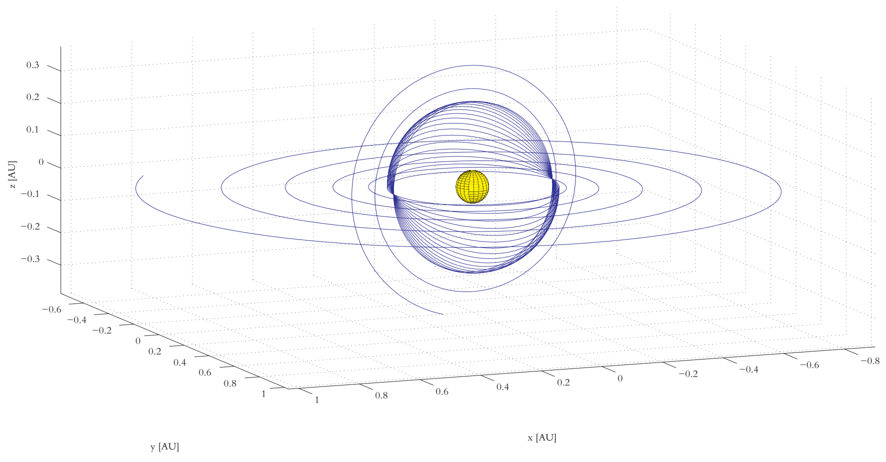
Improving Global Optimization Methods for Low Thrust Trajectories



Jasper Spaans
Delft University of Technology
Faculty of Aerospace Engineering
Section Astrodynamics and Satellite Systems

August 11, 2009

Faculty of Aerospace Engineering
Delft University of Technology
Kluyverweg 1, 2629 HS Delft
The Netherlands

TU Delft
Delft University of Technology

Contents

List of figures **ix**

List of tables **xiii**

List of listings **xv**

Glossary **xvii**

Summary **xxiii**

1 Introduction **1**
- 1.1 Optimization of trajectories . . . 1
- 1.2 Motivation . . . 2
- 1.3 Thesis Objectives . . . 3
- 1.4 Organization of this report . . . 3

2 Introduction to Optimization **5**
- 2.1 Introduction and terminology of optimization . . . 5
- 2.2 Steps in the optimization process . . . 7
- 2.3 Global one-shot optimizers . . . 8
  - 2.3.1 Nested do-loop . . . 8
  - 2.3.2 Regular sampling . . . 9
  - 2.3.3 Monte Carlo sampling . . . 9
  - 2.3.4 Sobol sampling . . . 9
- 2.4 Population based evolutionary optimizers . . . 9
  - 2.4.1 Genetic Algorithms . . . 9
  - 2.4.2 Evolutionary Algorithms . . . 10
  - 2.4.3 Particle Swarm Optimization . . . 11
  - 2.4.4 Differential Evolution . . . 11
- 2.5 Local optimizers . . . 11
  - 2.5.1 Newton-Raphson . . . 11
  - 2.5.2 Golden section search . . . 12
  - 2.5.3 Steepest descent method . . . 12
  - 2.5.4 Powell's quadratically convergent method . . . 13
- 2.6 Other algorithms . . . 16
  - 2.6.1 Interval Analysis . . . 16
  - 2.6.2 Simulated Annealing . . . 19
  - 2.6.3 Hybrid methods . . . 20
- 2.7 Selection of optimizers . . . 20

3 Evolutionary Computation **21**
3.1 History of Evolutionary Computation 21
3.1.1 Evolutionary programming 21
3.1.2 Genetic algorithms 21
3.1.3 Evolution strategies 22
3.1.4 Convergence of the optimizers 22
3.2 Introduction to evolutionary computing 22
3.3 Representation of a solution 22
3.3.1 Binary coding 22
3.3.2 Gray coding 23
3.3.3 Real coding 24
3.3.4 Permutations 24
3.3.5 Probability coding 24
3.4 Variation Operators 25
3.4.1 Crossover operators 25
3.4.2 Additional real-valued crossover operators 26
3.4.3 Mutation 28
3.5 Methods of selection 29
3.5.1 Reproduction operator 29
3.5.2 Fitness scaling 31
3.5.3 Fitness sharing 32
3.5.4 Rank-based allocation 33
3.6 Selection of representation and operators 33
3.6.1 Representation of the solution 33
3.6.2 Variation operators 33
3.6.3 Selection operator 34

4 Particle Swarm Optimization **35**
4.1 Introduction to PSO 35
4.2 Basic formulation of the velocity and position of particles 36
4.3 Tuning optimization parameters in PSO 37
4.3.1 Varying inertia weight 37
4.3.2 Constriction coefficient 38
4.3.3 Population size in PSO 38
4.4 Predator-Prey Optimization 39
4.5 Applying bounds on particle velocity and position 42
4.6 Examples of contracting swarms 43

5 Differential Evolution **47**
5.1 Introduction to Differential Evolution 47
5.2 Mutation vector generation 47
5.2.1 Scheme Tasoulis1 47
5.2.2 Scheme Tasoulis2 / DE1 48
5.2.3 Scheme Tasoulis3/DE2 49
5.2.4 Scheme Tasoulis4 49
5.2.5 Scheme Tasoulis5 49
5.2.6 Scheme Tasoulis6 49
5.2.7 Alternative naming of the schemes 50
5.3 Trial vector generation and selection 50
5.4 Usage of Differential Evolution 51
5.5 Selecting Differential Evolution strategies 51

6 Multi-Objective Optimization (moo) 59
6.1 Weighted objective function 60
6.2 Pareto front ranking 61
6.2.1 Classic Pareto front ranking algorithm 61
6.2.2 Non-dominated Sorting Genetic Algorithm II (NSGA-II) 62
6.3 Ranking individuals inside their front 62
6.4 Verification of the NSGA-II implementation 63

7 Verification of PSO and DE on mathematical test functions 67
7.1 Mathematical test functions 67
7.1.1 Himmelblau function 67
7.1.2 Griewank function 67
7.1.3 Schaffer's f_6 function 69
7.1.4 Deb-Tan function 69
7.2 Verifying PSO for OPTIDUS 70
7.2.1 Overview of the tested optimizer implementations 71
7.2.2 Himmelblau function results 71
7.2.3 Griewank function results 71
7.2.4 Schaffer's f_6 function results 72
7.2.5 Deb-Tan function results 73
7.2.6 Conclusions for the verification 73

8 Improving the Solar Polar Sail Mission Optimization 77
8.1 Introduction to solar sailing 77
8.2 Model of the Solar Polar Sail Mission 79
8.2.1 Geocentric Phase: Departure from Earth 79
8.2.2 Heliocentric phase 81
8.3 Optimizing the Geocentric Phase 82
8.3.1 Original results from [*Garot*, 2006] 83
8.3.2 Selecting an optimizer for the first phase 83
8.3.3 Improved results using OPTIDUS-PSO 83
8.4 Optimizing the complete mission 84
8.4.1 Settings for the optimization runs 86
8.4.2 Comparison of the results 86
8.5 Local optimization using random sampling 87
8.6 Local optimization using Powell's method 90
8.7 Conclusions and recommendations for the Solar polar sail mission 94

9 Third Global Trajectory Optimization Competition 97
9.1 Introduction to GTOC 97
9.2 The GTOC3 problem 97
9.3 Solutions of the GTOC3 problem 98
9.4 Simulation Model 102
9.4.1 First order differential equations 102
9.4.2 Integrator 102
9.4.3 Thrust direction parametrization 104
9.4.4 Thrust magnitude parametrization using Chebyshev polynomials . 104
9.4.5 On/off switching 105
9.5 Launch conditions 106
9.5.1 First leg 106
9.5.2 Other legs 106

9.6 Solution vector 106
9.7 Constraint handling 106
9.8 Optimization 107
9.8.1 Global optimization 107
9.8.2 Local optimization 107
9.9 Conclusions and recommendations for GTOC3 109

10 Conclusions and recommendations 111
10.1 Fulfillment of objectives 111
10.2 Conclusions 111
10.3 Recommendations 112

Bibliography 113

A Test functions 119
A.1 De Jong's functions 119
A.1.1 De Jong F1 119
A.1.2 De Jong F2, aka Rosenbrock 119
A.1.3 De Jong F3 119
A.1.4 De Jong F4 120
A.1.5 De Jong F4, modified 120
A.1.6 De Jong F5 120
A.2 Corana's parabola 120
A.3 Zimmermann's problem 121

B Numerical approximation of derivatives 123

C Parallel Processing in Fortran using OpenMP 125
C.1 What is parallel processing, and why is it needed? 125
C.2 Multithreaded Fortran using OpenMP 126
C.3 Performance caveats in a multithreaded environment 127
C.4 Pure functions 128

D api documentation 131
D.1 Design 131
D.2 Evolutionary Algorithms 131
D.2.1 RaceType 131
D.2.2 Individual 133
D.2.3 Population 135
D.2.4 Managing the population 135
D.2.5 Mutation 136
D.2.6 Crossover 136
D.2.7 Selection 137
D.2.8 Routines for GAVaPS 138
D.2.9 Other routines 140
D.3 PSO/PPO 141
D.3.1 Predator type 141
D.3.2 PSO/PPO routine 141
D.4 Differential Evolution methods 141
D.5 Multi Objective Routines 142
D.6 Random numbers 143
API index 146

API index 146

List of Figures

2.1 Example of the objective values for some solutions of a multi-objective problem. 6
2.2 Distribution of 3000 points picked using a pseudorandom number generator (left) and using a Sobol sequence (right) 10
2.3 Successive bracketing of a minimum using a golden section search. 13
2.4 Successive minimizations along coordinate directions in a long, narrow valley [*Press et al.*, 2007] 14
2.5 Comparison of local optimizers applied to the Himmelblau function. . . . 17
2.6 Comparison of local optimizers applied to the Himmelblau function. (zoomed in) 18
2.7 Example of the bounding intervals for a function 19

3.1 Example of a Gray coded sequence 24
3.2 Flowchart of a simple genetic algorithm 25
3.3 Example of a single point crossover 26
3.4 Example of a two-point crossover 26
3.5 Example of a uniform crossover 26
3.6 Example of a roulette wheel as used by fitness-proportionate selection. . . . 30
3.7 Example of two selections done using SUS 30
3.8 Transfer functions for the cases where condition (3.10) is satisfied (left) and not satisfied (right) 31

4.1 A flock of birds in Anchorage 35
4.2 Flowchart of a PSO algorithm 36
4.3 Fitness as a function of the population size for the Himmelblau function, using the aggregate results of 100 runs for each population size. 40
4.4 Fitness as a function of the population size for the Griewank function with dimensionality $N = 7$, using the aggregate results of 100 runs for each population size. 40
4.5 Fitness as a function of the population size for the Griewank function with dimensionality $N = 50$, using the aggregate results of 100 runs for each population size. 41
4.6 Calculating a particle's position when bouncing against a parameter boundary. 43
4.7 Movement of a particle swarm during an optimization of the Himmelblau function; the optimum is located at the coordinates $(3,2)$. The swarm sized used was $S = 40$. 44
4.8 Movement of a particle swarm during an optimization of the Himmelblau function, with a predator (indicated by the red dot); the optimum is located at the coordinates $(3,2)$. The swarm sized used was $S = 40$. 45

5.1 Flowchart of a basic Differential Evolution algorithm 48
5.2 Movement of a population of solutions during an optimization of the Himmelblau function using Differential Evolution, population size $S = 12$ and scheme Tasoulis1; the optimum is located at the coordinates $(3,2)$ 52
5.3 Movement of a population of solutions during an optimization of the Himmelblau function using Differential Evolution, population size $S = 12$ and scheme Tasoulis2; the optimum is located at the coordinates $(3,2)$ 53
5.4 Movement of a population of solutions during an optimization of the Himmelblau function using Differential Evolution, population size $S = 12$ and scheme Tasoulis3; the optimum is located at the coordinates $(3,2)$ 54
5.5 Movement of a population of solutions during an optimization of the Himmelblau function using Differential Evolution, population size $S = 12$ and scheme Tasoulis4; the optimum is located at the coordinates $(3,2)$ 55
5.6 Movement of a population of solutions during an optimization of the Himmelblau function using Differential Evolution, population size $S = 12$ and scheme Tasoulis5; the optimum is located at the coordinates $(3,2)$ 56
5.7 Movement of a population of solutions during an optimization of the Himmelblau function using Differential Evolution, population size $S = 12$ and scheme Tasoulis6; the optimum is located at the coordinates $(3,2)$ 57
5.8 Average distance of the population to the global optimum as a function of the generation during an optimization of the Himmelblau function using several DE schemes. 58
5.9 Fitness value of the best individual per generation, during an optimization of the Himmelblau function using several DE schemes. 58

6.1 Example of the objective values for some solutions of a multi-objective problem. 59
6.2 Example of a population in which the first four Pareto fronts have been ranked 61
6.3 The crowding distance calculation [*Deb et al.*, 2001]. 62
6.4 Analytical solution with the minimal values of f_2 from (6.6) 64
6.5 Exploration of the search space and results for optimizing (6.6) using the first NSGA algorithm, from [*Deb*, 1999] 64
6.6 Movement of a population of solutions while optimizing the problem given by (6.6). This optimization is done using Differential Evolution, and uses NSGA-II to rank the population. 66

7.1 Contour plot of the Himmelblau function. 68
7.2 Plots of Griewank's function in 1 dimension. 68
7.3 Histogram showing the amount of non-zero coordinates as a function of dimension i for 100 optimization runs of Griewank's function. 69
7.4 Plots of Schaffer's f_6 function in 1 dimension. 69
7.5 Plots of the Deb-Tan original and modified function in 1 dimension. . . . 70
7.6 Histograms of the best fitness values of the Himmelblau function found by the various implementations. 75
7.7 Histograms of the best fitness values of the 50-dimensional Griewank function found by the various implementations. 75
7.8 Histograms of the the difference between the theoretical optimum and best fitness values of Schaffer's f_6 function in 5 dimensions found by the various implementations. 76

7.9 Histograms of the difference between the theoretical optimum and the best fitness values of the 3-dimensional Deb-Tan function found by the various implementations. 76

8.1 A proposed solar sail. (Source: `http://www.u3p.net`) 77
8.2 Forces acting on a solar sail. (2-dimensional and not to scale) 78
8.3 Solar sail cone and clock angle in the local sail reference frame. Source: [*McInnes*, 1999] 78
8.4 The sailcraft's orbit in the Earth escape phase is divided into an acceleration, feathering and drag part. Source: [*Garot*, 2006] 79
8.5 The acceleration phase is divided into six sections. Source: [*Garot*, 2006] . 80
8.6 Trajectories for the best found solution for the solar polar sail mission in [*Garot*, 2006]. 82
8.7 Polynomials through the pitch angles that correspond to the shortest flight times for the geocentric phase. Source: [*Garot*, 2006] 83
8.8 Optimal pitch angles α_E as a function of the GTO-orientation ω. The circles are the outcomes of optimization runs, the black bars denote their standard deviation and average, and the line is a the fit to the sinusoidal function (8.17). 85
8.9 Comparison of the time of flight for the historical results (polynomial fit), free optimization (PSO) and optimization using the sine fit for the pitch angles (sine fit). 86
8.10 Average of the best fitness values for each optimizer/population size combination for the complete solar sailing mission. 88
8.11 Best of the best fitness values for each optimizer/population size combination for the complete solar sailing mission.. 88
8.12 Fitness versus the number of evaluations for the different optimizers used, and the Pareto-fronts of those points. 89
8.13 Histograms of the change in objective function value of the Solar polar sail mission, when applying a Monte Carlo sampling around the best found solution, using a Sobol random sequence (left) and a pseudo-random number generator (right) for different sampling area sizes (rows). 91
8.14 Distance from the spacecraft to the Moon during the geocentric phase as found using Powell's quadratically convergent method. 92
8.15 Trajectories of the spacecraft and the Moon during the geocentric phase as found using Powell's quadratically convergent method. 93
8.16 Trajectory of the spacecraft during the heliocentric phase as found using Powell's quadratically convergent method. 93

9.1 Orbits of the asteroids and Earth around the Sun, as given in the GTOC3 problem.. 99
9.2 Leg 1 of the original TUDelft GTOC3 solution (Earth to asteroid 96). 100
9.3 Leg 2 of the original TUDelft GTOC3 solution (asteroid 96 to 122). 100
9.4 Leg 3 of the original TUDelft GTOC3 solution (asteroid 122 to 85). 101
9.5 Leg 4 of the original TUDelft GTOC3 solution (asteroid 85 to Earth). 101
9.6 Definition of the thrust angles and transformation matrix for the thrust vector from the spacecraft oriented frame to the heliocentric frame. 104
9.7 Chebyshev polynomials 105
9.8 Optimal trajectory of the first leg for GTOC3 found using DE. 110
9.9 Optimal trajectory of the first leg for GTOC3 found using DE and a successive local optimization using Powell's method. 110

List of Tables

4.1 Comparison of optimal population sizes according to experimental data and equations (4.11) and (4.12). 39
4.2 Optimization parameters in PSO and PPO 42

5.1 Conversion between the DE scheme names used in this study and the “Storn” names. 50

6.1 Comparison of two different weights for weighted objective functions . . . 60

7.1 Average best fitness values for the Himmelblau function for as found by various optimizers. 72
7.2 Average and standard deviation of the best fitness values for the 50-dimensional Griewank function as found by various optimizers. 72
7.3 Average and standard deviation of the difference between the theoretical optimum and the best fitness values for Schaffer’s f_6 function in 5 dimensions as found by various optimizers. 73
7.4 Average and standard deviation of the difference between the theoretical optimum and the best fitness values for the 3-dimensional Deb-Tan function as found by various implementations. 73

8.1 Limits of the optimization parameters for the geocentric phase 83
8.2 Values of the coefficients of the sine-fit of (8.17). All coefficients are in degrees. 84
8.3 Optimal parameters for the solar sailing mission as found by the differential evolution runs. 87
8.4 Tabulated values of equation (8.18) 89
8.5 Top-5 of fitness values found by applying a random sampling local optimizer around the best solution of the solar polar sail mission. 90
8.6 Times of flight in days for the phases in the heliocentric phase. 92
8.7 Results of the optimization of the geocentric phase of the solar polar sail mission using several optimizers. 95

9.1 GTOC3 Spacecraft properties 98
9.2 Cash-Karp parameters for embedded Runge Kutta method, [*Press et al.*, 2007] 103
9.3 Solution vector, number of evaluations, fuel consumption and constraint violation of the best solution from the global and local optimizations for GTOC3. 108

C.1 Simultaneous addition in a multiprocessor environment without locks. . . . 128
C.2 Simultaneous addition in a multiprocessor environment with locks. 128

D.1 Execution order for different numbers of threads in a multi-core system.. . 144

Listings

2.1 Example of a nested do-loop optimizer. 8
6.1 Pareto front ranking algorithm from [*Safipour*, 2007] 64
6.2 NSGA-II Pareto front ranking algorithm 65
6.3 NSGA-II Crowding distance algorithm 65
B.1 Calculating the gradient numerically in Python 123

Glossary

Symbols and notations

α	Ch2: constant
α	Ch8: Sail pitch angle [°]
α_{drag}	Angle of attack in the drag phase [°]
α_E	Sail pitch angle in the Earth escape phase
α_S	Sail pitch angle in the heliocentric phase
β	Constant
β	Spread factor
δ	Ch2: Recency weighted running average coefficient
δ	Ch8: clock angle [°]
δ_i	Tolerance for equality constraint i
ε	Precision/tolerance
θ	Sail cone angle [°]
κ	Constant
λ	Change in position along gradient
μ_s	Sun's gravitational parameter, $\approx 1.3271 \cdot 10^{11}$kg^3 s^{-2}
ρ	Uniform crossover weight
ρ	Ch8: Atmospheric density [kg m^{-3}]
σ_f	Standard deviation of the fitness value for a population
σ_{share}	Sharing distance scale factor
Σ	Solution space
τ_μ	Trigonometric operator probability
τ_j	Stay time at asteroid j [days]
Φ	Ch2: Golden ratio conjugate ≈ 0.6180
Φ	Ch8: Transformation matrix from local to inertial reference frame
φ_1,φ_2	Uniform random number generator, output interval $[0, \varphi]$
χ	Ch2: Constriction coefficient
χ	Ch8: Angle between the velocity vector and the Sun line [°]
ω	Initial GTO orientation (with respect to the Sun line) [°]

Symbols and notations (continued)

∂	Partial derivative operator
∇	Gradient operator
A^I	Interval (for Interval Analysis)
A	Solar sail size [m^2]
$\mathbf{a}$	Acceleration of solution particle
$\mathbf{a}_{drag}$	Acceleration due to drag
$\mathbf{a}_{Moon}$	Acceleration due to gravitational pull by the Moon
$\mathbf{a}_{Sun}$	Acceleration due to gravitational pull by the Sun
a	Ch2: Starting point golden section search
a	Crossover weight
$a_0...a_4$	Coefficients for the sum of sines fit
B^I	Interval (for Interval Analysis)
b	Starting point golden section search
$b_1...b_4$	Coefficients for the sum of sines fit
C	Constant
C_1	Cost function [M€]
c^i	Expected number of times a solution may reproduce
$\bar{c}$	Average reproduction count for a population
c_{mult}	Reproduction count for the best solution
c_r	Position error [m]
$c_1...c_4$	Constants
c	Testing point golden section search
d	Testing point golden section search
$d(i,j)$	Distance between two solutions i and j
d	Size of random sampling box
$\mathbf{e}_i$	Unit vector, along direction i of the coordinate space
F	Mutation constant
f	(Set of) objective functions(s)
f'	Derivative of f
f''	Second derivative of f
f^i	Fitness value for solution i
f^{*i}	Scaled fitness value for solution i
$\bar{f}$	Average fitness value for a population
f_k	Objective function k
f_{min}	Minimum reproduction count
f_{max}	Minimum reproduction count
f_{pred}	Predator fear constant
G	Universal gravitational constant $\approx 6.668 \cdot 10^{-11}$[Nm^{2}kg^{-2}]
g_i	Equality constraint i
h_i	Inequality constraint i

Symbols and notations (continued)

h	Altitude above the Earth [m]
I_{sp}	Specific impulse [m s^{-1}]
i_{gen}	Generation number
l_k	Lower boundary for solution component k
K	Dimensionality of objective values
M	Number of knapsack items
M_{Earth}	Mass of Earth $\approx 5.9737 \cdot 10^{24}$[kg]
$M_{Mercury}$	Mass of Mercury $\approx 3.3022 \cdot 10^{23}$[kg]
M_{Venus}	Mass of Venus $\approx 4.8685 \cdot 10^{24}$[kg]
$\mathbf{m}$	Mutation vector
$\mathbf{m}$	Ch8: sail force vector
m_f	Final mass [kg]
m_i	Initial mass [kg]
N	Dimensionality of solution space
N	Ch8: Magnitude of the drag force
$\mathbf{n}$	Sail normal vector
n_g	Number of equality constraints
n_{eval}	Total number of evaluations
n_{gen}	Total number of generations
n_h	Number of inequality constraints
n_m	Number of manoeuvres
$\mathbf{P}_i$	Point in solution space at iteration i
P	Solar radiation pressure
P_i	Profit of knapsack item i
$\mathbf{p}$	Specific angular momentum [m^2s^{-1}]
p_1, p_2, p_3	Trigonometric weights
p_f	Predator (fear) probability
p_m	Mutation probability
p_r^i	Reproduction probability for solution i
p_{xover}	Crossover probability
q	Dynamic pressure [N m^{-2}]
$\mathbf{r}$	Sun vector [m]
$\hat{\mathbf{r}}$	Unit Sun vector [-]
r^i	Rank of individual i (based on fitness)
r_E	Sun-Earth distance [1 AU]
S	Population size
T_{launch}	Launch date
t_f	Time of flight [days]
$u()$	Uniform random number generator
$\mathbf{u}_i$	Optimization direction vector i

Symbols and notations (continued)

u_k	Upper boundary for solution component k
$\mathbf{V}$	Velocity vector [m s^{-1}]
$\mathbf{v}$	Velocity of solution particle
$\mathbf{v}_p$	Velocity of the predator
ΔV	Speed difference [m s^{-1}]
W_E	Solar flux at the Earth ≈ 1367.6[W m^{-2}]
W_i	Weight of knapsack problem i
w_k	Weighting function weight k
w	Inertia weight
X_i	Presence of knapsack item i
$\mathbf{x}$	Solution vector
$\mathbf{x}_{best}$	Best solution in the population
$\mathbf{x}_p$	Position of the predator
$\mathbf{x}*_i$	Historical best solution vector for solution i
$\mathbf{x}*_G$	Historical best solution vector for the population
Δx	Change in x
x_i	Solution vector component i
x_i, n	Solution vector component n of solution i
$\mathcal{F}$	Augmented objective function

Abbreviations

AU	Astronomical Unit $\approx 1.495979 \cdot 10^{11}$[m]
BLX	Blend Crossover
DE	Differential Evolution
EA	Evolutionary Algorithm
EC	Evolutionary Computation
EP	Evolutionary Programming
FORTRAN	Formula Translating System
GAVaPS	Genetic Algorithm with Variable Population Size
GA	Genetic Algorithm
GALOMUSIT	Genetic Algorithm Optimization of a Multiple Swing-by Interplanetary Trajectory
GCC	GNU Compiler Collection
GTO	Geostationary Transfer Orbit
GTOC	Global Trajectory Optimization Competition
IA	Interval Analysis
MJD	Modified Julian Date
MOO	Multi-Objective Optimization
NSGA	Non-dominated Sorting Genetic Algorithm
OPTIDUS	Optimization Tool for Interplanetary trajectories by Delft University Students
PPO	Predator-Prey Optimization

Abbreviations (continued)

PSO	Particle Swarm Optimization
SA	Simulated Annealing
SAPPO	Silva Adaptive PPO
SBX	Simulated Binary Crossover
SPSO	Standard PSO
SUS	Stochastic Universal Sampling
TSP	Travelling Salesman Problem

Summary

Optimization of spacecraft trajectories is an interesting area of research, where a lot of development takes place: in the past years, an optimizer called OPTIDUS was implemented, which is able to do optimization of trajectories (and other problems) using an evolutionary algorithm (EA). There are other optimization algorithms available which (according to literature) also perform well on trajectory optimizations. These are Particle Swarm Optimization (PSO) and Differential Evolution (DE).

A new version of OPTIDUS is implemented, which not only supports evolutionary algorithms, but also several variants of differential evolution and particle swarm optimization. These algorithms are also implemented in a way that can exploit the availability of multiple processing cores in a computer, which can give a very good improvement in the runtime of optimizations. Also, a local optimizer using Powell's quadratically convergent method is implemented.

This new version of OPTIDUS is benchmarked against several PSO implementations, and it is found that performance on several (mathematical) testing functions is comparable to existing implementations. However, it is not possible to appoint DE or PSO as the best algorithm in general, as that is problem dependent.

The software is also applied to improve the optimization of a Solar polar sailing mission. By comparing the number of function evaluations and the best fitness value found for a family of optimizers (several variants of PPO and DE and an EA), it is found that DE gives the best results, giving an reduction in mission cost almost 4% compared to the original result found using an EA. The use of a local optimizer on this best result is able to improve it even further by another 0.15%.

Finally, this software is used to find a solution to the GTOC3 problem, where the goal is to find a low-thrust trajectory for a spacecraft to rendezvous with several asteroids. The hard part of this mission are the rendezvous constraints: upon arrival at an asteroid, the location and velocity of the spacecraft relative to the asteroids have very low margins, making a very large part of the solution space infeasible. For that reason, an augmented objective function which takes these violations into account is used. Both an augmentive function in the form of a weighted sum (of violations) and a multi-objective function are considered. The multi-objective variants gives better results, but is not compatible with the local optimizer, Powell's method.

Only the first part of the mission, from Earth to the first asteroid, is optimized. Using DE, a solution is found, which almost satisfies the constraints, but not completely. By applying Powell's method to this solution, the constraint violations are further reduced, but still a small violation of the position remains.

It is found that doing global optimization using DE can generate solutions with quite large constraint violations, but that subsequent application of a local optimizer can reduce those violations, while not changing the solution considerably.

How the solution of this first leg compares to solutions by other teams is not known, as the per-leg results have not been published.

Chapter 1
Introduction

1.1 Optimization of trajectories

When sending a spacecraft into space, in most cases it will have to travel to a destination orbit. For example, the Cassini spacecraft is currently orbiting Saturn, but first had to fly there using an interplanetary transfer trajectory.

