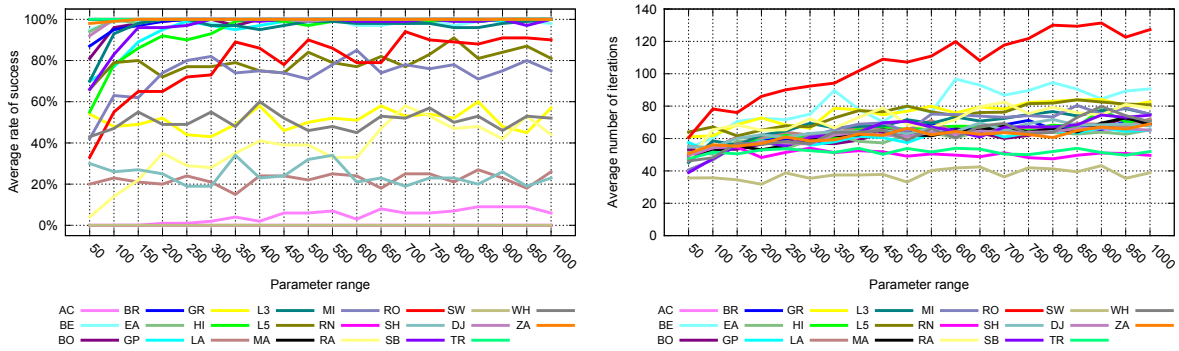
We conducted tests of all six parameters used to configure PBIL method:

- **population size** – Both success rate and iteration count were rising as this parameter was increased. Figure D.18a and Figure D.18b illustrate this behaviour.
- **encoding length** – there seemed to be a very small or no change at all while using different values for this parameter. Figure D.19a shows only that using 8 bit encoded numbers did not lead to very good results, however we expected that for higher values the results would be better. This might be due to a too low default population size for with that parameter increased the PBIL method achieved much better solutions. Iterations (Figure D.19b) were constant after moving from 8 bits to 16 bits which caused almost a



(a) Average success rate for SADE. (b) Average number of iterations taken by SADE.

Figure 4.31: SADE results for mutation rate = 0.00, ..., 1.00, step = 0.05.

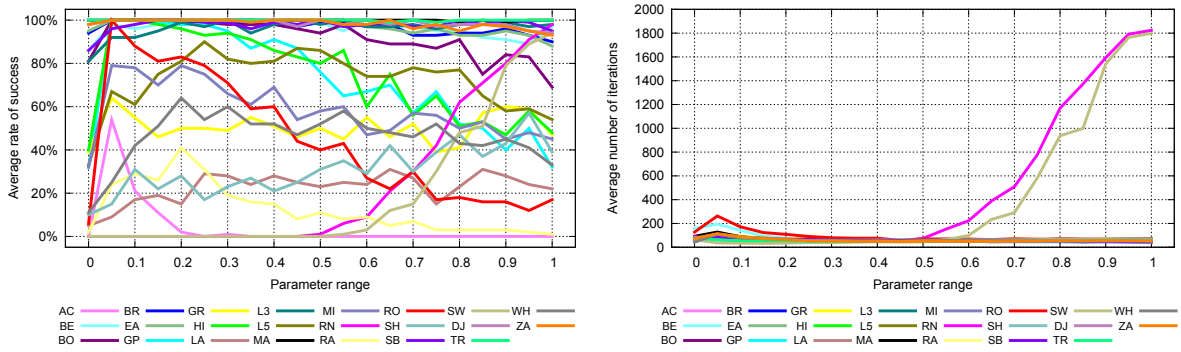


(a) Average success rate for SADE. (b) Average number of iterations taken by SADE.

Figure 4.32: SADE results for mutagen rate = 50, ..., 1000, step = 50.

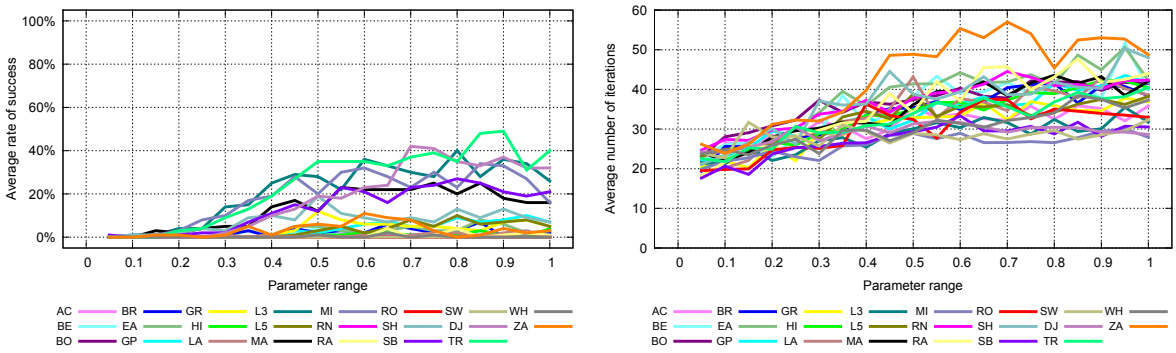
triplication in the numbers. This means that the searching was more successful and the algorithm was not terminated by the stopping condition too early.

- **learning rate** – since this parameter influences how much the newly generated population inclines to good solutions the results shown in Figure 4.34a and Figure 4.34b were quite expected. With higher values we get both better results and worse, higher iteration



(a) Average success rate for SADE. (b) Average number of iterations taken by SADE.

Figure 4.33: SADE results for crossover rate = 0.00, ..., 1.00, step = 0.05.



(a) Average success rate for PBIL.

(b) Average number of iterations taken by PBIL.

Figure 4.34: PBIL results for learning rate = 0.05, ..., 1.00, step = 0.05.

counts.

- **negative learning rate** – effects of this parameter are almost the same as of the previous one, see Figure 4.35a and Figure 4.35b. This parameter pushes the probability vector away from the worst solutions, which makes solution candidates being chosen from the area closer to the promising solutions. However we can see that for highly multimodal functions the PBIL algorithm usually does converges to some local minimum since this seems to be the "good" solution at that stage of the algorithm run.

- **mutation probability** – situation is quite clear when looking at both Figure 4.36a and Figure 4.36b: the optimal value is between 0.1 and 0.2 for all the test functions. This comes with the drawback of an increased number of iterations needed to terminate but this is natural for more successful search.

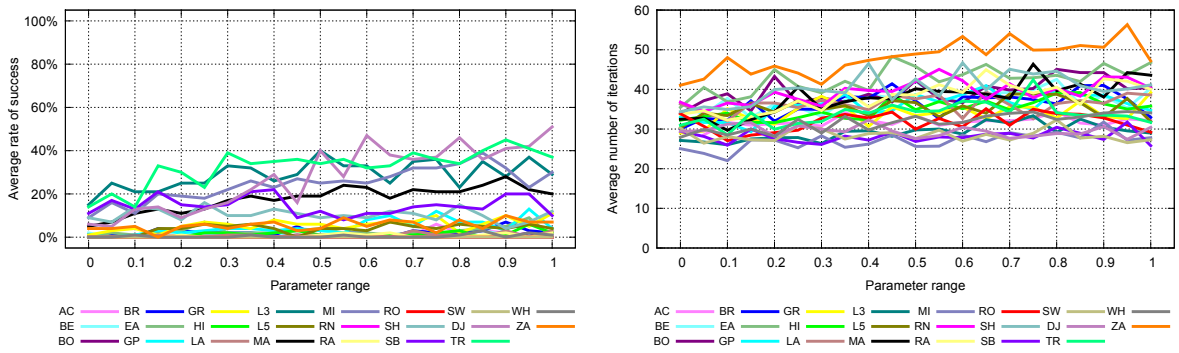
- **mutation shift** – using values between 0.20 and 0.35 should yield good results for all functions we tested, see Figure 4.37a. This is due to the way this parameter affects the probability vector. It indicates the amount of change propagated to the vector when mutation occurs. Since too much change would not closely reflect the change needed the success rate drops to zero for high values. Too low values are not advisable as well because the mutation would not change the probability vector significantly enough. Number of iterations taken is forming an almost symmetrical hill shape with center around a value of 0.4 (Figure 4.37b). Thanks to that we can use the suggested values and enjoy affordable number of iterations needed to find a solution.

4.1.17 HGAPSO

The last tested algorithm uses six parameters. The observations follow:

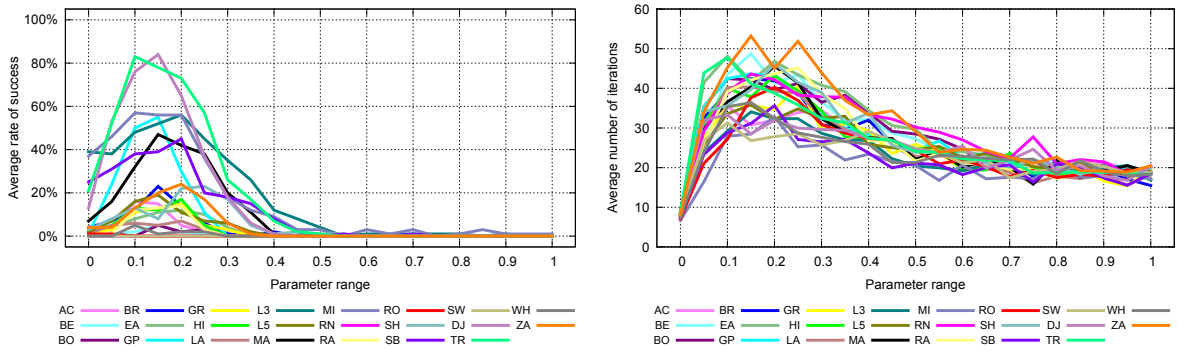
- **population size** – Figure 4.38a confirms that the more particles we use the better results are obtained. Iterations taken (Figure 4.38b) are increasing rapidly for low values and then decrease in the very same fashion. This is due to the fact that having more particles means more work done in each iteration and thus the algorithm can terminate faster.

- **elite ratio** – this parameter indicates how many individuals attend the PSO part of the algorithm. Accordingly we see that it is a good idea to use all of them because then we get a whole population enhanced (Figure 4.39a). Number of iterations is decreasing after



(a) Average success rate for PBIL. (b) Average number of iterations taken by PBIL.

Figure 4.35: PBIL results for negative learning rate = 0.00, ..., 1.00, step = 0.05.

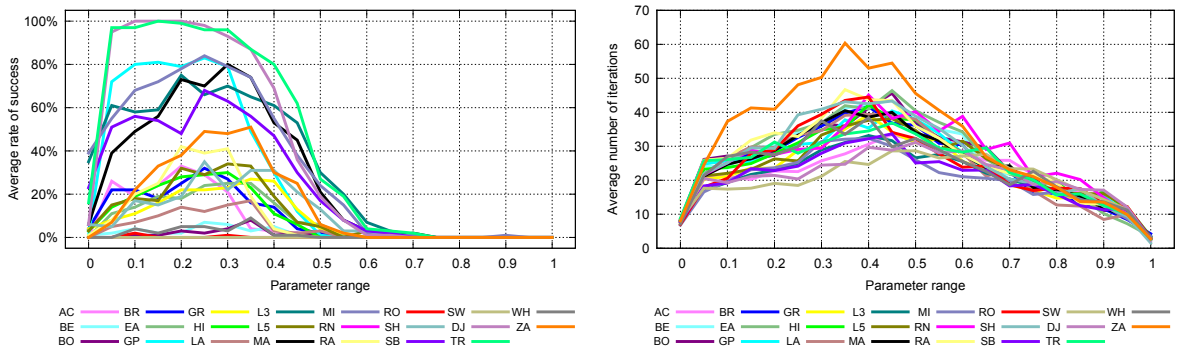


(a) Average success rate for PBIL. (b) Average number of iterations taken by PBIL.

Figure 4.36: PBIL results for mutation probability = 0.00, ..., 1.00, step = 0.05.

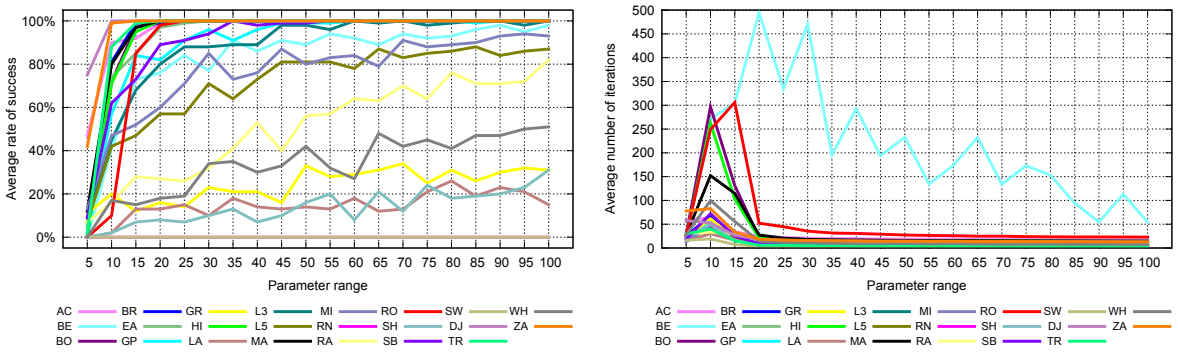
the initial jump and then keeps constant or increases slightly as more individuals are used for elites (Figure 4.39b).

- **cognitive acceleration coefficient** ϕ_1 – no interesting effects were observed while changing this parameter (see Figure D.20a). However the iteration count shows us that for the Beale's function the HGAPSO behaves almost randomly in the terms of iteration



(a) Average success rate for PBIL. (b) Average number of iterations taken by PBIL.

Figure 4.37: PBIL results for mutation shift = 0.00, ..., 1.00, step = 0.05.



(a) Average success rate for HGAPSO.

(b) Average number of iterations taken by HGAPSO.

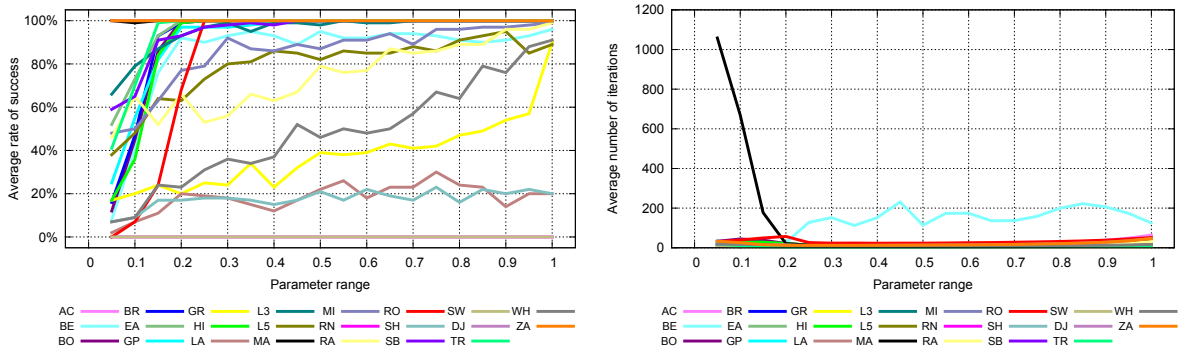
Figure 4.38: HGAPSO results for population size = 5, ..., 100, step = 5.

count, see Figure D.20b. This is because Beale's function makes optimization methods very prone to converge in a wrong direction since the valley hiding the solution is not easy to find. This can make the HGAPSO take wrong way and end up in a local minimum which is far away from the solution and the algorithm is thus taking a lot of unnecessary steps.

- **social acceleration coefficient** ϕ_2 – the higher the better except for the Griewangk's and the Rastrigin's functions, see Figure D.21a. These two should be optimized with this parameter very low or set to zero. This has exactly tha same reason as we stated for the original version of PSO, see Section 4.1.13. Iterations taken by the HGAPSO are affected in the same way as for they were for ϕ_1, see Figure D.21b.

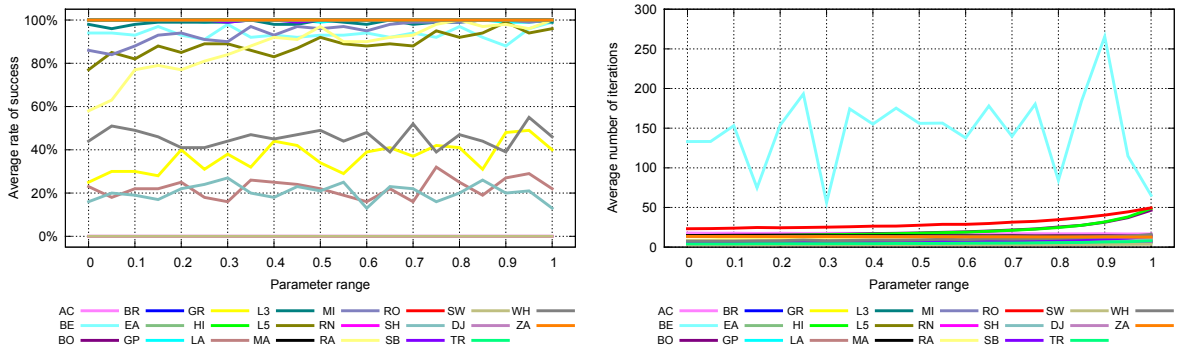
- **mutation probability** – it would seem that there is no effect of this parameter since for the majority of the test functions the average success rate was constant (Figure 4.40a). However we can notice two exceptions, the highly multimodal functions of Griewangk and Rastrigin. For these the HGAPSO's rate of success is improving steadily as this parameter was increased. Iteration count was constant or slowly increasing for high values. On Figure 4.40b we can also notice the Beale's function causing trouble to the HGAPSO algorithm.

- **velocity constraint** χ – except for the already mentioned functions (Griewangk, Rastrigin) we should use high values for this parameter. Figure 4.41a shows us that for highly multimodal functions values between 0.1 and 0.2 should be used. High values also lead to less iterations taken, although the optimum is in the middle of the tested parameter range (Figure 4.41b). Furthermore with high constraint on the velocity (allowing thus higher particle velocities) the HGAPSO again takes more and more steps for the Beale's function, presumably due to falling into a wrong direction.



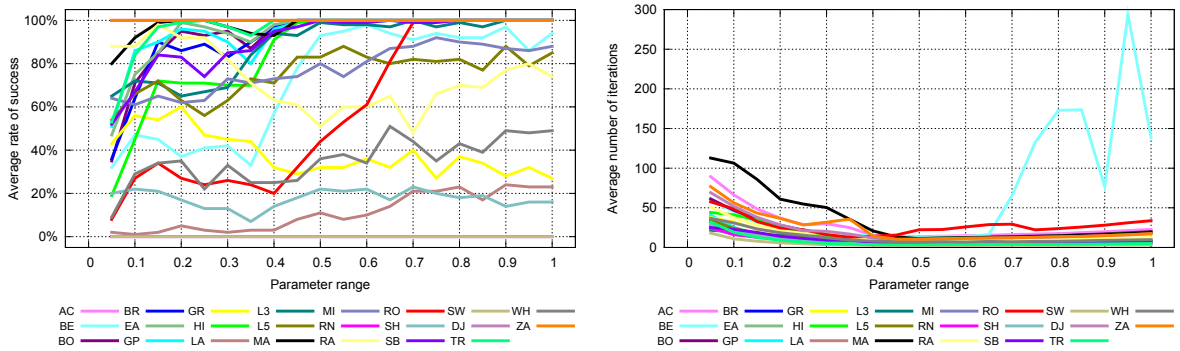
(a) Average success rate for HGAPSO. (b) Average number of iterations taken by HGAPSO.

Figure 4.39: HGAPSO results for elite ratio = 0.05, ..., 1.00, step = 0.05.



(a) Average success rate for HGAPSO. (b) Average number of iterations taken by HGAPSO.

Figure 4.40: HGAPSO results for mutation probability = 0.00, ..., 1.00, step = 0.05.



(a) Average success rate for HGAPSO. (b) Average number of iterations taken by HGAPSO.

Figure 4.41: HGAPSO results for χ = 0.05, ..., 1.00, step = 0.05.

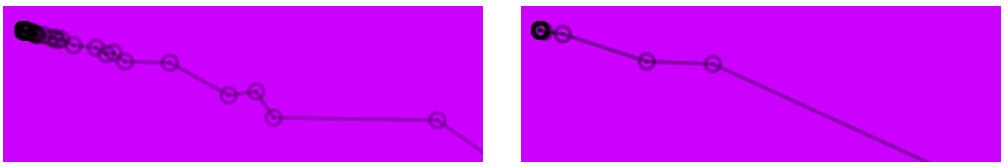
Chapter 5

Recommended parameter values

This chapter recommends optimal parameter settings of the optimization methods according to observed rates of convergence and presents a comparison of default parameter setting and the recommended one according to the best average success rate achieved.