These trajectories will have to be optimized to reduce mission cost. This can be done in several ways: for example, a Hohmann transfer orbit is the trajectory with the lowest ΔV budget to transfer an object between two coplanar circular orbits around a body using only impulsive shots at the beginning and end of the orbit. This trajectory can be determined using an analytical method.

In principle, a Hohmann orbit could be used to send a spacecraft from Earth to an outer planet, but the travel time will become very long: the travel time for a Hohmann transfer orbit scales cubically with the sum of the beginning and ending radius. A transfer from the Earth to Pluto this way will take more than 45 years.[1]

For this reason, other trajectories are needed. One possibility is to use a more direct trajectory – but this comes at the cost of requiring a larger ΔV.

For engines that deliver thrust by pushing away matter – this includes chemical rockets as well as electronically powered engines which shoot out ions and plasma – the ΔV for a given amount of fuel is proportional to the specific impulse (I_{sp}), which is a property of the propellant and engine.

Thus, by using a propulsion method with a higher specific impulse, a higher ΔV can be attained. According to [*Zandbergen*, 1997], a chemical rocket using liquid hydrogen and oxygen has a specific impulse in the order of 400 seconds, whereas for example electrostatic ion thrusters can reach much higher values, in the orer of 2000 seconds: ESA's SMART-1 mission used a ion engine with Xenon as propellant. This engine had a specific impulse of 1640s [*ESA*, 2004]. There is a drawback: because ion thrusters need electrical power to generate thrust, the amount of thrust is limited by available power. However, if power can be supplied over an extended period (for example by a solar cell or radioisotope thermoelectric generator), the engine can also be operated over a long time.

Another method is to make use of aerogravity-assisted flybys. This was used in the Cassini-Huygens mission. By passing a planet at a close distance, momentum will be exchanged, and the path of the spacecraft is deflected. This way, the speed and orientation can be changed for free (without consuming fuel), but requires that the trajectory brings the spacecraft close to the planet.

[1] Actually, this number refers to a transfer between the two circular orbits with equal semi-major-axis as the Earth and Pluto, and without a difference in inclination.

In the case of a Hohmann transfer, the engine is only fired at the beginning and at the end of the trajectory. It is possible to reduce the time of flight using so-called "deep space manoeuvres", where the engine is also fired at other moments. This can also be used to reach planets used for gravity-assists.

Finally, Solar sailing uses the radiation pressure from the Sun's light for propulsion. It does this by deflecting incident light using a "sail". Because the photons, which are deflected have momentum, a force is exerted on the sail, which can be used as a propulsion force. Because no fuel is used, this force can be used over long periods of time – but as the intensity of the Sun's light decreases quadratically with the distance from the Sun, its use is limited to the inner regions of the Solar system.

1.2 Motivation

In the last couple of years, two programs have been written at Delft University of Technology to enable the optimization of trajectories using the above three techniques. These two programs are GALOMUSIT and OPTIDUS.

GALOMUSIT (Genetic Algorithm Optimization of a Multiple Swing-by Interplanetary Trajectory) uses evolutionary algorithms to optimize analytical representations of trajectories. By using Lambert targeting, this enables it to optimize trajectories with high-thrust impulsive shots, gravity assists and deep space manoeuvres. An example of the usage of this software was the optimization of a trajectory from Earth to Neptune [*Melman*, 2007].

OPTIDUS (Optimization Tool for Interplanetary trajectories by Delft University Students) also uses evolutionary algorithms, but optimizes numerical representations of trajectories.

Because of this numerical representation, it needs to integrate the motion of a spacecraft over the complete trajectory (instead of only modelling impulsive thrusting or flybys). This makes it somewhat slower, but also more flexible: for example, in the past, a Solar sailing mission has been optimized [*Garot*, 2006].

OPTIDUS was also used to optimize the trajectories for the second Global Trajectory Optimization Contest (GTOC) problem [*GTOC2*, 2006]. This problem is the second one in a series of competitions, with the goal of testing (global) optimizers against each other. The second GTOC problem is to optimize a multiple asteroid rendezvous mission, where the participants can select four asteroids from a given set which they have to visit. The spacecraft to visit these asteroids can only use a low-thrust engine for its propulsion.

Unfortunately, [*Evertsz*, 2008] was not able to satisfy the mission constraints, especially the rendezvous constraints: when arriving at an asteroid, the speed difference between that asteroid and the spacecraft should be less than 1 m/s, while the distance between them should be less than 1000 km. Another constraint was that the mission should not take longer than 20 years.

OPTIDUS was also used by [*Orlando et al.*, 2007] in an attempt to solve the third GTOC problem [*Casalino et al.*, 2007]. The third GTOC problem is very similar to the second one, as again the participants are to select a set of asteroids to visit, but this time they have to return to the Earth at the end of the mission. Also, gravity-assists using the Earth may be used. The rendezvous constraints are the same as in the second GTOC problem. Again the rendezvous conditions at the asteroids were not satisfied.

In [*Vinkó et al.*, 2007], alternatives to evolutionary algorithms are benchmarked, and it was found that Particle Swarm Optimization (PSO) and Differential Evolution (DE) provide better optimization speed and robustness.

Finally, recent hardware requires software to make use of multithreading to fully utilize the capacity of that hardware.[2] The old version of OPTIDUS is not able to this. Especially

[2]Almost all current CPUs in computers contain more than one core.

population-based algorithms can benefit from this, because the evaluation of multiple individuals (which is where most of the processing time is spent) in a population can be done in a parallel way. This decreases the execution time of the optimization.

For the last two reasons, a complete overhaul of the OPTIDUS software will be done, and support for DE and PSO will also be implemented.

1.3 Thesis Objectives

- This thesis work will try to improve the performance of OPTIDUS by implementing PSO and DE. These algorithms will be benchmarked to an evolutionary algorithm.
- By making use of parallel computation, calculation time on systems with multiple cores can be reduced.
- This improvement will be benchmarked against historical results of the Solar polar sailing mission from [*Garot*, 2006].
- Also, an attempt will be made to find a solution to the third GTOC problem which does satisfy the constraints.

1.4 Organization of this report

This remainder of this report is organized as follows: chapter 2 introduces the concept of optimization and the associated terminology. Furthermore, several types of optimization algorithms (both evolutionary as well as the more mathematical oriented ones) will be discussed briefly. Chapter 3 describes population-based evolutionary algorithms and its building blocks in more detail. Chapter 4 contains a description of particle swarm optimization (PSO) and its variants, and introduces a method to restrict the movement of particles by letting them bounce against the boundaries of the solution space. Next, chapter 5 shows how differential evolution (DE) works. In chapter 6, multi-objective optimization is treated. A method to rank individuals in a multi-objective setting (NSGA-II) is introduced, and a proof of concept implementation is applied in a differential evolution optimization. Next, in chapter 7, PSO, DE and an EA are applied to four mathematical test functions, and convergence robustness for those algorithms is compared. In chapter 8, PSO, DE and an EA are used to optimize a solar polar sailing mission, for various population sizes. Furthermore, the solutions of these global optimizers are fed to several local optimizers (Monte Carlo sampling around the best found solution, and Powell's method), and the outcomes compared. In chapter 9, DE is used in an attempt to find solutions to the GTOC3 problem. Finally, chapter 10 contains the conclusions and recommendations for further research.

Chapter 2
Introduction to Optimization

This chapter introduces the concept of optimization and the associated terminology. Furthermore, it gives an overview of the different optimization methods that can be used to perform low-thrust trajectory optimization. Finally, a selection will be made of the algorithms to be implemented, so they can be compared.

2.1 Introduction and terminology of optimization

Optimization is the process of finding the optimal solution for a given problem. Using a more formal notation, this process can be described by having to find the optimal solution $\mathbf{x}$ in a solution space Σ, that optimizes a set of objective functions $f_k(\mathbf{x})$, where $k \in \{1, ..., K\}$. These solutions may not violate a set of constraints, usually divided into a set of n_g equality constraints $g_i(\mathbf{x}) = 0$ and a set of n_h inequality constraints $h_j(\mathbf{x}) \leq 0$.

For the case where $K = 1$, meaning the goal is to optimize just one objective function, this can be formulated as

$$\min_{\mathbf{x}\in\Sigma} f(\mathbf{x}) \text{ subject to } g_i(\mathbf{x}) = 0 \text{ and } h_j(\mathbf{x}) \leq 0 \text{ for } i \in \{1, ..., n_g\} \text{ and } j \in \{1, ..., n_h\} \quad (2.1)$$

where n_g is the number of equality constraints and n_h the number of inequality constraints.

The following paragraphs will explain the terms in the previous formulation using examples from trajectory optimization.

First of all, $\mathbf{x}$ is an abstract representation of the **solution**, which is composed of (independent) input variables, which might represent launch dates, travel times, spacecraft masses, initial velocities, etc., depending on the given problem. It is also possible to put discrete variables into the solution: for example, in the GTOC2 problem, an important part of the optimization was to determine which asteroids from a given set should be visited.

The components of $\mathbf{x}$ are bounded by the **search space** Σ. For example, the arrival date might be constrained by the position of the target body[1] or the initial velocity should be within the limits of the launcher.

The **objective function(s)** $f_k(\mathbf{x})$ is what evaluates the solution. This can be an integration over the complete trajectory of the mission using the input parameters given by the solution. A typical outcome of an objective function is a mission cost estimate [*Garot*, 2006] or a fuel mass [*Elvik*, 2004], both of which can be a target for minimization.

[1] An example of a body which poses constraints on the arrival date is Pluto, which has an orbital period of 248 years and a rather eccentric orbit ($e \approx 0.25$). This eccentricity causes it to be more than 1.5 times further away from the Sun at aphelion than at perihelion. This means that at aphelion, available solar power is much lower, and sending data back to Earth is also more difficult.

In such cases, where the outcome is only one number, it is quite clear what the term *optimizes* is about: to find a mission which satisfies the requirements and has a minimal cost or fuel consumption.[2]

It is also possible that a problem has more than one objective. An example of this is given in [*Safipour*, 2007], where a trajectory around Neptune and Triton was optimized, using the following objectives:

- Maximize the number of flybys along the inner moons of Neptune
- Minimize the required ΔV-budget to get into the final science orbit

This type of optimization is known as **multi-objective optimization** (MOO). It is, however, difficult to determine an optimal solution in this case; suppose we have a two solutions for the Neptune orbit, one which flies past 20 moons and costs 1 km/sec and another one which visits 30 moons but also needs 2 km/sec. The solutions are depicted in figure 2.1 as solutions A and B. The arrow points into the direction in which both objective values get better. Without further information, it is impossible to determine that one of these solutions is better than the other one.[3]

However, there are two strategies which allow solutions to be ordered. Suppose a third solution C exists, which visits 25 moons, and consumes 2.5 km/sec worth of fuel. This solution is better than solution A for the visited number of bodies, but is worse for fuel consumption. However, compared to solution B, it is worse on both objective values. For that reason, it is said to be **dominated** by solution B. If a solution is not dominated by any other solution, it is said to be **Pareto optimal**. The solutions which are Pareto optimal form the (first) **Pareto front**. If this process is repeated without the solutions of that Pareto front, a second Pareto front can be found, and so on, until all solutions have been put in a front.

Members not in the first Pareto front are (objectively) worse than those in it. However, it is not possible to compare the individuals within a single Pareto front with each other, and extra criteria are required to pick the best one. Chapter 6 contains more details on how to implement Pareto front ranking.

[2] It is also possible to state the problem such that $f(\mathbf{x})$ should be maximized, but in this report minimization will be used.

[3] This is the same problem as that of complex numbers: it is not possible to order complex numbers, because the comparison operator is not defined for them. [*Almering*, 1993]

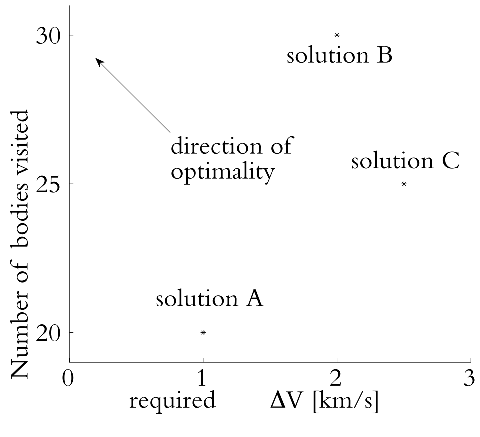


Figure 2.1 Example of the objective values for some solutions of a multi-objective problem.

Another method is to collapse the two (or more) objective functions into one using a **weighting function**.[4] This was done in [*Van der Pols*, 2006], where the objective function had two outcomes, the ΔV budget and the number of manoeuvres n_m necessary for station keeping in a halo orbit. These were combined using the following expression:

$$f = \frac{n_m}{\Delta V}$$

Note that this method requires extra knowledge about the problem in the form of the method in which the objective values are combined, or the weighting factors. This makes the use of weighting functions subjective.

Finally, the constraint functions $g_i(\mathbf{x})$ and $h_j(\mathbf{x})$ must be satisfied. Examples of the **inequality constraints** $h_i(\mathbf{x})$ are that the total mission lifetime should not exceed 20 years, or that the spacecraft may not come within a distance of 0.4 AU from the Sun during the mission. **Equality constraints** $g_i(\mathbf{x})$ are less common in trajectory optimization – an example can be fabricated for a rendezvous mission with a prolonged stay; upon arrival, the speed difference between the spacecraft and the target should be zero. In practice, the requirement is made less stringent, and stated as $|g_i(\mathbf{x})| < \delta_i$, making it an inequality constraint. This has been done in the GTOC3 problem, where the spacecraft should be closer than 1000 km to the target asteroid, and should have a speed difference of less than 1 m/s upon arrival.

How these constraints are enforced is another issue; a method commonly used is to create an **augmented objective function**. [*Michalewicz*, 1995] lists several options, from which [*Demeyer*, 2007] uses the following augmented objective function which was first given in [*Joines and Houck*, 1994]:

$$\mathcal{F} = F(X) + (C \cdot i_{gen})^{\alpha} c_r^{\beta} \tag{2.2}$$

In this equation, $F(X)$ is the original objective function, and represents the total ΔV budget, so it has to be minimized. The rest of the right hand side represents the penalty for not satisfying the constraints, and is influenced by i_{gen}, the number of the current generation, and c_r, the magnitude of the position-error at the end of the trajectory. The remaining terms, C, α an β are predefined constants. It is apparent from this equation that an increase in the position error c_r leads to an increase of the augmented objective function, and thus a less fit individual.

An alternative method to handle constraints is by giving members which do not satisfy the constraints the **death penalty**. This prevents them from entering the population completely. This might be a too strict method, as a solution which almost satisfies the constraints (and might evolve into one which does) can be better in terms of objective values. The augmented objective function does not suffer from this.

2.2 Steps in the optimization process

The goal of this thesis work is to find out if trajectory optimization using evolutionary methods can be improved. However, such optimizations are usually carried out by applying several methods sequentially:

According to [*Becerra et al.*, 2005] it is necessary to prune the search space. This means that the region(s) of the solution space where feasible solutions can be found should be identified. The easiest way to implement this is by sampling the solution space. Because the samples taken do not depend on the objective values of the other samples, and explore

[4]Note that a weighting function would normally be in the form of (6.1): $\sum_{k=1}^{K} w_k f_k$, but any function combining the outcomes of the objective functions into one number is acceptable.

the complete solution space, the methods used for this are also classified as *global one-shot optimizers*. Section 2.3 lists several of these optimizers.

Next, an evolutionary algorithm is to be used. In this thesis, only population-based evolutionary optimizers have been tested. These work by creating a population of solutions and evaluating their objective functions. The ones with good objective values are then combined and mutated to generate a new population. By selecting only the good individuals, the process tries to create "good" offspring, but no guarantees are given that the best solution will be found. Section 2.4 has a short overview of these optimizers.

The final step is to identify the exact optimum. This is necessary, as most evolutionary algorithms end up near a (local or global) optimum, but do not reach the exact optimum [*Deb*, 1995]. By employing local optimizers, which explore the solution space around a given solution, it is possible to further improve the value of the objective function. Several local optimizers are listed in section 2.5.

2.3 Global one-shot optimizers

A global one-shot optimizer is a method which tries to find an optimum for a function by evaluating it at multiple locations in the complete solution space. Because the optimizer does not reuse the output of those evaluations as input for other evaluations, it is called one-shot.

2.3.1 Nested do-loop

If the solution space is discrete (meaning that the components of the solution vector only take on discrete values), it is possible to check all possible solutions using a (nested) do-loop (also called factorial design or enumerative sampling), to determine the optimal value. For example, a function on a 10^2 grid can be sampled using listing 2.1.

However, in trajectory optimization, the solution space is either too big to be checked completely, or is not discrete. Therefore, this method is not used.

```
best = -1
best_i = -1
best_j = -1

do i = 1, 10
  do j = 1, 10
    obj = objective_function(i, j)
    if (obj > best) then
      best = obj
      best_i = i
      best_j = j
    end if
  end do
end do
```

Listing 2.1 Example of a nested do-loop optimizer.

2.3.2 Regular sampling

Regular (or grid) sampling is similar to nested do-loops, but only samples a subset of all possible solutions. This makes sampling of continuous functions possible, and can also lead to a big reduction in the number of solutions to be evaluated. The implementation is similar to the one given above.

Unfortunately, the large number of input parameters for orbit optimization problems prohibits the successful application of this method too: if a problem has 10 variables x_1 ... x_{10}, and each variable is sampled in 100 grid points, this means $100^{10} = 10^{20}$ possible solutions have to be evaluated, which is not doable on current hardware.

2.3.3 Monte Carlo sampling

Monte Carlo (or random) sampling chooses the points to be sampled randomly. This helps to prevent problems which might be caused by regular sampling.[5] Another feature of Monte Carlo sampling is that the number of samples can be extended, as opposed to grid sampling, where the layout of the grid determines the number of samples. Because pseudo-random numbers do not depend on the previous samples taken, generating new solutions is not a problem.

The downside is that populations generated using Monte Carlo sampling can suffer from clustering in the generated solutions, as shown in figure 2.2.

Although it uses a random generator to sample points, experiments are repeatable if a pseudorandom generator is used. By using a fixed seed value, the same points will be generated each time.

2.3.4 Sobol sampling

Sobol sampling is similar to Monte Carlo sampling, but instead of using a (pseudo-)random number sequence to generate the points to be sampled, these points are created using the Sobol quasirandom sequence. This helps to sample the random space more uniformly than random numbers [*Burkardt*, 2006]. Figure 2.2 shows that a set of points generated using a normal random number generator (left) has lots of clusters, whereas the points generated using a Sobol sequence (right) are distributed more evenly. Just as normal Monte Carlo sampling, continuing the sample sequence is possible, and the samples taken are repeatable.

2.4 Population based evolutionary optimizers

Population-based evolutionary optimizers are optimization methods which work by simulating a population of solutions, which - hopefully - evolve towards an optimal solution. This evolution takes place by combining and mutating individuals from the population, and using those to form a new generation. By selecting only the good individuals to form the next generation, the individuals inside that generation tend to improve.

2.4.1 Genetic Algorithms

Genetic algorithms (GAs) are probably the best known evolutionary optimizer; they make use of a population of solutions, which are allowed to evolve.

[5] For example, if a launch date is sampled with a resolution of exactly 24 hours, the launch point on Earth will always have the same orientation towards the Sun. If this orientation influences the outcome of the objective function, the regular sampling has sampled only one value of that orientation.

These solutions are represented using genomes, which are composed of genes. In genetic algorithms, these genes are built up from binary numbers, i.e. strings of 0 and 1's. [*Goldberg*, 1989].

Of course, it is only possible to optimize a function using this technique if this evolution process is directed. By evaluating each solution using one or more objective functions, it is possible to compare those solutions, and determine which ones in the population are good, resulting in a fitness value.

These good individuals then have a higher chance to procreate than the bad ones. During the process of procreation, several operators can be applied to the individuals: well-known operators are crossover and mutation. Because of selective pressure (caused by the chance to procreate), the quality of the solutions has a tendency to get better. Once an optimum is reached, the optimization is stopped.

There are several methods to determine the moment at which an optimization needs to be stopped: for example, the user might set a limit on the number of generations. Another option is to stop the optimization if the best solution has not been improved for the last couple of generations (where the user can choose how many generations of stagnation are allowed).

2.4.2 Evolutionary Algorithms

Evolutionary Algorithms (EAs) are similar to GAs, with the difference that they do not use genes built up from strings of zeros and ones, but instead use a vector of floating-point numbers for the genomes. This can allow for a much easier mapping of the problem to the real world, and allows a lot of extra operators. Because EAs are very similar to GAs, they are sometimes (mistakenly) called GAs. For example, the original implementation of OPTIDUS uses an algorithm which is called "Genetic Algorithm with Variable Population Size (GAVaPS)", however, in reality, it is an EA, as it uses floating-point numbers.

The main reason for using this GAVaPS algorithm is that the method of applying selective pressure is different: all solutions have an equal chance of procreating, but when a new solution has been formed, its lifetime is determined according to its fitness. According to [*Michalewicz*, 1996] this leads to faster convergence of the algorithm. In practice however, [*Garot*, 2006] encountered problems with explosions of the population size.

Note that because of the similarity between GAs and EAs, they will both be treated in chapter 3.

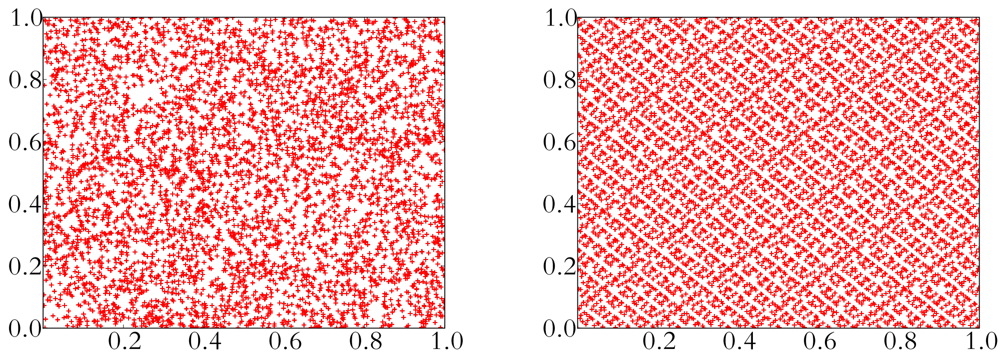


Figure 2.2 Distribution of 3000 points picked using a pseudorandom number generator (left) and using a Sobol sequence (right)

2.4.3 Particle Swarm Optimization

Particle Swarm Optimization (PSO) is an algorithm which tries to mimic the behavior of a flock of birds scavenging for food. Once food has been found, the flock of birds is attracted to that food.[6]

In PSO, this is modelled by having solutions fly[7] through the solution space. Each solution remembers its best position, and is attracted to that. Furthermore, the global best position is also known by each solution, and all solutions are attracted to that to. Finally, an inertia weight term is used to limit the speed of the solutions, and a random disturbance is applied to the speed at each generation, to maintain the diversity of the population. PSO can suffer from premature convergence, but methods to counter this (for example, introducing a predator), are known. PSO will be treated in more detail in chapter 4.

2.4.4 Differential Evolution

Differential Evolution (DE) is an algorithm that was introduced in [*Storn and Price*, 1997]. On the surface, it looks a lot like an EA as it uses populations of solutions, in which solutions are combined to give new solutions for the next generation. The big difference is in the operator used for combining individuals. For example, one operator listed in [*Tasoulis et al.*, 2004] uses the following method: each iteration starts by taking the difference between two randomly selected solutions and adding that difference to the best solution in the population. This way, a mutation vector is generated. This mutation vector is combined with a third random solution using a two point crossover, giving the trial solution. Next, this trial solution is evaluated, and if its fitness is better than that of the third random solution, it replaces that. If not, it is discarded. This process is repeated until convergence is reached or another termination criterion applies.

See chapter 5 for more information on DE.

2.5 Local optimizers

Local optimizers try to find the local optimum in the neighbourhood of a given starting point. [*Noomen*, 2007], [*Deb*, 1995] and [*Press et al.*, 2007] are good starting points for finding out more about local optimizers.

2.5.1 Newton-Raphson

For a problem that depends on one input parameter, a good method for finding local optima is applying the Newton-Raphson method on the derivative of the objective function. If the derivative is smooth enough, this method will converge quadratically [*Almering*, 1993].

Writing down the first order Taylor expansion of the derivative yields the following:

$$f'(x_i+\Delta x) = f'(x_i) + f''(x_i)\Delta x + O(\Delta x^2) \tag{2.3}$$

At an optimum $f'(x_i+\Delta x) = 0$, so after discarding $O(\Delta x^2)$, the following approximation for Δx is found:

$$\Delta x = -\frac{f'(x_i)}{f''(x_i)} \tag{2.4}$$

[6]This might not look like a pure EA, as there is no evolution involved, but because of the similarity with normal EAs (it uses a population of solutions which change over successive generations), it is classified as such.

[7]Flying in this context means that each solution is modelled as a particle having a position and a velocity in the solution space, and at each successive generation these are updated.

and the optimum can be found by iterating over:

$$x_{i+1} = x_i - \frac{f'(x_i)}{f''(x_i)} \tag{2.5}$$

In general, calculating the derivatives will have to be done numerically using a difference quotient, for example a first order central difference quotient: [*Vuik et al.*, 2006]:

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + O(h^2) \tag{2.6}$$

[*Vuik et al.*, 2006] also lists a second order central difference quotient:

$$f''(x) = \frac{f(x+h) - 2f(x) + f(x-h)}{h^2} + O(h^2) \tag{2.7}$$

Because computers only have limited accuracy, care should be taken when determining the stepsize h in the above equations. Appendix B shows how to do that.

2.5.2 Golden section search

Another method for finding the optimum of a problem with only one variable is the golden section search [*Deb*, 1995]. This method works by contracting a pair of points with the goal of finding a minimum.

The theory is as follows: if a (sufficiently smooth) function $f(x)$ is expected to reach a minimum between $x = a$ and $x = b$, there should be two testing points $x = c$ an $x = d$ where

$$f(c) < f(a) \wedge f(c) < f(b)$$
$$f(d) < f(a) \wedge f(d) < f(b)$$

For the golden section search, the location of points c and d are given by partitioning the interval $[a, b]$ using the golden ratio conjugate Φ:

$$\Phi = \frac{1}{\frac{1+\sqrt{5}}{2}} = \frac{2}{1+\sqrt{5}} = 0.6180... \tag{2.8}$$

Now, $R = 1 - \Phi = 0.3820...$ is used to construct the location of the testing points:

$$c = a + R(b-a) \tag{2.9}$$
$$d = b - R(b-a) \tag{2.10}$$

Now, the function is evaluated in c and d, and both objective values are compared. If $f(c) < f(d)$, the bracketing interval for the next step is set to $[a, d]$. If else, the interval is set to $[c, b]$.

With this new interval, the procedure is repeated. For example, the function in figure 2.3, has $f(d) < f(c)$. The new interval is then set to $[c, b]$ with testing points d and e. Because of the use of the golden section, d is still in the same location, and only one evaluation at e is needed. This procedure is repeated until the interval reaches a predefined minimum size, specified by the user.

2.5.3 Steepest descent method

The steepest descent optimizer is a method which can be used to optimize a function with more than one variable. It is given in [*Press et al.*, 2007]. The procedure is as follows:

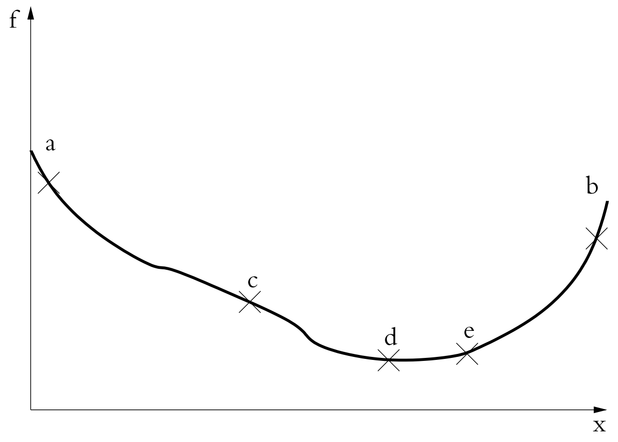


Figure 2.3 Successive bracketing of a minimum using a golden section search. See the text for the meaning of the points *a* to *e*.

1. Choose a starting point $\mathbf{P}_0$ and set $i = 0$
2. Determine the gradient of the function at that point: $\nabla f(\mathbf{P}_i)$
3. Determine the point $\mathbf{P}_{i+1}$ along the line $\mathbf{P}_i + \lambda \nabla f(\mathbf{P}_i)$ where f is optimal using a linear local optimizer (such as the Newton-Raphson method or a golden section search)[8]
4. Stop if convergence is reached (which occurs when λ is smaller than a tolerance ε specified by the user), else increase i and repeat from step 2.

This technique is simple to implement, but has a problem which is illustrated in figure 2.4. In a point P_{i+1}, the gradient is orthogonal to the gradient of that in point P_i, because that was the goal of the optimization. This causes the optimizer to "zig zag" with small steps, leading to slow convergence. If a good direction can be chosen, a much bigger stepsize is possible, and less function evaluations are needed, leading to faster convergence.

Figures 2.5(a) and 2.6(a) show the steps this algorithm takes to locally optimize the Himmelblau function (which will be introduced in section 7.1.1).

2.5.4 Powell's quadratically convergent method

[*Press et al.*, 2007, section 10.7] describes direction set methods, of which Powell's quadratically convergent method is considered to perform the best. It is built on the concept of conjugate directions, which are defined as follows:

Conjugate directions

Given a function $f(\mathbf{x})$, where $\mathbf{x} \in \mathbb{R}^N$. If this function is smooth, it can be approximated by its Taylor series around point $\mathbf{P}$:

$$\begin{aligned} f(\mathbf{P} + \delta\mathbf{x}) &= f(\mathbf{P}) + \sum_i \frac{\partial f}{\partial x_i} \delta x_i + \frac{1}{2} \sum_{i,j} \frac{\partial^2 f}{\partial x_i \partial x_j} \delta x_i \delta x_j + \ldots \\ &\approx c + b \cdot \delta\mathbf{x} + \frac{1}{2} (\mathcal{A} \cdot \delta\mathbf{x}) \cdot \delta\mathbf{x} \end{aligned} \tag{2.11}$$

[8]Even though the function is multi-dimensional, this is possible, because the problem is then reduced to finding the optimal value of λ for $f(\mathbf{P}_i + \lambda \nabla f(\mathbf{P}_i))$

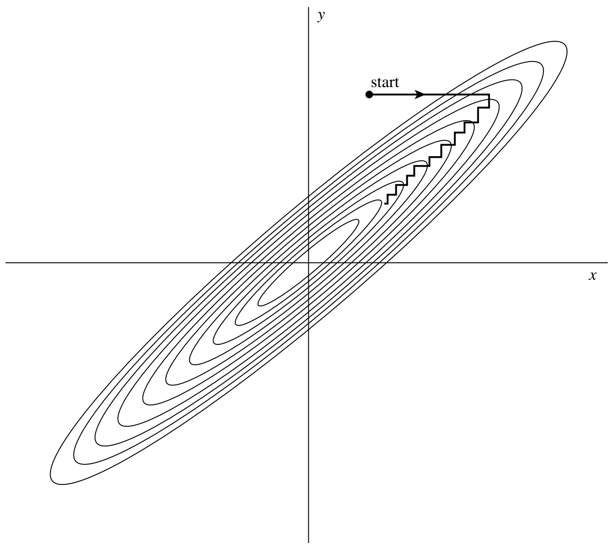


Figure 2.4 Successive minimizations along coordinate directions in a long, narrow valley [*Press et al.*, 2007]

where

$$c = f(\mathbf{P}) \quad \mathbf{b} = \nabla f|_{\mathbf{P}} \quad [\mathcal{A}]_{ij} = \frac{\partial^2 f}{\partial x_i \partial x_j}|_{\mathbf{P}} \tag{2.12}$$

The Taylor expansion of the gradient of f around point $\mathbf{P}$ is given by:

$$\nabla f|_{\mathbf{P}+\delta\mathbf{x}} = \mathbf{b} + \mathcal{A} \cdot \delta\mathbf{x} \tag{2.13}$$

So the change in gradient after moving along a certain direction δx then equals:

$$\delta(\nabla f|_{\mathbf{P}}) = \mathcal{A} \cdot \delta\mathbf{x} \tag{2.14}$$

Now, if the function has been optimized in a direction $\mathbf{u}$, the gradient along that direction should be 0:

$$\mathbf{u} \cdot \nabla f = 0 \tag{2.15}$$

If we next wish to move along a (non-zero) direction $\mathbf{v}$ to optimize further, we must make sure that the gradient in the $\mathbf{u}$ direction stays 0:

$$0 = \mathbf{u} \cdot \delta(\nabla f) = \mathbf{u} \cdot \mathcal{A} \cdot \mathbf{v} \tag{2.16}$$

If this holds for two vectors $\mathbf{u}$ and $\mathbf{v}$, they are said to be conjugate. This property is used in Powell's quadratically convergent method.

Powell's quadratically convergent method

If a set of N mutually conjugate vectors can be found, one pass of N optimizations will put the optimizer at the (quadratic form) optimum. Thus, if it is possible to create a set of mutually conjugate vectors, it is possible to locate an optimum.

Powell's method uses this to find an optimum as follows:

1. Initialize the set of directions $\mathbf{u}_i$ with the unit vectors in the solution space Σ: $\mathbf{u}_i = \mathbf{e}_i$
2. Save the starting position as $\mathbf{P}_0$.

3. Do one round of subsequent linear optimizations along the directions $\mathbf{u}_i$ for $i = 1,...,N$ using a linear local optimizer, giving the point $\mathbf{P}_N$ as the final point.

4. Drop $\mathbf{u}_1$, and shift all directions $\mathbf{u}_i$ left, so:

$$\mathbf{u}_i = \mathbf{u}_i + 1 \quad \text{for} \quad i = 1,...,N-1$$

5. Determine a new optimization direction vector: $\mathbf{u}_N = \mathbf{P}_N - \mathbf{P}_0$.

6. Go back to step 2 unless convergence was reached.

Now, (if the function is smooth), after k iterations of the above algorithm, the last k members of the set $\mathbf{u}_i$ will be mutually conjugate, thus, after N of these iterations, all vectors $\mathbf{u}_i$ are mutually conjugate, and a final round of optimizations then (in theory) is enough to locate the local minimum. This means, in theory, $N(N+1)$ linear optimizations are sufficient. Note that each linear optimization will incur multiple calls of the objective function.

However, problems can occur because similar vectors might appear in the list of direction vectors. If this happens, these vectors become linearly dependent, and do not span up the complete solution space anymore, thus making a region of the solution space unreachable. The heuristic solution proposed in [*Press et al.*, 2007] is to drop the term with the largest change in f in step 3 of the algorithm instead of $\mathbf{u}_1$.

Figures 2.5(b) and 2.6(b) show the steps this algorithm takes to locally optimize the Himmelblau function (which will be introduced in section 7.1.1). The first two iterations of this particular optimization are worked out here for clarity:

1. The set of directions is initialized with the unit vectors, so $\mathbf{u}_1 = [1,0]^T$ and $\mathbf{u}_2 = [0,1]^T$.

2. The starting position $\mathbf{P}_0 = [3.3,3]^T$ with $f(\mathbf{P}_0) = 36.44$

3. Now, the function is optimized along $\mathbf{u}_1$, giving $\mathbf{P}_1$:

$$\mathbf{P}_1 = [2.667, 3.000]^T \quad \text{with} f(\mathbf{P}_1) = 22.57 \quad f(\mathbf{P}_0) - f(\mathbf{P}_1) = 13.93$$

Optimizing along $\mathbf{u}_2$ yields the point $\mathbf{P}_2$:

$$\mathbf{P}_2 = [2.667, 2.173]^T \quad \text{with} f(\mathbf{P}_2) = 3.05 \quad f(\mathbf{P}_1) - f(\mathbf{P}_0) = 19.52$$

4. The second step caused the biggest drop in fitness value, so $\mathbf{u}_2$ is dropped from the list of direction vectors.

5. $\mathbf{P}_2 - \mathbf{P}_0 = [-0.631, -0.827]^T$ is inserted at the front of the direction vectors. Now, the new direction vectors are: $\mathbf{u}_1 = [-0.631, -0.827]^T$ and $\mathbf{u}_2 = [1,0]^T$

6. Go back to step 2 for another iteration:

2b. The new starting point is $\mathbf{P}_0 = [2.667, 2.173]^T$ with $f(\mathbf{P}_0) = 3.05$

3b. Optimize along $\mathbf{u}_1$, giving $\mathbf{P}_1$:

$$\mathbf{P}_1 = [2.764, 2.297]^T \quad \text{with} f(\mathbf{P}_1) = 2.21 \quad f(\mathbf{P}_0) - f(\mathbf{P}_1) = 0.84$$

Optimizing along $\mathbf{u}_2$ yields the point $\mathbf{P}_2$:

$$\mathbf{P}_2 = [2.915, 2.297]^T \quad \text{with} f(\mathbf{P}_2) = 1.46 \quad f(\mathbf{P}_1) - f(\mathbf{P}_0) = 0.75$$

4b. The second step caused the biggest drop in fitness value, so again $\mathbf{u}_2$ is dropped from the list of direction vectors.

5b. $\mathbf{P}_2 - \mathbf{P}_0 = [0.246, 0.124]^T$ is inserted at the front of the direction vectors.
Now, the new direction vectors are: $\mathbf{u}_1 = [-0.246, 0.124]^T$ and $\mathbf{u}_2 = [1, 0]^T$

6b. No convergence yet, so iterate again.

This example needs more than the theoretical $N(N+1)$ linear optimizations: this is because the golden section search implementation used does not grow its bracket by more than a factor 2, causing the last three rounds of this optimization to return points which are almost on a straight line.

2.6 Other algorithms

2.6.1 Interval Analysis

Interval Analysis (IA) is somewhat different from the rest of the algorithms in this chapter, as it does not try to optimize a single solution, but tries to find the best solution in the solution space by evaluating regions of the solution space at once.

It is a mathematical technique that can be used to solve certain classes of optimization problems. An introduction can be found in [*Caprani et al.*, 2002]. This method works by replacing numerical analysis by interval analysis, which can be used to calculate the output interval of a function, when given a certain input interval. An example of an interval operator is the addition operator:

$$A^I + B^I = [a_1 \quad a_2] + [b_1 \quad b_2] = [a_1 + b_1 \quad a_2 + b_2] \tag{2.17}$$

So, for example adding $A_I = 1 \pm 0.05$ and $B_I = 1 \pm 0.05$ gives:

$$[0.95 \quad 1.05] + [0.95 \quad 1.05] = [1.9 \quad 2.1] \tag{2.18}$$

Subtraction works in a similar manner:

$$A^I - B^I = [a_1 \quad a_2] - [b_1 \quad b_2] = [a_1 - b_2 \quad a_2 - b_1] \tag{2.19}$$

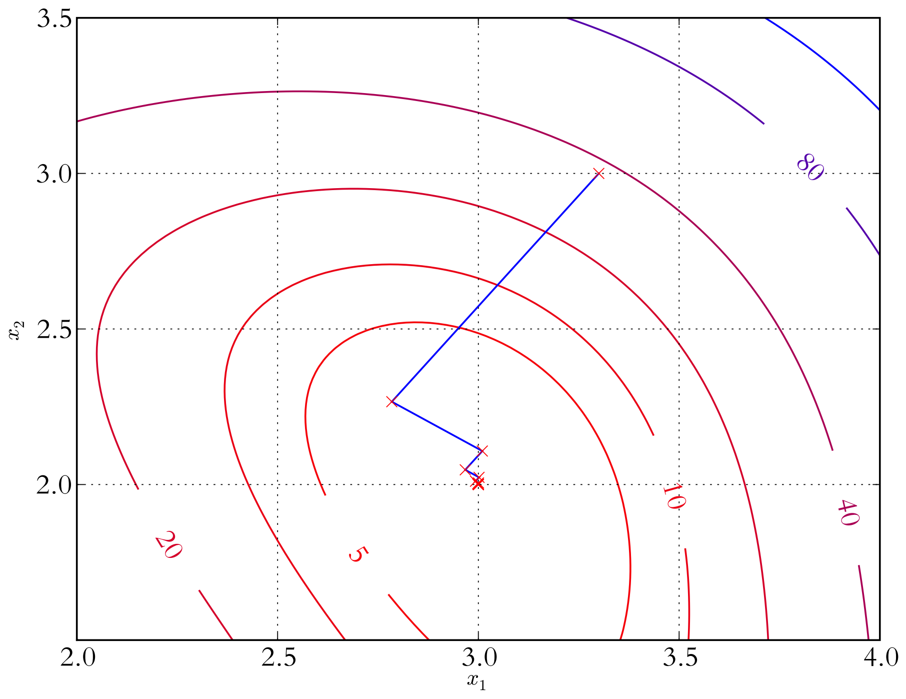
However, there are pitfalls: suppose we substract A^I from A^I. In that case, the expected result is 0. However, we get

$$[0.95 \quad 1.05] - [0.95 \quad 1.05] = [-0.1 \quad 0.1] \tag{2.20}$$

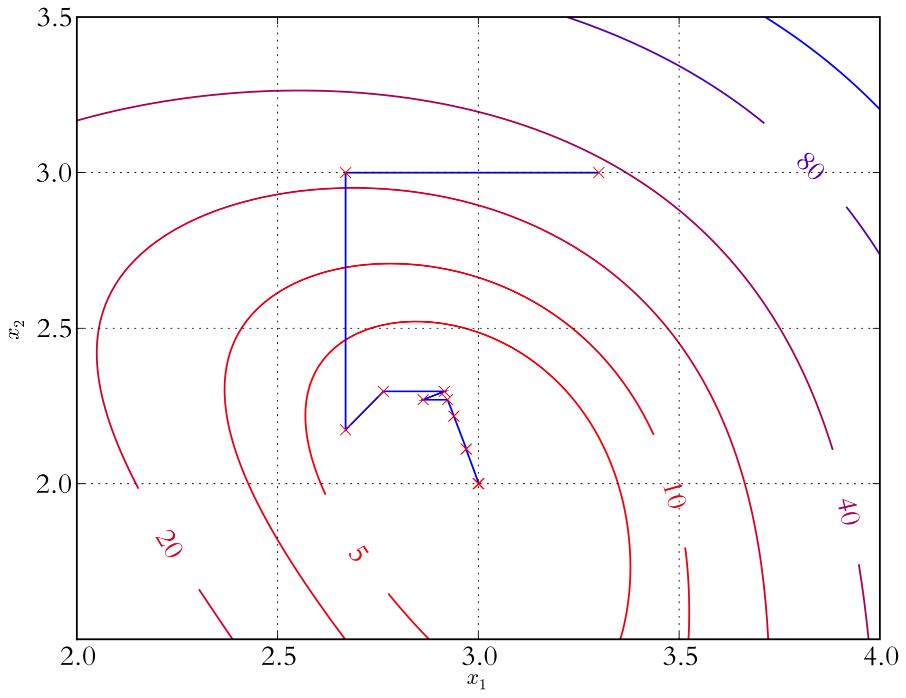
This is called the dependency problem.

By creating interval operators for all numerical functions, a simulation can be recreated in interval analysis. If this is done, the input interval can be split into several subintervals, and the output intervals of these subintervals can be determined.

By eliminating the input intervals for which the minimum is above the maximum of any other interval, the search space can be reduced. For example, in figure 2.7, it is obvious that interval I1 cannot contain the minimum, since all solutions in I3 have a lower value. Therefore, I1 can be eliminated. I2 and I4 can still contain the optimum, and thus are not eliminated.
After this step, the remaining intervals are divided into smaller intervals, and this procedure can be repeated, hopefully leading to an interval that contains the global maximum and is small enough that a (non-interval) solution which is close to the optimum can be determined.

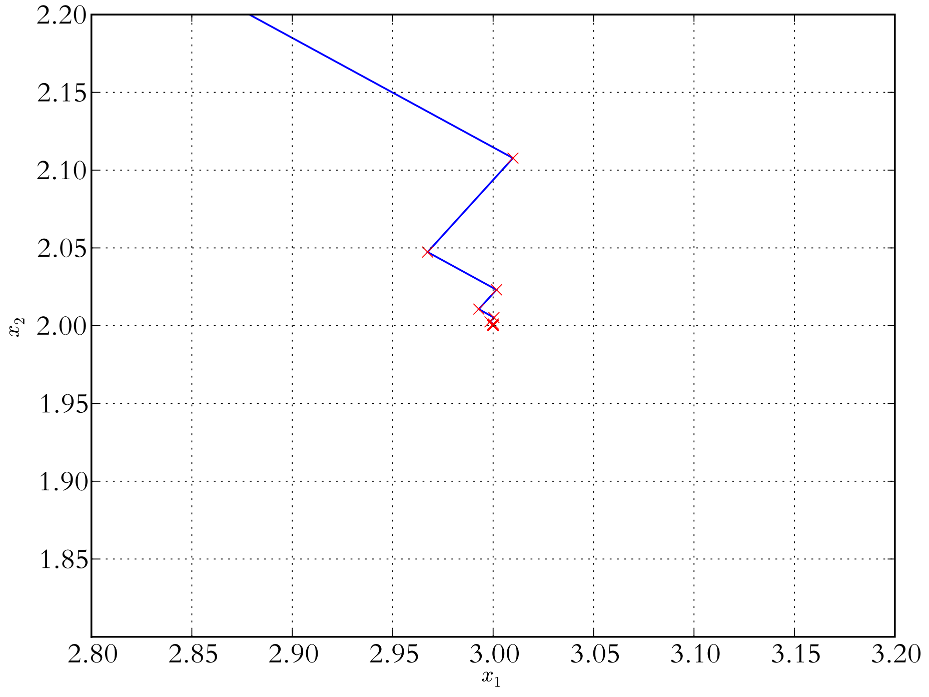


(a) Local optimization of the Himmelblau function using the steepest descent method

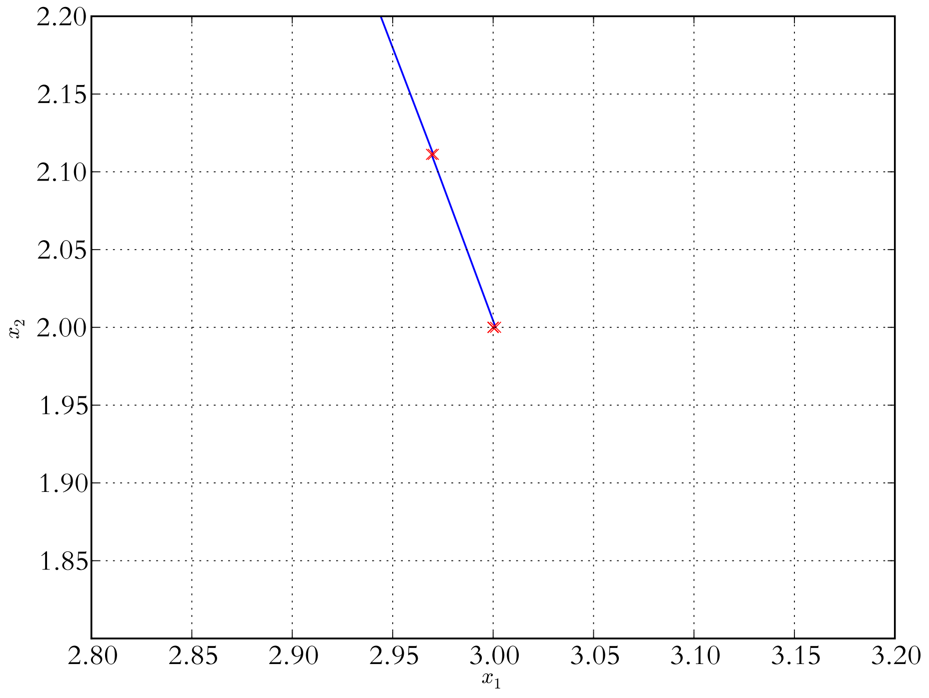


(b) Local optimization of the Himmelblau function using Powell's quadratically convergent method.

Figure 2.5 Comparison of local optimizers applied to the Himmelblau function. The upper figure shows how the steepest descent method moves through the solution space, the lower figure contains Powell's quadratically convergent method. The contours plotted are the values of the Himmelblau function.



(a) Local optimization of the Himmelblau function using the steepest descent method



(b) Local optimization of the Himmelblau function using Powell's quadratically convergent method.

Figure 2.6 Zoomed in versions of figure 2.5 displaying the final steps of both local optimizers.

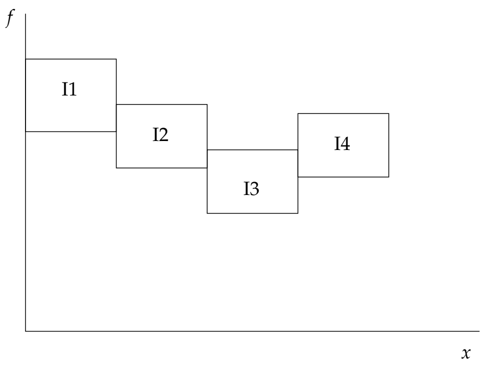


Figure 2.7 Example of the bounding intervals for a function

In [*Chu*, 2007], this technique is used to optimize several mathematical test functions successfully. Optimizing the reentry problem was less successful, with the dependency problem and the wrapping effect mentioned as causes for that.[9]

Another possible problem can arise from branching: suppose a problem containing the following pseudo-code is to be executed using interval analysis:[10]

```
do while (.true.)
  if (abs(angle) < 90) then
    call method1
  else
    call method2
  end if
end do
```

If the input angle interval now is $[80, 100]$, the algorithm will need to be executed twice, since there are two branches of execution to follow, one in which method1 is executed and another one in which method2 is called. This can lead to exponential growth of the number of branches, and thus also exponential growth of the calculation time. On the other hand, this branching also causes inherent parallelism in solving the problem (as the two branches of execution are independent and can be executed simultaneously), and a programming language which exploits this can profit from that.

2.6.2 Simulated Annealing

Simulated annealing (SA) is an algorithm in which (only) one solution is iteratively modified to find the best solution possible.

The idea behind SA is to mimic the annealing of metals. It simulates one individual solution, which is allowed to move randomly through the solution space, like an atom through a grid of a heated piece of metal. The distance this solution is allowed to move varies with the "temperature", which is gradually decreased as the process goes. The reason this seemingly random movement can be used for optimization is that in principle, only movements that increase the fitness value of the solution are accepted. However, to decrease the occurrence of premature convergence, movements that decrease the fitness of the solution are sometimes accepted - again as a function of the temperature. More information on SA can be

[9]The wrapping effect is caused by the fact that intervals are assumed to be independent, so the space spanned up by a set of n intervals is an n-dimensional hypercube. In reality, only a small part of this hypercube might be feasible, causing the input domain to be too large, which can cause the output domain to be too large.

[10]This pseudo code is inspired by an actual algorithm, namely the method to divide an orbit into intervals for a steering algorithm in [*Elvik*, 2004]

found in [*Kirkpatrick et al.*, 1983], [*Lu and Khan*, 1994], [*Nam and Park*, 2000] and [*Nam and Park*, 2002].

2.6.3 Hybrid methods

In [*Vinkó et al.*, 2007], a number of optimization methods is compared, including one named 'COOP'. This method combines DE and PSO, by alternating between N_1 iterations of DE, and N_2 iterations of PSO, passing the final population of each algorithm as the initial population for the following one at each switch. This can lead to faster and more robust optimization. Of course, other combinations of optimizers are also possible.

2.7 Selection of optimizers

From the optimization method listed in this chapter, the following have been implemented in the new version of OPTIDUS:

Monte Carlo sampling has been implemented as it is a method which can be used to quickly generate the initial population of any evolutionary optimizer. It can also be used to generate solutions for local optimizations at the end of a run.

Sobol sampling has been implemented because it is similar to Monte Carlo sampling, and might provide better coverage of the solution space than Monte Carlo sampling. It has been compared to Monte Carlo sampling only for local optimizations.

Evolutionary algorithms have also been implemented using floating point representation, crossover and mutation. Furthermore, several reproduction/selection schemes have been implemented. This was done, because it is the baseline optimizer to compare the other evolutionary optimizers against.

Particle Swarm Optimization has been implemented because it is a promising method, which has been found to work for several trajectory optimizations, [*Vinkó et al.*, 2007] and by varying the inertia weight, it is possible to have both global exploration as well as local convergence. [*Clerc and Kennedy*, 2002]

Differential Evolution has been implemented for the same reasons, that is: it has been found to work for trajectory optimizations [*Becerra et al.*, 2005]. Furthermore, it is claimed to have good local convergence at the end of optimization. [*Storn and Price*, 1997]

Powell's quadratically convergent method has been implemented because it is an analytical local optimizer which does not need the calculation of gradients, and has good convergence for smooth functions.

Hybrid methods have not been implemented directly, but by using the same data structures to store populations for EA, PSO and DE, it is possible to reuse the current data structures when constructing an optimizer chaining more than one optimization method.

Chapter 3
Evolutionary Computation

Evolutionary Computation (EC) is an alternative to the more traditional optimization methods such as enumeration, random selection (Monte Carlo) and calculus based methods (steepest descent).

This chapter will give an overview of the main building blocks of EC, such as the representation of solutions and evolutionary operators. A selection of these operators will be implemented, and the rationale for that is also given.

3.1 History of Evolutionary Computation

A history of the development of EC is given in [*Bäck et al.*, 1997]. Three groups came up with three different sets of ideas; in the USA, Fogel started out with *evolutionary programming* [*Fogel et al.*, 1966], and Holland developed *genetic algorithms* [*Holland*, 1967]. In Europe, Schwefel was part of a group of students that developed the concept of Evolution Strategies [*Schwefel*, 1977].

3.1.1 Evolutionary programming

In evolutionary programming (EP), *state machines* are constructed which can convert a series of input values into a series of output values. By comparing the actual output with the expected output, a fitness can be determined for that machine.

A population of machines is generated, and each generation, offspring machines are created by mutating the parents. The parents and offspring were combined, and the half of the population with the worst fitness was discarded.

In the mid-1980s the general EP procedure was extended to alternative representations including ordered lists for the traveling salesman problem, and real-valued vectors for continuous function optimization [*Bäck et al.*, 1997].

3.1.2 Genetic algorithms

Genetic algorithms (GA) where developed with natural adaptive systems in mind; the key feature in these systems was the the successful use of competition and innovation to provide the "ability to dynamically respond to unanticipated events and changing environments" [*Bäck et al.*, 1997].

The group of Holland developed systems making use of genomes, which evolved using variation methods borrowed from nature: mutation and crossover. During the following years, elitism and multi-point crossover were pioneered by this group.

3.1.3 Evolution strategies

The theory of Evolution strategies was developed originally to control a robot which was to be constructed to minimize the drag of a three-dimensional body in a wind tunnel.

This optimization was first done manually by adjusting only one variable at a time, but that method got stuck in a local minimum. The use of dice to have random changes in the control variables finally allowed this optimization to succeed.[1]

Note that in this case, a population of only one individual was evolving. Later, this group also used population sizes bigger than one, and also explored self-adaptation of the optimization parameters.

3.1.4 Convergence of the optimizers

Finally, in the early 1990s, these three groups came into contact with each other during several conferences and formed the journal *Evolutionary Computation*. This information sharing ultimately led to the development Differential Evolution (DE) [*Storn and Price*, 1997] and Particle Swarm Optimization (PSO) [*Kennedy and Eberhart*, 1995].

3.2 Introduction to evolutionary computing

The general idea behind the use of EC is to mimic evolution in nature, by simulating a population of solutions. In this population, individual solutions combine (mate) with each other to form new solutions (reproduce). If those new solutions are better than the existing solutions, they are kept at the expense of worse solutions, thus increasing the general level of the solutions.

When this process is repeated indefinitely, this algorithm should – at least in theory – find the best solution to a problem, but this is somewhat hindered by the fact that the size of the population and the number of iterations are limited by computing power, so the optimal solution is not always found within a reasonable amount of time.

According to [*Fogel and Angeline*, 2000], the user has control over at least four aspects of the approach, and these four aspects will be treated in this chapter: the representation of the solution (in section 3.3), variation operators (in section 3.4), methods of selection (in section 3.5) and ordering of multi-objective functions (in chapter 6).