5.1 CG

The CG method is parametrized through the formula for computing parameter β. Although there is no general consent in the terms of which one should be used and usually the choice is based on some external heuristic, it was observed during the experiments that the Fletcher–Reeves formula causes the CG method to take excessively more algorithm iterations. The reason we see as a cause is the fact that it measures only the current step gradient information to the old one instead of incorporating some kind of *current-to-old* gradient information measured to the old one as the other formulas do. This will then lead to the method more similar to SD than to the Newton's method as the original intention of CG was. This is illustrated by Figure 5.1 where the Fletcher–Reeves formula is compared to the Polak–Ribière when both applied on the Booth's function with the same starting point and both finding the global minimum. As we can see the former update formula takes us on rather indirect path to the solution, resembling the SD method, whereas the latter chooses more direct steps.



(a) Fletcher–Reeves formula (b) Polak–Ribière formula

Figure 5.1: CG with different update formulas.

Therefore the recommended update formula would be any of the two other combined with Brent's line search method without the use of derivatives. The comparison can be made directly based on the results in previous chapter. This choise comes even when we see that this formula might be superior in some cases to the other two since for other cases it might be as well inferior. We would then prefer a stable formula resulting in generally good results within the expectation.

5.2 API

Table 5.1 summarizes the recommended parameter values for the API method based on observations made in previous chapter. These values may be used with no regard of the optimized function. Values changed are the population size and the starvation parameter. Value for the population size will be from now on recommended always the same – to be set to the maximum used in the benchmarking (100 individuals) with the only exception of the ACO* method (100 or 30 individuals recommended depending on the objective function).

Population size:	≥ 100
Move nest to the best point every:	30 generations
Number of hunting sites for each ant:	3
Number of iterations without solution before removing hunting site:	8

Table 5.1: Recommended parameter values for API.

5.3 AACA

Recommended parameter values for the AACA method based on the experiments are collected in Table 5.2. Again there was no significant difference in the results over the tested functions and thus this should be regarded as a general recommendation. The encoding length should be higher than the default value as well as the evaporation factor. The pheromone index is recommended to be set higher too, however there was no significant change in the success rate for different values of this parameter.

Population size:		≥ 100
Encoding length:		32 bits
Evaporation factor	λ:	0.9
Pheromone index	β:	0.5
Cost index	δ:	0.0

Table 5.2: Recommended parameter values for AACA.

5.4 ACO*

In Table 5.3 there are recommended parameter values for the ACO* method as obtained after using a cluster analysis for benchmark function types. Parameters should be set differently for functions with only one global minimum and for functions with more global minima, valleys or with randomly placed minima.

Function type:		Type 1	Type 2
Population size:		100	30
# of ants replaced each iteration:		100	30
Deviation parameter	ω:	0.05	0.05
Convergence parameter	σ:	0.2	0.35
Use standard deviation:		$false$	$false$
Force diversity:		$false$	$false$
Type 1:	unimodal or multimodal with one global minimum		
Type 2:	multimodal with local or random minima or a valley		

Table 5.3: Recommended parameter values for ACO*.

5.5 DACO

Since the DACO method uses only two parameters we recommend values in Table 5.4 for all function types tested. Compared to default settings we recommend to lower the value of λ as the DACO then yields better results.

Population size:		≥ 100
Evaporation factor	λ:	0.2

Table 5.4: Recommended parameter values for DACO.

5.6 PSO

Three variants of velocity update formula for PSO were tested and the recommended parameter values are divided accordingly. For all variants the parameters ϕ_1 and ϕ_2 should be used higher or even set to maximum possible value (see Table 5.5).

5.6.1 Original PSO

Number of particles:		≥ 100
Cognitive acceleration coefficient	ϕ_1:	1.0
Social acceleration coefficient	ϕ_2:	1.0

Table 5.5: Recommended parameter values for original PSO.

5.6.2 PSO–FI

There was difference in the attained success rate for individual types of test functions when using the FI update formula for PSO method. In Table 5.6 there are two recommended settings as a result of a cluster analysis of the observed convergence. Note that for $m = 1$ the fully informed variant of PSO in not equal to the original PSO although there is only one individual

used for the global candidate solution. In the fully informed version, this individual is always different for each particle opposite the shared individual in the original version.

Function type:		Type 1	Type 2
Number of particles:		≥ 100	≥ 100
Cognitive acceleration coefficient	ϕ_1:	1.0	1.0
Neighbourhood distance	m:	1	3
Type 1:	functions with small area of global minimum or with multiple local and global minima		
Type 2:	unimodal, multiple local minima or multiple global minima, the rest		

Table 5.6: Recommended parameter values for PSO–FI.

5.6.3 PSO–C

For the canonical velocity update formula there was no need for a change. However we recommend to use more particles as well since this leads to much better results, see Table 5.7.

Number of particles:		≥ 100
Cognitive acceleration coefficient	ϕ_1:	1.0
Social acceleration coefficient	ϕ_2:	1.0
χ parameter	κ:	1.0

Table 5.7: Recommended parameter values for PSO–C.

5.7 DE

The DE parameters again divided tested functions into two groups – highly multimodal functions such as Levy 3, Michalewicz or Rastrigin – and the rest. For these the crossover seemed to be unnecessary or even harmful in the terms of success rate. For the rest (unimodal functions or *not so much* multimodal, e.g. with multiple global minima and no local ones) of the functions the highest crossover rate proved to be most useful. Difference was also in the setting of the mutation rate as this should be used higher the more local and global minima optimized function presents, see Table 5.8.

Function type:	Type 1	Type 2
Population size:	≥ 100	≥ 100
Crossover rating:	0.0	1.0
Mutation constant:	1.0	0.6
Type 1:	highly multimodal functions	
Type 2:	the rest (unimodal or few local / global minima)	

Table 5.8: Recommended parameter values for DE.

5.8 SADE

According to the results for the radioactivity parameter we divided the tested functions in two classes, one once again contains highly multimodal functions and the second the rest (see Table 5.9). The second group does not seem to need the radioactivity to be successfully optimized, however for the first one we observed better results when the radioactivity was set higher then the default setting. Note that for the both classes the local radioactivity is lowered, on the other hand the mutagen rate was set to the maximum value. Crossover rate differs slightly for each class.

Function type:	Type 1	Type 2
Pool size:	≥ 100	≥ 100
Radioactivity:	0.3	0.0
Local radioactivity:	0.2	0.2
Mutation rate:	0.5	0.5
Mutagen rate:	1000.0	1000.0
Crossover rate:	0.2	0.1
Type 1:	highly multimodal functions	
Type 2:	the rest (unimodal or few local / global minima)	

Table 5.9: Recommended parameter values for SADE.

5.9 PBIL

Table 5.10 shows recommended settings for the PBIL method common for all tested functions. These include longer bit strings used for encoding the probability vector as well as increased both learning rate and negative learning rate according to the discussion in the previous chapter. Both the mutation probability and mutation shift were set to a half of the default values.

Population size:	≥ 100
Encoding length:	48 bits
Learning rate:	0.85
Negative learning rate:	1.0
Mutation probability:	0.15
Mutation shift:	0.25

Table 5.10: Recommended parameter values for PBIL.

5.10 HGAPSO

Parameters for the HGAPSO method set to their optimal values as a result of cluster analysis of the data presented in the previous section are summarized in Table 5.11. Elite ratio was set to a maximal value as was expected since this means that the whole population is enhanced by the PSO algorithm before the crossover occurs. ϕ_1 was set to a half of the default value and both χ and the mutation probability were set to maximum values.

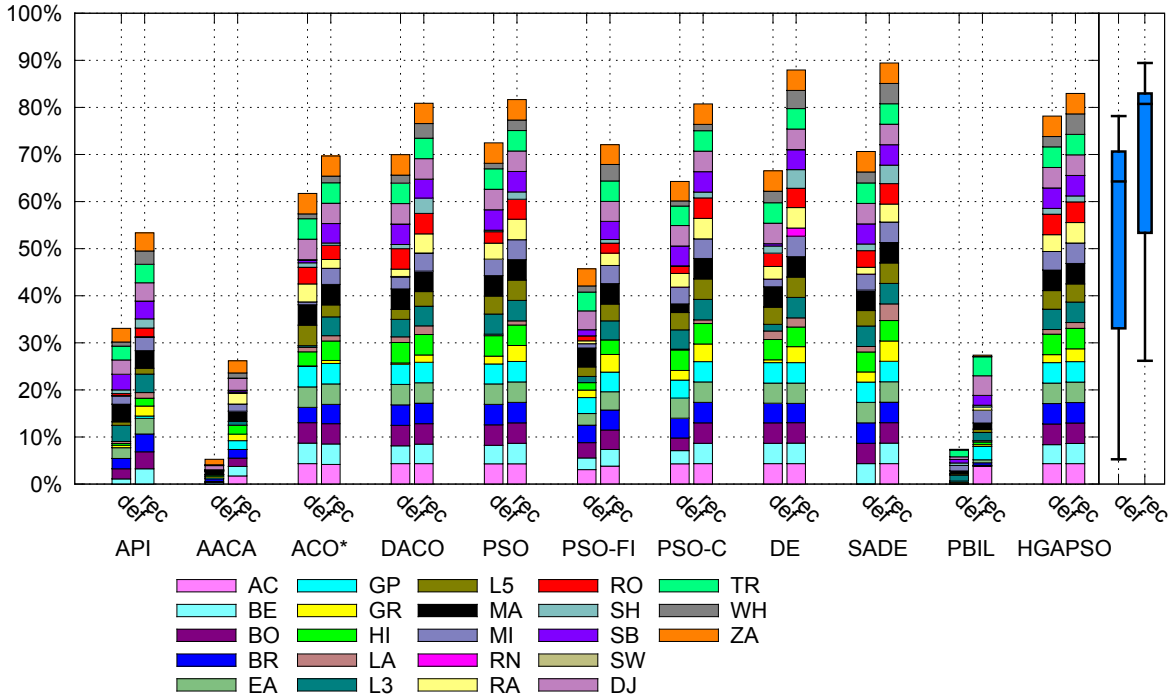


Figure 5.2: Comparison of the average success rate for default parameter setting and for recommended setting.

Population size :		≥ 100
Elite ratio:		1.0
Cognitive acceleration coefficient	ϕ_1:	0.5
Social acceleration coefficient	ϕ_2:	1.0
Parameter	χ:	0.9
Mutation probability:		1.0

Table 5.11: Recommended parameter values for HGAPSO.

5.11 Comparison of the default and the recommended setting

Now that we have a complete set of new settings for implemented algorithms it would be a good idea to verify these values. We optimized the algorithm parameters one by one and thus there is no real guarantee that a combination of these results (i.e. setting all parameters to their individually optimal values) will lead to overall more successful algorithm runs. Therefore we compared the average success rate of each algorithm on every benchmark function using the default setting against the recommended setting. Number of algorithm runs was set to 200, maximal number of iterations to 2000 and the stopping condition to 100 iterations. Figure 5.2 summarizes recorded results.

Two columns correspond to each algorithm. The one on the left is for the default parameter setting (labeled "def"), the one on the right represents results obtained for the setting recommended in this chapter (labeled "rec"). Different recommended settings were used for

different function types if appropriate. Column height represents summation of success rates as recorded for each benchmark function, these are represented by individual color boxes. The ideal algorithm would reach 100% which would mean a 100% success rate for each test function. On the far right there are two boxplots, the left one for average success rates with default parameter setting, the right one for average success rates with recommended setting. We can clearly see that there was a significant improvement.

Average increase in the success rate was 16.09%, the lowest 4.78% for the HGAPSO algorithm, which was already giving very good results, and the highest was 26.35% for PSO with FI velocity update formula. Unfortunately we have no reference to say how good these numbers are besides the overall performance of the given method since we do not know the whole algorithm parameter space and even if we knew we would be bounded by our benchmark set which, although assembled in order to reflect many different function types, can not cover every continuous optimization problem these algorithms are ever to solve. However, given these results we conclude this chapter saying that the suggested parameters resulted in significantly better results and that the experiments were successful since the goal of finding better parameter setting was achieved.

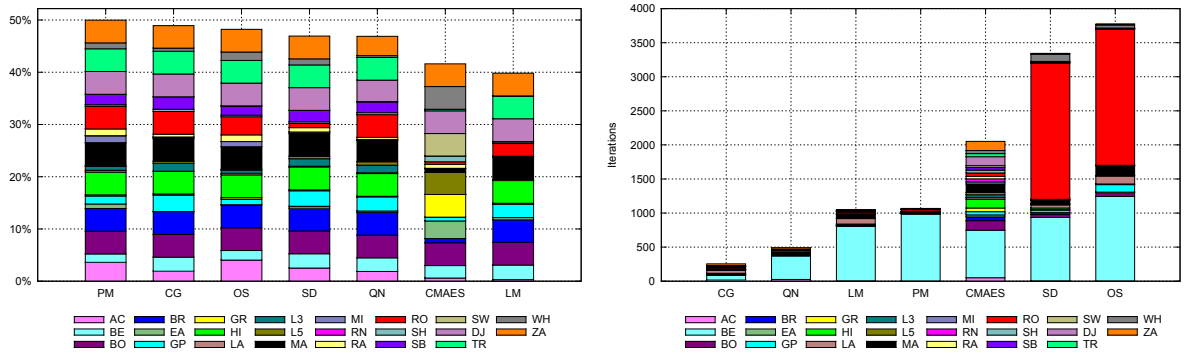
Chapter 6

Comparison of methods

This chapter presents a brief comparison of implemented methods based on iteration count, rate of convergence and on number of function, gradient and hessian evaluations. Besides the discussed rate of convergence in Chapter 2 these indicators should give a researcher an advantage when in need of an optimization method for unknown continuous task. The comparison is first limited to numerical respective nature inspired methods as these can be compared directly and we can focus on common parameters used and then overall comparison concludes the chapter and this thesis.

6.1 Numerical methods

Numerical methods can be compared either by their average rate of success or by the number of iterations taken. This is connected to the number of function, gradient and hessian evaluations since the nature inspired methods do not use this information. Moreover numerical methods are expected to offer only a local convergence since they do not use any global information about the function surface. The first and second order derivatives (or their approximation in the case of QN method) gives the algorithm some information about local area but in the global scope this is not sufficient to guarantee the property of global convergence. In fact no method guarantees global convergence under all circumstances.



(a) Average success rate for numerical methods

(b) Average number of iterations taken by numerical methods

Figure 6.1: Results of numerical optimization.

Figure 6.1 summarizes the results achieved by numerical methods. In Figure 6.1a there is a stacked column of average succes rates for 200 algorithm runs on every benchmark function (represented by color boxes). Iteration limit was set to 2000 iterations, stopping condition to 20 iterations. We can clearly see that in general the numerical methods are not successful in optimizing multimodal functions since they usually converge into a local minimum, depending on the starting point of optimization. Although the average success rates seem to be equal and even the simplest method of SD ranked right in the middle according to the overall success, Figure 6.1b gives us quite different point of view. Now we can see that sophisticated methods such as the CQ and QN have their use as these are unmatched by the simple ones (SD, OS) in the terms of algorithm iterations. Both these methods proved to have nearly quadratic convergence rate for unimodal and convex functions as the theory presented in Chapter 2 suggested.

The iteration count reveals another fact already mentioned before – that the OS and SD methods have very slow convergence when it comes to narrow valleys. This is documented by both methods reaching the limit of 2000 iterations while the others used only a few steps in the case of the Rosenbrock's valley. Other interesting function is the Beale's function as we noticed several times in the Chapter 4. Given the iteration counts are very high for all methods except for the CG method, the Beale's function makes optimization methods take steps in a wrong direction as the global optimum is hidden in a hard to reach valley and surrounding area is slowly decreasing in a directions leading away from the correct solution. However the CG method tends to stop earlier even if it takes the wrong direction. This goes by the average success rate that is comparable among the methods, meaning that the CG does not reach the solution either.

The CMA-ES method differs from the rest of the examined algorithms by that it uses a population of points instead of optimizing just one point at a time. This helps it to overcome local minima and converge into a global one for multimodal functions (e.g. Easom's function – small area of unique solution, Griewangk's function, Whitley's function, Schwefel's function). However it lacks good final convergence on some functions considered as simpler then these multimodal, most noticable the Matyas' function and Branin's function. This also comes with an additional cost in form of iteration count as this method is not as precise as other numerical methods are. We can notice a stack of although low iteration counts for most of the functions yielding in overall last rank among the more sophisticated methods. If we would exclude the Rosenbrock's valley from the test set, the CMA-ES method would certainly rank last. Finaly the De Jong's function proved to be of use as it revealed that CMA-ES with its circa 130 iterations needed to reach the optimum is surely the least effective in optimizing simple functions. This is due to the fact that it uses population to sample the function surface and thus has much lower accuracy then other numerical methods here presented. It takes too many iterations to reach the solution with high enough accuracy even though it is a very simple task.

Last comparison of numerical methods is summarized in Table 6.1. This table offers generalized numbers of function, gradient and hessian evaluations taken by numerical methods each iteration. We consider only the Brent's line search method without the use of derivatives. ls is a substitution for the number of function evaluations taken by the line search method. This number will vary from method to method as it depends on the function area where the current line search is conducted. n is the number of dimensions of the optimized function. Last column states the average number of function evaluations as measured during the experiment and is presented as a proof that the number of function evaluations taken by line search is variable (see values for e.g. SD and QN methods).

The method of choice for unimodal functions regarding previous results is the CG method. QN method is ranked second as an almost equal alternative. These two methods have the advantage of having multiple update formulas to choose from, which might lead to even better

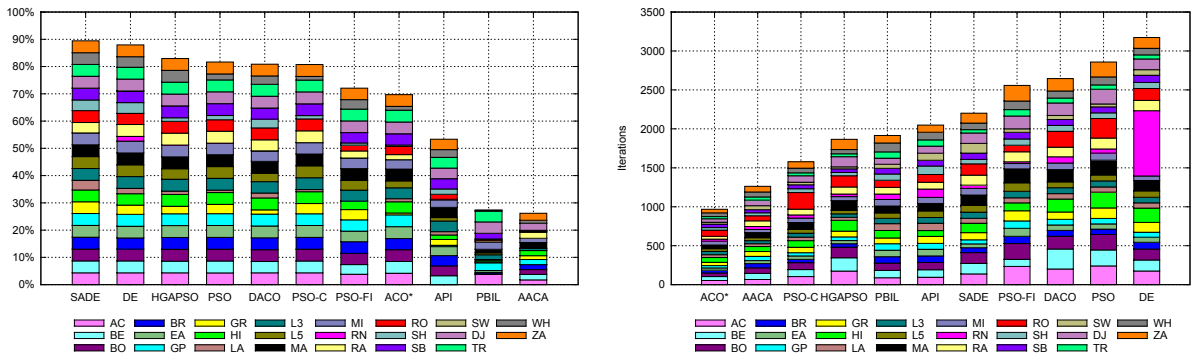
Evaluations per iteration:	function	gradient	hessian	AVG function
SD	ls	1	0	20.15
CG	ls	1	0	34.68
LM	$1 + ls$	1	1	35.98
QN	ls	1	0	43.13
OS	$n \cdot ls$	1	0	32.85
PM	$2 + (n + 1) \cdot ls$	1	0	72.42
CMA-ES	$4 + 3 \cdot \ln(n)$	1	0	6

Table 6.1: Summary of function, gradient and hessian evaluations taken by numerical methods.

results in specific situations not covered by this thesis. For multimodal functions the CMA-ES method should be used.

6.2 Nature inspired methods

Methods inspired by nature usually do not use gradient nor hessian information to examine the objective function we want to optimize. Instead they use set of points spread in the function parameter space to gather information on the function characteristics. Since all the methods used for the population size the recommended value (with the only exception of ACO* for multimodal functions with multiple global minima) of 100 individuals we can compare them directly. All other benchmarking conditions were set equally: 200 algorithm runs, limit of 2000 algorithm iterations per run and stopping criterion of 50 consecutive iterations with no improvement in the function value (this value is higher since these methods usually need *more time* to improve the current best solution candidate). Local search method of algorithms using this technique was also implemented identically for all of them, see Section 2.3.1.1.



(a) Average success rate for methods inspired by nature (b) Average number of iterations taken by methods inspired by nature

Figure 6.2: Results of optimization by nature inspired methods.

Figure 6.2a compares the average success rates of nature inspired methods for the whole benchmark set under these conditions. Success rate for individual functions is represented by color boxes. These were added together to obtain an indicator of overall performance (column height). The recommended parameter values for optimization methods were used. As for these results the genetic algorithms SADE and DE ranked top while methods using bit strings to encode real valued function parameters ranked last. The only function that resisted

Method	Function evaluations per iteration
AACA	ps
ACO	$rs = 1, ..., ps$
API	ps
DACO	ps
DE	ps
SADE	$3 \cdot n \cdot ps$
PBIL	ps
PSO	ps
HGAPSO	$ps \cdot ((1 - er) \cdot 4 + er \cdot PSO)$

Table 6.2: Summary of function evaluations taken by methods inspired by nature.

optimization efforts of these algorithms was the Schwefel's function since no method succeeded at all. Also the Rana's function was problematic in general, only the DE method was somehow successful. However besides these results the two top algorithms achieved almost 90% success rate which makes them very effective means of continuous optimization.

It is obvious that nature inspired methods do have a property of global convergence. Although not guaranteed, this makes them number one choice for multimodal functions for these can not be successfully optimized by numerical methods. However local convergence of nature inspired methods is rather poor since they are much less accurate and their local search is often random (see Section 2.3.1.1 and Section 2.3.1.2). This leads to slower final convergence as mentioned in previous section in connection with the CMA-ES method.

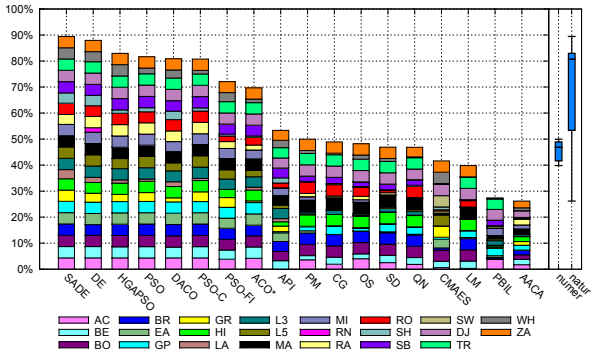
However there is the question of number of algorithm iterations taken. These are illustrated by Figure 6.2b. Although very successful according to the solutions found, genetic algorithms SADE and DE need rather more iterations to find them. Of course we could neglect the Rana's function in the sake of DE as it was the only one successful in finding its solution, however for high price in the iteration count. Without the Rana's function DE would rank on 8^{th} right after the SADE method. On the other side the ACO* algorithm ranks top and with its average success rate nearing 70% is a good candidate for overall method of choice. Given ranks in both success rate and iteration count there is however another method which should be recommended for function optimization. The HGAPSO method ranked 3^{rd} considering the success rate with over 80% average success rate and 4^{th} in the iteration count ladder. That makes it the overall rank number one. The AACA algorithm used only a few iterations even though it ranked bottom regarding the success rate. This suggests that the algorithm stops early due to the stopping criterion and should be examined under looser conditions allowing it to run longer.

Number of function evaluations for each algorithm is summarized in Table 6.2. These are presented in the terms of population size ps, number of replaced ants rs (ACO, recommended value $rs = ps$ was used), number of dimensions of the objective function n, elite ratio $er \in \langle 0, 1 \rangle$ and number of iterations of particle swarm optimization used to enhance elites PSO (both used in HGAPSO). Value for the PSO method applies to all velocity update formulas.

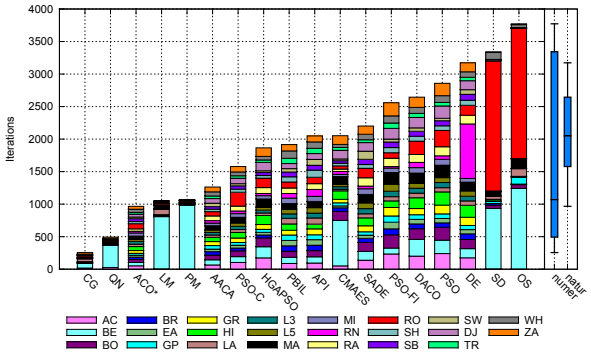
6.3 Overall comparison

Comparing numerical methods and methods inspired by nature can not be done in a direct way as it was done until now. We can try to compare both their success rates (Figure 6.3a)

and numbers of iterations needed to terminate (Figure 6.3b), however we would miss some very important facts by doing so. If only these measures would be used, the numerical methods would rank poorly in terms of average success rate and somehow better considering the iteration count.



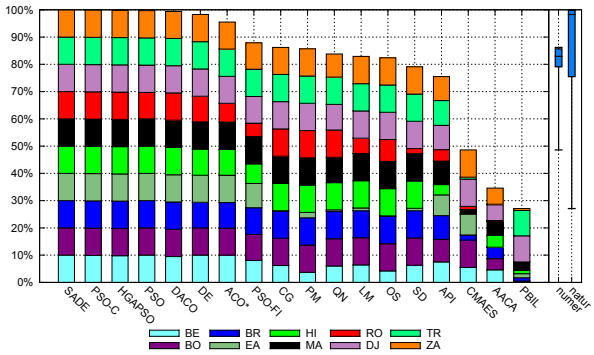
(a) Average success rate for implemented methods



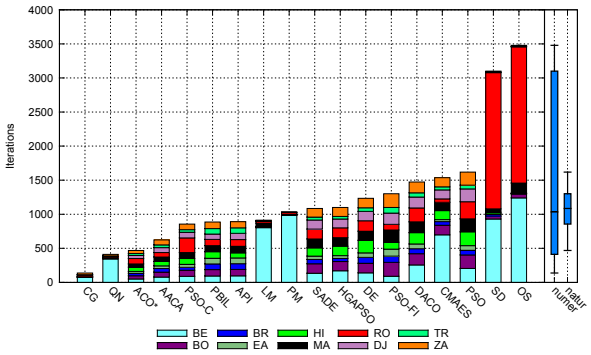
(b) Average number of iterations taken by implemented methods

Figure 6.3: Results of optimization by implemented methods.

Certainly better way to compare numerical methods offering local convergence and nature inspired methods offering global convergence is to focus separately on their performance exclusively on unimodal functions or functions with no local minima and exclusively on multimodal functions with local minima. Results for unimodal functions are represented by Figure 6.4, for multimodal by Figure 6.5. According to these we should use numerical methods for unimodal functions or multimodal functions with no local minima even though they might perform little worse (this is due to the Easom's function which is optimized by a set of points with much advantage over the methods using single point) for they offer very good iteration counts (as mentioned before, this is due to their near quadratic convergence). However the ACO* is equally efficient as it offers better rate of success with a slightly higher number of iterations taken.



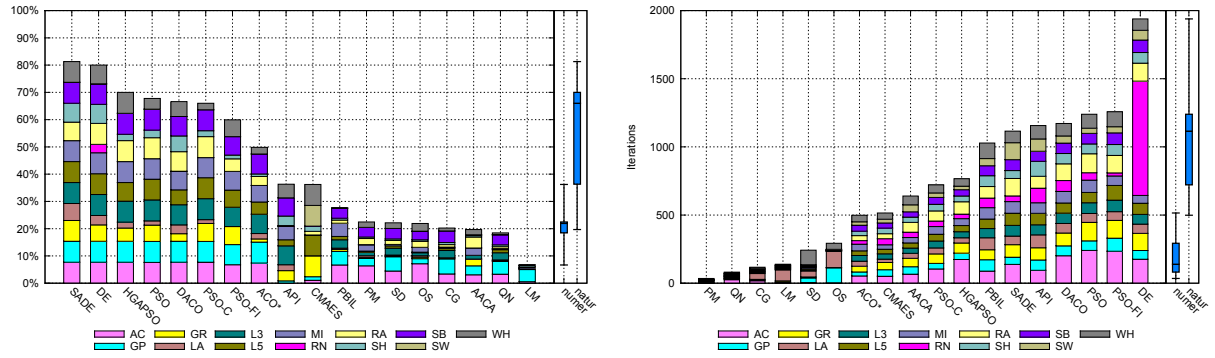
(a) Average success rate on unimodal functions



(b) Average number of iterations taken on unimodal functions

Figure 6.4: Results of optimizing unimodal functions.

For multimodal functions we must recommend nature inspired methods because the numerical methods lack global convergence. We can see that clearly from their success rate compared to the numerical methods. The only exception is the AACA method, which uses bit strings



(a) Average success rate on multimodal functions

(b) Average number of iterations taken on multimodal functions

Figure 6.5: Results of optimizing multimodal functions.

to encode the solution candidates. The AACA method was discussed in previous section with regards to its poor performance under strict stopping conditions. The only advantage of the numerical methods when optimizing multimodal functions is that they take just a few steps and then terminate. However we would not know whether we discovered a local or a global minimum. Therefore we should try to restart the method several times and see if the solution is changing or not. This increases the chance that we will discover different values which should be a signal to switch to nature inspired methods. This might be affordable since the numerical methods need usually only a few steps to converge into a local minimum.

So far it might seem that there is no use for numerical methods except for unimodal functions, functions with only global minima and a very simple way of obtaining meta information on the objective function we want to minimize. This would be very premature conclusion since there is a great potential in the numerical methods: their numerical precision. In our benchmarking we used an arbitrary small value to prevent an exclusion of a solution candidate with lower numerical precision from the successful result set. The value we used was relatively high especially on account of the nature inspired methods. Since these use various heuristics to optimize their population of points, they posses different numerical precision. This is usually much lower than the precision of numerical methods for most of which we would find the need of such a tolerance on accuracy of solution unnecessary. The numerical precision is illustrated by Figure 6.6 and Figure 6.7. They show solutions found for the Booth's function by the DACO method (Figure 6.6) and the SD method (Figure 6.7). Despite the DACO method was used with the recommended parameter values the accuracy is not absolute.

Results

Solution	100 point with minimim: 1.2503445347753037E-27@[1.0000000000000167,2.9999999999999742]
Stopped on	[Convergence detected at 1.2503445347753037E-27]
Value evaluations	16500
Gradient evaluations	0
Hessian evaluations	0

Figure 6.6: Solution for Booth's function found by DACO.

Results

Solution	0.0@[1.0,3.0]
Stopped on	[Convergence detected at 0.0]
Value evaluations	237
Gradient evaluations	11
Hessian evaluations	0

Figure 6.7: Solution for Booth's function found by SD.

Individuals spread around the solution are shown in a detailed view of the area of the global minimum in Figure 6.8. In contrast to this the SD found the solution with absolute precision, in fewer steps and using fewer function evaluations.

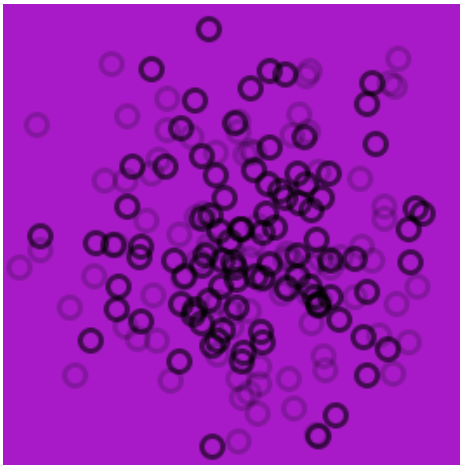


Figure 6.8: Detail of the solution found by DACO.

Therefore we should bear in mind that numerical methods solve suggested tasks both very quickly (quadratic convergence of CG and QN) and efficiently (sophisticated methods have absolute numerical precision if the function can be modelled by quadratic forms near the minimum). Moreover, the numerical methods evaluate the function, gradient or hessian only once each iteration if we do not count the evaluations needed for line searches. Opposite this the nature inspired methods evaluate the function value in general many times more since they work with a set of points. This of course depends on the population size, however the number of function evaluations taken by line searches are lower then recommended values for populations size as shown in previous section.

Chapter 7

Summary and conclusions

In this thesis I presented a unified description of several numerical optimization methods as well as several different optimization methods inspired by nature. These include Ant algorithms, Particle swarm methods and Genetic algorithms. I described mechanics of each method and explained their benefits and drawbacks. I implemented these methods into the JCool environment and adjusted them according to conducted experiments and previous work done on FAKE GAME software.

Furthermore I implemented numerous objective functions acknowledged for their useful characteristics as a benchmark functions and offered their complete description. I selected them in order to cover a wide variety of continuous optimization tasks and to contain both simple functions and those considered as hard tasks. These now form JCool benchmarking set of objective functions, which I used to evaluate the rate of convergence and parameter dependent behaviour of implemented optimization methods. I tested all constrained algorithm parameters using values from their full range as well as I tested unconstrained algorithm parameters with values generally accepted as giving reasonable results. I discussed parameter dependent behaviour of individual optimization methods in detail and derived a set of recommended algorithm parameter values which should be used instead of the default ones which comes from source articles or the FAKE GAME software.

Results obtained for newly set algorithm parameter values were significantly better in comparison to the default settings for every algorithm examined and proved that the evaluation of parameter dependent behaviour was correct and successful. Moreover I suggested a simple classification of objective functions in connection to recommended parameter settings and outlined a basic division of algorithm parameters usable for meta-optimization where experiment results were appropriate. These were also experimentally verified as correct.

My thesis is concluded by rigorous method comparison which offers a collation of both numerical methods and methods inspired by nature and finally their united confrontation. I described their usefulness and weaknesses and evaluated their overall performance using different measures such as objective function evaluations, gradient and hessian matrix evaluations, number of algorithm iterations and average success rate in finding global minimum. A recommendation on ideal usage of these optimization methods is one of the outcomes of my thesis.

7.1 Future work

Although this thesis aimed to cover a lot of different optimization methods over standard benchmark problems there will always be a way to take and continue with the research this

thesis presented. Although we described the behaviour of individual existing algorithms we only broached the problem of meta-optimization. The previous chapter presented a framework usable when facing a new, unknown task and being confronted with need of choice of suitable algorithm. This framework gives the basic idea which one of the presented algorithms might prove fast and effective and which ones most likely would not. This whole area of research is very actual and is in the center of attention of many researchers. I see a great opportunity in this field of research as it opens many ways to expand the results obtained through this thesis as well as it suggests possible improvements to the JCool framework.

The next step we will take is to adapt the JCool environment for **automatic benchmarking** of selected algorithms on a subset of disposable test functions. This will require extension of the source code of the annotations to **specify a step-size and range** for individual algorithm parameters which are to be tested. **A summarizing report** is due to be created, storing the results in a preferably portable format (either XML of HTML are of consideration). New visualisations need to be created as well to give a better insight how does current algorithm work and whether are its iterations effective. This covers both the pheromone levels for ant algorithms as well as directions in the particle swarm optimization.

References

[1] M. Hvizdoš, "Knihovna pro spojitou optimalizaci," Master's thesis, Fakulta elektrotechnická, České vysoké učení technické v Praze, Karlovo nám. 13, 121 35 Praha 2, 2009.

[2] P. Kordík, *Fully Automated Knowledge Extraction using Group of Adaptive Models Evolution*. PhD thesis, Fakulta elektrotechnická, České vysoké učení technické v Praze, Karlovo nám. 13, 121 35 Praha 2, 2006.

[3] R. P. Brent, "Algorithms for minimization without derivatives," 1973.

[4] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical recipes in C (2nd ed.): the art of scientific computing*. New York, NY, USA: Cambridge University Press, 1992.

[5] J. R. Shewchuk, "An introduction to the conjugate gradient method without the agonizing pain," tech. rep., Carnegie Mellon University, 1994.

[6] R. Fletcher and C. M. Reeves, "Function minimization by conjugate gradients," *The Computer Journal*, vol. 7, no. 2, pp. 149–154, 1964.

[7] E. Polak, *Computational methods in optimization; a unified approach [by] E. Polak*. Academic Press, New York,, 1971.

[8] E. Beale, "A derivation of conjugate gradients." Numer. Math. non-lin. Optim., Conf., Univ. Dundee 1971, 39-43 (1972)., 1972.

[9] H. W. Sorenson, "Comparison of some conjugate direction procedures for function minimization," *Journal of the Franklin Institute*, vol. 288, no. 6, pp. 421–441, 1969.

[10] M. R. Hestenes and E. Stiefel, "The method of conjugate gradients for solving linear systems," *J. Res. Nat. Bur. Standards*, pp. 409–436, 1952.

[11] K. Levenberg, "A method for the solution of certain non-linear problems in least squares," *Quarterly Journal of Applied Mathmatics*, vol. II, no. 2, pp. 164–168, 1944.

[12] D. W. Marquardt, "An algorithm for least-squares estimation of nonlinear parameters," *SIAM Journal on Applied Mathematics*, vol. 11, no. 2, pp. 431–441, 1963.

[13] J. E. Dennis and R. B. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Prentice-Hall Series in Computational Mathematics, Englewood Cliffs, New Jersey: Prentice-Hall, 1983.

[14] C. G. Broyden, "A class of methods for solving nonlinear simultaneous equations," *Mathematics of Computation*, vol. 19, no. 92, pp. 577–593, 1965.

[15] W. C. Davidon, "Variable metric method for minimization," *SIAM Journal on Optimization*, vol. 1, no. 1, pp. 1–17, 1991.

[16] R. Fletcher and M. J. D. Powell, "A rapidly convergent descent method for minimization," *j-COMP-J*, vol. 6, pp. 163–168, July 1963.

[17] C. G. Broyden, "The convergence of a class of double-rank minimization algorithms," *j-J-INST-MATH-APPL*, vol. 6, pp. 76–90, 1970.

[18] R. Fletcher, "A new approach to variable metric algorithms," *j-COMP-J*, vol. 13, pp. 317–322, Aug. 1970.

[19] D. Goldfarb, "A family of variable metric updates derived by variational means," *Mathematics of Computation*, vol. 24, pp. 23–26, 1970.

[20] D. F. Shanno, "Conditioning of quasi-newton methods for function minimization," *Mathematics of Computation*, vol. 24, pp. 647–656, July 1970.

[21] R. Fletcher, *Practical methods of optimization; (2nd ed.)*. New York, NY, USA: Wiley-Interscience, 1987.

[22] M. J. D. Powell, "An efficient method for finding the minimum of a function of several variables without calculating derivatives," *The Computer Journal*, vol. 7, pp. 155–162, February 1964.

[23] N. Hansen and A. Ostermeier, "Adapting arbitrary normal mutation distributions in evolution strategies: The covariance matrix adaptation," in *Proceedings of the 1996 IEEE International Conference on Evolutionary Computation*, pp. 312–317, 1996.

[24] N. Hansen, "The CMA evolution strategy: a comparing review," in *Towards a new evolutionary computation. Advances on estimation of distribution algorithms* (J. Lozano, P. Larranaga, I. Inza, and E. Bengoetxea, eds.), pp. 75–102, Springer, 2006.

[25] N. Hansen, "The cma evolution strategy: A tutorial," tech. rep., Laboratoire de Recherche en Informatique, 2010.

[26] M. Dorigo, G. D. Caro, and L. M. Gambardella, "Ant algorithms for discrete optimization," *Artificial Life*, vol. 5, pp. 137–172, 1999.

[27] O. Kovářík, "Ant colony optimization for continuous problems," Master's thesis, Fakulta elektrotechnická, České vysoké učení technické v Praze, Karlovo nám. 13, 121 35 Praha 2, 2006.

[28] G. Bilchev and I. C. Parmee, "The ant colony metaphor for searching continuous design spaces," *Lecture Notes in Computer Science*, vol. 993, pp. 25–39, 1995.

[29] G. Venturini and M. Slimane, "On how pachycondyla apicalis ants suggest a new search algorithm," *Future Generation Computer Systems*, vol. 16, pp. 937–946, 2000.

[30] M. Kong and P. Tian, "Introducing a binary ant colony optimization," in *ANTS Workshop*, pp. 444–451, 2006.

[31] K. Socha, "Aco for continuous and mixed-variable optimization," in *ANTS Workshop*, pp. 25–36, 2004.

[32] K. Socha and M. Dorigo, "Ant colony optimization for continuous domains," *European Journal of Operational Research*, vol. 185, pp. 1155–1173, March 2008.

[33] M. Kong and P. Tian, "A direct application of ant colony optimization to function optimization problem in continuous domain," in *ANTS Workshop*, pp. 324–331, 2006.

[34] *Particle swarm optimization*, vol. 4, August 2002.

[35] *A modified particle swarm optimizer*, 1998.

[36] R. Mendes, J. Kennedy, and J. Neves, "The fully informed particle swarm: Simpler, maybe better," *IEEE Transactions on Evolutionary Computation*, vol. 8, pp. 204–210, June 2004.

[37] M. Clerc and J. Kennedy, "The particle swarm: Explosion, stability, and convergence in a multi-dimensional complex space," *IEEE Transactions on Evolutionary Computation*, pp. 58—-73, February 2002.

[38] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley Professional, January 1989.

[39] M. Mitchell, *An Introduction to Genetic Algorithms (Complex Adaptive Systems).* The MIT Press, February 1998.

[40] R. Storn and K. Price, "Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces," *J. of Global Optimization*, vol. 11, no. 4, pp. 341–359, 1997.