3.3 Representation of a solution

In all known (living) organisms on Earth, nature stores its genetic information in chromosomes, which are long strings of amino-acids that can be represented by a 4-letter alphabet. Genetic Algorithms use a similar coding method, however, using a binary alphabet. EAs also make use of real-coded information, and other representations such as probability based codings are also available. The choice of a coding (or representation) should depend on the type of problem to solve. To make this choice easier, this section gives an overview of the different ways to represent a solution, based on the information in [*Goldberg*, 1989], [*Bäck et al.*, 2000*a*], [*Bäck et al.*, 2000*b*], [*Baluja and Caruana*, 1995], and [*Michalewicz*, 1996].

3.3.1 Binary coding

The best-known coding for GAs is binary coding; this coding follows from the minimum alphabets principle: [*Goldberg*, 1989]

[1]And the robot was constructed to execute this optimization!

The user should select the smallest alphabet that permits a natural expression of the problem.

According to [*Bäck*, 2000], this works best for problems whose solutions are binary in nature, such as pseudo-Boolean optimization problems like the knapsack problem.

The knapsack problem is a combinatorial problem [*Whitley et al.*, 1997]: given n objects with weight W_i and profits P_i, the goal is to find a set of objects with a combined weight not exceeding the capacity of the knapsack M, which maximizes the sum of the benefits:

$$\sum_{i=1}^{M} X_i W_i \leq M \quad \text{and} \quad \sum_{i=1}^{M} X_i P_i \text{ is maximal} \tag{3.1}$$

in which X_i can be 0 or 1, indicating if an object is put into the knapsack.

A simple example: given a set of objects with weights $1, 4, 4, 9, 11, 15$, and profits equal to their weights, $P_i = W_i$. Furthermore, a knapsack with capacity $M = 17$ is considered. In this case, even though there are $2^6 = 64$ possible combinations of objects, there is only one optimal solution, using the objects with weights: 4, 4 and 9. As the size of the problem space scales exponentially with the amount of objects to pack, this problem can be quite challenging.[2]

However, also for problems which are not binary in nature, such as parameter optimization problems, binary codings can be used. For this to work, the user must choose a domain and precision for the parameters, which then gives the number of bits needed to express that parameter in binary:

If a parameter falls in an interval $[a \quad b]$, and needs a precision of ε, $(b-a)/\varepsilon$ positions need to be encoded by this number, so $\mathrm{ceiling}(\log_2 \frac{b-a}{\varepsilon})$ bits are needed to express this.[3]

For example, if a number can range from 0 to 100, and is stored with a resolution of 0.1, it can take on 1001 values. This means that $\mathrm{ceiling}(\log_2 1001) \approx \mathrm{ceiling}(9.97) = 10$ bits are necessary to represent that number in binary.

3.3.2 Gray coding

One of the problems that can show up when using binary coding for parameter optimization problems is referred to as 'Hamming cliffs'. This name is related to the Hamming distance, which is defined as the number of differing bits between two numbers. Hamming cliffs occur when two adjacent numbers have a large Hamming distance, so the mutation from one value to the other is rather unlikely, for example:

If a binary coded variable has the value 3, its binary representation is 011. It is unlikely that a mutation will change its value to 4, which has the representation 100, since that number would require the mutation of 3 bits at once (and hence has a Hamming distance of 3).

A solution to this problem is to make use of Gray coding. Gray coding is a one-to-one mapping between binary representations of values, keeping the number of bits (and the size of the solution space) the same as that of normal binary coding. [*Michalewicz*, 1996] gives an implementation of a simple Gray encoder in pseudo-code, [*Heitkoetter and Beasley*, 2001] gives an implementation in C. An example of how a 3-bit number can be represented using Gray coding is given in figure 3.1

[2]In fact, the knapsack problem is *NP-complete* meaning it is not possible to locate the optimum quickly, but it can be verified quickly. This is the reason the knapsack problem is also used as the basis for some cryptographic algorithms [*Schneier et al.*, 1996].

[3]$\mathrm{ceiling}(X)$ is a Fortran function which returns the least integer greater than or equal to the number X.

Gray codes solve the problem above because they have the property that the Hamming distance between two adjacent values is 1: in the coding given in table 3.1, the number 3 is represented as 010, which differs only one bit from the coding for 4: 110.

3.3.3 Real coding

When taking a closer look at the minimum alphabets principle, it states that a minimum alphabet should be selected that *permits a natural expression* of the problem. Thus, in the case of parameter optimization, the representation does not need to be binary.

[*Michalewicz*, 1996] states that real coding of floating point numbers gives better results than binary (or Gray) coding.[4] The reasons given are that on the one hand, floating point numbers can represent a large domain with a limited amount of bits, and on the other hand, that they have a higher precision.

Note that real coding cannot use the traditional binary genetic operators such as crossover and mutation, but needs custom versions of those.

3.3.4 Permutations

[*Whitley*, 1997] describes some techniques to code permutations. In this context, a permutation of a set is an arrangement of its elements into a row. Given a set of n objects, $n!$ permutations of this set exist.

This can be used to solve traveling salesman problems (TSP), which are focused on finding the optimal path to visit a given set of locations, where the length of the total path is minimal. This has a similarity to the GTOC2 and GTOC3 problems, in which the problem is to find an optimal trajectory visiting a subset of a collection of asteroids which can be chosen by the optimizer.

[*Whitley*, 1997] contains several methods to map a sequence into an integer, but this is not natural for a problem like GTOC3: in GTOC3, only 4 asteroids have to be selected, and they can be indexed by integers. The only important thing is that care has to be taken that an asteroid does not occur twice in the same sequence.

If problems become much more complicated (for example, because a much larger sequence of asteroids is to be visited), use of this methods might become beneficial.

3.3.5 Probability coding

[*Baluja and Caruana*, 1995] states that the bits should be replaced by probabilities, and that the population of each generation should consist of individuals that are represented by bits (which are chosen based on the probabilities in the probability string). During the reproduction operator, the probabilities should be adjusted in the direction of the best individuals of that generation, and the whole population regenerated based on those new probabilities.

This method is claimed to work best with static problems with relatively small populations, in which niching is not needed (since only one super-individual exists). The applicability of this coding might be rather limited for trajectory-optimization problems.

[4]Where better means: smaller variance in the solutions, faster results, and a better 'best' result for a given implementation. The metrics of the quality of solutions will be explored further down the road.

Number	0	1	2	3	4	5	6	7
Bit 1	0	0	0	0	1	1	1	1
Bit 2	0	0	1	1	1	1	0	0
Bit 3	0	1	1	0	0	1	1	0

Figure 3.1 Example of a Gray coded sequence [*Spaans*, 2008]

3.4 Variation Operators

The conceptual implementation of an EC algorithm is quite simple: it should consist of a loop of some operators; the code should then run through this loop, until a termination condition is met, at which point the iteration stops. Examples of termination conditions are ones that check if the population is still improving, or one that fires when a (preset) maximum number of generations has been reached.

According to [*Goldberg*, 1989], a minimal implementation of a GA should at least contain the operators depicted in figure 3.2: reproduction, crossover and mutation. Since the algorithm is that simple, the implementation of the operators is what makes or breaks an algorithm. For that reason, an overview is given of the different types of genetic variation operators, in the example of figure 3.2 the crossover and mutation operators. The reproduction operator in that figure is not a variation operator, but a selection operator, and hence it will be postponed until section 3.5.

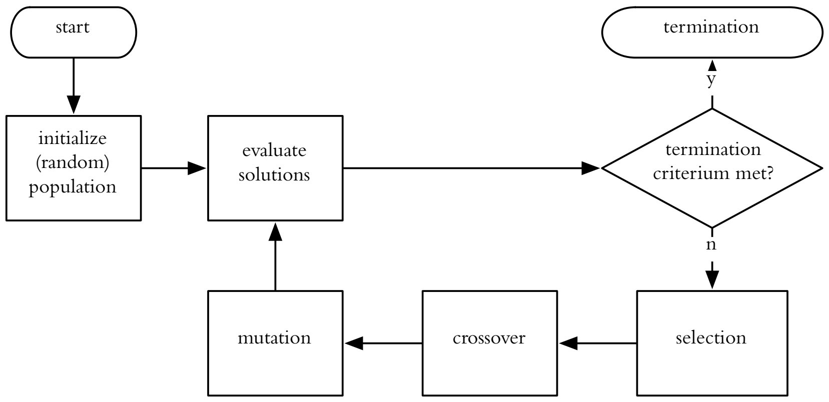


Figure 3.2 Flowchart of a simple genetic algorithm

3.4.1 Crossover operators

The crossover operator is the operator that causes the population to change; by combining the genes of two parents,[5] a pair of new children is created which has features of both parents. If the good parts of the parents are mixed into the child, it will probably have a better chance of surviving, so those good features are kept in the gene pool at the cost of worse individuals, causing the quality of the population to increase.

Single point crossover

[*Goldberg*, 1989] describes single point crossover; it is the first variant of crossover; in this variant, two parent-strings are placed next to another, a cut is made on a random point somewhere in the string, and the two 'tails' are swapped. This is illustrated in figure 3.3. This operator can be applied to both binary and real-valued strings, but behaves differently: in the case of binary crossover, the cut lies between the bits, so a gene can be split into two parts (and cause two new values for the gene which is split to be introduced into the population). In the case of real-valued genomes, the cuts are on the boundary between two genes and the genes themselves are not changed.

[5] [*Eiben et al.*, 1995] describe multi-parent crossover, where more than two parents are combined, but that kind of crossover is not used in this thesis.

Parent 1:	$a_1\ a_2\ a_3\ a_4\ a_5\ a_6\ a_7$	$a_8\ a_9$
Parent 2:	$b_1\ b_2\ b_3\ b_4\ b_5\ b_6\ b_7$	$b_8\ b_9$

$\Downarrow$

Child 1:	$a_1\ a_2\ a_3\ a_4\ a_5\ a_6\ a_7$	$b_8\ b_9$
Child 2:	$b_1\ b_2\ b_3\ b_4\ b_5\ b_6\ b_7$	$a_8\ a_9$

Figure 3.3 Example of a single point crossover

Multi-point crossover

Multi-point crossover is the logical extension of single-point crossover. It is the repeated application of single-point crossover in a single generation. For the specific case of two-point crossover, this offers the option of only exchanging a small part of the string. An example is given in figure 3.4. As with normal crossover, it can be applied to both binary as well as real-valued genes, with different behavior.

Parent 1:	$a_1\ a_2\ a_3\ a_4\ a_5$	$a_6\ a_7$	$a_8\ a_9$
Parent 2:	$b_1\ b_2\ b_3\ b_4\ b_5$	$b_6\ b_7$	$b_8\ b_9$

$\Downarrow$

Child 1:	$a_1\ a_2\ a_3\ a_4\ a_5$	$b_6\ b_7$	$a_8\ a_9$
Child 2:	$b_1\ b_2\ b_3\ b_4\ b_5$	$a_6\ a_7$	$b_8\ b_9$

Figure 3.4 Example of a two-point crossover

Uniform crossover

[*Syswerda*, 1989] introduces uniform crossover. It is the extreme form of multi-point crossover, where the genes in the parent-strings have an equal chance of ending up in the child string. There is no coupling between the individual bit strings, which makes this method less successful in binary coded algorithms (since it tends to break up genes irrespective of their goodness) than in real-coded algorithms.

Parent 1:	$a_1\ a_2\ a_3\ a_4\ a_5\ a_6\ a_7\ a_8\ a_9$
Parent 2:	$b_1\ b_2\ b_3\ b_4\ b_5\ b_6\ b_7\ b_8\ b_9$

$\Downarrow$

Child 1:	$a_1\ b_2\ b_3\ a_4\ b_5\ a_6\ b_7\ b_8\ a_9$
Child 2:	$b_1\ a_2\ a_3\ b_4\ a_5\ b_6\ a_7\ a_8\ b_9$

Figure 3.5 Example of a uniform crossover

DE uses a variation of this crossover operator, where the chance is biased. This can be used to control the amount of change in the trial vector.

3.4.2 Additional real-valued crossover operators

The following additional crossover operators are defined for real-valued genes. Note that because they are not usable in DE and PSO they have not been implemented in the new OPTIDUS.

Arithmetical crossover

In [*Michalewicz*, 1996] the arithmetical crossover operator is introduced, which mixes two real-coded genes by assigning a weighted average of both parents to the children:

$$\begin{aligned} x_i^{child_1} &= a x_i^{parent_1} + (1-a) x_i^{parent_2} \\ x_i^{child_2} &= a x_i^{parent_2} + (1-a) x_i^{parent_1} \end{aligned} \tag{3.2}$$

In this equation a is the weight which can be made constant (uniform arithmetical crossover) or dependent on time (non-uniform arithmetical crossover). a should be between 0 and 1, and three special cases can be identified easily:

$a = 0$ Causes the two genes to be swapped.

$a = 1$ Causes nothing to happen.

$a = 0.5$ Causes both child genes to get the same value, being the average of the values of both parent genes.

By letting the value of a increase from 0 to 1 during an optimization, the amount of change caused by this operator decreases, possibly allowing for better local convergence.

Flat crossover

[*Radcliffe*, 1991] describes a crossover operator called 'flat crossover'. It essentially is the same as arithmetical crossover, but the weight a is picked randomly each time.

Simulated Binary Crossover

[*Deb and Agrawal*, 1995] describe simulated binary crossover (SBX). The general idea behind SBX is that it is an operator which has the same statistical properties as normal binary crossover, yet can be applied to real-coded genes. The properties taken into account are keeping the average values of the parents and children equal, and also to have a similar spread-factor distribution.

The spread-factor is the ratio of the distances of the represented values of the children and the parents, so if the child-genes are more alike, the spread-factor is < 1, and the crossover is said to be contracting. In the opposite case, the spread-factor is > 1, and the crossover is said to be expanding.

For example, if the two parents have values 3 and 7, and the children have values 3.5 and 6.5, the spread factor $\beta = \frac{6.5-3.5}{7-3} = 0.75$.

The distribution of spread factors for binary cases is derived in [*Deb and Agrawal*, 1995] and given as a pair of probability density functions $c(\beta)$.

$$c(\beta) = \begin{cases} 0.5(n+1)\beta^n & \text{for } 0 < \beta \leq 1 \\ 0.5(n+1)\frac{1}{\beta^{n+2}} & \text{for } \beta > 1 \end{cases} \tag{3.3}$$

This function depends on a non-negative number n (to be chosen by the user); for increasing values of n, values of β close to 1 will have a bigger probability, so the change in values between the parents and children will become lower. Values of n between 2 and 5 mimic the behavior of real binary genomes the closest.

It can be shown that $\int_{\beta=0}^{\infty} c(\beta)\mathrm{d}\beta = 1$, so random values of β conforming to this distribution can be generated by picking a random number $0 \leq r \leq 1$ and then finding the value of β by integrating (3.3) and equating it with r:

$$r = \int_{x=0}^{\beta} c(x)\mathrm{d}x = \begin{cases} 0.5\beta^{n+1} & \text{for } 0 < \beta \leq 1 \\ 1 - 0.5\beta^{-(n+1)} & \text{for } \beta > 1 \end{cases} \tag{3.4}$$

Now that β has been determined, the values of the child genes are given by:

$$x_{1,2}^{child} = \frac{x_1^{parent} + x_2^{parent}}{2} \pm \beta(x_1^{parent} - x_2^{parent}) \tag{3.5}$$

Of course, care has to be taken that the values may take on have to be respected.

Blend Crossover

[*Deb and Agrawal*, 1995] briefly mention blend crossover (BLX). For two parents with values x_i and x_j (picked such that $x_i < x_j$) a child value is picked randomly (in each direction) from the range

$$[x_i - a(x_j - x_i), x_j + a(x_j - x_i)] \tag{3.6}$$

The best performance is reported for $a = 0.5$, which is logical, as otherwise the blending would be biased. A performance comparison between BLX and SBX is given in [*Deb and Agrawal*, 1995]. The conclusion regarding BLX is that it is inferior to SBX when faced with problems with multiple optimum points.

Crossover probability

[*Deb*, 2001] notes that it is important to keep part of the population intact when using a crossover operator that shakes up the strings a lot, such as multi-point and uniform crossovers. The intention of that is to maintain a good balance between exploration and exploitation, so that on the one hand, good individuals are not lost by accident, and on the other hand, no premature convergence occurs. This can be done by keeping the crossover probability low enough to not break too much of the population. According to [*Grefenstette*, 1986], the optimal setting for this probability is linked to the mutation probability and population size. Reported values range from 0.3 for large populations ($S = 80$) to 0.9 for small populations ($S = 30$).

3.4.3 Mutation

Mutation is the operator which can introduce new features into the gene pool; it does this by changing the values of individual bits or genes at random. Of course, there are several ways to do this.

For the binary coded case it is a rather simple operator:

Binary mutation Binary mutation is the mutation-operator of choice for binary representations. Most of the time, it is implemented as an operator which has a probability p_m of occurring for a certain bit. If it is applied to a bit, its value is flipped from 0 to 1 or vice versa.

For real-coded representations, [*Michalewicz*, 1996] gives the following two mutation operators:

Uniform mutation In uniform mutation, an element of a chromosome may be replaced by a completely random value (that is, there is no relation between the value before and after the mutation).

Suppose an individual is given by a vector $\mathbf{x}^i$. Now, its kth element x_k^i is chosen to mutate. After uniform mutation, this element is replaced by a random value chosen from a uniform distribution between l_k and u_k, the lower and upper boundaries for element k.

Non-uniform mutation In non-uniform mutation, the contents of the element before and after mutation are related; it works as follows: after an element has been selected for mutation, a coin is flipped to determine the direction in which the value will change, after which the amount by which the element changes is determined according to:

$$x*_k^i = \begin{cases} x_k^i + \Delta(t, u_k - v_{i,k}) & \text{if direction is up} \\ x_k^i - \Delta(t, v_{i,k} - l_k) & \text{if direction is down} \end{cases}$$

with l_k and u_k again the lower and upper bounds for x_k. $\Delta(i_{gen}, n)$ is a random number generator with output domain $[0, n]$, and a distribution that may vary with the generation number i_{gen}. This can be used to control the amount of mutation over time – for example, at the end of the optimization, it might be useful to have mutations which only cause small changes.

3.5 Methods of selection

3.5.1 Reproduction operator

The reproduction operator is the operator that selects which parents are allowed to reproduce, and how much children they will get. This operator works in two steps: first of all, it calculates a (non-negative) fitness value for the current generation, and after that, based on the fitness value and possibly chance, the individuals which will be represented in the next generation are selected. There are several methods to do this, and they are described below.

Fitness-proportionate selection

Fitness-proportionate selection is described in [*Goldberg*, 1989] as the "biased roulette wheel" method. This name comes from the fact that this method can be visualized by a roulette wheel with slots whose sizes are proportional to the fitness value of the individual they represent – so an individual whose fitness value is twice the average, will have a chance twice as large to capture the ball.

A mathematical description of this algorithm is also rather easy: if an individual has a fitness f^i, the chance that it will be selected to reproduce equals

$$p_r^i = f^i / \sum_{n=1}^{N} f^n \tag{3.7}$$

An example of a roulette wheel filled with the relative size of the slots (which equals the chance the individual will be selected) is given in figure 3.6.

The expected number of times an individual may reproduce then equals:

$$c^i = Nf^i / \sum_{n=1}^{N} f^n \tag{3.8}$$

This is the amount of offspring an individual will probably have in the next generation.

Stochastic Universal Sampling

Stochastic Universal Sampling (SUS) is a sampling technique introduced in [*Baker*, 1987] which is similar to fitness-proportionate selection, but makes sure that individuals are selected in a fairer way. It starts out similar to fitness-proportionate selection, by setting up

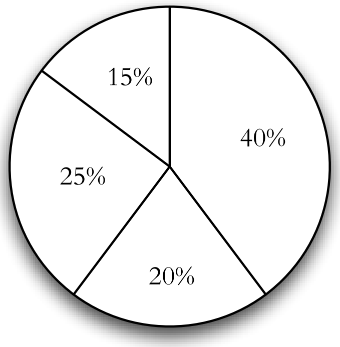


Figure 3.6 Example of a roulette wheel as used by fitness-proportionate selection.

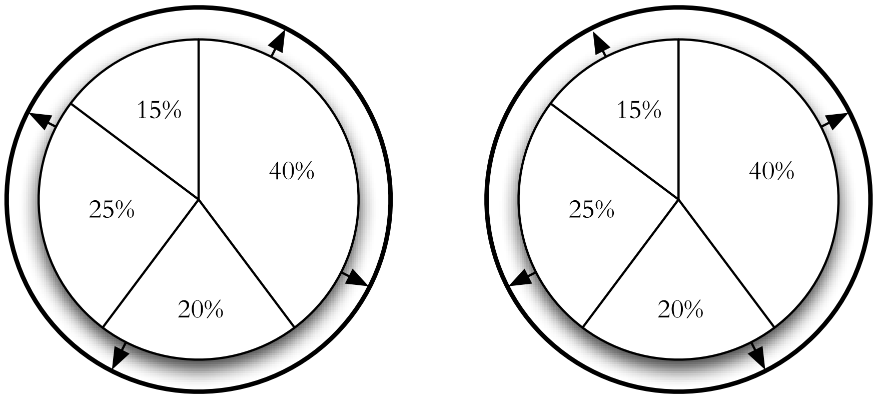


Figure 3.7 Example of two selections done using SUS

a roulette wheel. Next, a random starting point at the edge is chosen, and from there the edge of the wheel is divided into N parts of equal size.

Figure 3.7 shows how this method selects individuals graphically. In this case, 4 individuals are to be selected, so the arrows which indicate the selected individuals are spaced 90 °apart. It can be seen that the outcome depends on the position of the starting point: in the left case, the individual with a 15% chance of being selected is not, and in the right case it is.

SUS guarantees that the amount of times an individual is either $\text{floor}(c_i)$ or $\text{ceiling}(c_i)$.[6] This makes it more fair than simple roulette wheel selection.

Tournament selection

Tournament selection is described in detail in [*Bäck et al.*, 1997]. In principle, it works as follows: pick q individuals at random from a population; let the picked individuals duel, and let the winner reproduce. This process is repeated λ times, giving λ winners.

The main benefits of this scheme are that the calculations for this operator can be carried out quickly, and that it is does not need scaling functions, since rank is compared and not the value of the fitness functions. Another benefit is that no global comparison of the population is needed, so this operator is well suited for parallelization.

Note that increasing q will reduce the diversity of the population, as the chance that a (winning) high-ranking individual is picked gets higher if q increases.

[6] floor and ceiling are the functions which round a real number to a normal number, in the downward and upward direction respectively.

3.5.2 Fitness scaling

One of the problems with a simple biased roulette wheel implementation is that when a population has been evolving for some time, it will probably consist of individuals with fitness values that are in the same range. In that case, c^i will be close to 1 for all individuals, and selective pressure will be too low, which means that evolutionary speed is also low. One way to resolve this is to use a method called fitness scaling.

This scaling is done by applying a linear transformation to the fitness function which satisfies the constraints that the average reproduction count $\bar{c} = 1$ and $c^i \geq 0 \quad \forall i$. The first constraint causes the population size to be constant; the second one prevents the occurrence of negative probabilities which would give problems with fitness proportionate selection, as the slots in the roulette wheel would have negative sizes.

The applied transformation is given in [*Goldberg*, 1989] as:

$$c^i = af^i/\bar{f} + b \tag{3.9}$$

When the user chooses the (expected) reproduction count for the best individual c_{mult}, the parameters a and b can be calculated as follows: first, it is tested if the population satisfies the following condition:

$$\bar{f} - f_{min} < \frac{f_{max} - f_{min}}{c_{mult}} \tag{3.10}$$

If this is satisfied, then the minimal fitness is high enough to allow the expected reproduction count of the worst individual to be positive if $c_{max} = c_{mult}$. In that case, a and b follow from:

$$\begin{aligned} a = & \quad (c_{mult} - 1)\frac{\bar{f}}{f_{max} - \bar{f}} \\ b = & \quad \bar{f}\frac{(f_{max} - \bar{f}c_{mult})}{f_{max} - \bar{f}} \end{aligned} \tag{3.11}$$

If (3.10) is not satisfied, the following alternative is given:

$$\begin{aligned} a = & \quad \frac{\bar{f}}{\bar{f} - f_{min}} \\ b = & \quad \frac{-f_{min}\bar{f}}{\bar{f} - f_{min}} \end{aligned} \tag{3.12}$$

A graphical example of this scaling function is shown in figure 3.8.

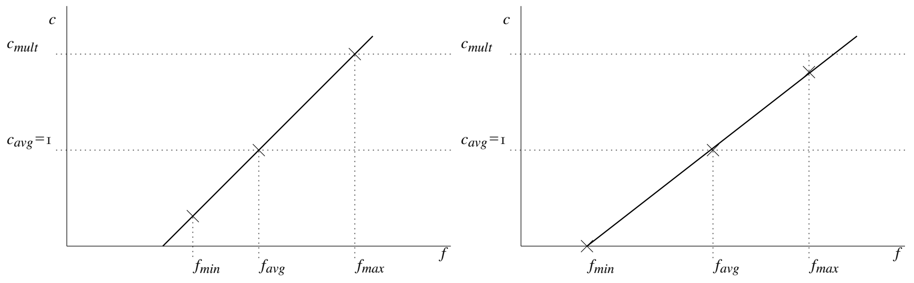


Figure 3.8 Transfer functions for the cases where condition (3.10) is satisfied (left) and not satisfied (right)

Other methods are described in [*Bäck et al.*, 1997, C2.2.2]: the first one is a time-varying linear transformation, for example:

$$f^{*i} = f^i - \beta(i_{gen}) \tag{3.13}$$

where $\beta(i_{gen})$ is the worst value seen in the past generations. However, this might be problematic when there have been bad individuals in the population, which cause the value of β to be too low for the whole optimization run.

One method to prevent this is by defining β as a recency-weighted running average:

$$\beta_i = \delta\beta_{i-1} + (1-\delta)f_{min} \tag{3.14}$$

with, for example, $\delta = 0.1$. This will cause the value of β_t to be representative of more recent values of f_{min}, so a bad individual will be smoothed out of β over time.

Normal tournament selection can then be performed on the scaled fitness values f^{*i}.

Sigma scaling

Another method mentioned in [*Goldberg*, 1989] is sigma scaling, which uses the following transformation:

$$f^{*i} = \begin{cases} f^i - (\bar{f} - c\sigma_f) & \text{if } f^i > \bar{f} - c\sigma_f \\ 0 & \text{otherwise} \end{cases} \tag{3.15}$$

in which σ_f is the standard deviation of the fitness values of the population, and c is a constant which can be selected by the user, with a suggested value of $c = 2$. This kills off individuals that score below $c\sigma_f$ from the average of the population. The idea is that $\bar{f} - c\sigma_f$ represents a lower threshold for any reproducing individual. As the population improves, this statistic tracks the improvement, yielding a level of selective pressure that is sensitive to the spread of performance values in the population [*Bäck et al.*, 1997].

3.5.3 Fitness sharing

Fitness sharing is one of the niching methods, and is explained in [*Bäck et al.*, 1997, C6.1]. What fitness sharing does is to penalize the fitness value of an individual, when it is too close to other individuals. The formula given is:

$$f^{*i} = \frac{f^i}{\sum_{j=1}^{S} \mathrm{sh}(d(i,j))} \tag{3.16}$$

in which sh() is the sharing function, which is a function with the constraints $sh(0) = 1$, and decreases to 0 as the distance increases. A common sharing function is:

$$\mathrm{sh}(d) = \begin{cases} 1 - (d/\sigma_{share})^{\alpha} & \text{if } d < \sigma_{share} \\ 0 & \text{otherwise} \end{cases} \tag{3.17}$$

Typically, $\alpha = 1$ and σ_{share} should be small enough to allow discrimination between local optima - so it should be less than half the distance between two local optima. This distance is not known beforehand, so some experimentation is needed per problem.

$d(i,j)$ is the distance between two individuals. For binary coded individuals,the Hamming distance is a good distance metric. For real-valued individuals, the Euclidian distance of the two vectors can be taken: $d(i,j) = |\mathbf{x}^i - \mathbf{x}^j|$

The effect is that there is selective pressure against getting individuals that are too close to eachother, and increases diversity in the population.

3.5.4 Rank-based allocation

The opposite of too low selective pressure is premature convergence; this can happen when a population contains a lot of so-called "super-individuals" that eat up all space of the roulette wheel, and force the solution into the direction of a (local) optimum. One way to get around this problem is to use rank-based allocation, introduced in [*Baker*, 1985] and improved upon in [*Whitley et al.*, 1989].