[41] R. Storn, "On the usage of differential evolution for function optimization," in *in NAFIPS'96*, pp. 519–523, IEEE, 1996.

[42] O. Hrstka and A. Kučerová, "Improvements of real coded genetic algorithms based on differential operators preventing premature convergence," *Adv. Eng. Softw.*, vol. 35, no. 3-4, pp. 237–246, 2004.

[43] S. Baluja, "Population based incremental learning: A method for integrating genetic search based function optimization and competitve learning," tech. rep., School of Computer Science, Carnegie Mellon University, 1994.

[44] M. Sebag and A. Ducoulombier, "Extending population-based incremental learning to continuous search spaces," in *PPSN V: Proceedings of the 5th International Conference on Parallel Problem Solving from Nature*, (London, UK), pp. 418–427, Springer-Verlag, 1998.

[45] C.-F. Juang and Y.-C. Liou, "On the hybrid of genetic algorithm and particle swarm optimization for evolving recurrent neural network," in *2004 IEEE International Joint Conference on Neural Networks, 2004. Proceedings*, pp. 2285 – 2289 vol.3, 2004.

[46] J. J. Moré, B. S. Garbow, and K. E. Hillstrom, "Testing unconstrained optimization software," *ACM Trans. Math. Softw.*, vol. 7, no. 1, pp. 17–41, 1981.

[47] K. A. D. Jong, *An analysis of the behavior of a glass of genetic adaptive systems.* PhD thesis, University of Michigan, 1975.

[48] E. Easom, "A survey of global optimization techniques," m.eng. thesis, University of Louisville, Louisville, KY, 1990.

[49] H. H. Rosenbrock, "An automatic method for finding the greatest or least value of a function," *The Computer Journal*, vol. 3, pp. 175–184, March 1960.

[50] Y.-W. Shang and Y.-H. Qiu, "A note on the extended rosenbrock function," *Evol. Comput.*, vol. 14, no. 1, pp. 119–126, 2006.

[51] A. Neumaier, "Some hard global optimization test problems." `http://www.mat.univie.ac.at/~neum/glopt/my_problems.html`, May 2010.

[52] D. Ackley, "An empirical study of bit vector function optimizacion," in *Genetic Algorithms and Simulated Annealing*, pp. 170–215, 1987.

[53] F. K. Branin, "A widely convergent method for finding multiple solutions of simultaneous nonlinear equations," *IBM J. Res. Develop.*, pp. 504–522, September 1972.

[54] A. A. Goldstein and I. F. Price, "On descent from local minima," *Math. Comput*, vol. 25, no. 115, 1971.

[55] T. Back, D. B. Fogel, and Z. Michalewicz, eds., *Handbook of Evolutionary Computation*. Bristol, UK, UK: IOP Publishing Ltd., 1997.

[56] H. Bersini, M. Dorigo, S. Langerman, G. Seront, and L. M. Gambardella, "Results of the first international contest on evolutionary optimisation (1st iceo)," in *International Conference on Evolutionary Computation*, pp. 611–615, 1996.

[57] Z. Michalewicz, *Genetic algorithms + data structures = evolution programs (2nd, extended ed.)*. New York, NY, USA: Springer-Verlag New York, Inc., 1994.

[58] D. Whitley, D. Garrett, and J. paul Watson, "Quad search and hybrid genetic algorithms," in *In Genetic and Evolutionary Computation - GECCO 2003*, pp. 1469–1480, Springer, 2003.

[59] L. A. Rastrigin, "Extremal control systems," *Nauka*, 1974.

[60] H.-P. Schwefel, *Numerical Optimization of Computer Models*. New York, NY, USA: John Wiley & Sons, Inc., 1981.

[61] D. M. Gay, "Some convergence properties of broyden's method," Working Paper 175, National Bureau of Economic Research, April 1977.

[62] T. Bäck, *Evolutionary algorithms in theory and practice: evolution strategies, evolutionary programming, genetic algorithms*. Oxford, UK: Oxford University Press, 1996.

Appendix A

Changes to the JCool source code

The JCool environment by Martin Hvizdoš is a well designed piece of software. While working with it I enjoyed an easy to understand architecture and intuitive interfaces. However as I implemented first algorithms and test functions I discovered few incidentally overlooked mistakes that needed to be corrected. Some of these were rather of a minor importance but some would manifest themselves as fatal if left uncorrected.

First and perhaps one of the two most crucial corrections was carried out on a routine for numerical computation of the Hessian matrix. JCool employs this routine to supply Hessian matrix when requested for a function without an analytic formula. In this case the JCool environment uses the *Strategy* design pattern which delegates the call for evaluation of the Hessian matrix to this routine. At present the JCool framework offers only the central difference method for computing numerical Hessian matrix.

The problem was that the function value was obtained all the time only for the given point and not for the updated point in the forward and backward direction. See Listing A.1, lines 16 and 18 where the original value of `point` was used instead of the changed `Point.at(p[])`. This error was introduced most likely due to a refactoring.

Next serious problem became evident while I was working with algorithms using a set of points – population based methods. These sometimes use a set of `ValuePoint`s to store current state of the algorithm. However whenever any of these algorithms needed to get a **sorted** set of these pairs of value–point, they behaved oddly enough to get my attention. The set after sorting were *not quite* sorted. Although there was no obvious error on the first glance, when examining the sorted values I noticed that values differing in range $\delta \in \langle 0, 1)$ were sometimes garbled.

All was resolved when the `ValuePoint.compareTo` was examined – as we can see in Listing A.2, line 8, the original code tried to simulate the *signum* function applied on the difference between sorted values. However the typecasting to `int` did not bring desired effect as this rounds numbers with difference of $-1 < \delta < 0$ to zero. Obviously, close values yielding in this difference were then treated as equal.

The rest of the changes made is just extension of the original functionality. Listing A.3 shows added static methods `Point.random` which supplies a new static instance of `point` initialized randomly in the given bounds (optional) and of given dimension (mandatory). If no bounds were given then the full range of `Double` is used.

Last addition to the JCool environment I contributed is the full pivoted Gauss–Jordan elimination (Wilhelm Jordan, 1887) for square matrices as presented by [4]. The code was adapted for square matrix ($n \times n$) and a vector (i.e. $n \times 1$ matrix) and to use indices starting

```
1  public class CentralDifferenceHessian implements NumericalHessian {
2    public Hessian hessianAt(ObjectiveFunction function, Point point) {
3      int dimension       = function.getDimension();
4      double[][] hessian  = new double[dimension][dimension];
5      double[] h          = new double[dimension];
6      double[] fplus      = new double[dimension];
7      double[] fminus     = new double[dimension];
8      double tolerance    = Math.pow(MachineAccuracy.EPSILON, 0.25);
9      double[] p          = point.toArray();
10     double valueAtPoint = function.valueAt(point);
11     double xh, oldx, oldy, fxx, tH;
12     for (int i = 0; i < dimension; i++) {
13       h[i] = tolerance * (Math.abs(p[i]) + MachineAccuracy.EPSILON);
14       oldx = p[i];
15       p[i] = oldx + h[i];
16       fplus[i] = function.valueAt(Point.at(p));
17       p[i] = oldx - h[i];
18       fminus[i] = function.valueAt(Point.at(p));
19       p[i] = oldx;
20     }
21     for (int i = 0; i < dimension; i++) {
22       hessian[i][i] = h[i] * h[i];
23       for (int j = 0; j < dimension; j++) {
24         tH = h[i] * h[j];
25         hessian[i][j] = tH;
26         hessian[j][i] = tH;
27       }
28     }
29     for (int i = 0; i < dimension; i++) {
30       for (int j = 0; j < dimension; j++) {
31         if (i == j)
32           hessian[i][j] = (fplus[i] + fminus[j] - 2 * valueAtPoint) / hessian[i][j];
33         else {
34           oldx = p[i];
35           oldy = p[j];
36           p[i] = oldx + h[i];
37           p[j] = oldy - h[j];
38           fxx = function.valueAt(Point.at(p));
39           p[i] = oldx;
40           p[j] = oldy;
41           hessian[i][j] = (fplus[i] + fminus[j] - valueAtPoint - fxx) / hessian[i][j];
42         }
43       }
44     }
45     for (int i = 0; i < dimension; i++) {
46       for (int j = 0; j < dimension; j++) {
47         tH = (hessian[i][j] + hessian[j][i]) / 2.0;
48         hessian[i][j] = tH;
49         hessian[j][i] = tH;
50       }
51     }
52     return Hessian.valueOf(hessian);
53   }
54 }
```

Figure A.1: Final code of `CentralDifferenceHessian.hessianAt` routine.

from `0`, see Listing A.4. For full details of the method please refer to [4].

```
1  public class ValuePoint implements Comparable<ValuePoint> {
2    private final Point   point;
3    private final double  value;
4    ...
5    public int compareTo(ValuePoint o) {
6      //return (int) (o.getValue() - this.getValue());
7      if (this.getValue() > o.getValue())
8        return 1;
9      if (this.getValue() < o.getValue())
10       return -1;
11     return 0;
12   }
13 }
```

Figure A.2: Corrected `ValuePoint.compareTo`.

```
1  public class Point {
2    ...
3    private Point(double[] array) {
4      this(array.length);
5      System.arraycopy(array, 0, this.array, 0, this.array.length);
6      StringBuilder builder = new StringBuilder();
7      builder.append("[");
8      for (int i = 0; i < array.length; i++) {
9        builder.append(array[i]);
10       if(i < array.length - 1) {
11         builder.append(",");
12       }
13     }
14     this.toString = builder.append("]").toString();
15   }
16   public static final Point random(int dimension) {
17     return Point.random(dimension, -Double.MAX_VALUE, Double.MAX_VALUE);
18   }
19   public static final Point random(int dimension, double min, double max) {
20     double[] array = new double[dimension];
21     for (int i = 0; i < array.length; i++)
22       array[i] = Math.random() * (max - min) + min;
23     return new Point(array);
24   }
25   ...
26 }
```

Figure A.3: Extended class `Point`.

```
1  public class VectorAlgebra {
2    public static double[] GaussJordanElimination(double[][] A, double[] b, int dimension)
         {
3      double[] x = new double[dimension];
4      System.arraycopy(b, 0, x, 0, dimension);
5      int[] indexColumn = new int[dimension];
6      int[] indexRow    = new int[dimension];
7      int[] indexPivot  = new int[dimension];
8      int row = 0;
9      int col = 0;
10     Arrays.fill(indexPivot, 0);
11     for (int i = 0; i < dimension; i++) {
12       double big = 0.0;
13       for (int j = 0; j < dimension; j++)
14         if (indexPivot[j] != 1)
15           for (int k = 0; k < dimension; k++)
16             if (indexPivot[k] == 0)
17               if (Math.abs(A[j][k]) >= big) {
18                 big = Math.abs(A[j][k]);
19                 row = j;
20                 col = k;
21               }
22       if (row != col) {
23         for (int j = 0; j < dimension; j++) {
24           double temp = A[row][j];
25           A[row][j] = A[col][j];
26           A[col][j] = temp;
27         }
28         double swap = x[row];
29         x[row] = x[col];
30         x[col] = swap;
31       }
32       indexRow[i] = row;
33       indexColumn[i] = col;
34       double pivotInverse = 1.0 / A[col][col];
35       A[col][col] = 1.0;
36       for (int j = 0; j < dimension; j++)
37         A[col][j] *= pivotInverse;
38       x[col] *= pivotInverse;
39       for (int j = 0; j < dimension; j++)
40         if (j != col) {
41           double tmp = A[j][col];
42           A[j][col] = 0.0;
43           for (int k = 0; k < dimension; k++)
44             A[j][k] -= A[col][k] * tmp;
45           x[j] -= x[col] * tmp;
46         }
47     }
48     for (int i = dimension - 1; i >= 0; i--)
49       if (indexRow[i] != indexColumn[i])
50         for (int j = 0; j < dimension; j++) {
51           double swap = A[j][indexRow[i]];
52           A[j][indexRow[i]] = A[j][indexColumn[i]];
53           A[j][indexColumn[i]] = swap;
54         }
55     return x;
56   }
57 }
```

Figure A.4: Added code of Gauss–Jordan elimination for square matrices.

Appendix B

Default method parameters

The default parameters of implemented algorithms follow. Each optimization algorithm had the starting point (or staring points if population based) bounded by constant limits on the coordinates. These were

$$x_{i_{\mathrm{MIN}}} = \max(-10, f_{\mathrm{bounds}_{\mathrm{MIN}}})$$
$$x_{i_{\mathrm{MAX}}} = \min(10, f_{\mathrm{bounds}_{\mathrm{MAX}}})$$

and the starting point was then randomly generated by `Point.random(`$\vec{x}_{\mathrm{MIN}}$`,` $\vec{x}_{\mathrm{MAX}}$`,` d`)` using uniform distribution withing given bounds. d is the number of dimensions, $i = 1, ..., d$. f_{bounds} are the function bounds if the function is bounded.

B.1 QN

Update formula:	Broyden—Fletcher—Goldfarb—Shanno
ϕ for Broyden Family formula:	0.5

B.2 OS

Stochastic search:	$false$

B.3 CMA-ES

Initial standard deviation	σ:	0.2
Population size	λ:	$4 + \lfloor 3 \ln d \rfloor$
Number of parents	μ:	$\lfloor \frac{\lambda}{2} \rfloor$
Recombination weights	w_i:	$\frac{\ln(\frac{\lambda}{2}+0.5)-\ln i}{\sum_{j=1}^{\mu}(\ln(\frac{\lambda}{2}+0.5)-\ln j)}$ $i = 1, ..., \mu$
Variance effective selection mass	μ_{eff}:	$\frac{1}{\sum_{i=1}^{\mu} w_i^2}$
Learning rate for the cumulation for the step-size control	c_σ:	$\frac{\mu_{\text{eff}}+2}{d+\mu_{\text{eff}}+5}$
Damping parameter for step-size update	d_σ:	$1 + 2\max\left(0, \sqrt{\frac{\mu_{\text{eff}}-1}{d+1}} - 1\right) + c_\sigma$
Learning rate for cumulation for the rank-one update of the covariance matrix	c_c:	$\frac{4+\frac{\mu_{\text{eff}}}{d}}{d+4+2\frac{\mu_{\text{eff}}}{d}}$
Learning rate for the rank-one update of the covariance matrix update	c_1:	$\frac{2}{(d+1.3)^2+\mu_{\text{eff}}}$
Learning rate for the rank-μ update of the covariance matrix update	c_μ:	$\min\left(1 - c_1, \alpha_\mu \frac{\mu_{\text{eff}}-2+\frac{1}{\mu_{\text{eff}}}}{(d+2)^2+\frac{\alpha_\mu \mu_{\text{eff}}}{2}}\right)$ $\alpha_\mu = 2$

B.4 CACO

Number of directions:	20
Initial pheromone level:	0.1
Minimal pheromone level:	0.1
Pheromone increase:	0.3
Pheromone evaporation:	0.3
Initial search radius:	50.0
Search radius decrease speed:	0.8
Generations before radius decrease:	30

B.5 API

Population size:	20
Move nest to the best point every:	30 generations
Number of hunting sites for each ant:	3
Number of iterations without solution before removing hunting site:	5

B.6 AACA

Population size:		20
Encoding length:		16 bits
Evaporation factor	λ:	0.8
Pheromone index	β:	0.1
Cost index	δ:	0.0

B.7 ACO*

Population size:		60
# of ants replaced each iteration:		60
Deviation parameter	ω:	0.8
Convergence parameter	σ:	0.4
Use standard deviation:		*true*
		(AVG used otherwise)
Force diversity:		$false$
		Limits the neighbourhood for the deviation computation
Neighborhood size for force diversity:		0.1

B.8 DACO

Population size:		40
Evaporation factor	λ:	0.5

B.9 PSO

Number of particles:		20
Velocity update method:		original
Maximal particle velocity	$\vec{v}_{\max}$:	0.5
Cognitive acceleration coefficient	ϕ_1:	0.5
Social acceleration coefficient	ϕ_2:	0.5
Neighbourhood distance	m:	2
χ parameter	κ:	1.0

B.10 DE

Population size:	16
Crossover rating:	0.85
Mutation constant:	0.6

B.11 SADE

Pool size:	20
Radioactivity:	0.2
Local radioactivity:	0.25
Mutation rate:	0.5
Mutagen rate:	400.0
Crossover rate:	0.2

B.12 PBIL

Population size:	50
Encoding length:	16 bits
Learning rate:	0.5
Negative learning rate:	0.3
Mutation probability:	0.3
Mutation shift:	0.5

B.13 HGAPSO

Number of particles/population size :		80
Elite ratio:		0.5
Number of PSO iterations:		5
Cognitive acceleration coefficient	ϕ_1:	1.0
Social acceleration coefficient	ϕ_2:	1.0
Maximal particle velocity	$\vec{v}_{\max}$:	2.0
Parameter	χ:	0.8
Mutation probability:		0.1

Appendix C

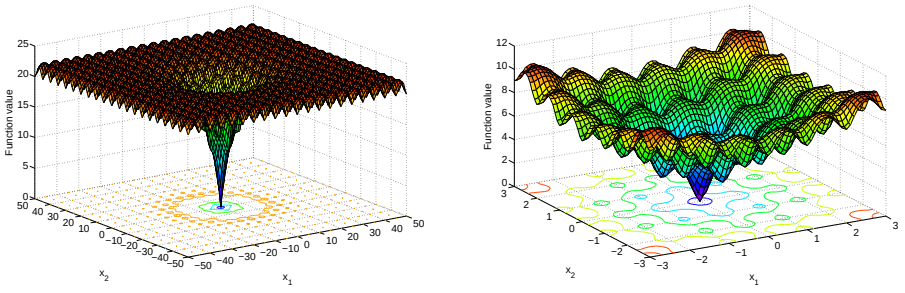
Test functions detailed

In this chapter details of test functions implemented in JCool environment are presented. Not all these functions were used when benchmarking the algorithms this thesis examines but still they were included in the implemented test set as they are well known or introduce interesting features to the optimization tasks.

Many of these functions are generalized to offer high dimensional problems, have multiple formula or are parametrized. This set will support any researcher in the need of thorough benchmarking of their algorithms.

C.1 Ackley's path

Dimensions:	$d =$	n
Formula:	$f(\vec{x}) =$	$-20e^{-0.2\sqrt{\frac{\sum_{i=1}^{d} x_i^2}{d}}} - e^{\frac{\sum_{i=1}^{d} \cos(2\pi x_i)}{d}} + 20 + e$
Function bounds:		unconstrained
Local minima:		large number of local minima evenly distributed
Global minimum:	$f(\vec{x}_{\min}) =$	0
	$\vec{x}_{\min} =$	$(0, ..., 0)$

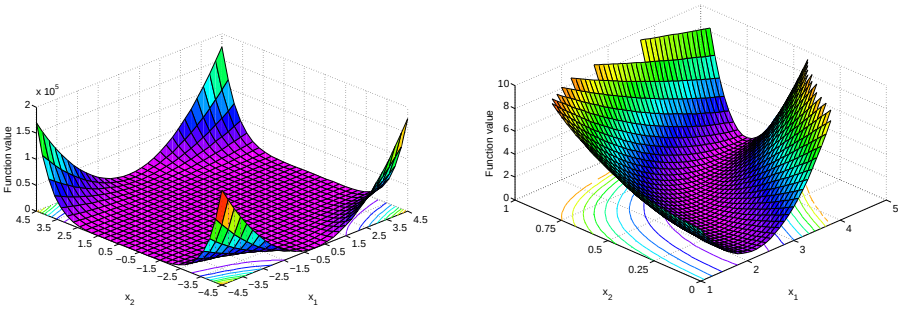


(a) Full scale (b) Detail of the global minimum

Figure C.1: Ackley's path for $d = 2$.

C.2 Beale function

Dimensions:	$d =$	2
Formula:	$f(x_1, x_2) =$	$(1.5 - x_1(1 - x_2))^2 + (2.25 - x_1(1 - x_2^2))^2 + +(2.625 - x_1(1 - x_2^3))^2$
Function bounds:	$-4.5 \leqq x_i \leqq 4.5$	$i = 1, ..., d$
Local minima:		unimodal
Global minimum:	$f(\vec{x}_{\min}) =$	0
	$\vec{x}_{\min} =$	$(3, 0.5)$

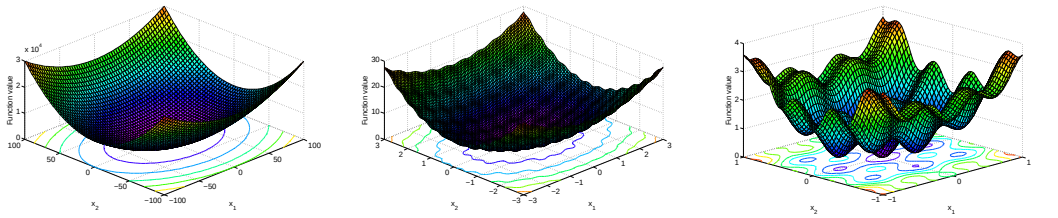


(a) Full scale (b) Detail of the global minimum

Figure C.2: Beale function.

C.3 Bohachevsky's function

Dimensions:	$d =$	2
Formula:	$f(x_1, x_2)_1 =$	$x_1^2 + 2x_2^2 - 0.3\cos(3\pi x_1) - 0.4\cos(4\pi x_2) + 0.7$
	$f(x_1, x_2)_2 =$	$x_1^2 + 2x_2^2 - 0.3\cos(3\pi x_1)\cos(4\pi x_2) + 0.3$
	$f(x_1, x_2)_3 =$	$x_1^2 + 2x_2^2 - 0.3\cos(3\pi x_1 + 4\pi x_2) + 0.3$
Function bounds:		unconstrained
Local minima:		several symmetrically spaced local minima
Global minimum:	$f(\vec{x}_{\min}) =$	0
	$\vec{x}_{\min} =$	$(0, ..., 0)$



(a) Full scale (b) Medium scale (c) Detail of the global minimum

Figure C.3: Bohachevsky's function.

C.4 Booth's function

Dimensions:	$d =$	2
Formula:	$f(x_1, x_2) =$	$(x_1 + 2x_2 - 7)^2 + (2x_1 + x_2 - 5)^2$
Function bounds:		unconstrained
Local minima:		unimodal
Global minimum:	$f(\vec{x}_{\min}) =$	0
	$\vec{x}_{\min} =$	$(1, 3)$

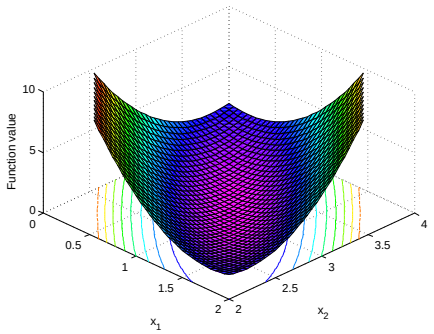


Figure C.4: Booth's function.

C.5 Branin's function

Dimensions:	$d =$	2
Formula:	$f(x_1, x_2) =$	$\left(x_2 - \frac{5.1}{4\pi^2}x_1^2 + \frac{5}{\pi}x_1 - 6\right)^2 + 10\left(1 - \frac{1}{8\pi}\right)\cos(x_1) + 10$
Function bounds:		$-5 \leqq x_1 \leqq 10$
		$0 \leqq x_2 \leqq 15$
Local minima:		no local minima
Global minimum:	$f(\vec{x}_{\min}) =$	0.397887
	$\vec{x}_{\min_1} =$	$(-\pi, 12.275)$
	$\vec{x}_{\min_2} =$	$(\pi, 2.275)$
	$\vec{x}_{\min_3} =$	$(9.42478, 2.475)$

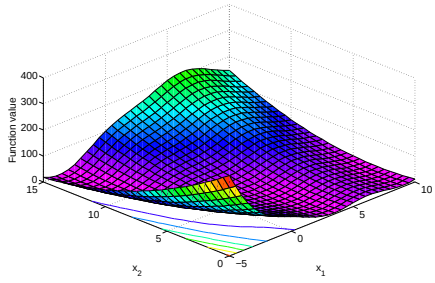


Figure C.5: Branin's function.

C.6 Colville's function

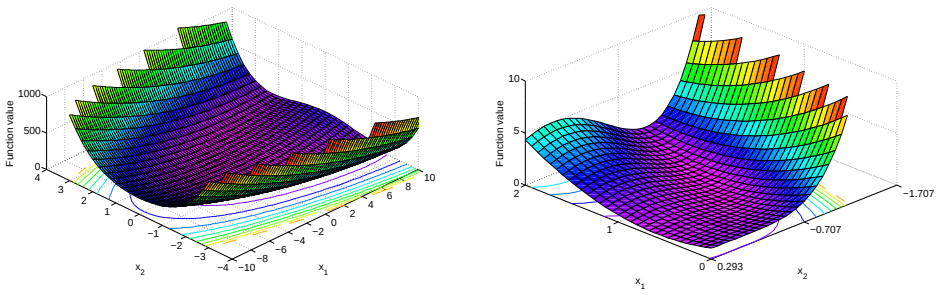
Dimensions:	$d =$	4
Formula:	$f(x_1, x_2, x_3, x_4) =$	$100(x_1^2 - x_2)^2 + (x_1 - 1)^2 + (x_3 - 1)^2 + 90(x_3^2 - x_4)^2 + +10.1((x_2 - 1)^2 + (x_4 - 1)^2) + 19.8x_2^{-1}(x_4 - 1)$
Function bounds:	$-10 \leqq x_i \leqq 10$	$i = 1, ..., d$
Local minima:		unimodal
Global minimum:	$f(\vec{x}_{\min}) =$	0
	$\vec{x}_{\min} =$	$(1, 1, 1, 1)$

C.7 De Jong's (sphere) function

Dimensions:	$d =$	n
Formula:	$f(\vec{x}) =$	$\sum_{i=1}^{d} x_i^2$
Function bounds:		unconstrained
Local minima:		unimodal
Global minimum:	$f(\vec{x}_{\min}) =$	0
	$\vec{x}_{\min} =$	$(0, ..., 0)$

C.8 Dixon–Price function

Dimensions:	$d =$	n
Formula:	$f(\vec{x}) =$	$(x_1 - 1)^2 + \sum_{i=2}^{d} i(2x_i^2 - x_{i-1})^2$
Function bounds:	$-10 \leqq x_i \leqq 10$	$i = 1, ..., d$
Local minima:		unimodal
Global minimum:	$f(\vec{x}_{\min}) =$	0

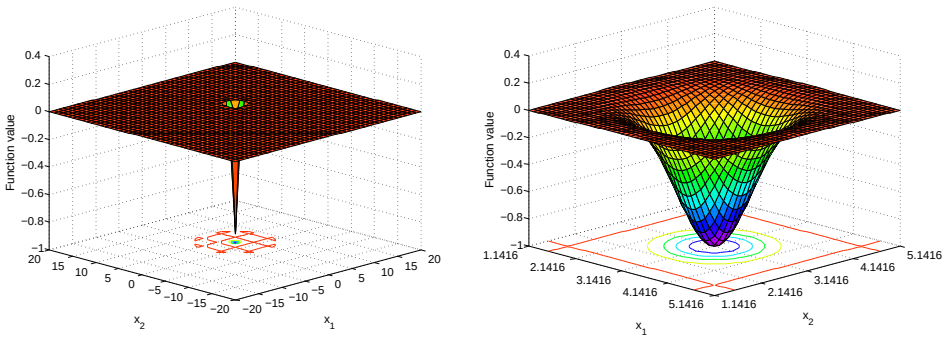


(a) Full scale (b) Detail of the global minimum

Figure C.6: Dixon–Price function.

C.9 Easom's function

Dimensions:	$d =$	2
Formula:	$f(x_1, x_2) =$	$-\cos(x_1)\cos(x_2)e^{-(x_1-\pi)^2-(x_2-\pi)^2}$
Function bounds:		unconstrained
Local minima:		unimodal
Global minimum:	$f(\vec{x}_{\min}) =$	-1
	$\vec{x}_{\min} =$	(π, π)



(a) Full scale (b) Detail of the global minimum

Figure C.7: Easom's function.

C.10 Goldstein–Price function

Dimensions:	$d =$	2
Formula:	$f(x_1, x_2) =$	$(1 + (x_1 + x_2 + 1)^2 \cdot$
		$\cdot(19 - 14x_1 + 3x_1^2 - 14x_2 + 6x_1x_2 + 3x_2^2))$
		$(30 + (2x_1 - 3x_2)^2 \cdot$
		$\cdot(18 - 32x_1 + 12x_1^2 + 48x_2 - 36x_1x_2 + 27x_2^2))$
Function bounds:	$-2 \leqq x_i \leqq 2$	$i = 1, ..., d$
Local minima:		several local minima near the global optimum
Global minimum:	$f(\vec{x}_{\min}) =$	3
	$\vec{x}_{\min} =$	$(0, -1)$

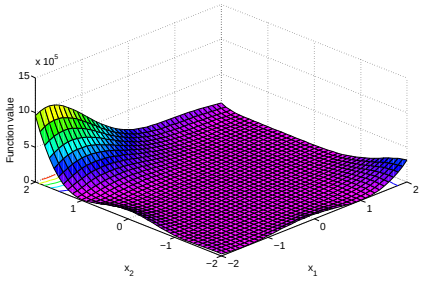
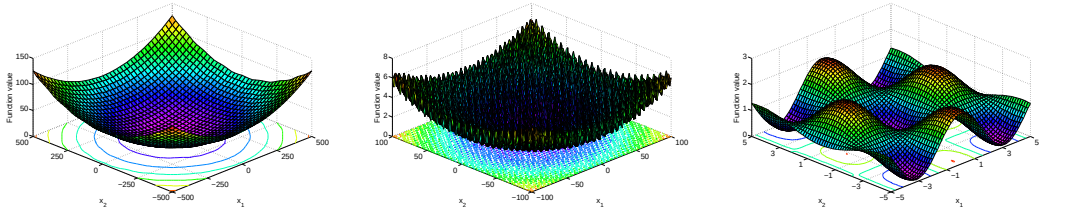


Figure C.8: Goldstein–Price function.

C.11 Griewangk's function

Dimensions:	$d =$	n
Formula:	$f(\vec{x}) =$	$\sum_{i=1}^{d} \frac{x_i^2}{4000} - \prod_{i=1}^{d} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1$
Function bounds:		unconstrained
Local minima:		large number of regularly located local minima
Global minimum:	$f(\vec{x}_{\min}) =$	0
	$\vec{x}_{\min} =$	$(0, ..., 0)$



(a) Full scale (b) Medium scale (c) Detail of the global minimum

Figure C.9: Griewangk's function for $d = 2$.

C.12 Hartmann's functions

Dimensions:	$d_{3,4} =$	3
	$d_{6,4} =$	6
Formula:	$f(\vec{x})_{3,4} =$	$-\sum_{i=1}^{4} \alpha_i e^{-\sum_{j=1}^{3} \mathbf{A}_{i,j}(x_j - \mathbf{P}_{i,j})^2}$
	$f(\vec{x})_{6,4} =$	$-\sum_{i=1}^{4} \alpha_i e^{-\sum_{j=1}^{6} \mathbf{B}_{i,j}(x_j - \mathbf{Q}_{i,j})^2}$
Function bounds:	$0 \leqq x_i \leqq 1$	$i = 1, ..., d$
Local minima:		4 local minima for $d = 3$
		6 local minima for $d = 6$
Global minimum:	$f(\vec{x}_{\min_{3,4}}) =$	-3.86278
	$\vec{x}_{\min_{3,4}} =$	$(0.1146, 0.5557, 0.8526)$
	$f(\vec{x}_{\min_{6,4}}) =$	-3.32237
	$\vec{x}_{\min_{6,4}} =$	$(0.2017, 0.1500, 0.4769, 0.2753, 0.3117, 0.6573)$

$$\vec{\alpha} = \begin{pmatrix} 1.0 & 1.2 & 3.0 & 3.2 \end{pmatrix}$$

$$\mathbf{A} = \begin{pmatrix} 3.0 & 10.0 & 30.0 \\ 0.1 & 10.0 & 35.0 \\ 3.0 & 10.0 & 30.0 \\ 0.1 & 10.0 & 35.0 \end{pmatrix}$$

$$\mathbf{B} = \begin{pmatrix} 10.00 & 3.00 & 17.00 & 3.05 & 1.70 & 8.00 \\ 0.05 & 10.00 & 17.00 & 0.10 & 8.00 & 14.00 \\ 3.00 & 3.50 & 1.70 & 10.00 & 17.00 & 8.00 \\ 17.00 & 8.00 & 0.05 & 10.00 & 0.10 & 14.00 \end{pmatrix}$$

$$\mathbf{P} = \begin{pmatrix} 0.6890 & 0.1170 & 0.2673 \\ 0.4699 & 0.4387 & 0.7470 \\ 0.1091 & 0.8732 & 0.5547 \\ 0.0381 & 0.5743 & 0.8828 \end{pmatrix}$$

$$\mathbf{Q} = \begin{pmatrix} 0.1312 & 0.1696 & 0.5569 & 0.0124 & 0.8283 & 0.5886 \\ 0.2329 & 0.4135 & 0.8307 & 0.3736 & 0.1004 & 0.9991 \\ 0.2348 & 0.1451 & 0.3522 & 0.2883 & 0.3047 & 0.6650 \\ 0.4047 & 0.8828 & 0.8732 & 0.5743 & 0.1091 & 0.0381 \end{pmatrix}$$

C.13 Himmelblau function

Dimensions:	$d =$	2
Formula:	$f(x_1, x_2) =$	$(x_1^2 + x_2 - 11)^2 + (x_1 + x_2^2 - 7)^2$
Function bounds:	$-6 \leqq x_i \leqq 6$	$i = 1, ..., d$
Local minima:		no local minima
Global minimum:	$f(\vec{x}_{\min}) =$	0
	$\vec{x}_{\min_1} =$	$(3, 2)$
	$\vec{x}_{\min_2} =$	$(-2.805118, 3.131312)$
	$\vec{x}_{\min_3} =$	$(-3.779310, -3.283185)$
	$\vec{x}_{\min_4} =$	$(3.584428, -1.848126)$

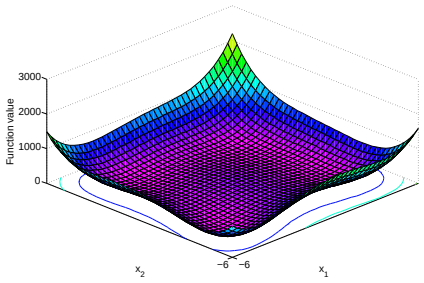


Figure C.10: Himmelblau function.

C.14 Hump's function

Dimensions:	$d =$	2
Formula:	$f(x_1, x_2) =$	$1.0316285 + 4x_1^2 - 2.1x_1^4 + \frac{x_1^6}{3} + x_1x_2 - 4x_2^2 + 4x_2^4$
Function bounds:		$-1.9 \leqq x_1 \leqq 1.9, \quad -1.1 \leqq x_2 \leqq 1.1$
Local minima:		4 symmetric local minima
Global minimum:	$f(\vec{x}_{\min}) =$	0
	$\vec{x}_{\min_1} =$	$(0.0898, -0.7126)$
	$\vec{x}_{\min_2} =$	$(-0.0898, 0.7126)$

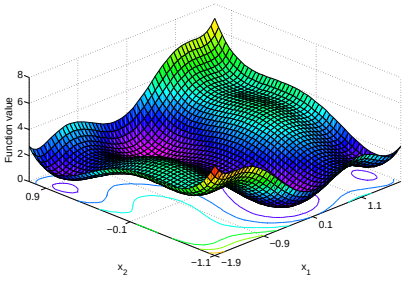


Figure C.11: Hump's function.

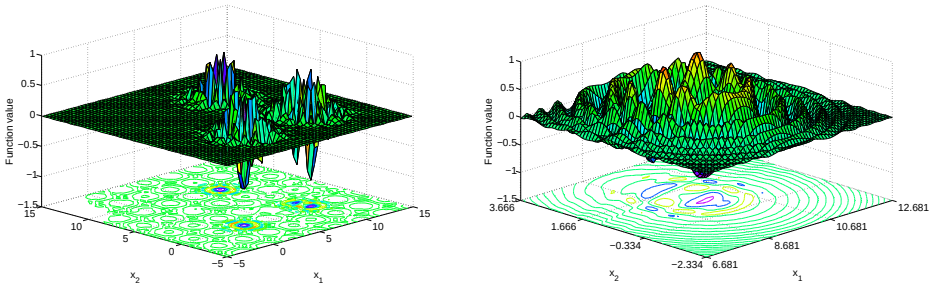
C.15 Langerman's function

Dimensions:	$d =$	$1, ..., 10$
Formula:	$f(\vec{x}) =$	$-\sum_{i=1}^{m} c_i e^{-\frac{\sum_{j=1}^{d}(x_j - \mathbf{A}_{i,j})^2}{\pi}} \cos\left(\pi \sum_{j=1}^{d}(x_j - \mathbf{A}_{i,j})^2\right)$
		$m = 1, ..., 5$
Function bounds:		unconstrained
Local minima:		several local minima randomly in the search space and several around the global minimum
Global minimum:	$f(\vec{x}_{\min_{d=2}}) =$	-1.08093846
	$\vec{x}_{\min_{d=2}} =$	$(9.681, 0.666)$
	$f(\vec{x}_{\min_{d=5}}) =$	-0.96499991
	$\vec{x}_{\min_{d=5}} =$	$(8.074, 8.777, 3.467, 1.863, 6.707)$

Values referenced by [62] were used:

$$\mathbf{A} = \begin{pmatrix} 9.681 & 0.667 & 4.783 & 9.095 & 3.517 & 9.325 & 6.544 & 0.211 & 5.122 & 2.020 \\ 9.400 & 2.041 & 3.788 & 7.931 & 2.882 & 2.672 & 3.568 & 1.284 & 7.033 & 7.374 \\ 8.025 & 9.152 & 5.114 & 7.621 & 4.564 & 4.711 & 2.996 & 6.126 & 0.734 & 4.982 \\ 2.196 & 0.415 & 5.649 & 6.979 & 9.510 & 9.166 & 6.304 & 6.054 & 9.377 & 1.426 \\ 8.074 & 8.777 & 3.467 & 1.863 & 6.708 & 6.349 & 4.534 & 0.276 & 7.633 & 1.567 \end{pmatrix}$$

$$\vec{c} = \begin{pmatrix} 0.806 & 0.517 & 0.100 & 0.908 & 0.965 \end{pmatrix}$$

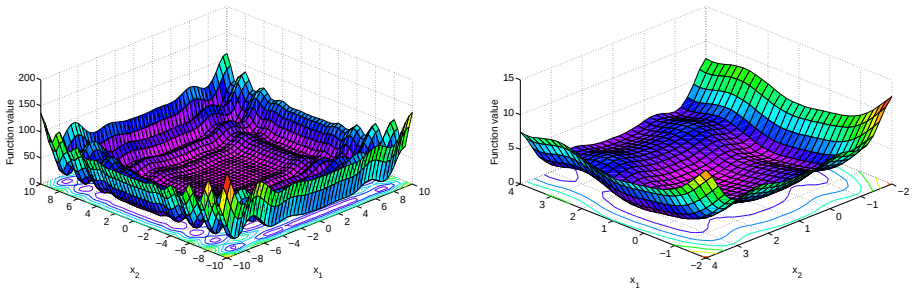


(a) Full scale (b) Detail of the global minimum

Figure C.12: Langerman's function.

C.16 Levy's function

Dimensions:	$d =$	n
Formula:	$f(\vec{x}) =$	$\sin^2(\pi y_1) + \sum_{i=1}^{d-1}((y_i-1)^2(1+10\sin^2(\pi y_i+1))+$
		$+(y_d-1)^2(1+10\sin^2(2\pi y_n))$
		$y_i = 1 + \frac{x_i-1}{4}$
Function bounds:	$-10 \leqq x_i \leqq 10$	$i = 1, ..., d$
Local minima:		several local minima
Global minimum:	$f(\vec{x}_{\min}) =$	0
	$\vec{x}_{\min} =$	$(1, ..., 1)$

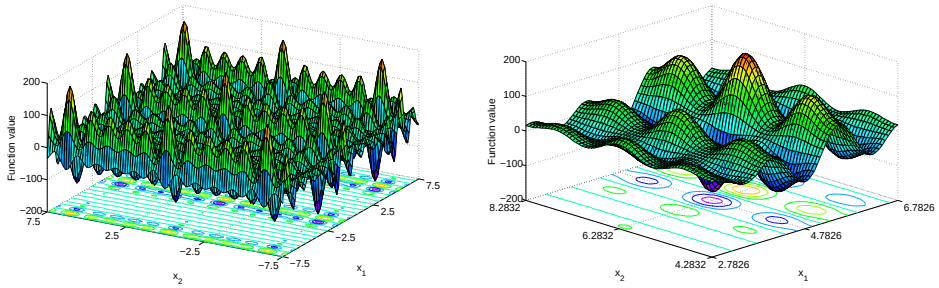


(a) Full scale (b) Detail of the global minimum

Figure C.13: Levy's function.