Rank-based allocation is a relatively simple concept: the individuals are sorted on the fitness value, and then the reproduction count is determined according to the rank of the individual.

[*Whitley et al.*, 1989] claims that this is good because fitness values are not an exact measure of fitness, so using that value directly to calculate a reproduction count is not the way to go. Their argument for this is that fitness-proportionate allocation can lead to super-individuals.

Furthermore, ranking removes the need for scaling algorithms, and its inherent need for custom parameters, however, the implementer of rank-based allocation must also pick a method to convert rank to probabilities. The advice is to use a linear method to map between rank and reproduction count; a method to do this is given in [*Bäck et al.*, 1997, C2.4]:

$$f^{*i} = \frac{\alpha + r^i/(N-1)(\beta - \alpha)}{N} \tag{3.18}$$

where $\alpha = 2 - \beta$ if the population size is to be kept constant; α is the expected number of offspring for the worst individual. r_i is the (base zero) rank of individual i, so it runs from 0 to $N-1$.

An example of an exponential ranking function is also given, for example:

$$f^{*i} = \alpha(1-\alpha)^{N-1-r^i} \tag{3.19}$$

3.6 Selection of representation and operators

The new version of OPTIDUS makes use of the following structures and operators.

3.6.1 Representation of the solution

Real coding is used because of two reasons. The first one is that the parameters defining the solutions are best represented using real numbers, and the second one is that PSO and DE require real coding.

3.6.2 Variation operators

Single point crossover has been implemented because it is one of the standard operators in GAs, and allows for quick distribution of information between individuals.

Multi point crossover has been implemented because it is one of the standard operators in GAs. It allows for information sharing between individuals, but might cause less bad individuals as it can also swap smaller pieces of genomes.

Uniform crossover has been implemented as it is needed for DE. It has not been used in any of the other optimizers.

Non-uniform mutation has been implemented as it is the mutation operator used in the original version of OPTIDUS. It is used in most of the operators, as it allows the amount of mutation to be controlled, so that the optimizer can converge.

3.6.3 Selection operator

Roulette wheel selection has been implemented because it is the standard operator which has also been used in the previous version of OPTIDUS.

Stochastic universal sampling has been implemented as it is a variant of the roulette wheel, which is supposed to be more fair, which should have a positive effect on the optimization process, as good individuals are guaranteed to survive.

Rank based allocation has been implemented as it is a selection operator which should provide some resistance against solution getting dominated by super-individuals, which might lead to premature convergence.

Tournament selection has also been implemented – with two restrictions: the amount of duels is equal to the size of the population, and in each duel, two individuals participate.

Chapter 4
Particle Swarm Optimization

This chapter contains general information about Particle Swarm Optimization (PSO) which is an evolutionary algorithm for optimizing non-linear problems. It was introduced by [*Kennedy and Eberhart*, 1995], and is modelled after the flocking of birds searching for food. This chapter contains an introduction to PSO, the tuning of the parameters in PSO, and how to handle constraints with PSO. Also, a variant using a "predator" called Predator-Prey Optimization (PPO) is shown, and a sample optimization using both PSO and PPO is shown.

4.1 Introduction to PSO

Particle Swarm Optimization (PSO) is modelled after flocks of birds: the behavior modelled is that inside flocks, information is communicated regarding the position where food has been found, and that birds can remember where they found good food themselves. During the search for food, individual birds are then attracted to the (overall) best place where food has been found, and the best places where they have found food themselves.

Figure 4.1 A flock of birds in Anchorage, Alaska. Based on `http://flickr.com/photos/johnblumenberg/2331700365/`

This model can also be applied to optimization problems: replace birds by solutions, and food by fitness. A flowchart of this is given in figure 4.2.

4.2 Basic formulation of the velocity and position of particles

This section handles two boxes from figure 4.2 which are characteristic for PSO: the boxes labeled "calculate new velocities" and "calculate new positions". These velocities and positions are associated with solutions, and will vary over time. These solutions act as particles, which obey Newtonian mechanics: The velocity of the particles is influenced by several accelerating forces acting on it:

$$\frac{\mathrm{d}\mathbf{v}_i}{\mathrm{d}t} = \sum \mathbf{a} \tag{4.1}$$

and the position of a particle over time is related to its velocity by

$$\frac{\mathrm{d}\mathbf{x}_i}{\mathrm{d}t} = \mathbf{v}_i \tag{4.2}$$

The velocity of a particle is influenced by three factors: first of all, a friction factor w' is introduced to limit the speed of the particles. This term can be modelled as:

$$\mathbf{a}_{fr} = -w'\mathbf{v}_i \tag{4.3}$$

However, this term is usually written as an inertia weight w:

$$\mathbf{a}_{fr} = (w-1)\mathbf{v}_i \tag{4.4}$$

Typical values of w lie between 0.8 and 1.2 (see section 4.3.1 for more details on this value). From this it can be seen why it is called an inertia term: if $w = 1$, the friction term is zero,

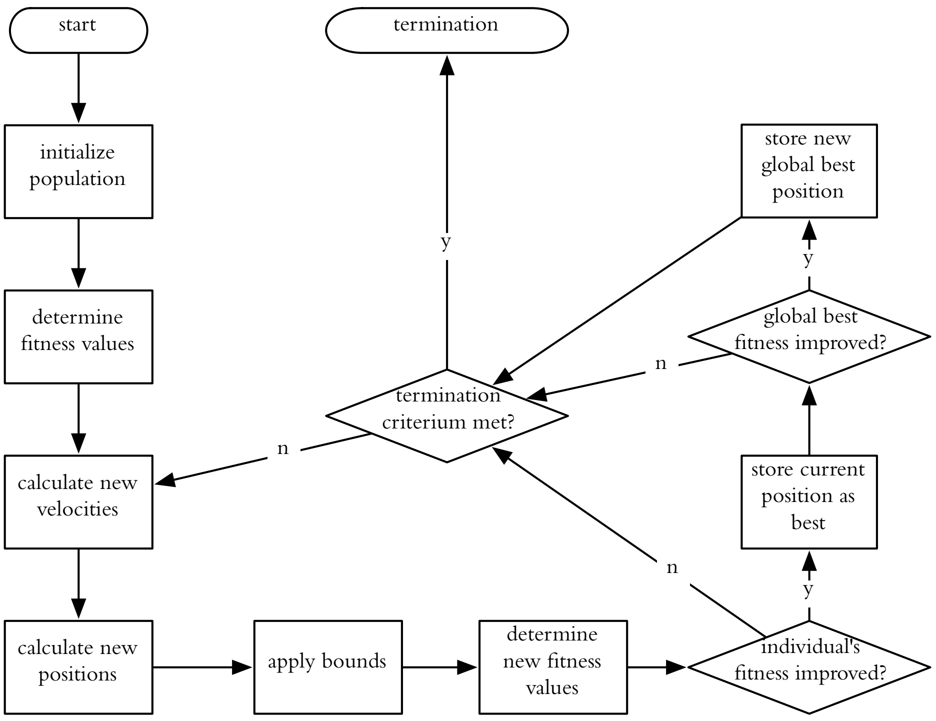

start
termination
initialize population
determine fitness values
calculate new velocities
calculate new positions
apply bounds
determine new fitness values
individual's fitness improved?
store current position as best
global best fitness improved?
store new global best position
termination criterium met?
y
n
y
n
y
n

Figure 4.2 Flowchart of a PSO algorithm.

and the particle will keep on moving in its current direction. If $w < 1$, this term will cause a deceleration in the current direction, and the particle will slowly come to a stop, as if there is less inertia. For $w > 1$, the opposite happens.

The second acceleration is the attraction by the best position a particle has seen before, denoted by $\mathbf{x}_i^*$. The magnitude of this acceleration is proportional to the distance between the current and the best position, and is scaled by a constant c_1 and a number coming form a random number generator $u()$:

$$\mathbf{a}_{pb} = c_1 \cdot u() \cdot (\mathbf{x}_i^* - \mathbf{x}_i) \tag{4.5}$$

The third acceleration is caused by the attraction by the best position the flock has seen before, denoted by $\mathbf{x}_G^*$. The model is similar to the one for the particle's best position:

$$\mathbf{F}_{gb} = c_2 \cdot u() \cdot (\mathbf{x}_G^* - \mathbf{x}_i) \tag{4.6}$$

In PSO literature, the accelerations from equations (4.1) and (4.4)-(4.6) are then summed and added to the current velocity, giving the new velocity [*Kennedy and Eberhart*, 1995]:

$$\mathbf{v}_{i,new} = w \cdot \mathbf{v}_i + c_1 \cdot u() \cdot (\mathbf{x}_i^* - \mathbf{x}_i) + c_2 \cdot u() \cdot (\mathbf{x}_G^* - \mathbf{x}_i) \tag{4.7}$$

In the same way, equation (4.2) can be integrated[1] giving:

$$\mathbf{x}_{i,new} = \mathbf{x}_i + \mathbf{v}_i \tag{4.8}$$

With formulas (4.7) and (4.8) the behavior of the particles is determined.

4.3 Tuning optimization parameters in PSO

Equation (4.7) contains 3 parameters which can be tuned, being w, c_1 and c_2. The values of these parameters have a profound influence on the speed and robustness of the optimization.This section will highlight some of the schools of thought regarding the choice of these parameters.

4.3.1 Varying inertia weight

[*Shi and Eberhart*, 1998*b*] use the original form of (4.7), and analyze the effect of choosing w on the robustness of the algorithm. It was found that PSO performs best for their test problems if $0.9 < w < 1.2$. For lower values of w, the number of evaluations before finding the global optimum (per run) was lower, but during a lot of runs no optimum was found. For higher values of w, the number of evaluations was higher, as was the number of runs in which no optimum was found.

When looking at the behavior of the swarm for different values of w, it follows that a high value of w increases the velocities of particles, which means that more space will be explored, and that the opposite holds for low values of w. This inspired the authors to try an experiment in which the value of w is set to 1.4, and is decreased during the run, called the varying inertia weight. This gave good results: all runs converged to the optimum result, and the average number of evaluations was lower than that for $w = 0.9$, which was the lower bound for robust results.

[*Shi and Eberhart*, 1998*a*] did an analysis of the performance of PSO for Schaffer's f_6 function (treated in more detail in section 7.1.3) when setting an upper limit on the velocity

[1]In both cases, no real integration takes place, as the integration period is conveniently set to 1, and the accelerations and velocity are assumed constant over the integration. If these particles really would have been obeying Newton's laws of motion, these equations would be different.

of particles. This leads to some recommendations on the selection of w: if the maximum speed is small, $w \approx 1$ is a good choice; if the maximum speed is not small, $w = 0.8$ is a good choice.

In this recommendation, the value of "small" depends on the problem. The advice in general is to set the maximum speed equal to the size of the parameter domain ($v_{max} = x_{max}$), and $w = 0.8$.

Another observation in this paper was that when performing the optimization with a decreasing inertia weight, that the variance of iterations required to find the global optimum is smallest, even when no maximum velocity is applied.

4.3.2 Constriction coefficient

In [*Clerc and Kennedy*, 2002], another notation for the velocity adjustment (4.7) is suggested, based on an analysis of the motions of a particle around a stationary best solution in a one-dimensional space. This led to the following formulation:

$$\mathbf{v}_{i,new} = \chi \left(\mathbf{v}_i + \varphi_1(\mathbf{x}_i^* - \mathbf{x}_i) + \varphi_2(\mathbf{x}_G^* - \mathbf{x}_i)\right) \tag{4.9}$$

In this form, φ_1 and φ_2 are picked randomly from a uniform distribution $[0, \varphi]$ at each evaluation.

For the analysis of the one-dimensional case, the value of φ determines the behavior of the particle:

$0 < \varphi < 4$ causes the solution to oscillate around the "optimal" point (which is the center of attraction, in real a normal PSO the point halfway the best global and best individual solutions)

$\varphi > 4$ does not show cyclic behavior, and instead causes the solution to move away from the optimal point.

$\varphi = 4$ is a special case, where the solution will (ultimately) move away from the optimal point.

According to [*Clerc and Kennedy*, 2002], this analysis for the values of φ also holds for the PSO formulation. This mean the user can select a value for φ, but special care should be taken when $\varphi > 4$. This is done using the constriction coefficient χ, which can be calculated using:

$$\chi = \begin{cases} \dfrac{\kappa}{\left|1 - \frac{\varphi}{2} - \frac{\sqrt{\varphi(\varphi-4)}}{2}\right|} & \text{for } \varphi > 4 \\ \kappa & \text{otherwise} \end{cases} \tag{4.10}$$

with $0 < \kappa < 1$, with a suggested value of $\kappa = 0.8$.

This formulation has the advantage that it only needs two coefficients, namely κ and ϕ, to be chosen by the user, as opposed to the three (w, c_1 and c_2) in the case of the formulation of (4.7).

4.3.3 Population size in PSO

[*Clerc*, 2008*a*] gives the following rule of thumb for determining the population size (S) for optimizing a problem using PSO where the number of variables to be optimized is N:

$$S = \text{ceiling}(10 + 2\sqrt{N}) \tag{4.11}$$

Dimensionality N	Optimal population size S		
	Experimental	Clerc1 (4.11)	Clerc2 (4.12)
2	6	13	7
7	16	16	13
50	24	25	23

Table 4.1 Comparison of optimal population sizes according to experimental data and equations (4.11) and (4.12).

In [*Clerc*, 2008*b*], another rule is given:

$$S = \text{ceiling}\left(\frac{\ln(1-0.5^{1/N})}{\ln(1-\alpha)}\right) \quad \text{with } \alpha = 0.17 \tag{4.12}$$

These rules of thumb seem to have some credibility: for example, the Himmelblau function (given in 7.1.1) was optimized using PSO while varying the population size between 1 and 30 individuals. Figure 4.3 shows the results of these optimizations. For each population size, 100 runs where done with a maximum number of function evaluations of 180 per run.[2] From each run, the best fitness value was taken and is shown as a red circle in that figure. Also, from the best fitness values, the average values (per population size) are taken and plotted with the green line. Finally, the best of the best fitness values are plotted by the blue line.

The same procedure was repeated for the Griewank function, with dimensionality $N = 7$ and $N = 50$, however, this time with 1800 evaluations per run. The results of these runs are shown in figures 4.4 and 4.5.

The metric used to determine the optimum population size in this case is the average of the best fitness values. The best of the best fitness values is not discriminating enough, because it is influenced too much by luck: if one of the 100 runs per population size hits a good value, and the rest of the runs return bad solutions, that population size should not be considered good.

This effect is most noticeable in figure 4.3, where the blue line is almost horizontal, but the green line reaches a clear minimum for $S = 6$ (and $S = 7$ and $S = 8$ as runners-up).

For both Griewank functions, this effect is less noticeable, as the blue lines have much more resemblance to the shape of the green lines. For $N = 7$, the optimal population size is found at $S = 16$ (with $S = 14$ and $S = 17$ in second and third place), and for $N = 50$, at $S = 24$ (and $S = 25$ and $S = 21$ as runners-up).

For easy comparison, these numbers, and the outcome of equations (4.11) and (4.12) are shown in table 4.1. From this, it can be seen that both equations give reasonable estimates of the optimal population size. Because of its simplicity, (4.11) is used in chapter 7.

4.4 Predator-Prey Optimization

In [*Silva et al.*, 2002], Predator-Prey Optimization (PPO) is introduced as a variant of PSO to help avoid premature convergence.

[2]Normally, the optimization would be terminated at the end of each generation, so the amount of evaluations would be a multiple of the population size. This would not be fair: a population of 29 individuals is stopped after 7 generations (for a total of 203 evaluations) whereas a population of 30 individuals is stopped after 6 generations (for a total of 180 evaluations). In this implementation, this was fixed, by returning invalid fitness values after 180 evaluations have been executed, thus causing the results of the extra solutions not to be used. As we are only interested in the best individual encountered during the run, this does not impact the results negatively.

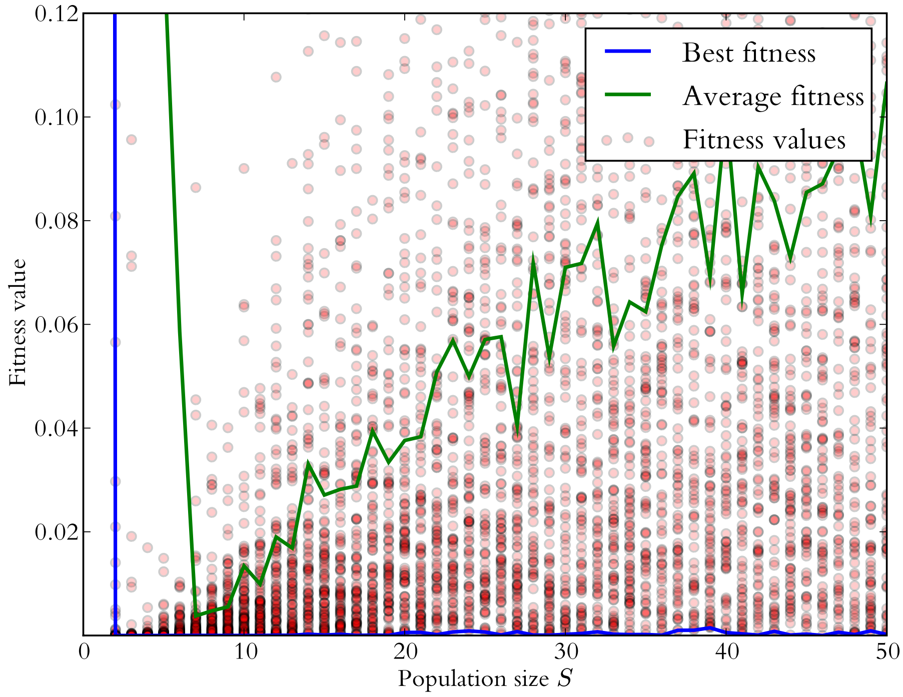


Figure 4.3 Fitness as a function of the population size for the Himmelblau function, using the aggregate results of 100 runs for each population size.

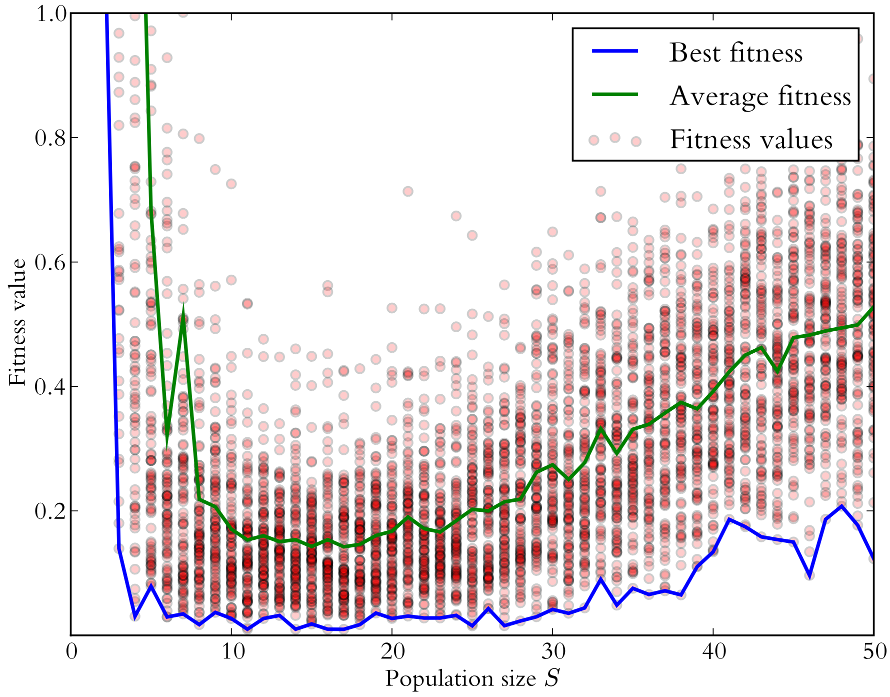


Figure 4.4 Fitness as a function of the population size for the Griewank function with dimensionality $N = 7$, using the aggregate results of 100 runs for each population size.

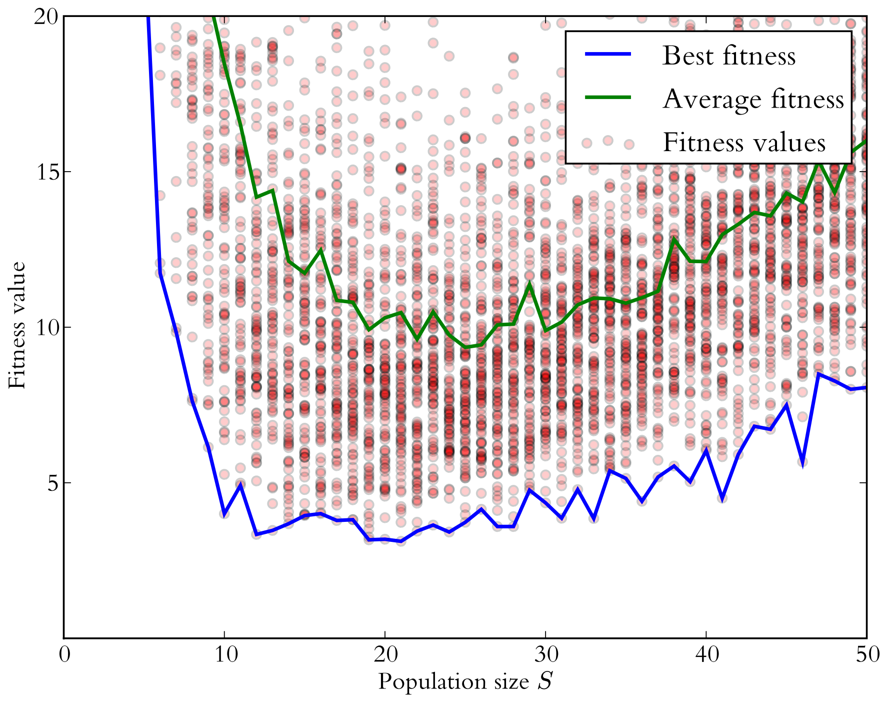


Figure 4.5 Fitness as a function of the population size for the Griewank function with dimensionality $N = 50$, using the aggregate results of 100 runs for each population size.

This works by introducing a predator into the population. This predator tries to hunt down the best particle in the population, and scares away any particle that comes close. This keeps those good individuals moving, and should lower the occurrence of premature convergence. According to [*Silva et al.*, 2002], this allows lower values for the inertia weight w, thus forcing a faster convergence, while relying on the predator to maintain population diversity.

This predator is also modelled as a particle, but uses another set of equations of motion:

$$\begin{cases} \mathbf{v}_{p,new} = c_4 \cdot u() \cdot (\mathbf{x}^*_G - \mathbf{x}_p) \\ \mathbf{x}_{p,new} = \mathbf{x}_p + \mathbf{v}_p \end{cases} \tag{4.13}$$

where c_4 is a constant that controls how fast the predator is able to catch up with the best individual.

This predator would be useless if it did not influence the motion of the normal particles. This is done by adding a term $D(d)$ to the equations of motion of the normal solutions, with a probability p_f. This term $D(d)$ is given in [*Silva et al.*, 2002] as:

$$D(d) = ae^{-bd} \tag{4.14}$$

where a can be tuned to set the maximum amplitude of the predator effect and b influences the distance at which the effect is still significant. d is the (Euclidean) distance between the prey and the predator in solution space.

This term $D(d)$ is put into equation (4.7) by modifying it for a randomly chosen direction k:

$$v_{i,k,new} = wv_{i,k} + c_1 \cdot u() \cdot (x^*_{i,k} - x_{i,j}) + c_2 \cdot u() \cdot (x^*_{G,k} - x_{i,j}) + c_3 \cdot u() \cdot D(d) \tag{4.15}$$

(a) Global parameters

Constant	Value
c_1	2.0
c_2	2.0
c_3	1.0
c_4	0.1
w_{max}	0.9 / 0.7
w_{min}	0.4 / 0.0

(b) Per-function parameters

Function	a	b	p_f
f_1	$2.0\mathbf{x}_{max}$	$10/\mathbf{x}_{max}$	0.001
f_2	$2.0\mathbf{x}_{max}$	$10/\mathbf{x}_{max}$	0.001
f_3	$0.1\mathbf{x}_{max}$	$10/\mathbf{x}_{max}$	0.04
f_4	$0.1\mathbf{x}_{max}$	$10/\mathbf{x}_{max}$	0.06
f_5	$0.1\mathbf{x}_{max}$	$10/\mathbf{x}_{max}$	0.04
f_6	$2.0\mathbf{x}_{max}$	$10/\mathbf{x}_{max}$	0.06

Table 4.2 Optimization parameters in PSO and PPO (source: [*Silva et al.*, 2002])

whereas the position update function is not modified.

Table 4.2 lists the values of the optimization parameters as used in [*Silva et al.*, 2002] for various problems. These values are based on knowledge of the optimized functions and experiments done by the authors.

[*Silva et al.*, 2002] benchmark normal PSO against PPO, and show that given equal population size and inertia weight factors, PPO arrives at better solutions, and converges faster for 5 out of 6 selected problems, and has a worse solution for 1 out of 6 selected problems.

In the conclusions of that paper, the authors mention the intention to develop a multi-predator variant of this algorithm, however, no follow-up studies were published. This could be used in the case of MOO (see chapter 6) as in that case, multiple best solutions can be identified. A number predators could then be added to the population, chasing the Pareto optimal individuals. Which of these individuals each predator then chases can then be determined using the distance between the predator and the solutions, or any other random factor. However, if the goal is to have the individuals spread out evenly over the Pareto front, using an algorithm such as NSGA-II gives the same result.

4.5 Applying bounds on particle velocity and position

When the new position of a particle is determined in PSO, bounds on the search space are (in general) not taken into account. This leads to the situation in which parameters which are impossible or forbidden (for example, payload having a negative weight or a launch date outside the permitted launch window) might appear as part of a solution. To prevent this, a method to apply bounds to solutions is proposed here:

Consider a parameter x_{ij} that is bound by the interval $[l_j, u_j]$. If it is moved outside of those bounds its speed and position should be adjusted as if it had bounced elastically against a wall. For the lower bound, this situation is depicted in figure 4.6.

When looking at this figure, the position of the particle after it has bounced equals

$$x'_{ij,new} = l_j + f \cdot d$$

where f is an elasticity factor (with $0 \leq f \leq 1$) which determines how much of the velocity (momentum) is kept during the bounce. d is the distance the particle would have travelled if no bounce would have occurred, and is given by

$$d = l_j - x_{ij,new}$$

Combining these two equations and repeating this procedure for the upper bound then

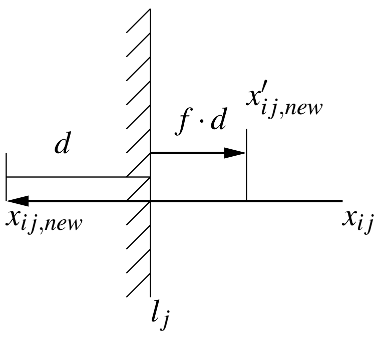


Figure 4.6 Calculating a particle's position when bouncing against a parameter boundary.

gives the following formula for the bound position $x'_{ij,new}$:

$$x'_{ij,new} = \begin{cases} x_{ij,new} & \text{if } l_j \le x_{ij,new} \le u_j \\ (1+f)l_j - f \cdot x_{ij,new} & \text{if } x_{ij,new} < l_j \\ (1+f)u_j - f \cdot x_{ij,new} & \text{if } x_{ij,new} > u_j \end{cases} \tag{4.16}$$

Of course, in case of a bounce, the velocity also should be adjusted:

$$v'_{ij,new} = \begin{cases} v_{ij,new} & \text{if } l_j < x_{ij,new} < u_j \\ -f \cdot v_{ij,new} & \text{otherwise} \end{cases} \tag{4.17}$$

Thus, by applying (4.16) and (4.17), particles can be kept inside the bounds of the solution space. In OPTIDUS, a default value of $f = 0.8$ is used, but it might be interesting to investigate the influence of this parameter. (4.16) shows that when f decreases, particles will end up closer to the bounds. This is logical, as a lower value of f causes the particle to lose more of its momentum.

4.6 Examples of contracting swarms

This section shows an example how a swarm moves through the solution space from generation to generation, and how it reaches the global optimum after several generations.