C.17 Levy function no. 3

Dimensions:	$d =$	2
Formula:	$f(x_1, x_2) =$	$\sum_{i=1}^{4}(i\cos((i+1)x_1+i))\sum_{i=1}^{4}(i\cos((i+1)x_2)+i)$
Function bounds:	$-10 \leqq x_i \leqq 10$	$i = 1, ..., d$
Local minima:		around 760 regularly distributed local minima
Global minimum:	18: $f(\vec{x}_{\min}) =$	-174.542

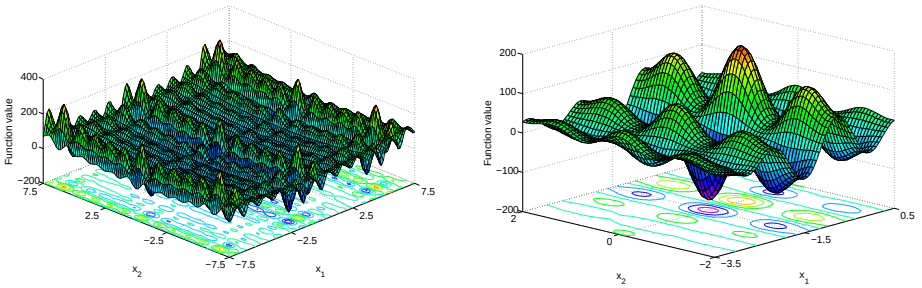


(a) Full scale (b) Detail of one of the global minima

Figure C.14: Levy function no. 3.

C.18 Levy function no. 5

Dimensions:	$d =$	2
Formula:	$f(x_1, x_2) =$	$\sum_{i=1}^{4}(i\cos((i+1)x_1+i))\sum_{i=1}^{4}(i\cos((i+1)x_2)+i)$
Function bounds:	$-10 \leqq x_i \leqq 10$	$i = 1, ..., d$
Local minima:		around 760 regularly distributed local minima
Global minimum:	$f(\vec{x}_{\min}) =$	-174.0399
	$\vec{x}_{\min} =$	$(-1.5, 0)$



(a) Full scale (b) Detail of one of the global minima

Figure C.15: Levy function no. 5.

C.19 Matyas' function

Dimensions:	$d =$	2
Formula:	$f(x_1, x_2) =$	$0.26(x_1^2 + x_2^2) - 0.48x_1x_2$
Function bounds:		unconstrained
Local minima:		unimodal
Global minimum:	$f(\vec{x}_{\min}) =$	0
	$\vec{x}_{\min} =$	$(0, 0)$

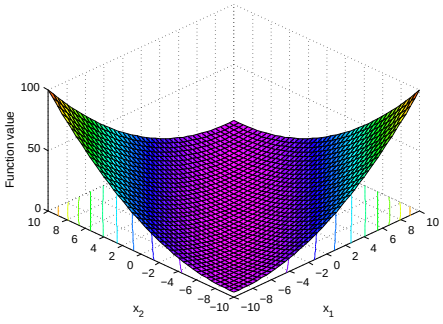
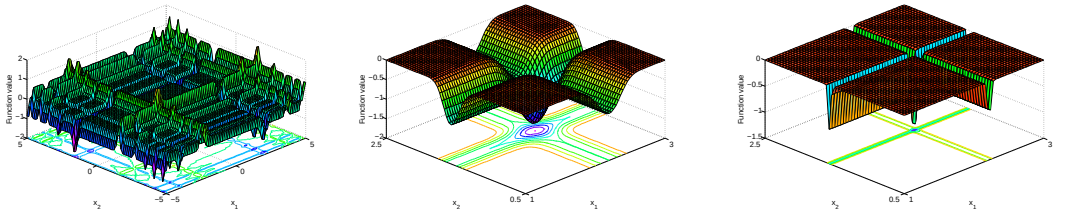


Figure C.16: Matyas function.

C.20 Michalewicz's function

Dimensions:	$d =$	n
Formula:	$f(\vec{x}) =$	$-\sum_{i=1}^{d} \sin(x_i) \left(\sin\left(\frac{ix_i^2}{\pi}\right)\right)^{2m}$
		$m > 0$
Function bounds:	$0 \leqq x_i \leqq \pi$	$i = 1, ..., d$
Local minima:		highly multimodal with $d!$ local minima
Global minimum:	$f(\vec{x}_{\min_{d=2}}) =$	-1.8013
	$f(\vec{x}_{\min_{d=5}}) =$	-4.687658
	$f(\vec{x}_{\min_{d=10}}) =$	-9.66015

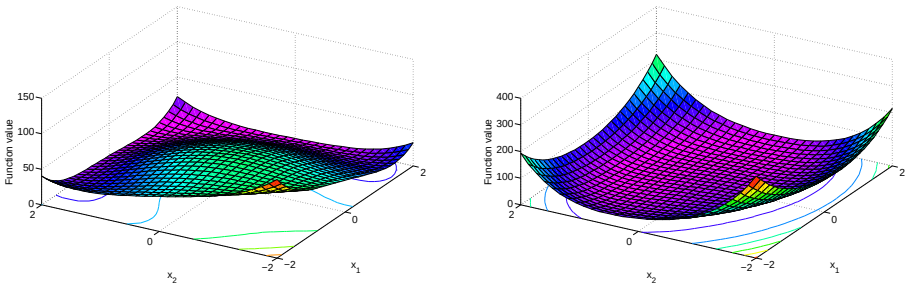


(a) Full scale, $m = 10$ (b) Detail of the global minimum, $m = 10$ (c) Detail of the global minimum, $m = 500$

Figure C.17: Michalewitcz's function for $d = 2$.

C.21 Perm's functions

Dimensions:	$d =$	n
Formula:	$f(\vec{x})_{n,\beta} =$	$\sum_{i=1}^{d} \left(\sum_{j=1}^{d} (j^i + \beta) \left(\left(\frac{x_j}{j}\right)^i - 1 \right) \right)^2$
	$f(\vec{x})^0_{n,\beta} =$	$\sum_{i=1}^{d} \left(\sum_{j=1}^{d} (j + \beta) \left(x_j^i - \left(\frac{1}{j}\right)^i \right) \right)^2$
Function bounds:	$-d \leqq x_i \leqq d$	$i = 1, ..., d$
Local minima:		unimodal
Global minimum:	$f(\vec{x}_{\min}) =$	0
	$\vec{x}_{\min_{n,\beta}} =$	$(1, ..., n)$
	$\vec{x}_{\min^0_{n,\beta}} =$	$\left(1, \frac{1}{2}, ..., \frac{1}{n}\right)$



(a) Perm's function $f_{n,\beta}$ (b) Perm's function $f^0_{n,\beta}$

Figure C.18: Perm's functions for $d = 2$, $\beta = 0.5$.

C.22 Powell's function

Dimensions:	$d =$	$n \geqq 4$
Formula:	$f(\vec{x}) =$	$\sum_{i=1}^{\frac{d}{4}} (x_{4i-3} + 10x_{4i-2})^2 + 5(x_{4i-1} - x_{4i})^2 + +(x_{4i-2} - x_{4i-1})^4 + 10(x_{4i-3} - x_{4i})^4$
Function bounds:	$-4 \leqq x_i \leqq 5$	$i = 1, ..., d$
Local minima:		several local minima
Global minimum:	$f(\vec{x}_{\min}) =$	0
	$\vec{x}_{\min} =$	$(3, -1, 0, 1, ..., 3, -1, 0, 1)$

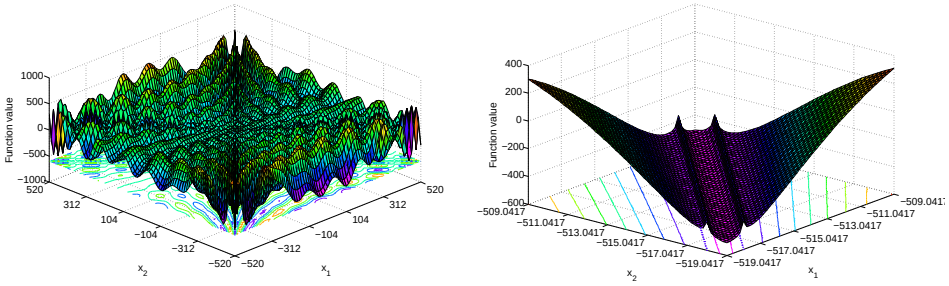
C.23 Power sum function

Dimensions:	$d =$	4
Formula:	$f(\vec{x}) =$	$\sum_{i=1}^{d} \left(\left(\sum_{j=1}^{d} x_j^i \right) - b_i \right)^2$
Function bounds:	$0 \leqq x_i \leqq n$	$i = 1, ..., d$
Local minima:		no local minima, several global ones
Global minimum:	$f(\vec{x}_{\min}) =$	0

$\vec{b} = (\ 8\ 18\ 44\ 114\ )$

C.24 Rana's function

Dimensions:	$d =$	n
Formula:	$f(\vec{x}) =$	$\frac{1}{d}\sum_{i=1}^{d} x_i \sin(\sqrt{\lvert x_j + 1 - x_i\rvert})\cos(\sqrt{\lvert x_j + 1 + x_i\rvert}) +$ $+(x_j+1)\cos(\sqrt{\lvert x_j + 1 - x_i\rvert})\sin(\sqrt{\lvert x_j + 1 + x_i\rvert})$ $j = (i+1) \bmod d$
Function bounds:	$-520 \leqq x_i \leqq 520$	$i = 1, ..., d$
Local minima:		multiple symmetrically spaced local minima
Global minimum:	$f(\vec{x}_{\min}) =$	-512.75316
	$\vec{x}_{\min} =$	$(-514,041683, ..., -514,041683)$

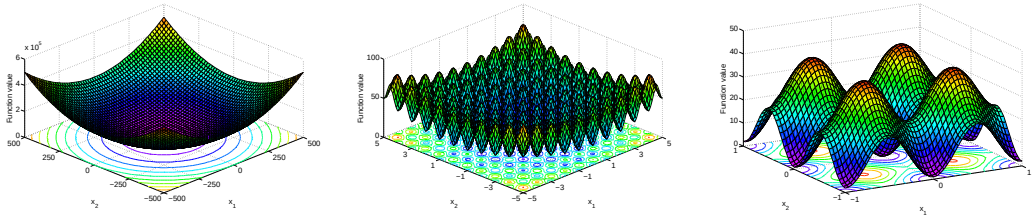


(a) Full scale (b) Detail of the global minimum

Figure C.19: Rana's function for $d = 2$.

C.25 Rastrigin's function

Dimensions:	$d =$	n
Formula:	$f(\vec{x}) =$	$10d + \sum_{i=1}^{d}(x_i^2 - 10\cos(2\pi x_i))$
Function bounds:		unconstrained
Local minima:		large number of regularly located local minima
Global minimum:	$f(\vec{x}_{\min}) =$	0
	$\vec{x}_{\min} =$	$(0, ..., 0)$

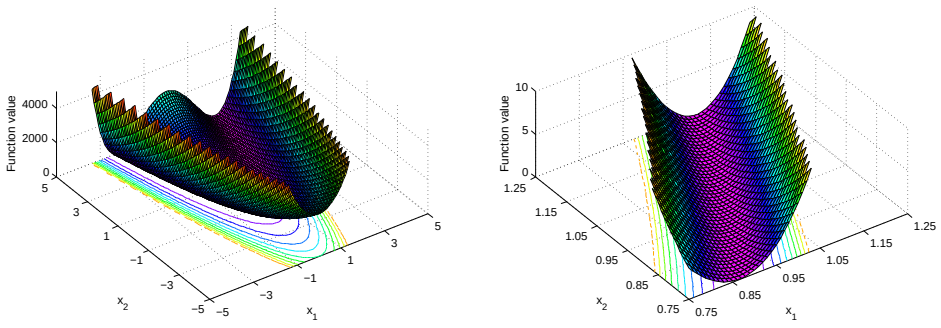


(a) Full scale (b) Medium scale (c) Detail of the global minimum

Figure C.20: Rastrigin's function for $d = 2$.

C.26 Rosenbrock's valley

Dimensions:	$d =$	n
Formula:	$f(\vec{x}) =$	$\sum_{i=1}^{d-1} 100(x_{i+1} - x_i^2)^2 + (1 - x_i)^2$
Function bounds:		unconstrained
Local minima:		unimodal
Global minimum:	$f(\vec{x}_{\min}) =$	0
	$\vec{x}_{\min} =$	$(1, ..., 1)$



(a) Full scale (b) Detail of the global minimum

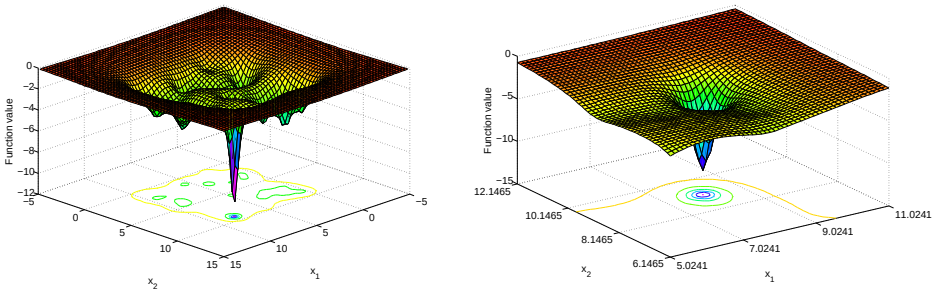
Figure C.21: Rosenbrock's valley for $d = 2$.

C.27 Shekel's foxholes

Dimensions:	$d =$	n
Formula:	$f(\vec{x}) =$	$-\sum_{i=1}^{m} \frac{1}{\sum_{j=1}^{d}(x_j - \mathbf{C}_{i,j})^2 + \beta_i}$
		$m = 1, ..., 30$
Function bounds:	unconstrained	
Local minima:		multiple areas containing local minima
Global minimum:	$f(\vec{x}_{\min_{d=2,m=30}}) =$	-12.119
	$\vec{x}_{\min_{d=2,m=30}} =$	$(8.024, 9.147)$
	$f(\vec{x}_{\min_{d=5,m=30}}) =$	-10.406
	$\vec{x}_{\min_{d=5,m=30}} =$	$(8.025, 9.152, 5.114, 7.621, 4.564)$

$$\mathbf{C} = \begin{pmatrix}
9.681 & 0.667 & 4.783 & 9.095 & 3.517 & 9.325 & 6.544 & 0.211 & 5.122 & 2.020 \\
9.400 & 2.041 & 3.788 & 7.931 & 2.882 & 2.672 & 3.568 & 1.284 & 7.033 & 7.374 \\
8.025 & 9.152 & 5.114 & 7.621 & 4.564 & 4.711 & 2.996 & 6.126 & 0.734 & 4.982 \\
2.196 & 0.415 & 5.649 & 6.979 & 9.510 & 9.166 & 6.304 & 6.054 & 9.377 & 1.426 \\
8.074 & 8.777 & 3.467 & 1.863 & 6.708 & 6.349 & 4.534 & 0.276 & 7.633 & 1.567 \\
7.650 & 5.658 & 0.720 & 2.764 & 3.278 & 5.283 & 7.474 & 6.274 & 1.409 & 8.208 \\
1.256 & 3.605 & 8.623 & 6.905 & 4.584 & 8.133 & 6.071 & 6.888 & 4.187 & 5.448 \\
8.314 & 2.261 & 4.224 & 1.781 & 4.124 & 0.932 & 8.129 & 8.658 & 1.208 & 5.762 \\
0.226 & 8.858 & 1.420 & 0.945 & 1.622 & 4.698 & 6.228 & 9.096 & 0.972 & 7.637 \\
7.305 & 2.228 & 1.242 & 5.928 & 9.133 & 1.826 & 4.060 & 5.204 & 8.713 & 8.247 \\
0.652 & 7.027 & 0.508 & 4.876 & 8.807 & 4.632 & 5.808 & 6.937 & 3.291 & 7.016 \\
2.699 & 3.516 & 5.874 & 4.119 & 4.461 & 7.496 & 8.817 & 0.690 & 6.593 & 9.789 \\
8.327 & 3.897 & 2.017 & 9.570 & 9.825 & 1.150 & 1.395 & 3.885 & 6.354 & 0.109 \\
2.132 & 7.006 & 7.136 & 2.641 & 1.882 & 5.943 & 7.273 & 7.691 & 2.880 & 0.564 \\
4.707 & 5.579 & 4.080 & 0.581 & 9.698 & 8.542 & 8.077 & 8.515 & 9.231 & 4.670 \\
8.304 & 7.559 & 8.567 & 0.322 & 7.128 & 8.392 & 1.472 & 8.524 & 2.277 & 7.826 \\
8.632 & 4.409 & 4.832 & 5.768 & 7.050 & 6.715 & 1.711 & 4.323 & 4.405 & 4.591 \\
4.887 & 9.112 & 0.170 & 8.967 & 9.693 & 9.867 & 7.508 & 7.770 & 8.382 & 6.740 \\
2.440 & 6.686 & 4.299 & 1.007 & 7.008 & 1.427 & 9.398 & 8.480 & 9.950 & 1.675 \\
6.306 & 8.583 & 6.084 & 1.138 & 4.350 & 3.134 & 7.853 & 6.061 & 7.457 & 2.258 \\
0.652 & 2.343 & 1.370 & 0.821 & 1.310 & 1.063 & 0.689 & 8.819 & 8.833 & 9.070 \\
5.558 & 1.272 & 5.756 & 9.857 & 2.279 & 2.764 & 1.284 & 1.677 & 1.244 & 1.234 \\
3.352 & 7.549 & 9.817 & 9.437 & 8.687 & 4.167 & 2.570 & 6.540 & 0.228 & 0.027 \\
8.798 & 0.880 & 2.370 & 0.168 & 1.701 & 3.680 & 1.231 & 2.390 & 2.499 & 0.064 \\
1.460 & 8.057 & 1.336 & 7.217 & 7.914 & 3.615 & 9.981 & 9.198 & 5.292 & 1.224 \\
0.432 & 8.645 & 8.774 & 0.249 & 8.081 & 7.461 & 4.416 & 0.652 & 4.002 & 4.644 \\
0.679 & 2.800 & 5.523 & 3.049 & 2.968 & 7.225 & 6.730 & 4.199 & 9.614 & 9.229 \\
4.263 & 1.074 & 7.286 & 5.599 & 8.291 & 5.200 & 9.214 & 8.272 & 4.398 & 4.506 \\
9.496 & 4.830 & 3.150 & 8.270 & 5.079 & 1.231 & 5.731 & 9.494 & 1.883 & 9.732 \\
4.138 & 2.562 & 2.532 & 9.661 & 5.611 & 5.500 & 6.886 & 2.341 & 9.699 & 6.500
\end{pmatrix}$$

$$\vec{\beta} = \begin{pmatrix}
0.806 & 0.517 & 0.100 & 0.908 & 0.965 & 0.669 & 0.524 & 0.902 & 0.531 & ... \\
... & 0.876 & 0.462 & 0.491 & 0.463 & 0.714 & 0.352 & 0.869 & 0.813 & ... \\
... & 0.811 & 0.828 & 0.964 & 0.789 & 0.360 & 0.369 & 0.992 & 0.332 & ... \\
... & 0.817 & 0.632 & 0.883 & 0.608 & 0.326 & & & &
\end{pmatrix}$$

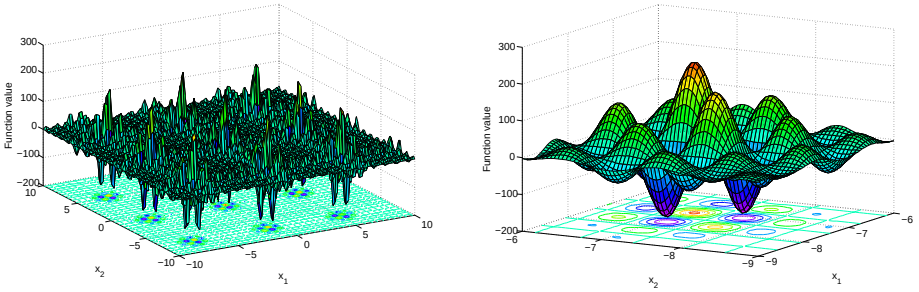


(a) All the foxholes (b) Detail of the global minimum

Figure C.22: Shekel's foxholes for $d = 2$, $m = 30$.

C.28 Shubert's function

Dimensions:	$d =$	2
Formula:	$f(x_1, x_2) =$	$\sum_{i=1}^{5} i\cos((i+1)x_1 + i) \sum_{i=1}^{5} i\cos((i+1)x_2 + i)$
Function bounds:	$-10 \leqq x_i \leqq 10$	$i = 1, ..., d$
Local minima:		multiple equally spaced local minima and maxima
Global minimum:	18: $f(\vec{x}_{\min}) =$	-186.7309088



(a) Full scale (b) Detail of one of the global minima

Figure C.23: Shubert's function.

C.29 Schwefel's function

Dimensions:	$d =$	n
Formula:	$f(\vec{x}) =$	$418.9829d - \sum_{i=1}^{d} -x_i \sin(\sqrt{\lvert x_i \rvert})$
Function bounds:	$-500 \leqq x_i \leqq 500$	$i = 1, ..., d$
Local minima:		numerous local minima, geometrically distant over the parameter space from the global minimum
Global minimum:	$f(\vec{x}_{\min}) =$	0
	$\vec{x}_{\min} =$	$(420.9687, ..., 420.9687)$

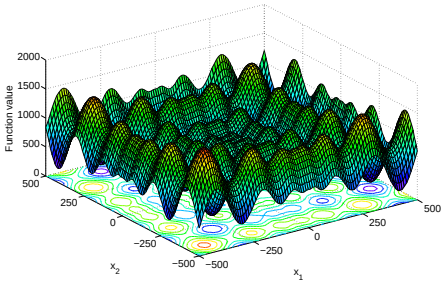


Figure C.24: Schwefel's function.

C.30 Trid function

Dimensions:	$d =$	n
Formula:	$f(\vec{x}) =$	$\sum_{i=1}^{d}(x_i - 1)^2 - \sum_{i=2}^{d} x_i x_{i-1}$
Function bounds:	$-d^2 \leqq x_i \leqq d^2$	$i = 1, ..., d$
Local minima:		unimodal
Global minimum:	$f(\vec{x}_{\min_{d=2}}) =$	-2
	$\vec{x}_{\min_{d=2}} =$	$(2, 2)$
	$f(\vec{x}_{\min_{d=6}}) =$	-50
	$\vec{x}_{\min_{d=6}} =$	$(id - i(i-1), ..., id - i(i-1))$
	$f(\vec{x}_{\min_{d=10}}) =$	-210
	$\vec{x}_{\min_{d=10}} =$	$(id - i(i-1), ..., id - i(i-1))$
	$i =$	$0, ..., \frac{d}{2}, \frac{d}{2}, ..., 0$

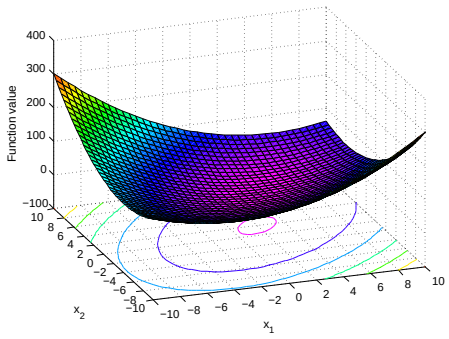
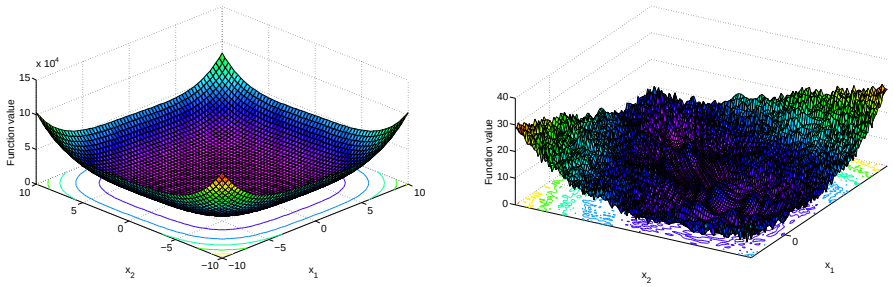


Figure C.25: Trid function.

C.31 Whitley's function

Dimensions:	$d =$	n
Formula:	$f(\vec{x}) =$	$\sum_{i=1}^{d}\sum_{j=1}^{d}\left(\frac{(100(x_i^2 - x_j)^2 + (1 + x_j)^2)^2}{4000} - \right.$
		$\left. - \cos(100(x_i^2 - x_j)^2 + (1 - x_j)^2) + 1\right)$
Function bounds:		unconstrained
Local minima:	several deep local minima in the area close to the global minimum	
Global minimum:	$f(\vec{x}_{\min}) =$	0
	$\vec{x}_{\min} =$	$(1, ..., 1)$

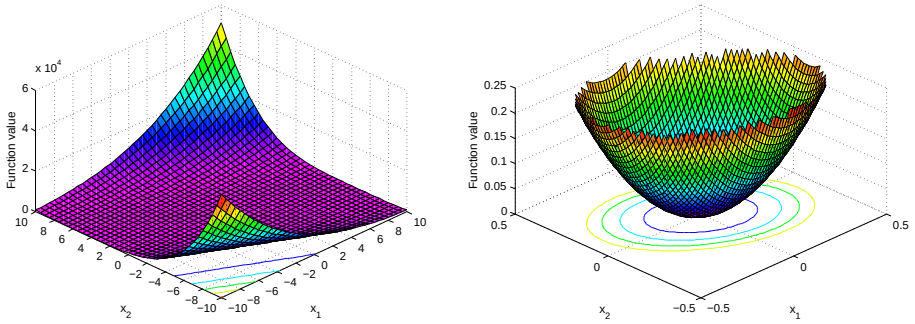


(a) Full scale (b) Detail of the global minimum

Figure C.26: Whitley's function for $d = 2$.

C.32 Zakharov's function

Dimensions:	$d =$	n
Formula:	$f(\vec{x}) =$	$\sum_{i=1}^{d} x_i^2 + \left(\sum_{i=1}^{d} \frac{ix_i}{2}\right)^2 + \left(\sum_{i=1}^{d} \frac{ix_i}{2}\right)^4$
Function bounds:		unconstrained
Local minima:		unimodal
Global minimum:	$f(\vec{x}_{\min}) =$	0
	$\vec{x}_{\min} =$	$(0, ..., 0)$



(a) Full scale (b) Detail of the global minimum

Figure C.27: Zakharov's function for $d = 2$.

Appendix D

Additional results of experiments

Machine accuracy as obtained on the computer running the tests:

$$\varepsilon = 2.22044604925031310^{-}16$$
$$\sqrt{\varepsilon} = 1.490116119384765610^{-}8$$

where ε is the difference between 1 and the smallest exactly representable number greater than one.

Additional results of experiments follow. These are considered self-explaining and therefore are not commented.

D.1 CACO

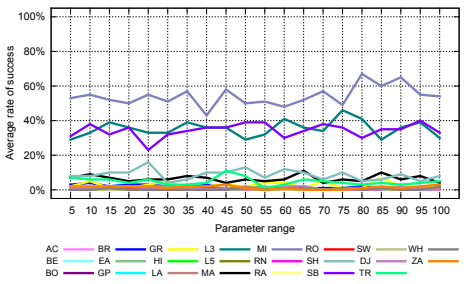


Figure D.1: CACO results for population size = 5, ..., 100, step = 5.

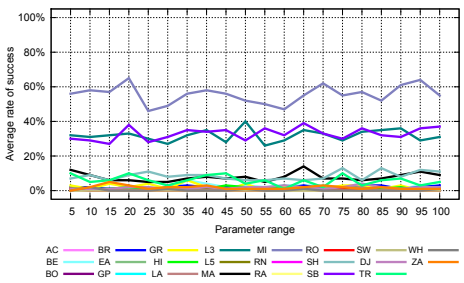


Figure D.2: CACO results for search radius = 5, ..., 100, step = 5.

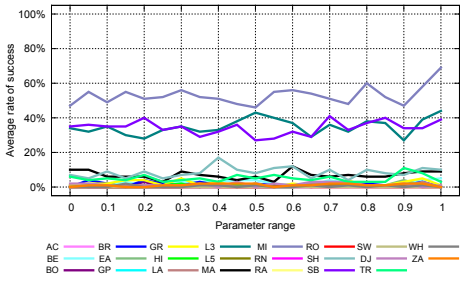


Figure D.3: CACO results for radius multiplier = 0.0, ..., 1.0, step = 0.05.

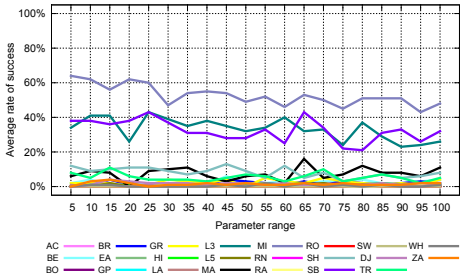


Figure D.4: CACO results for generations before radius decrease = 5, ..., 100, step = 5.

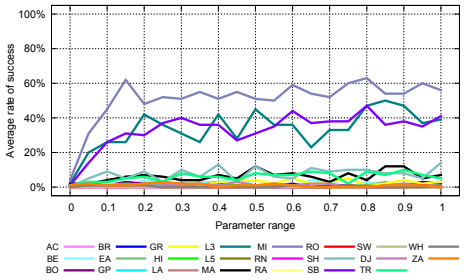


Figure D.5: CACO results for evaporation factor = 0.0, ..., 1.0, step = 0.05.

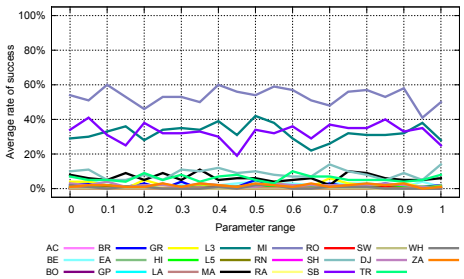


Figure D.6: CACO results for initial pheromone level = 0.0, ..., 1.0, step = 0.05.

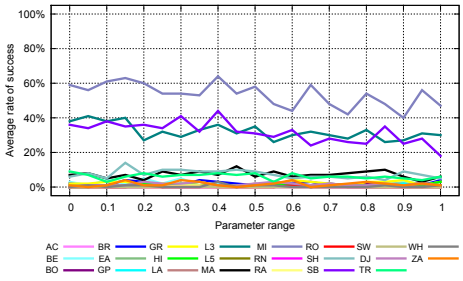


Figure D.7: CACO results for pheromone increase = 0.0, ..., 1.0, step = 0.05.

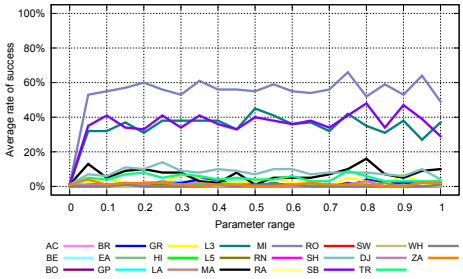
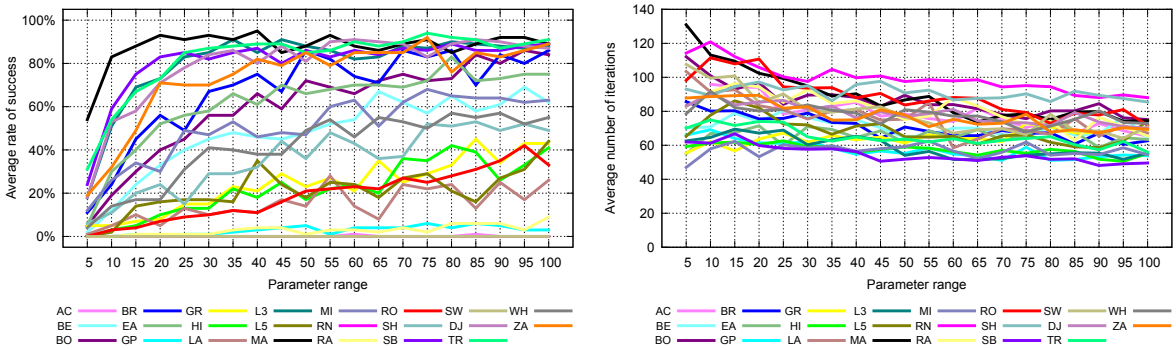


Figure D.8: CACO results for minimal pheromone level = 0.0, ..., 1.0, step = 0.05.

D.2 API

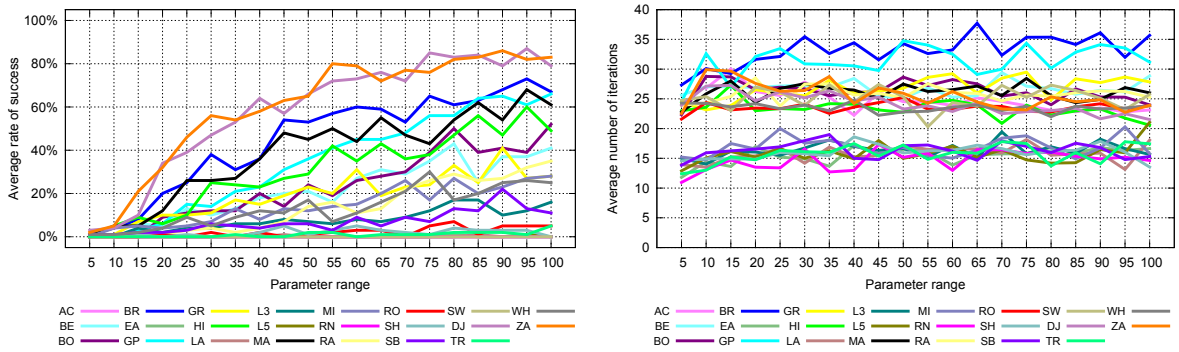


(a) Average success rate for API

(b) Average number of iterations taken by API

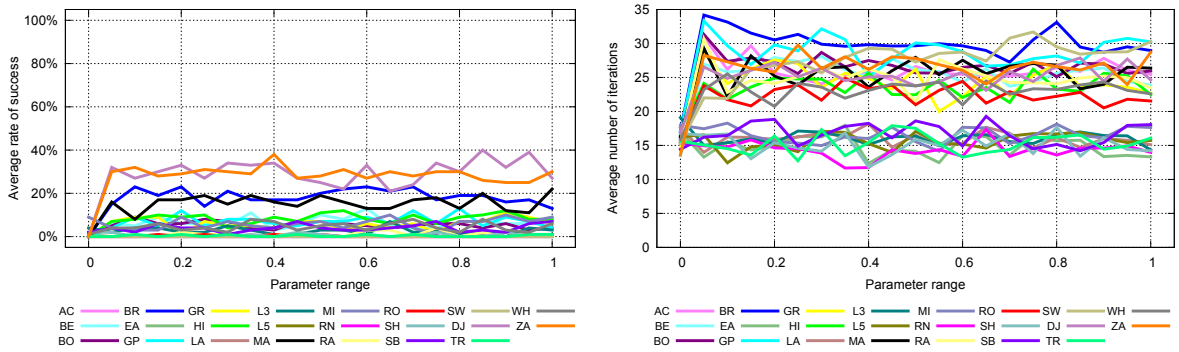
Figure D.9: API results for population size = 5, ..., 100, step = 5.

D.3 AACA



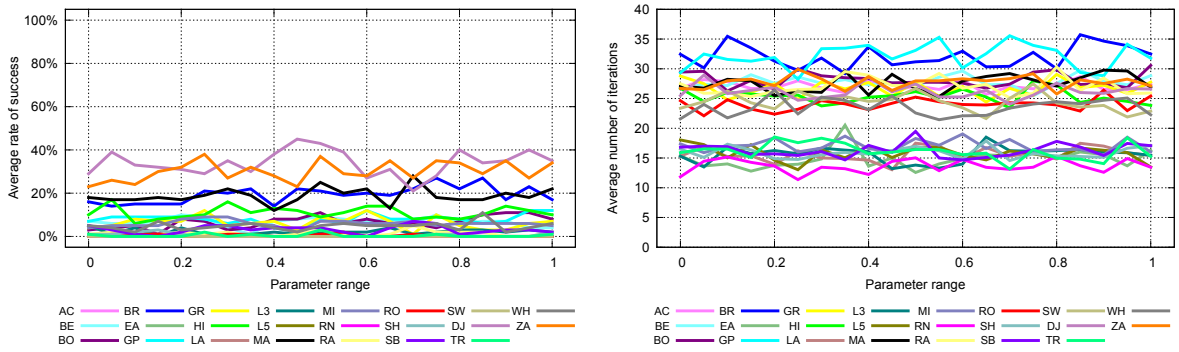
(a) Average success rate for AACA (b) Average number of iterations taken by AACA

Figure D.10: AACA results for population size $= 5, ..., 100$, step $= 5$.



(a) Average success rate for AACA (b) Average number of iterations taken by AACA

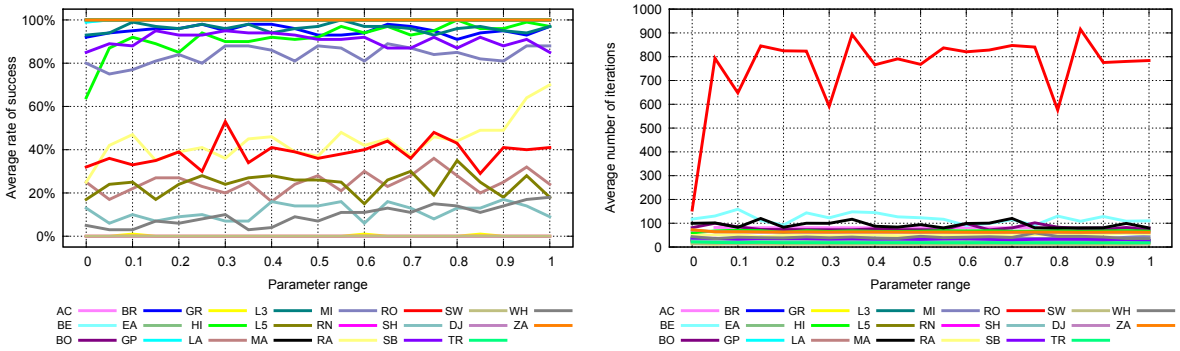
Figure D.11: AACA results for pheromone index $= 0.0, ..., 1.0$, step $= 0.05$.



(a) Average success rate for AACA (b) Average number of iterations taken by AACA

Figure D.12: AACA results for cost index $= 0.0, ..., 1.0$, step $= 0.05$.

D.4 ACO*

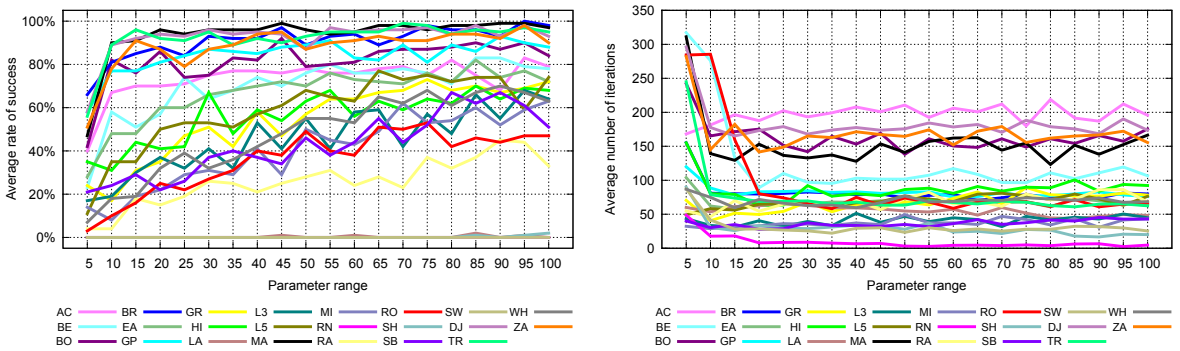


(a) Average success rate for ACO* (b) Average number of iterations taken by ACO*

Figure D.13: ACO* results for neighborhood size for force diversity $= 0.0, ..., 1.0$, step $= 0.05$, force diversity $= true$

D.5 PSO

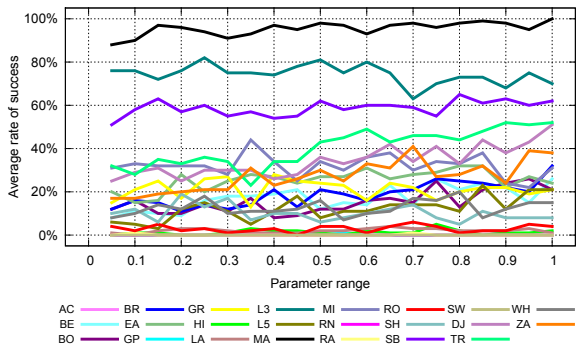
D.5.1 PSO–FI



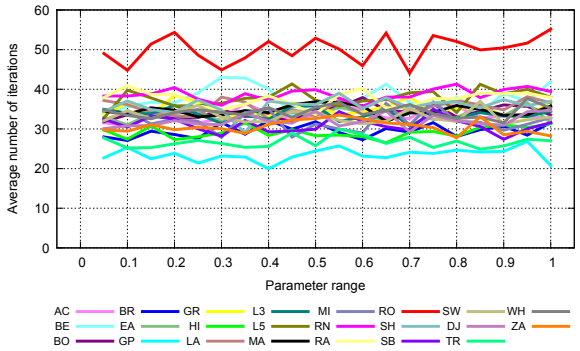
(a) Average success rate for PSO–FI (b) Average number of iterations taken by PSO–FI

Figure D.14: PSO results for population size $= 5, ..., 100$, step $= 5$. FI update formula used.