The function optimized here again is the Himmelblau function (given in section 7.1.1) with a swarm size $S = 40$. Figure 4.7 shows how the swarm moves: in the first generation, the particles are spread out randomly over the solution space. Already in the second generation, some clustering seems to be happening, and in the 10^{th} generation it is apparent that the position near $(3,2)$ is attracting the particles. In generation 40, almost all particles are near the optimum.

Figure 4.8 shows what happens when a predator is introduced. In that case, the population does contract, but every now and then, an individual is scared away along one of the axes. This behavior can be observed very clearly in generation 40, where the population takes the form of a cross with the optimum location as its center. This is the result of the individuals getting close to the optimum, where the predator lives, and then being scared away along one of the axes.[3] This also shows that using PPO for a unimodal problem like the Himmelblau function is overkill.

[3]The solution at approximately $(2,1.5)$ probably got scared twice.

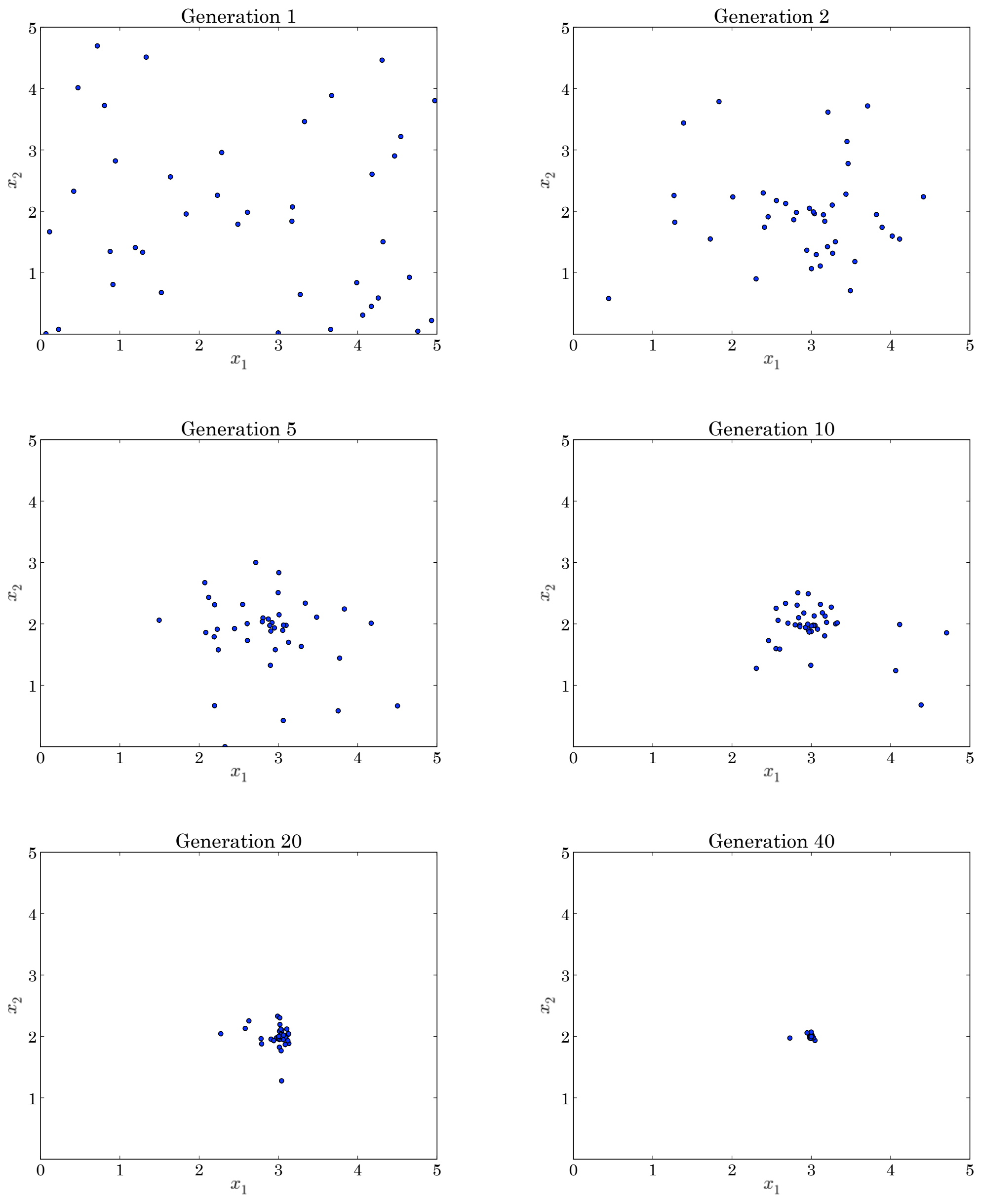


Figure 4.7 Movement of a particle swarm during an optimization of the Himmelblau function; the optimum is located at the coordinates $(3,2)$. The swarm sized used was $S = 40$.

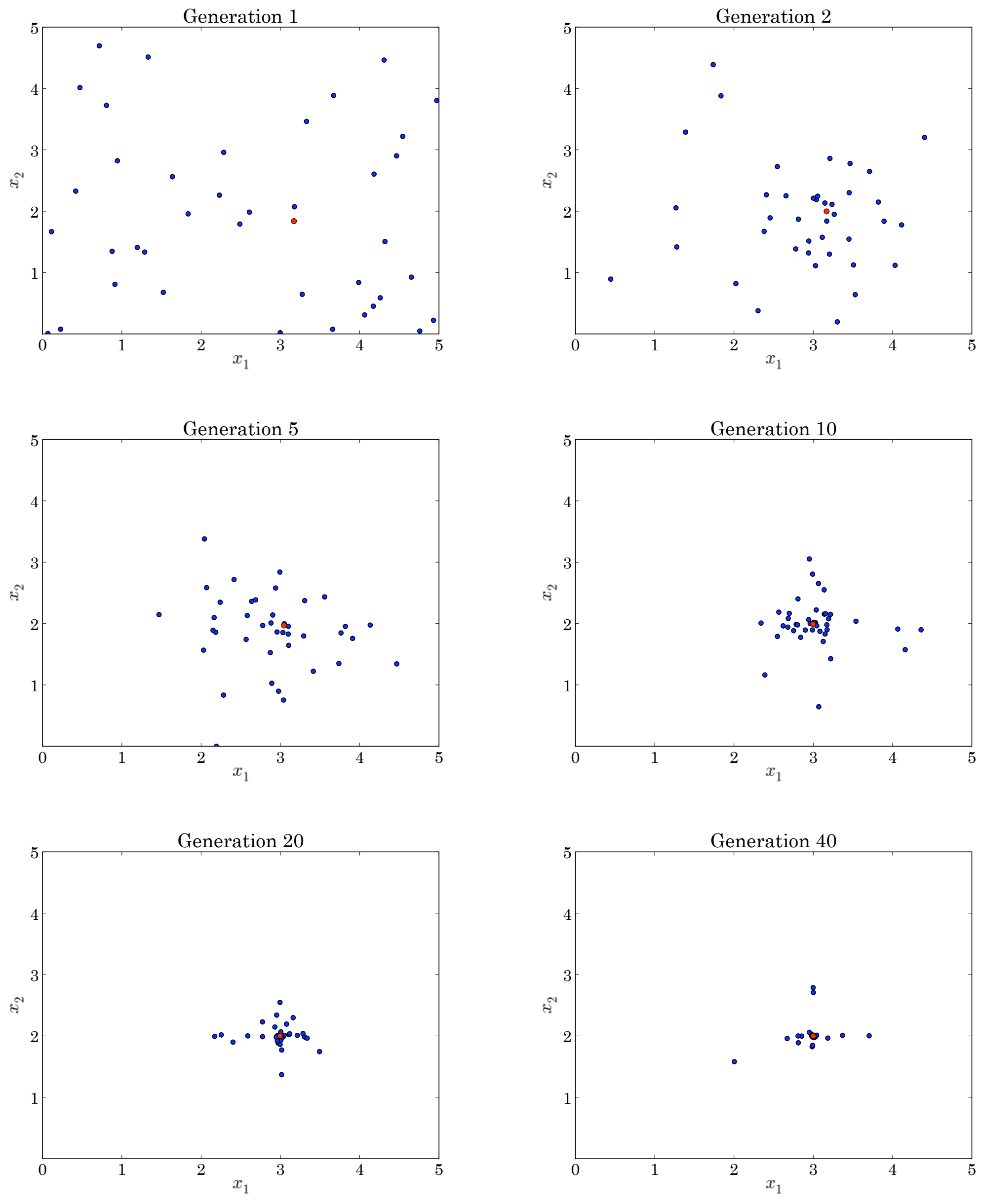


Figure 4.8 Movement of a particle swarm during an optimization of the Himmelblau function, with a predator (indicated by the red dot); the optimum is located at the coordinates $(3, 2)$. The swarm sized used was $S = 40$.

Chapter 5

Differential Evolution

This chapter treats Differential Evolution (DE), an evolutionary algorithm which was first introduced in [*Storn and Price*, 1997]. The general idea behind DE is to simulate a population of solutions, which can combine to produce offspring using several mathematical formulas. If such an interaction produces an individual which is better than the one from which it descends, it will replace its parent. By repeating this process for several generations, an optimal solution can be found.

5.1 Introduction to Differential Evolution

Differential Evolution (DE) is a population based evolutionary algorithm first proposed in [*Storn and Price*, 1997]. Just as with other evolutionary algorithms, a population of solutions is simulated which explore the population space. This is done by combining the values of several solutions to form a *mutation vector*, which is then combined with a solution using a crossover operator to form the *trial vector*.

A implementation detail was discovered after this step: it was necessary to apply the bounding algorithm from section 4.5 to make sure that all parameters are within the solution space. Even though this algorithm was designed for PSO, it can be applied to a DE problem when assuming all velocities are zero.

This trial vector is then compared with the original vector, and if its fitness value is better, it replaces that vector. This is summarized in figure 5.1.

Because of this comparison, DE can only be used when there is a way to sort solutions. This means that special methods will have to be applied in the case of MOO.

The following notation is used to represent a solution (which is a vector of parameters):

$$\mathbf{x}_i = [x_{i,0} \quad x_{i,1} \quad \dots \quad x_{i,N}]^T \quad \text{with } i \in \{1, \dots, S\} \tag{5.1}$$

in which S is the size of the population, and N the number of parameters from which the vector is built up.

5.2 Mutation vector generation

This section contains several methods for filling in the step "generate mutation vector" in figure 5.1.

5.2.1 Scheme Tasoulis1

The first scheme listed in [*Tasoulis et al.*, 2004] is called Tasoulis1. For each individual $\mathbf{x}_i$ in the population, it randomly picks two (distinct) individuals from the population, giving the individuals $\mathbf{x}_{r_1}$ and $\mathbf{x}_{r_2}$. These are combined with the best solution, $\mathbf{x}_{best}$, to form a mutation vector:

$$\mathbf{m}_i = \mathbf{x}_{best} + F(\mathbf{x}_{r_1} - \mathbf{x}_{r_2}) \tag{5.2}$$

where F is a real parameter, called mutation constant, which controls the amplification of the difference between two individuals so as to avoid the stagnation of the search progress. Based on the values given in [*Tasoulis et al.*, 2004], OPTIDUS uses $F = 0.7$ by default.

The movement of a population of solutions through the solution space for an optimization of the Himmelblau function can be seen in figure 5.2.

5.2.2 Scheme Tasoulis2 / DE1

The second scheme listed in [*Tasoulis et al.*, 2004] is the same as the first scheme listed in [*Storn and Price*, 1997], and is rather similar to Tasoulis1. The difference lies in the function used to generate the mutation vector, which uses a randomly picked individual instead of the best individual.

$$\mathbf{m}_i = \mathbf{x}_{r_1} + F(\mathbf{x}_{r_2} - \mathbf{x}_{r_3}) \tag{5.3}$$

The generation of the trial vector is the same.

The movement of a population of solutions through the solution space for an optimization of the Himmelblau function can be seen in figure 5.3.

Because a randomly chosen individual is used as the basis of the mutation vector instead of the best individual, the chance that the trial vector is rejected is bigger. This leads to slower movement, which is evident when comparing figures 5.3 and 5.2.

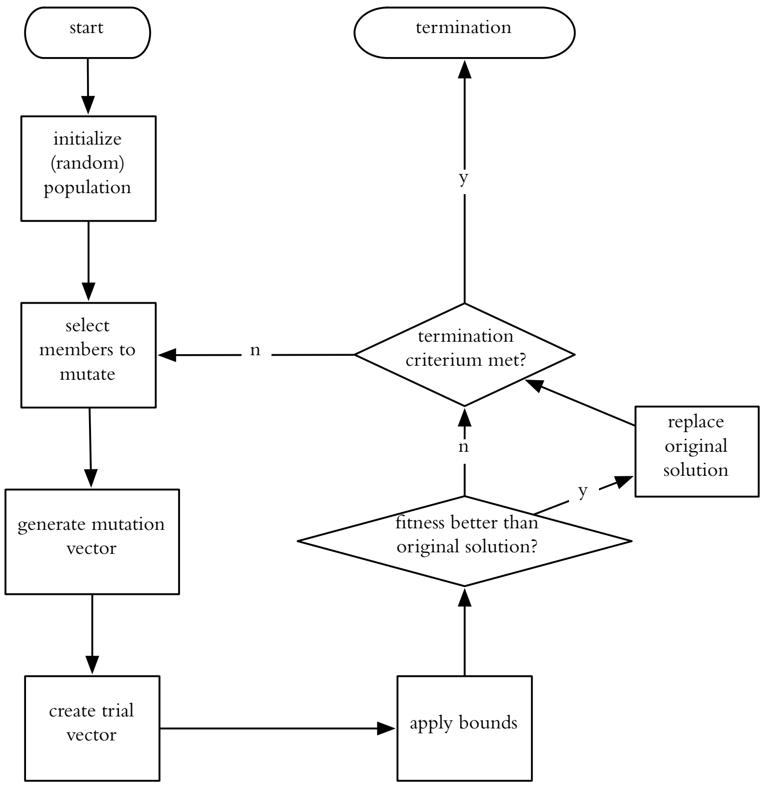

start
termination
initialize (random) population
y
select members to mutate
n
termination criterium met?
replace original solution
n
y
generate mutation vector
fitness better than original solution?
create trial vector
apply bounds

Figure 5.1 Flowchart of a basic Differential Evolution algorithm

5.2.3 Scheme Tasoulis3/DE2

The scheme DE2 is mentioned in [*Storn and Price*, 1997] and uses the following method to construct the mutation vector:

$$\mathbf{m}_i = \mathbf{x}_{r_i} + \lambda(\mathbf{x}_{best} - \mathbf{x}_{r_i}) + F(\mathbf{x}_{r_1} - \mathbf{x}_{r_2}) \tag{5.4}$$

In this equation, λ is a parameter like F which can be used to enhance the greediness of the algorithm. According to [*Storn and Price*, 1997], this feature can be useful for non-critical objective functions. Note that setting $\lambda = 1$ results in the same mutation vector as Tasoulis1.

By increasing λ, solutions tend to move in the direction of the best individual in the population, which might lead to premature convergence, as the diversity of the population then decreases. Setting λ to zero yields a scheme similar to Tasoulis2, except that a random individual $\vec{x_{r_1}}$ is picked as the basis of the mutation vector, instead of the original solution $\vec{x_{r_i}}$.

In [*Tasoulis et al.*, 2004], λ is given the value of F, yielding the scheme Tasoulis3:

$$\mathbf{m}_i = \mathbf{x}_{r_i} + F(\mathbf{x}_{best} - \mathbf{x}_{r_i} + \mathbf{x}_{r_1} - \mathbf{x}_{r_2}) \tag{5.5}$$

Figure 5.4 shows how a population subjected to Tasoulis3 explores the Himmelblau function.

5.2.4 Scheme Tasoulis4

Tasoulis4 is given in [*Tasoulis et al.*, 2004] and combines the parameters of 4 individuals to form a mutation vector using the following solution:

$$\mathbf{m}_i = \mathbf{x}_{best} + F(\mathbf{x}_{r_1} - \mathbf{x}_{r_2} + \mathbf{x}_{r_3} - \mathbf{x}_{r_4}) \tag{5.6}$$

Figure 5.5 shows how this scheme optimizes the Himmelblau function.

5.2.5 Scheme Tasoulis5

Tasoulis5 is almost the same as Tasoulis4, but uses a fifth random individual instead of the best to base the mutant individual upon.

$$\mathbf{m}_i = \mathbf{x}_{r_1} + F(\mathbf{x}_{r_2} - \mathbf{x}_{r_3} + \mathbf{x}_{r_4} - \mathbf{x}_{r_5}) \tag{5.7}$$

Figure 5.6 shows how this scheme optimizes the Himmelblau function. When comparing the location of the individuals in generation 20 for the different schemes, this one is the least converged.

5.2.6 Scheme Tasoulis6

The final scheme mentioned in [*Tasoulis et al.*, 2004] is Tasoulis6, which is also known as the trigonometric mutation. It was first introduced in [*Fan and Lampinen*, 2003]. This scheme uses a method which randomly chooses one of two methods for calculating mutation vectors.

With probability τ_μ it uses the trigonometric mutation operator, which is defined as:

$$\begin{aligned}\mathbf{m}_i = {} & (\mathbf{x}_{r_1} + \mathbf{x}_{r_2} + \mathbf{x}_{r_3})/3 + (p_2 - p_1)(\mathbf{x}_{r_1} - \mathbf{x}_{r_2}) + \\ & (p_3 - p_2)(\mathbf{x}_{r_2} - \mathbf{x}_{r_3}) + (p_1 - p_3)(\mathbf{x}_{r_3} - \mathbf{x}_{r_1})\end{aligned} \tag{5.8}$$

where the weights p_1, p_2, p_3 have the following values:

$$p_i = |f(\mathbf{x}_{r_i}|/p' \quad i = 1,2,3$$
$$p' = \sum_{i=1}^{3} |f(\mathbf{x}_{r_i})|$$

This trigonometric mutation results in a sort of weighted average over the three individuals. There is a difference: the generated points do not need to lie within the triangle spanned up by the original individuals in the solution space, so some exploration of the solution space is possible.

The other possibility (with probability $(1-\tau_\mu)$) is using the mutation strategy from Tasoulis2.

[*Fan and Lampinen*, 2003] suggests setting $\tau_\mu = 0.05$, which was chosen as the default setting for OPTIDUS.

Figure 5.7 shows the evolution of a population optimizing the Himmelblau function using this scheme.

5.2.7 Alternative naming of the schemes

[*Storn and Price*, 1997] introduces another naming method, in which algorithms are named as DE/x/y/z.

- x specifies the vector to be mutated which can be *rand* (a randomly chosen population vector) or *best* (the vector of lowest cost from the current population) or *self* (the vector against with which the mutation vector will be combined)
- y is the number of difference vectors used.
- z denotes the crossover scheme. For example "bin" for uniform crossover.

Table 5.1 shows the names of the schemes listed in this chapter according to this naming method. Note that Tasoulis6 cannot be named as it does not use difference vectors, and that the crossover field is not filled in.

5.3 Trial vector generation and selection

This section describes how the step "create trial vector" in figure 5.1 is implemented.

The trial vector u is generated by combining the mutation vector m_i generated in the previous step and the original solution vector x_i using a crossover operator.

[*Storn and Price*, 1997] uses uniform crossover to generate the trial vector. Each element of the trial vector is determined at random, having a chance of ρ of being copied from the mutation vector, and a chance of $(1-\rho)$ of being copied from the original solution vector. Values given for ρ vary from 0.3 to 0.5. OPTIDUS uses a value of $\rho = 0.5$.

Once the trial vector is computed, the objective function is then evaluated for it. If its outcome is better than that of the original vector $\mathbf{x}_i$, the original vector in the population is replaced by the trial vector.

Tasoulis	Stone
Tasoulis1	DE/best/1/...
Tasoulis2	DE/rand/1/...
Tasoulis3	DE/self/2/...
Tasoulis4	DE/best/2/...
Tasoulis5	DE/rand/2/...

Table 5.1 Conversion between the DE scheme names used in this study and the "Storn" names.

5.4 Usage of Differential Evolution

In [*Storn and Price*, 1997] the schemes DE1 and DE2 are compared to Adaptive SA and the Annealed Nelder & Mead approach, using 9 test functions, amongst which the 5 De Jong's functions, Corona's parabola, Zimmermann's problem (all listed in appendix A) and Griewank's function (see 7.1.2). Differential Evolution is able to find good solutions for all cases, but needs more function evaluations than the other two algorithms for some problems.

In [*Tasoulis et al.*, 2004], all 6 schemes are compared by performing an optimization of several mathematical test functions including Griewank's function (described in section 7.1.2). Tasoulis1 performed best for all tested functions.

In [*Vinkó et al.*, 2007], a comparison of several optimization methods was performed, including DE, EA and PSO. The problems benchmarked were multiple gravity assist missions with the possibility of deep space manoeuvres. The conclusion was that using a method combining several algorithms led to the best results, but DE was the best individual algorithm. The strategy used was Tasoulis2 with uniform crossover.

5.5 Selecting Differential Evolution strategies

The following pages (containing figures 5.2–5.7) show how the populations of a DE optimizer applied to the Himmelblau function evolve, when using the six schemes from section 5.2. The convergence speed is plotted in figures 5.8 and 5.9.

As can be seen, all 6 schemes reach the global optimum at $(3, 2)$, but the speed with which this happens differs. Tasoulis1 and Tasoulis3 both have reached convergence around generation 30, giving the best performance. This is in accordance with the results from [*Tasoulis et al.*, 2004], where the Sphere test function (De Jong F1, see A.1, which like the Griewank function is a unimodal function) was optimized quickest by these two algorithms.

For this reason, **Tasoulis1** was selected for solving the Solar polar mission, which will be treated in chapter 8.

Tasoulis6 was also selected, even though it did not perform very well on the Himmelblau function. However, it shows good results on the functions from [*Tasoulis et al.*, 2004]. Finally, it was selected because it is quite different from Tasoulis1.

The other schemes have not been selected; according to [*Tasoulis et al.*, 2004], Tasoulis schemes 2, 4 and 5 perform badly. Also, because scheme Tasoulis3 is quite similar to Tasoulis1, it has not been included.

Tasoulis1 and Tasoulis6 will also be used in the verification tests in the next chapter.

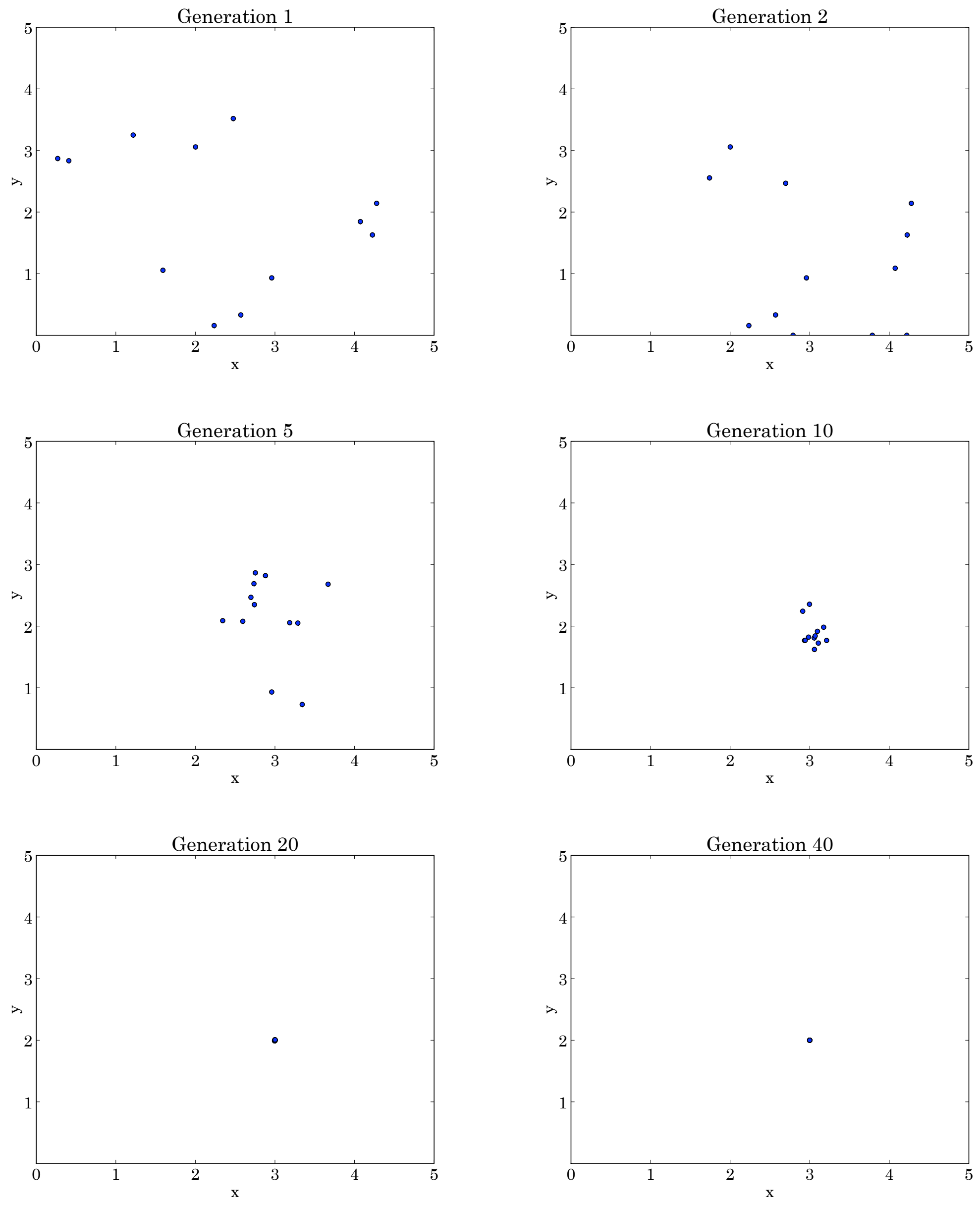


Figure 5.2 Movement of a population of solutions during an optimization of the Himmelblau function using Differential Evolution, population size $S = 12$ and scheme Tasoulis1; the optimum is located at the coordinates $(3, 2)$

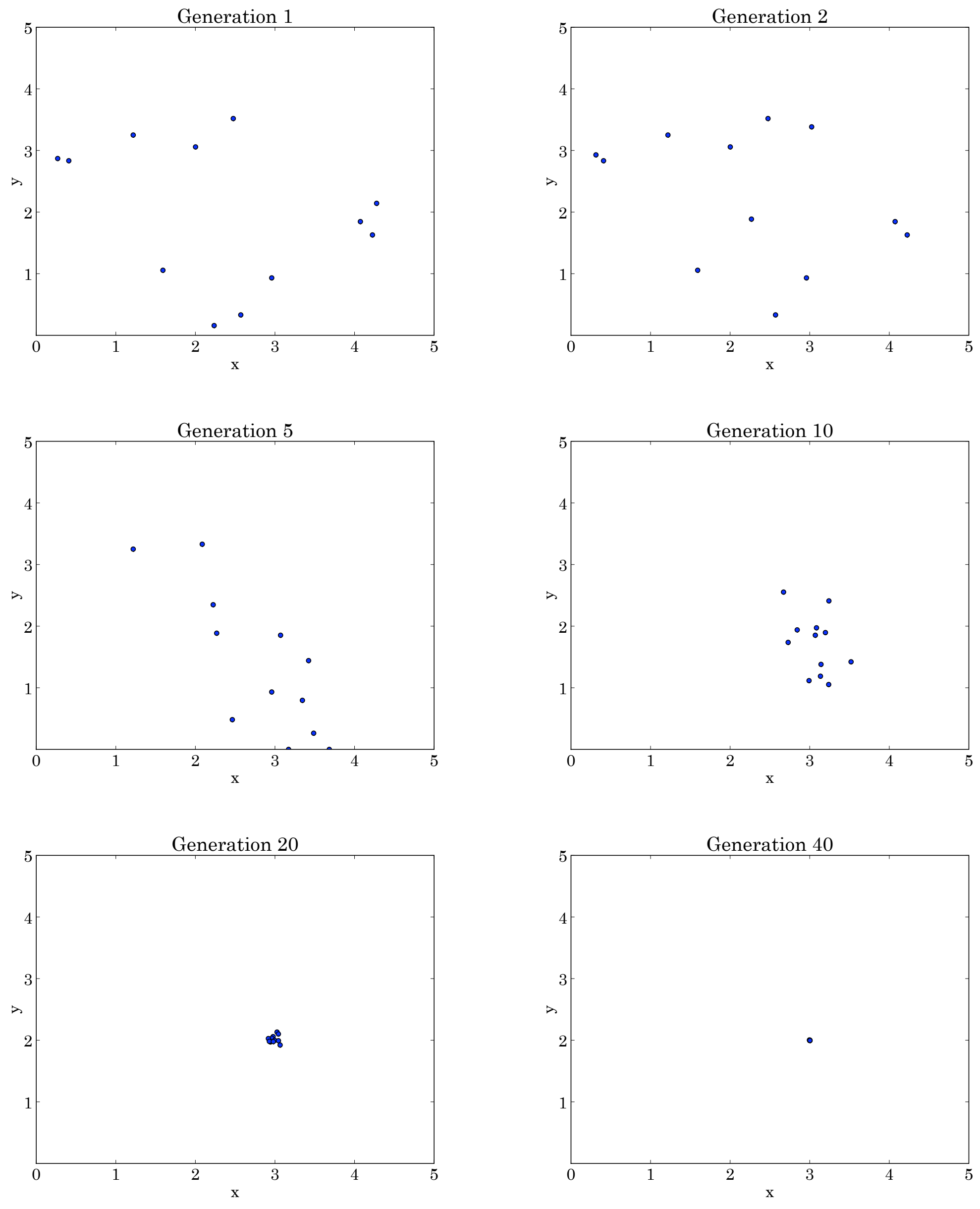


Figure 5.3 Movement of a population of solutions during an optimization of the Himmelblau function using Differential Evolution, population size $S = 12$ and scheme Tasoulis2; the optimum is located at the coordinates $(3, 2)$

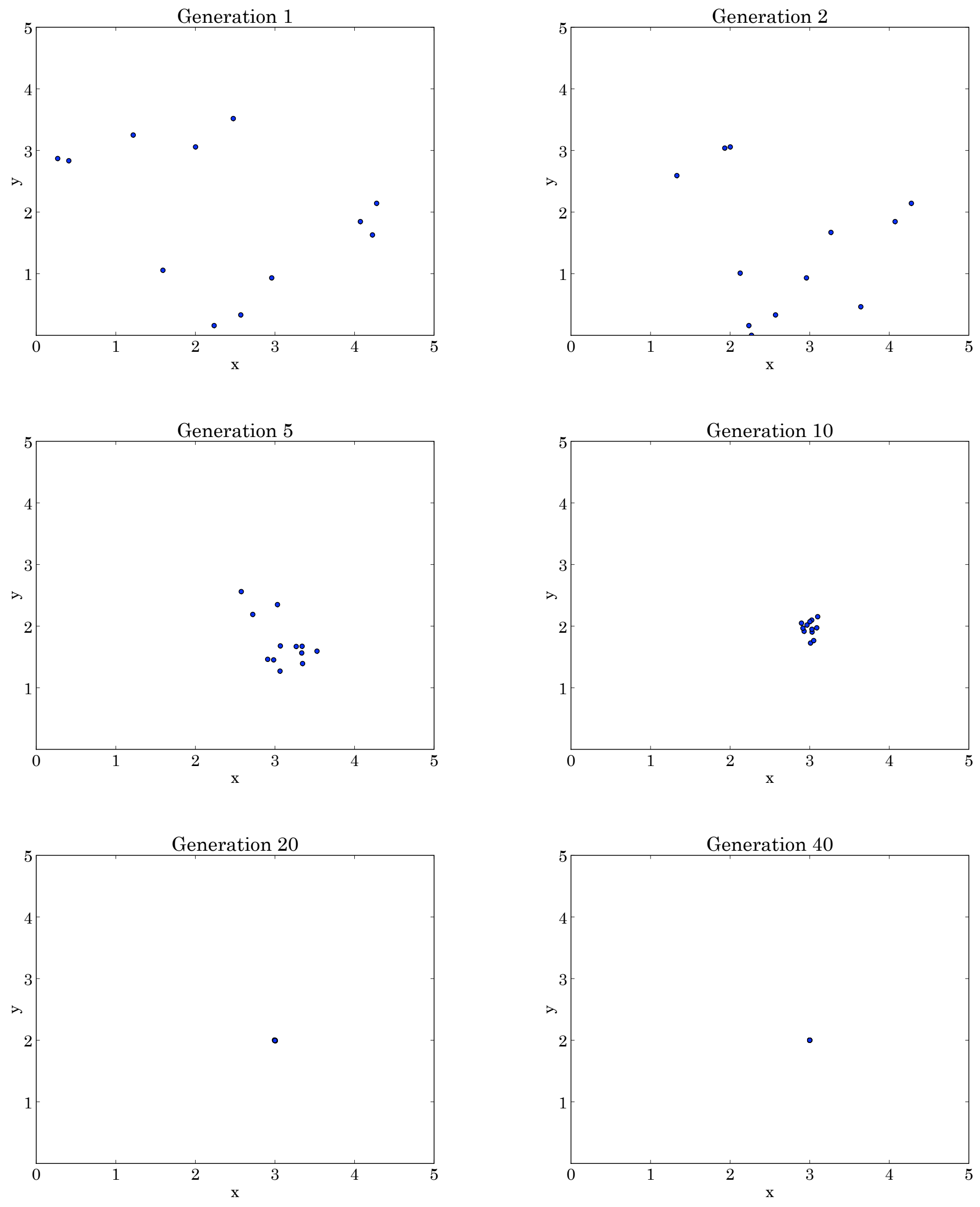


Figure 5.4 Movement of a population of solutions during an optimization of the Himmelblau function using Differential Evolution, population size $S = 12$ and scheme Tasoulis3; the optimum is located at the coordinates $(3, 2)$

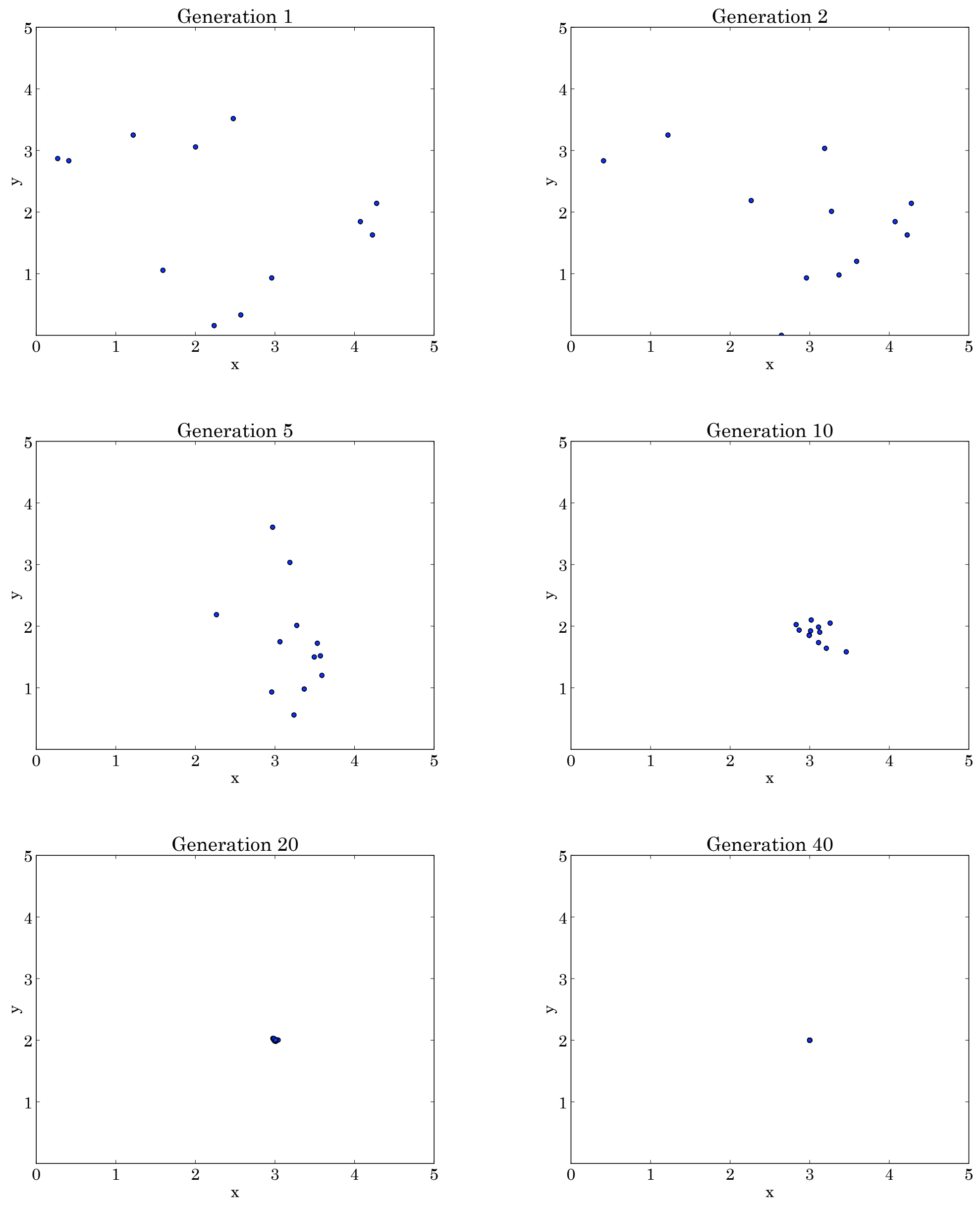


Figure 5.5 Movement of a population of solutions during an optimization of the Himmelblau function using Differential Evolution, population size $S = 12$ and scheme Tasoulis4; the optimum is located at the coordinates $(3, 2)$

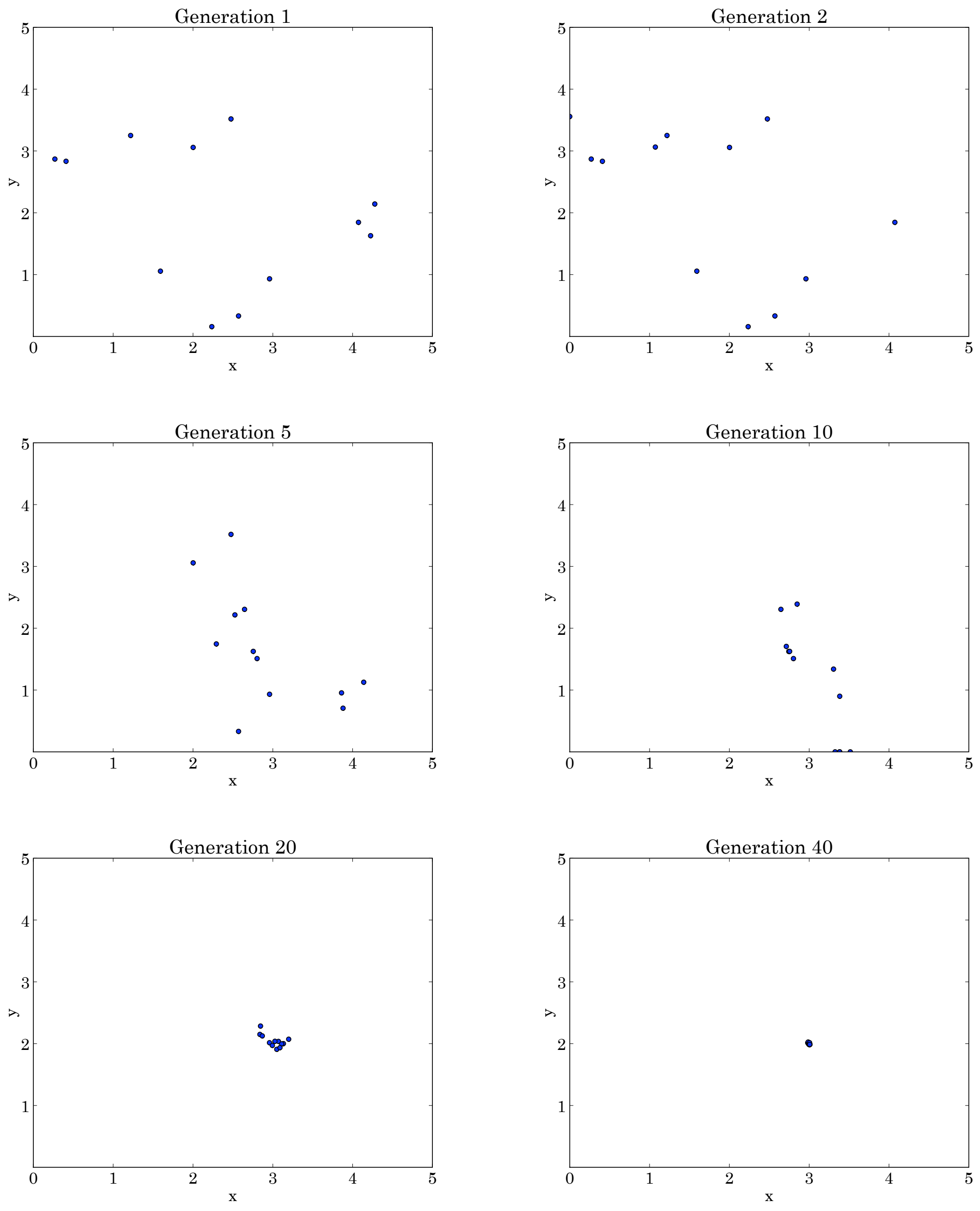


Figure 5.6 Movement of a population of solutions during an optimization of the Himmelblau function using Differential Evolution, population size $S = 12$ and scheme Tasoulis5; the optimum is located at the coordinates $(3, 2)$

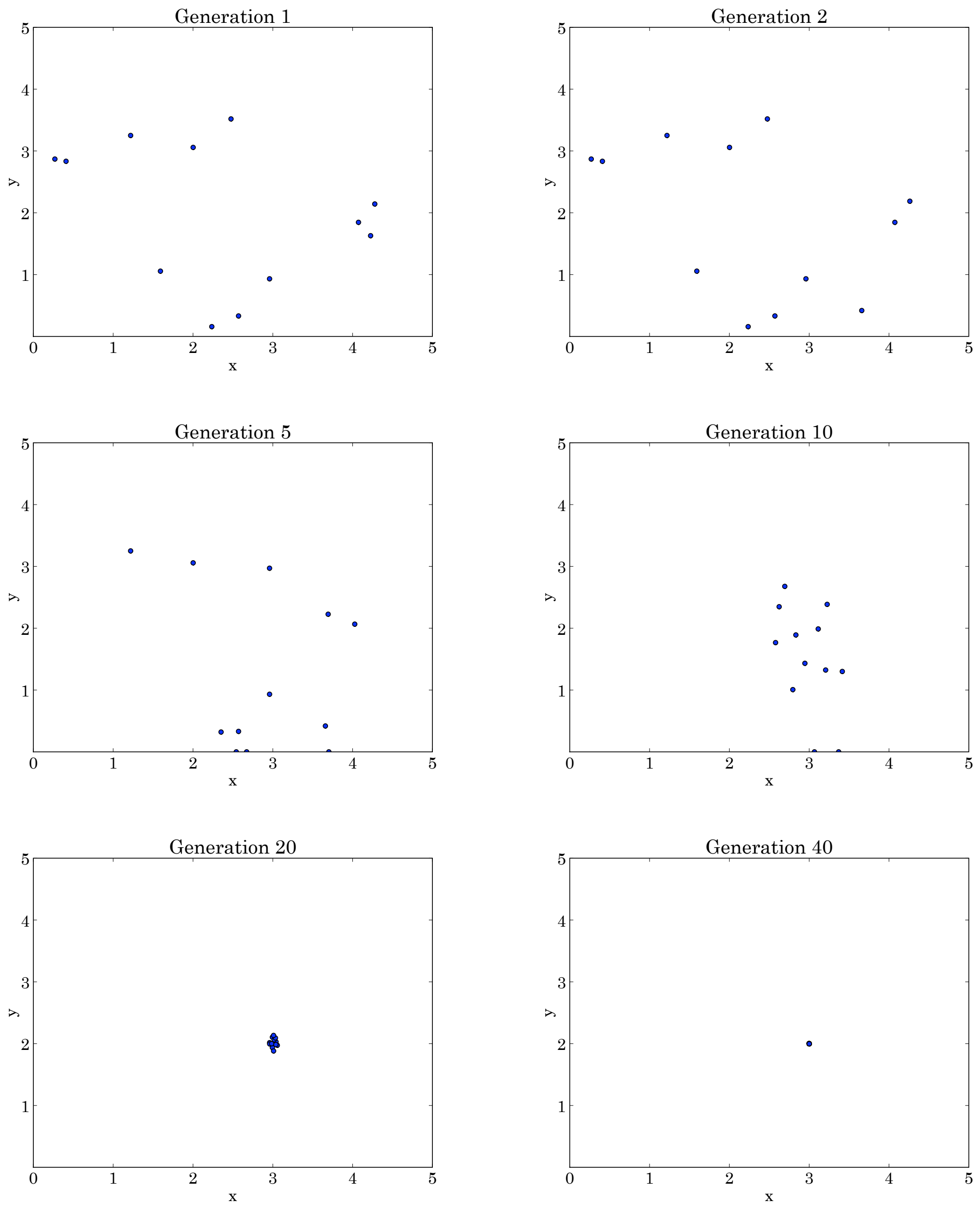


Figure 5.7 Movement of a population of solutions during an optimization of the Himmelblau function using Differential Evolution, population size $S = 12$ and scheme Tasoulis6; the optimum is located at the coordinates $(3, 2)$

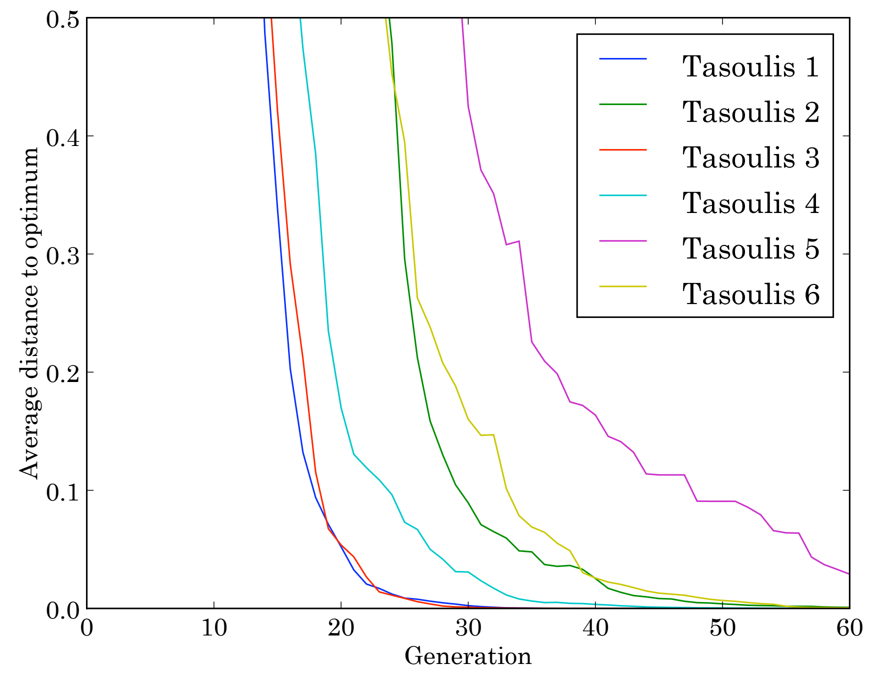


Figure 5.8 Average distance of the population to the global optimum as a function of the generation during an optimization of the Himmelblau function using several DE schemes. The increase in distance in Tasoulis5 around generation 34 is not an error; the distance can increase while the fitness decreases.

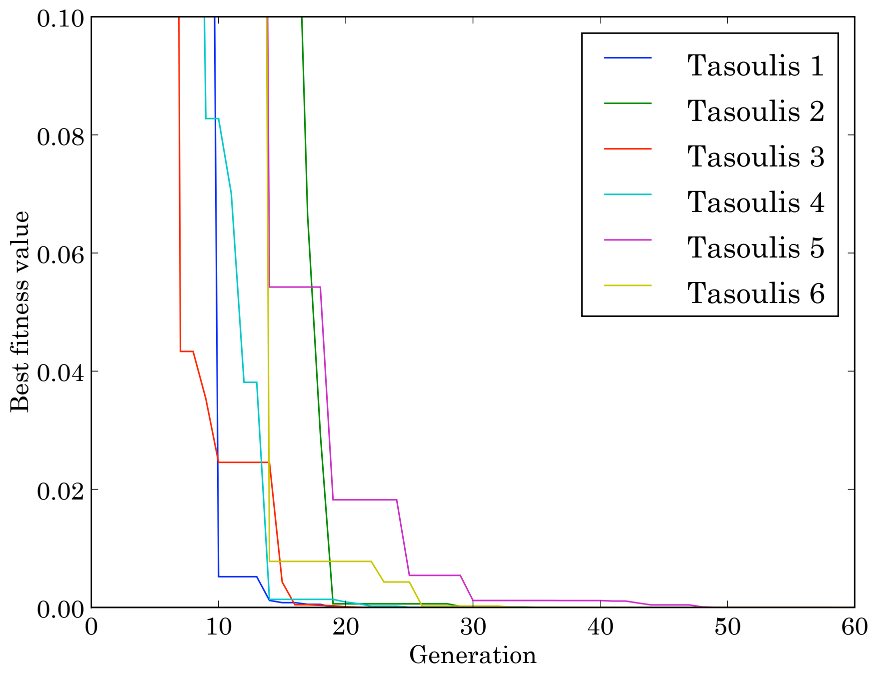


Figure 5.9 Fitness value of the best individual per generation, during an optimization of the Himmelblau function using several DE schemes.

Chapter *6*

Multi-Objective Optimization (MOO)

This chapter treats multi-objective optimization (MOO). MOO is a technique to handle problems which have more than one objective to be optimized. For example, consider the optimization done in [*Safipour*, 2007]: in that thesis, a trajectory around Neptune and Triton was optimized with two objectives:

- Maximize the number of flybys n_f along the inner moons of Neptune
- Minimize the required ΔV-budget to get into the final science orbit

Three (fictional) solutions for this problem have been plotted in figure 2.1, which is repeated below. The question is if it is possible to identify the best of these solutions.

What can be seen in this figure, is that solution A is the best when only considering the required ΔV budget, and B is the best when only considering the number of bodies visited. The comparisons using each objective thus contradict each other.

What cannot be disputed is that solution C is worse than B on both objectives, and thus cannot be the best solution.

This chapter will explore two methods to handle multi-objective optimization. The first one makes use of a weighted objective function, the second one uses Pareto front ranking. The Pareto front ranking will be extended with intra-front ranking using an algorithm called NSGA-II, which acts as a sort of fitness sharing to make sure solutions are spread out over the solution space [*Deb et al.*, 2001].

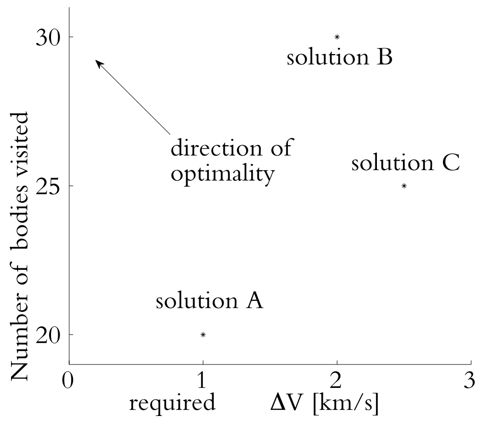


Figure 6.1 Example of the objective values for some solutions of a multi-objective problem.

6.1 Weighted objective function

One possible method to solve MOO problems is to construct a function which reduces the multiple objectives into one function.

An example of this is using a weighted objective function, for example:

$$f = \sum_{k=1}^{K} w_k f_k \tag{6.1}$$

This function combines K different objective functions are combined to give one grand total. The only problem with this is that the weights w_k have to be determined by the user.

An example of how this can be used is making use of a cost (or benefit) function. For example, in the problem of [*Safipour*, 2007], the ΔV parameter could be given a cost of 20 M€/(km/s), and visiting an extra moon a value of 2.5M€. This translates into a cost function of:

$$C_1 = 20[\text{M€}/(\text{km/s})]\Delta V - 2.5[\text{M€}]n_f \tag{6.2}$$

However, if the valuation of extra flybys is much lower, for example only 1M€per flyby, the cost function changes to:

$$C_2 = 20[\text{M€}/(\text{km/s})]\Delta V - 1[\text{M€}]n_f \tag{6.3}$$

These two cost functions were applied to the three solutions, giving the costs in table 6.1. From this, the influence of the weights is apparent: when using the weights of C_1, solution B has the lowest cost, but when using the weights of C_2, solution A has the lowest cost.

This illustrates the problem with using weights: if they can be chosen arbitrarily, the outcome of the optimization can be influenced.

Of course, a function to combine objective values is not restricted to linear combinations of the multiple objective values. For example, in [*Van der Pols*, 2006], the objective function used two numbers: the ΔV budget and the number of manoeuvres n_m necessary for station keeping around in a halo orbit. The were combined using the following expression:

$$f = \frac{n_m}{\Delta V}$$

Another example, using both a division and a weight, can be found in the GTOC3 problem definition [*Casalino et al.*, 2007]:
"Objective of the optimisation is to maximise the nondimensional quantity

$$J = \frac{m_f}{m_i} + K\frac{\min_{j=1,3}(\tau_j)}{\tau_{max}}$$

where m_i and m_f are the spacecraft initial and final mass, respectively; τ_j, with $j = 1,3$, represents the stay-time at the j-th asteroid in the rendezvous sequence and $\min_{j=1,3}(\tau_j)$ is the shortest asteroid stay-time; $\tau_{max} = 10$ years is the available trip time, and $K = 0.2$."

Solution	ΔV [km/s]	n_f [-]	C_1 [M€]	C_2 [M€]
A	1.0	20	−30.0	0.0
B	2.0	30	−35.0	10.0
C	2.5	25	−12.5	25.0

Table 6.1 Comparison of two different weights for weighted objective functions

6.2 Pareto front ranking

Pareto front ranking is a method to make it possible to give an objective ranking of results in MOO problems – where objective means that the ranking cannot be influenced by tuning weights or other factors.

Pareto front ranking is based on the concept of domination: if two solutions are compared, one is said to dominate the other if the outcomes of not all objective functions are equal, and all the outcomes of solution A are equal or better then those of solution B. In a more mathematical form:

When comparing a solution i and a solution j with objective values f_i and f_j respectively, it is said that $i \prec j$ if:[1]

$$\forall k \in \{1,..,K\} : f_{i,k} \leq f_{j,k} \tag{6.4}$$

and

$$\exists k \in \{1,...,K\} : f_{i,k} \neq f_{j,k} \tag{6.5}$$

This domination is then used to group the solutions into so-called Pareto fronts; ranking of the solutions (for example to assign reproduction probabilities, or to make use of elitism) is then based on the rank of the Pareto front the solutions are in.

Figure 6.2 shows an example of a population of solutions, trying to minimize two objective values. The first four Pareto fronts have been ranked and the solutions which make up those fronts have been connected by lines.[2]

6.2.1 Classic Pareto front ranking algorithm

In [*Safipour*, 2007], MOO support was introduced in OPTIDUS. There were two variants introduced; the "Classic" one is able to determine the front for all individuals in the population

[1]Following the notation from [*Deb et al.*, 2001], $i \prec j$ means solution i dominates solution j

[2]Note that the lines crossing each other is not an error – curved lines could also have been used to connect members of the same front, and can be drawn not to cross each other as the number of crossings is even.