D.5.2 PSO–C

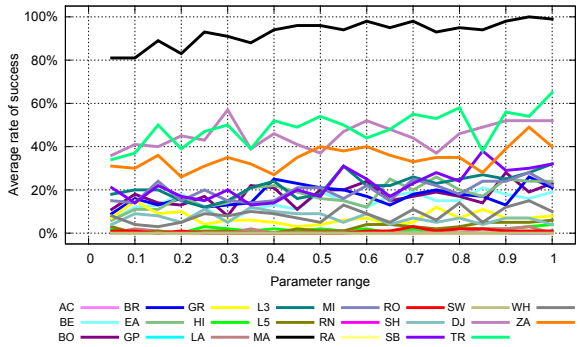


(a) Average success rate for PSO–C

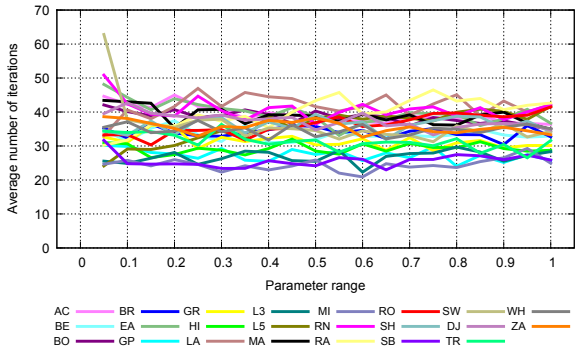


(b) Average number of iterations taken by PSO–C

Figure D.15: PSO results for $\phi_1 = 0.05, ..., 1.00$, step $= 0.05$. C update formula used.



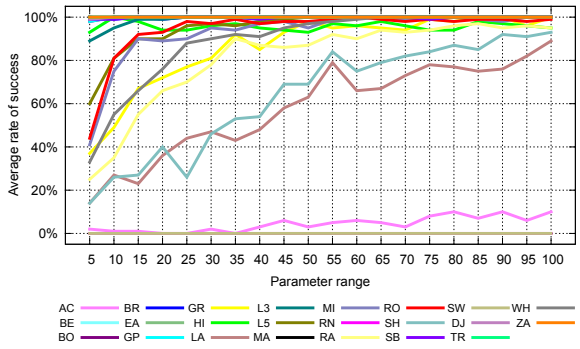
(a) Average success rate for PSO–C



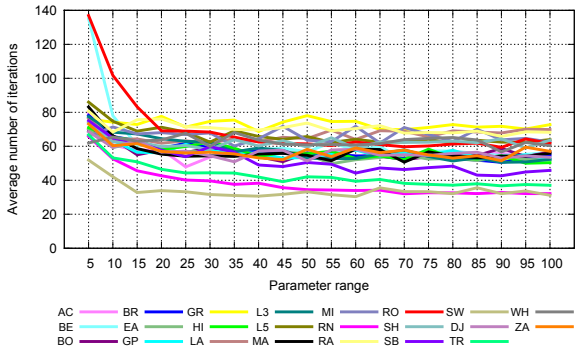
(b) Average number of iterations taken by PSO–C

Figure D.16: PSO results for $\phi_2 = 0.05, ..., 1.00$, step $= 0.05$. C update formula used.

D.6 SADE



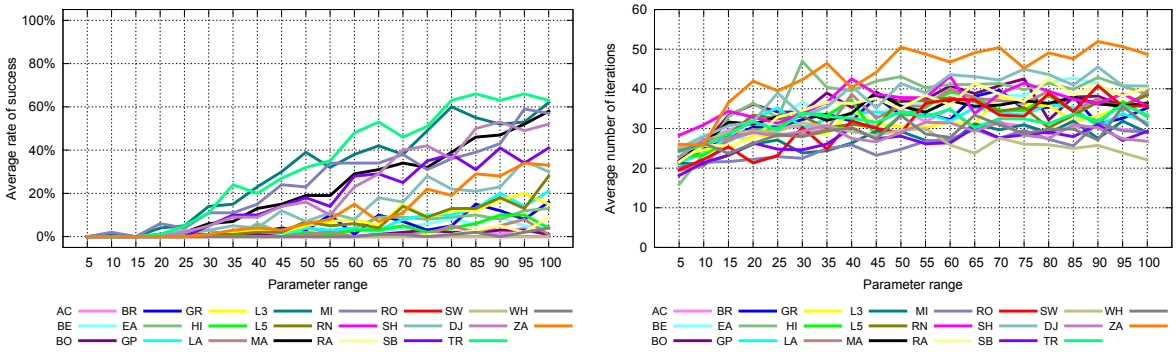
(a) Average success rate for SADE



(b) Average number of iterations taken by SADE

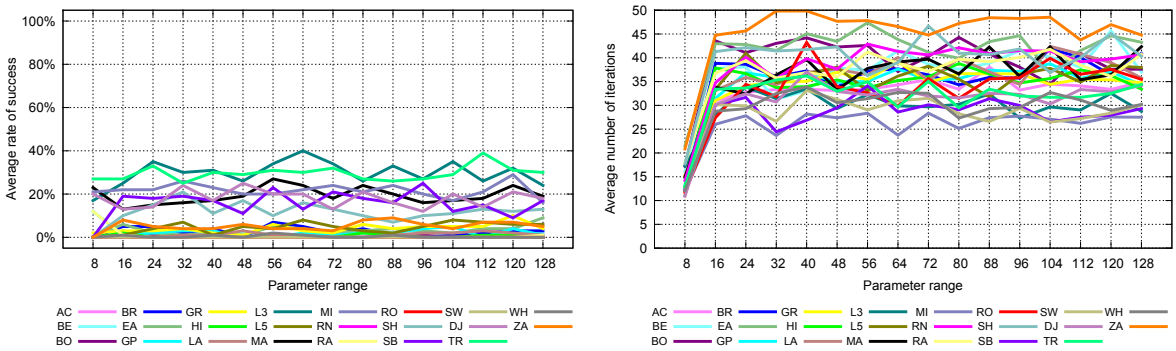
Figure D.17: SADE results for population size $= 5, ..., 100$, step $= 5$.

D.7 PBIL



(a) Average success rate for PBIL (b) Average number of iterations taken by PBIL

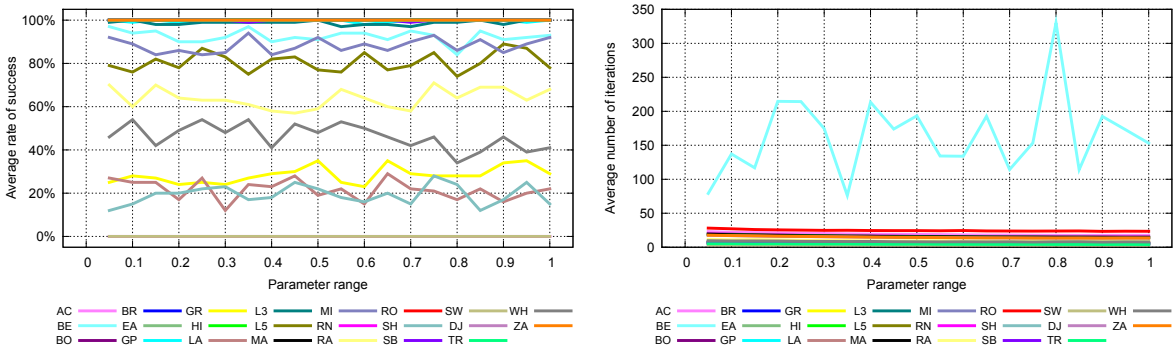
Figure D.18: PBIL results for population size = 5, ..., 100, step = 5.



(a) Average success rate for PBIL (b) Average number of iterations taken by PBIL

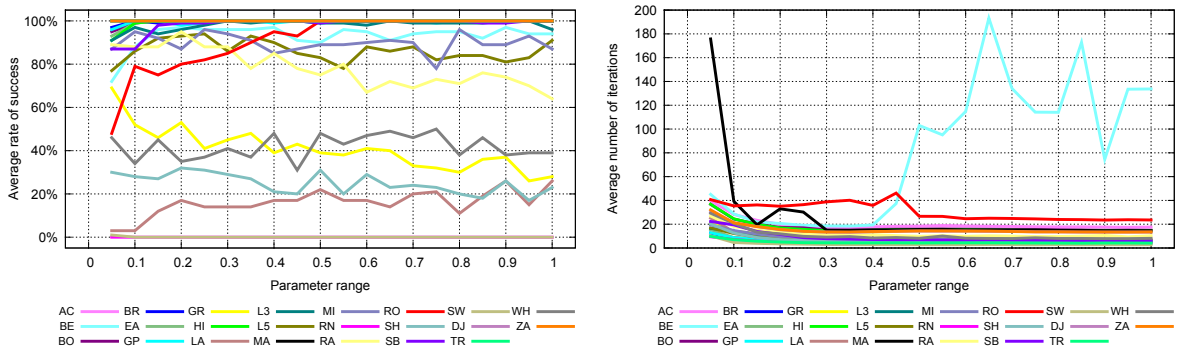
Figure D.19: PBIL results for encoding length = 8, ..., 128, step = 8.

D.8 HGAPSO



(a) Average success rate for HGAPSO (b) Average number of iterations taken by HGAPSO

Figure D.20: HGAPSO results for $\phi_1 = 0.05, ..., 1.00$, step = 0.05.



(a) Average success rate for HGAPSO

(b) Average number of iterations taken by HGAPSO

Figure D.21: HGAPSO results for $\phi_2 = 0.05, ..., 1.00$, step $= 0.05$.

Appendix E

Contents of the attached CD

Directory	Description
bin/	Binary distribution of JCool with the implemented optimization methods and objective functions
lib/	Libraries required by JCool
jcool	Shell script for running JCool platform
jcool.bat	Batch file for running JCool platform
JCool.jar	Main JCool JAR file
documentation/	JavaDoc documentation of the implemented optimization methods and objective functions
source/	Java source codes related to this thesis
functions/	Java source codes of the implemented objective functions
methods/	Java source codes of the implemented optimization methods
other/	Other related Java source codes contributed to JCool
text/	Text of this thesis
LaTeX/	LaTeX source files of this thesis
PDF/	This thesis in PDF format
readme.txt	Basic instructions on usage

Table E.1: Contents of the attached CD.

List of Abbreviations

AACA	Adaptive Ant Colony Algorithm
AC	Ackley's path function
ACO*	Extended Ant Colony Optimization
API	*Pachycondyla apicalis* method
AVG	average value
B	Brent's method for line search
BACA	Basic Ant Colony Algorithm
BE	Beale's function
BF	Broyden family update formula for QN
BFGS	Broyden-–Fletcher-–Goldfarb-–Shanno update formula for QN
BO	Booth's function
BR	Branin's function
BRO	Broyden's update formula for QN
BSHS	Beale–Sorenson–Hestenes–Stiefel update formula for CG
BWD	Brent's method for line search using first derivatives
C	Canonical update formula for PSO
CACO	Continuous Ant Colony Optimization
CG	Conjugate Gradient method
CMA-ES	Covariance Matrix Adaptation Evolution Strategy
DACO	Direct Ant Colony Optimization
DE	Differential Evolution
DFP	Davidon—Fletcher—Powell update formula for QN
DJ	De Jong's (sphere) function
EA	Easom's function

FAKE GAME	Fully Automated Knowledge Extraction using Group of Adaptive Models Evolution
FI	Fully Informed update formula for PSO
FR	Fletcher–Reeves update formula for CG
GA	Genetic algorithm
GP	Goldstein–Price function
GR	Griewangk's function
HGAPSO	Hybrid of the GA and the PSO
HI	Himmelblau function
HTML	HyperText Markup Language
JCool	Java Continuous Optimization Library
L3	Levy function no. 3
L5	Levy function no. 5
LA	Langerman's function
LM	Levenberg–Marquardt method
MA	Matyas' function
MI	Michalewicz's function
MSB	most significant bit
OS	Orthogonal search
PBIL	Population-Based Incremental Learning
PM	Powell's method
PR	Polak—Ribière update formula for CG
PSO	Particle Swarm Optimization
QN	quasi-Newton method
RA	Rastrigin's function
RN	Rana's function
RO	Rosenbrock's valley function
SADE	Simplified Atavistic Differential Evolution
SB	Shubert's function
SD	Steepest Descent method
SH	Shekel's foxholes

SW	Schwefel's function
TR	Trid function
WH	Whitley's function
XML	Extensible Markup Language
ZA	Zakharov's function

Index

Adaptive Ant Colony Algorithm, **20**
Algorithm, 15, 27
  Adaptive Ant Colony, *see* Adaptive Ant Colony Algorithm
  Ant algorithm, *see* Ant algorithm
  Basic Ant Colony, *see* Basic Ant Colony Algorithm
  genetic, *see* Genetic algorithm
Ant algorithm, 17
Ant Colony Optimization, 17
Average, 14, 15, 32

Basic Ant Colony Algorithm, **20**
Beale–Sorenson–Hestenes–Stiefel formula, **9**
Binary string, 18, 20, 24, 25
Bounds, *see* Function bounds
Brent
  Brent's method, **6**, 65
  Brent's method using first derivatives, **6**
Broyden family formula, 12
Broyden's formula, 12
Broyden—Fletcher—Goldfarb—Shanno formula, 12

Canonical search formula, 22
Central difference, 3
Children, 24
Cholesky decomposition, 11
Code
  source code, 3
Coefficient
  cognitive acceleration, *see* Cognitive acceleration coefficient
  constriction, *see* Constriction coefficient
  mutation, 25
  social acceleration, *see* Social acceleration coefficient
Cognitive acceleration coefficient, **23**
Competitive learning, 25
Condition, 13, 14, 24
  quasi-Newton, *see* *quasi*-Newton
  Stop condition, *see* Stop condition, 33
Conjugate Gradient, **8**, 13, 65
Constriction coefficient, **23**
Continuous Ant Colony Optimization, **18**
Convergence, 8, 13
  global, 9, 27
  local, 27
  premature, 22, 25, 29
  Rate of convergence, 13, 27, 32, 65
Coupling, 29
Covariance Matrix Adaptation Evolution Strategy, 7, **15**
Crossover, 23, 26

Davidon—Fletcher—Powell formula, 12
Descent, 7
  steepest, *see* Steepest Descent
Differential Evolution, **24**
Differential operator, 24
Direct Ant Colony Optimization, **21**
Distribution, 16
  normal, 15, 20
  Probability distribution, 20, 21
  Search distribution, 15

Elite, 26
Encoding, 18, 20, 24, 25
Enhancement, 26
Evaporation, 17, 22
Evolution path, 16
Extended Ant Colony Optimization, **20**

FAKE GAME, 1, 18
Fletcher–Reeves formula, **8**, 65
Formula
  Beale–Sorenson–Hestenes–Stiefel, *see* Beale–Sorenson–Hestenes–Stiefel formula
  Broyden family, *see* Broyden family formula
  Broyden's formula, *see* Broyden's formula
  Broyden—Fletcher—Goldfarb—Shanno, *see* Broyden-–Fletcher-–Goldfarb-–Shanno formula
  Canonical search, *see* Canonical search formula

Davidon—Fletcher—Powell, *see* Davidon—Fletcher—Powell formula
Fletcher–Reeves, *see* Fletcher–Reeves formula
Fully Informed, *see* Fully Informed search formula
Polak—Ribière, *see* Polak—Ribière formula
Fully Informed search formula, 22
Function
Ackley's path, 29
Beale's function, 28
Benchmark function, *see* Test function, 27
Booth's function, 28
bounds, 3, 29, 31
Branin's function, 29
convex, 28
De Jong's function, 28
Easom's function, 28
fitness, 23, 25
Gaussian function, **21**
Goldstein–Price function, 30
Griewangk's function, 30
Himmelblau function, 30
Langerman's function, 30
Levy function no. 3, 30
Levy function no. 5, 30
Matyas' function, 28
Michalewicz's function, 31
multidimensional, 5
multimodal, 27, 29
Objective function, 2, 7, 9, 10, 14
Rana's function, 31
Rastrigin's function, 31
Rosenbrock's valley, 28
Schwefel's function, 32
Shekel's foxholes, 31
Shubert's function, 31
Test function, 2
Trid function, 29
unidimensional, 13
unimodal, 27
Whitley's function, 32
Zakharov's function, 29

Generation, 16, 23, 26
Genetic algorithm, **23**, 24, 25
Global best, **6**, 22
Gradient, **7**, 7, 11, 12, 28, 32, 65
analytic, 16
conjugate, *see* Conjugate Gradient
numerical, 3
Hessian, **6**, 7, 10–12, 32
analytic, 10, 16
numerical, 3, 11
Hessian matrix, *see* Hessian
Hunting site, 18
Hybrid of the GA and the PSO, **25**

Implementation, 2
Individual, 16, 24–26
Interface, 2
Iteration, 7, 10, 12, 22, 32, 65

JavaDoc, 5
JCool, 1, 18, 27, 32, 33

Learning rate, 16, 25
negative, **25**
Levenberg–Marquardt, **9**, 10, 12
Line search, **6**, 7, 8, 10, 13, 14
Linear combination, 8
Linear dependance, 14
Linear system, 9
Local best, **6**, 22

Machine accuracy, 8
Matrix
Covariance matrix, 15, 16
diagonal, 10
Hessian matrix, *see* Hessian
Identity matrix, 10, 12
inverse, 11, 12
singular, 10
Maximization, *see* Minimization
Mean value, 15, 21, 22
Meta-optimization, 33
Method
Brent's, *see* Brent's method
Brent's method using first derivatives, *see* Brent's method using first derivatives
Central difference method, *see* Central difference
Conjugate Gradient method, *see* Conjugate Gradient
gradient, 7
Levenberg–Marquardt method, *see* Levenberg–Marquardt
Nature inspired method, 15, **16**, 23
Newton's method, *see* Newton's method
numerical, **7**, 15
Optimization method, *see* Optimization method

Orthogonal search method, *see* Orthogonal search
Pachycondyla apicalis method, *see Pachycondyla apicalis*
Powell's method, *see* Powell's method
quasi-Newton method, *see quasi*-Newton
Steepest Descent method, *see* Steepest Descent
Minimization, 5, 27
Minimizer, *see* Minimum
Minimum, 8, 9, 13
global, **6**, 29
local, **6**, 25, 29
Mutation, 23, 26

Nest, 18
Newton's method, **9**, 9, 10, 12, 65
Damped Newton's method, **10**, 10

Optimization
Ant Colony Optimization, *see* Ant Colony Optimization
Continuous Ant Colony Optimization, *see* Continuous Ant Colony Optimization
Direct Ant Colony Optimization, *see* Direct Ant Colony Optimization
discrete, 17
global, **6**
local, **6**
meta-optimization, *see* Meta-optimization
method, 2
Particle Swarm Optimization, *see* Particle Swarm Optimization
Optimization method, 16
Oriented graph, 20
Orthogonal search, **13**
stochastic, **13**

Pachycondyla apicalis, **18**
Parent, 15, 24, 26
Particle, *see* Individual
Particle Swarm Optimization, **22**, 25
Pheromone, **17**, 18, 20–22
Point, 5, 12
saddle, **6**, 10
stationary, **6**, 7, 10
Polak–Ribière formula, 65
Polak—Ribière formula, **8**
Population, **6**, 15, 16, 21, 23, 25, 26, 28, 32
diversity, 24
size, 16, 24
Population-Based Incremental Learning, **25**
Powell's method, **13**
Probability density function, **20**

Quadratic form, **9**, 10, 12, 14, 28, 29
quasi-Newton, 9, **10**, 12
condition, 11

Radiation field, 25

Sampling, 15, 21, 25
probabilistic, 18, 20
Search direction, **5**, 8, 11, 13, 30
Search radius, 18
Local search radius, **19**
Selection, 23
tournament, 26
Simplified Atavistic Differential Evolution, **24**
Social acceleration coefficient, **23**
Solution, 14, 16, 22, 23
Standard deviation, 15, 16, 21, 22
Steepest Descent, **7**, 8, 10, 12, 65
Step, 8, 13
direction, **7**, 8
length, **7**, 8, 15, 16
Newton step, 10
Stiefel's cage, **8**, 13, 29, 31
Stochastic orthogonal search, *see* Orthogonal search
Stop condition, 2

Taylor expansion, 6, 7, 9, 11
Trust region, 10

Update
rank-μ, 16
rank-one, 12, 16
rank-two, 12

Valley, 28, 31

Wiki, 5

Zig–zag, **8**, *see* Stiefel's cage