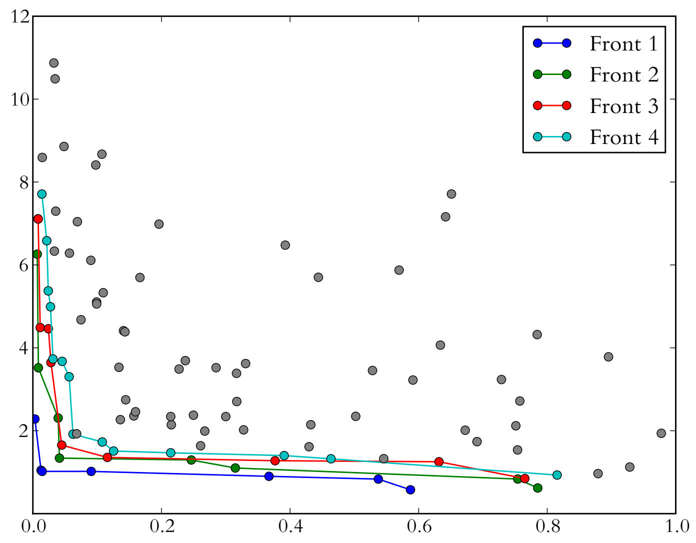


Figure 6.2 Example of a population in which the first four Pareto fronts have been ranked

using an algorithm which is given in condensed form in listing 6.1. This algorithm does not perform optimal; it needs O(KS^2) comparisons per ranked front (where K is the number of objectives and S the population size). The other variant given is the "Modern" one, which basically uses the same algorithm. The only difference is that it is limited to ranking no more than 5 fronts.

6.2.2 Non-dominated Sorting Genetic Algorithm II (NSGA-II)

In [*Deb et al.*, 2001], a faster algorithm is proposed, requiring O(KS^2) comparisons for the initialisation step, and O(S) operations for each front generated. It is called the "Non-dominated Sorting Genetic Algorithm II" (NSGA-II)[3] and is given in condensed form in listing 6.2.

This listing is in pseudo-code, which differs fundamentally from the Fortran implementation; because Fortran95 does not have built-in support for lists, the algorithm has been rewritten to make use of a domination matrix. Because of this, it requires O(S^2) operations per ranked front.

Comparing the Fortran implementation of NSGA-II with the classic implementation shows that the NSGA-II one needs a factor K less comparisons in the per-front loop, and should be faster. Another speed improvement can be found in the initialization of the domination matrix: this loop can be executed in parallel, so on a system with more than one core available, it should give faster results.

6.3 Ranking individuals inside their front

To be able to order the solutions inside a Pareto front (for example to compare two individuals in the same front for DE), an additional ordering method is provided in [*Deb et al.*, 2001], based on the concept of crowding. This is done by calculating a crowding distance, which is the sum of the sides of the box around the solution in which no neighboring solutions are found. Figure 6.3 shows what the box looks like.

The algorithm used to calculate this box is given in listing 6.3. In that listing, the solutions at the extremities of the front get a crowding distance of infinity. This is to make sure that those solutions are ranked higher than the other solutions.

Now, to increase chances of the solutions being spread out evenly over the solution space, when a choice is to be made between two members of the same Pareto front, the one with the biggest crowding distance is selected.[4]

[3] Despite its name, this algorithm is also usable for other optimization methods than genetic algorithms.

[4] [*Deb et al.*, 2001] does not tell us what happens when two individuals with the same crowding distance are

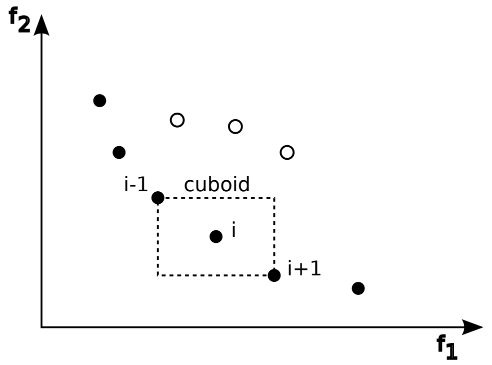


Figure 6.3 The crowding distance calculation [*Deb et al.*, 2001].

6.4 Verification of the NSGA-II implementation

To make sure the implementation of the NSGA-II implementation works, it was used in the optimization of a (mathematical) 2-objective function using differential evolution. This function was used in [*Deb*, 1999] to show the performance of the first NSGA algorithm.

The objective functions f_1 and f_2 are defined as:

$$\begin{aligned} f_1 &= x_1 \\ f_2 &= g \cdot h \\ g &= 1 + 10x_2 \\ h &= 1 - (x_1/g)^2 - (x_1/g)\sin(2\pi \cdot 4x_1) \end{aligned} \tag{6.6}$$

with the input domain

$$x_1, x_2 \in [0,1]$$

Writing out the expression for f_2 yields:

$$f_2 = g - x_1^2/g - x_1 \sin(2\pi \cdot 4x_1)$$

From this, it is clear that g must be minimized to minimize f_2, so it is reasonable to expect that the first Pareto front should be on the line where g is minimal. This line is plotted in figure 6.4.

Furthermore, two adjacent points in the first Pareto front should be connected by a line which goes from top-left to bottom-right, because else the objectives of one of these solutions is worse in both directions, and that solution becomes dominated. Thus, the points of the Pareto front are located on the parts of the line in figure 6.4 which move downward and are lower than the lowest point reached yet when going from left to right.

Figure 6.5 shows the first Pareto front found by an implementation of the first NSGA algorithm [*Deb*, 1999], with a population size $S = 200$ and 300 generations, using a normal GA.

Figure 6.6 shows how the evolution of a population of solutions using DE(Tasoulis1) and NSGA-II, after only 40 generations with a population size $S = 100$. It has already developed a Pareto-front which looks a lot like the front described above, and the points are spread out evenly over the front.

This shows that the NSGA-II method works, and that the implementation in Fortran also works correctly. Furthermore, the GA implementation needs much more generations and a bigger population than the DE implementation to reach a good result - GA uses a factor 15 more evaluations in this case.

chosen to battle each other, but this should not happen often. If it does happen, picking the winner at random will suffice.

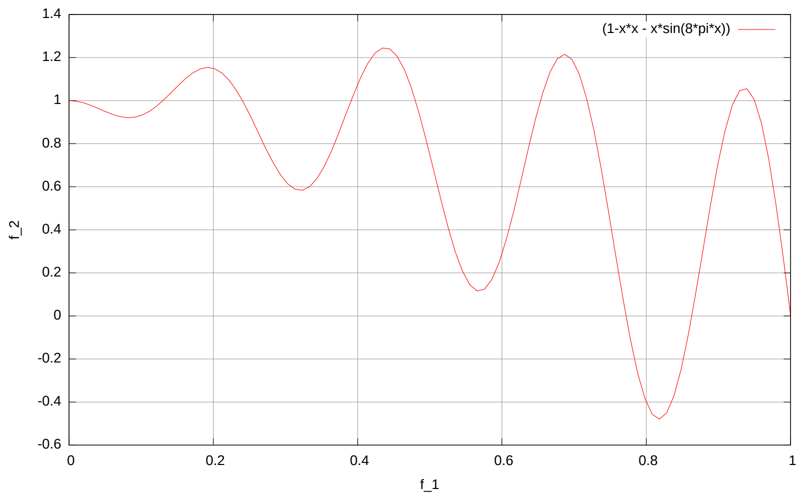


Figure 6.4 Analytical solution with the minimal values of f_2 from (6.6)

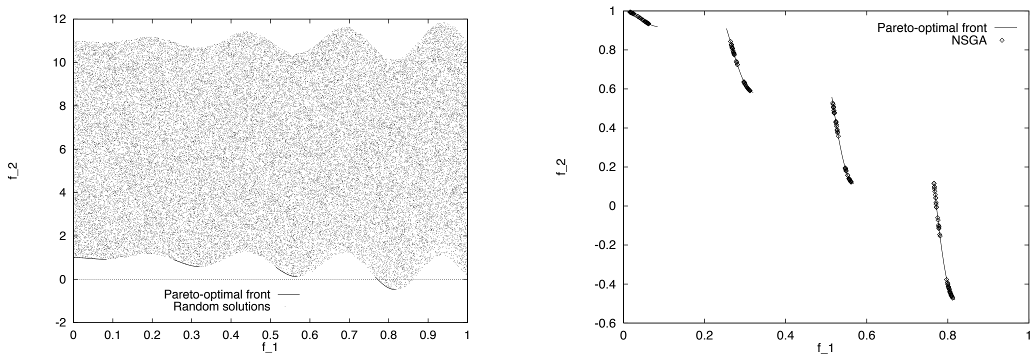


Figure 6.5 Exploration of the search space and results for optimizing (6.6) using the first NSGA algorithm, from [*Deb*, 1999]

```
for front in {1, ..., number of fronts}
  for each p in population
    if p not ranked yet
      set p%dominated = 0
      for each q in population not ranked yet
        if q $\prec$ p
          set p%dominated = 1
      if p%dominated == 0
        set p%front = front
  exit if all individuals have been ranked
```

Listing 6.1 Pareto front ranking algorithm from [*Safipour*, 2007]

```
% initialize dominated_by lists and dominators counters
for each p in population
  for each q in population
    if p $\prec$ q
       append q to dominated_by(p)
    if q $\prec$ p
      increase dominators(p) by 1
  if dominators(p) == 0
    front(1).append(p)
set f = 1
% main loop
do
  if fronts(f) is empty
    exit
  for each p in fronts(f)
    for each q in dominated_by(p)
      decrease dominators(q) by 1
      if dominators(q) ==  0
        set q%front = f+1
        append q to front(f+1)
  increase f by 1
```

Listing 6.2 NSGA-II Pareto front ranking algorithm

```
for each m in objectives
  sort population using key m
  population(1)%distance = infinity
  population(S)%distance = infinity
  for individual = 2 to S-1
    increase distance(individual) by
                    $(f(individual+1)_k - f(individual-1)_k)$
```

Listing 6.3 NSGA-II Crowding distance algorithm

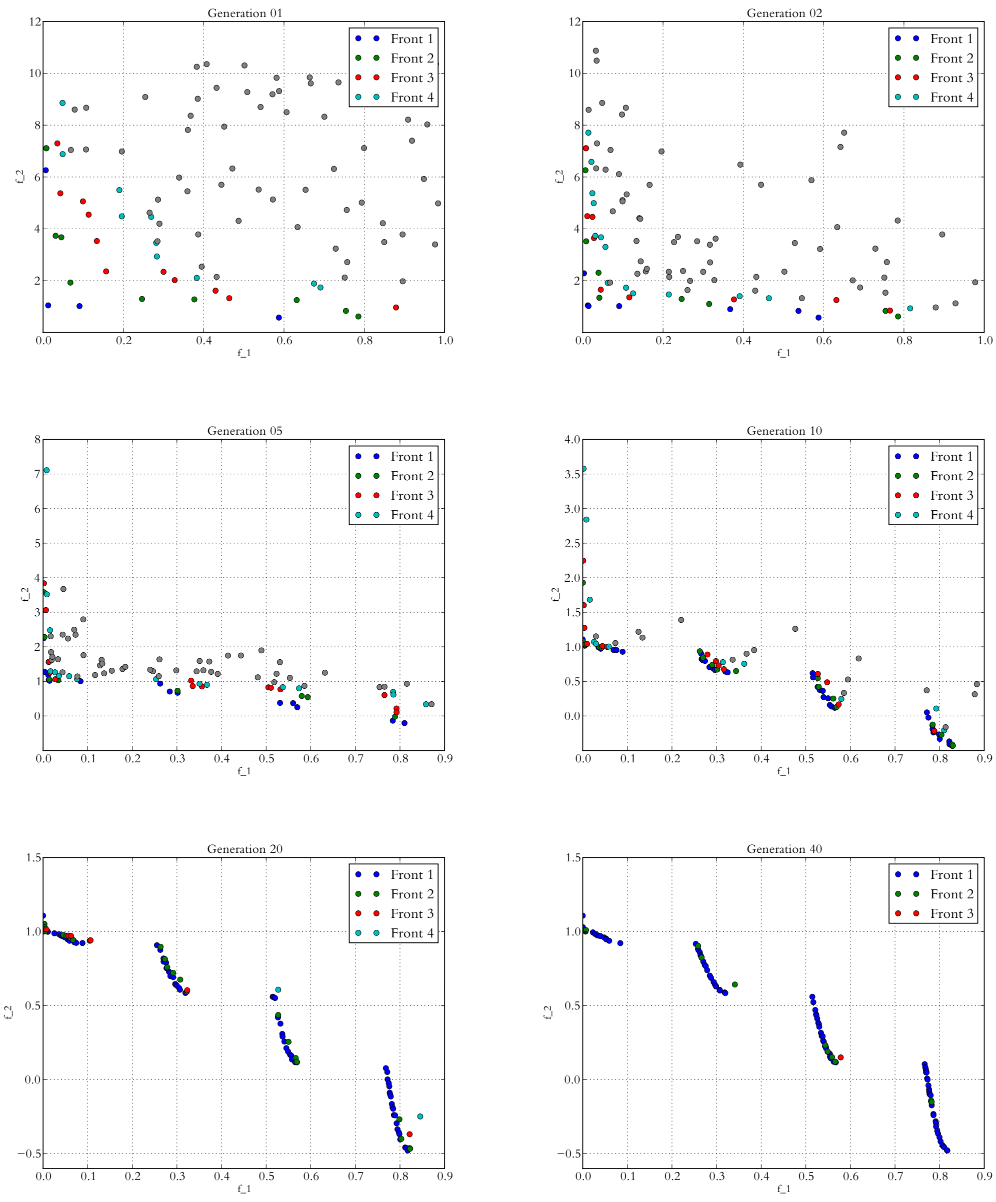


Figure 6.6 Movement of a population of solutions while optimizing the problem given by (6.6). This optimization is done using Differential Evolution, and uses NSGA-II to rank the population.

Chapter *7*

Verification of PSO and DE on mathematical test functions

This chapter contains the results of optimizing 4 mathematical test functions using several particle swarm optimization schemes, and the Tasoulis1 and Tasoulis6 schemes of DE. This was done to make sure that the various parameters which can be tuned in PSO and PPO are set to the right values. First, the mathematical functions will be introduced, and later the 6 optimizers which are compared. Finally, these optimizers are applied to test functions, and the outcomes discussed.

7.1 Mathematical test functions

This section contains the details of the four mathematical test functions chosen and optimized in the rest of this chapter.

7.1.1 Himmelblau function

The Himmelblau function is one of the well-known functions for testing optimizers. It has a 2-dimensional domain and a single output value and is given as:

$$f(x,y) = (x^2 + y - 11)^2 + (x + y^2 - 7)^2 \tag{7.1}$$

A contour plot of this equation is given in figure 7.1. This function reaches a global minimum at $f(3,2) = 0$ and has no other optima for $x, y \geq 0$. For the other quadrants, [*Wikipedia*, 2008*b*] locates the following optima:

$$f(-3.779310, -3.283186) = 0$$
$$f(-2.805118, 3.131312) = 0$$
$$f(3.584428, -1.848126) = 0$$

Because it is not possible to discriminate between these four function values, optimizers could end up in any of these four points. To prevent this, the requirement that $x, y \geq 0$ is used in the benchmarks in this chapter.

7.1.2 Griewank function

The Griewank function was first proposed in [*Griewank*, 1981] and is a popular function for benchmarking PSO algorithms. The dimensionality of the domain can be set to any value, but it does have a single output value. For the n-dimensional case, it is given as:

$$f(x) = \frac{1}{4000}\sum_{i=1}^{n}(x_i)^2 - \prod_{i=1}^{n}\cos\left(\frac{x_i}{\sqrt{i}}\right) + 1 \tag{7.2}$$

Two plots showing the behavior of the 1-dimensional version of this function are shown in figure 7.2. As can be seen clearly, it globally behaves like a parabola with a single optimum. Locally it has lots of optima due to the cosine term. The two combined give a global optimum $f(0) = 0$.

Note that the $\sqrt{i}$ in the denominator of this cosine term lowers the frequency of this disturbance function for higher values of i, which will make it easier to find the optimum function value for those terms. This can be seen when looking at the output of high-dimensional optimization runs, where more errors are found in the lower-dimensional coordinates. Figure 7.3 shows a histogram of the occurrence of non-zero (and non-optimal) coordinates as a function of i, and clearly demonstrates this behavior.

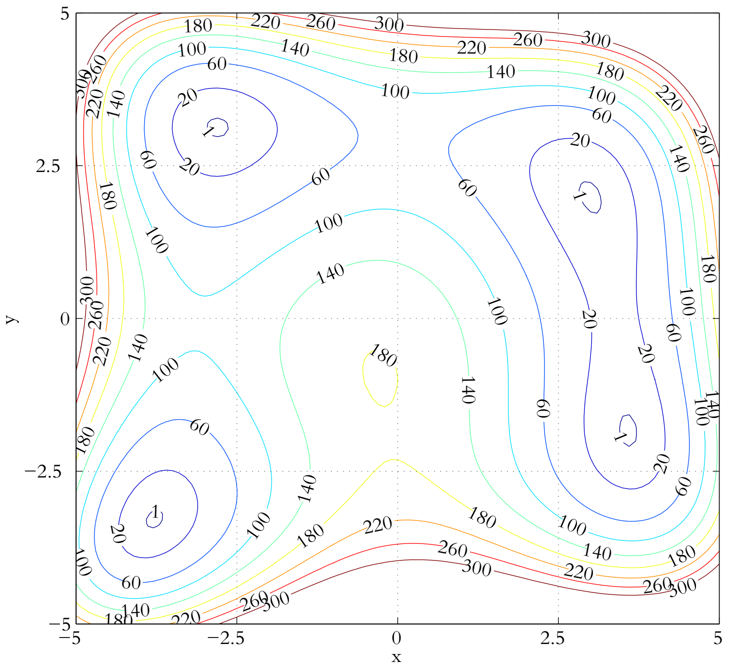
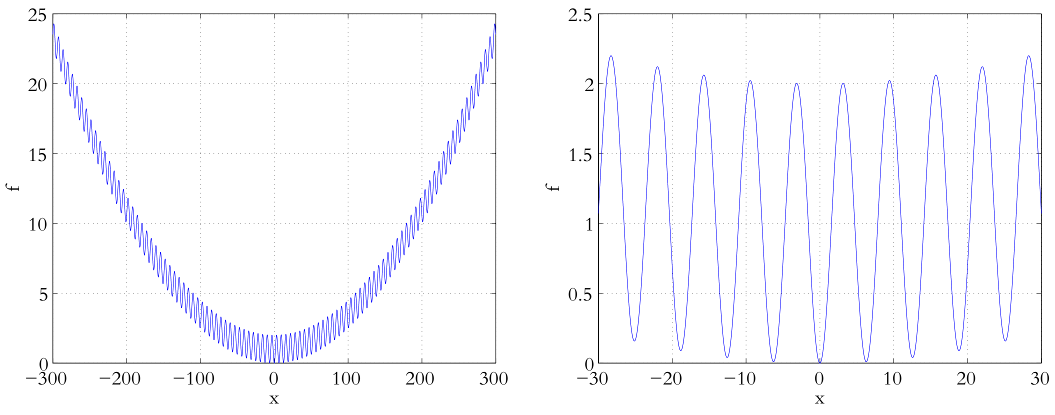


Figure 7.1 Contour plot of the Himmelblau function.



(a) Global view (b) Near optimum

Figure 7.2 Plots of Griewank's function in 1 dimension.

7.1.3 Schaffer's f_6 function

Schaffer's f_6 function was first given in [*Schaffer et al.*, 1989] and is a function that has an n-dimensional domain and a single valued result. It is given as:

$$f(x) = 1 + \left(\sum_{i=1}^{n} x_i^2\right)^{0.25} \left[\sin^2\left(50\left(\sum_{i=1}^{n} x_i^2\right)^{0.1}\right) + 1\right] \tag{7.3}$$

A plot of this function (with a 1-dimensional domain) is shown in figure 7.4. Just as the Griewank function, it has a global trend showing a global optimum, and lots of local optima. The global optimum is given by $f(0) = 1$.

7.1.4 Deb-Tan function

The Deb-Tan function is based on a function shown in [*Deb*, 1999], and was modified and made multidimensional in [*Tan et al.*, 2003]. The (modified) 1-dimensional version is given as:

$$f(x) = 2 - e^{-\left(\frac{x-0.1}{0.004}\right)^2} - 0.8e^{-\left(\frac{x-0.9}{0.4}\right)^2} \tag{7.4}$$

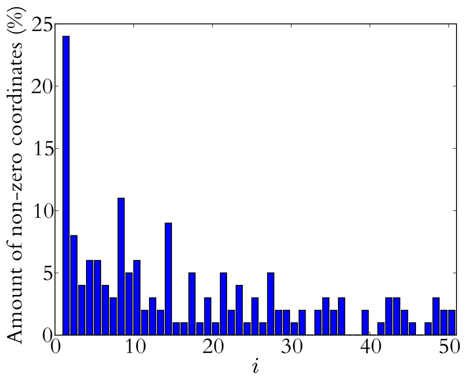


Figure 7.3 Histogram showing the amount of non-zero coordinates as a function of dimension i for 100 optimization runs of Griewank's function.

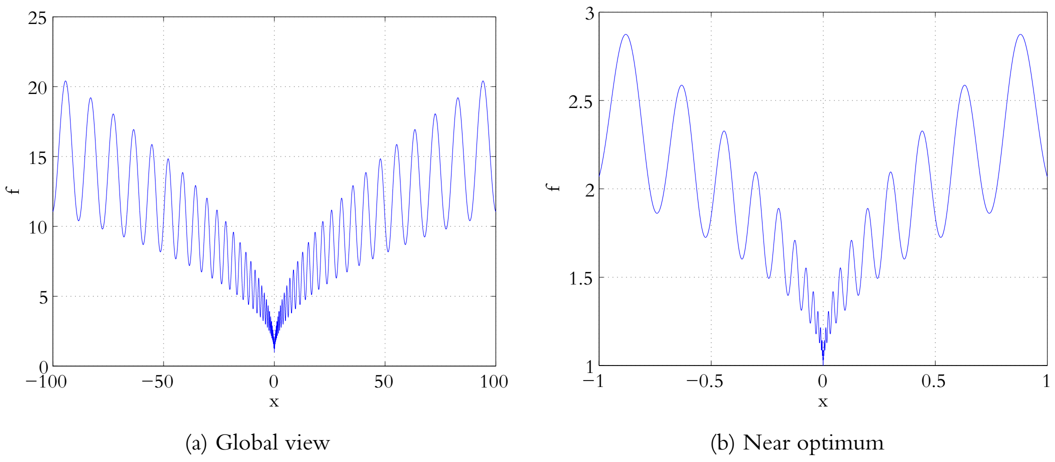


Figure 7.4 Plots of Schaffer's f_6 function in 1 dimension.

The multidimensional version can be constructed by taking the product of this function evaluated for the values in each dimension:

$$f^n(x) = \prod_{i=1}^{n} f(x_i) \tag{7.5}$$

A plot of the one-dimensional version is shown in figure 7.5(a). In that plot, it is clearly visible that for this domain the function has two optima: a global optimum at approximately $f(0.1) = 1$ and a local optimum at approximately $f(0.9) = 1.2$.

[*Tan et al.*, 2003] states that the optima are at exactly these locations, but that is an error. This can be shown by evaluating the derivative of the function in the supposed extremes, which is

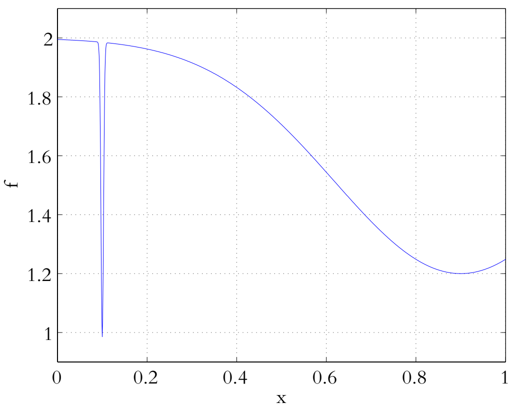
$$\frac{\mathrm{d}f}{\mathrm{d}x} = 2\frac{x-0.1}{0.004^2}e^{-\left(\frac{x-0.1}{0.004}\right)^2} + 1.6\frac{x-0.9}{0.4^2}e^{-\left(\frac{x-0.9}{0.4}\right)^2}$$

At the given points, one of these terms will be zero, but the other will not, thus the optima cannot lie there. However, an optimizer does not care about this, and should find the real optimum.

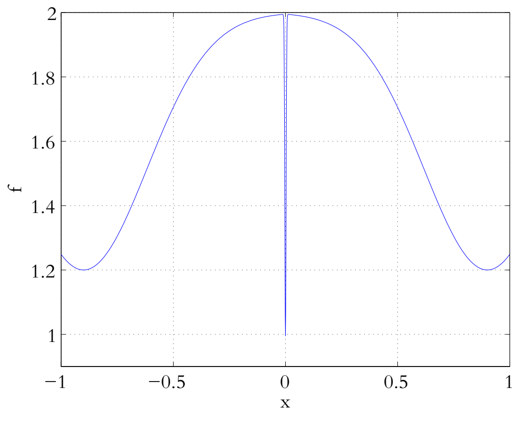
The big challenge when optimizing this function is that the optimizer needs to find the basin of attraction of the global optimum, which is very small and surrounded by high values.

However, when considering a solution which starts in the region left of the global minimum, the curve to the (global) optimum is a monotonic decreasing function, which in principle can be found by "sliding down the path". To prevent this, the following function is proposed, with boundaries $[-1, 1]$. This function is plotted in figure 7.5(b).

$$f(x) = 2 - e^{-\left(\frac{|x|}{0.004}\right)^2} - 0.8e^{-\left(\frac{|x|-0.9}{0.4}\right)^2} \tag{7.6}$$



(a) Original



(b) Modified

Figure 7.5 Plots of the Deb-Tan original and modified function in 1 dimension.

7.2 Verifying PSO for OPTIDUS

Because the control parameters of the PSOimplementation need to be tuned, a benchmark was done comparing the result of this implementation with other PSOimplementations and two DE schemes. Finally, a EA was also tested, but this failed to converge at the global optimum for 2 out of 4 problems.

7.2.1 Overview of the tested optimizer implementations

In this chapter, the following optimizer implementations where compared:

spso-07 is is the reference implementation that can be found at [*Clerc*, 2008*a*]. This version was tested without modifications. This is an implementation which has been specifically made available to benchmark against.

spso-07⋆ is a modified variant of this implementation. The only modification in this variant was that it has the parameter $p = 1$. This parameter controls the chance of an information link existing between two particles. By setting it to 1, all particles inform each other, which in practice means that all particles will be influenced by the global best individual.

optidus-pso mimics the behavior of SPSO-07⋆, except for the fact that it contains and a slightly modified formula for the determination of the speed.

optidus-ppo is the same as OPTIDUS-PSO, but adds a predator to the simulation to keep the swarm moving.

optidus-de1 is an implementation of the Tasoulis1 differential evolution scheme in OPTIDUS.

optidus-de6 is an implementation of the Tasoulis6 differential evolution scheme in OPTIDUS.

optidus-ea is an implementation of a EA in OPTIDUS. It uses immigration, crossover, mutation and stochastic universal sampling.

pso is the baseline PSO algorithm used in [*Silva et al.*, 2003]. No source code was available, and only one function from the set in section 7.1 was tested in that paper.

sappo is the improved algorithm presented in [*Silva et al.*, 2003]. It is a PPO algorithm which tries to adapt the control parameters in the optimization.

These tests were run on a MacBook with a CoreDuo processor running OSX 10.5.5, using GCC 4.0.1 (Apple Inc. build 5484) and GNU Fortran 4.4.0 20080909 (nightly build) as the compilers for the C and Fortran sources.

7.2.2 Himmelblau function results

No external Himmelblau results are known for PSO; therefore, the Himmelblau problem was added to the SPSO-07 program and OPTIDUS. The swarm size was set to 12, in accordance with (4.11) and because it's a simple problem, a maximum of 15 generations (180 function evaluations) are allowed. The results of this are shown in table 7.1 and figure 7.6.

OPTIDUS-DE1 is the clear winner, with the rest trailing behind.

Note that both OPTIDUS-PSO and OPTIDUS-PPO both

7.2.3 Griewank function results

In [*Silva et al.*, 2003], the Griewank function has been optimized for a 50-dimensional search space bounded by $-300 < x_i < 300 \quad \forall i \in \{1, ..., 50\}$. The reported number is the average value of the best fitness taken over 100 runs, with 5000 generations each. The swarm size was set to 30, giving a total of 150000 function evaluations. The results of these runs are summarized in table 7.2 and figure 7.7.

Implementation	$\bar{f}_{best}$	σ
Theoretical	0.000000	-
SPSO-07	0.036606	0.042298
SPSO-07⋆	0.023834	0.024082
OPTIDUS-PSO	0.010954	0.013596
OPTIDUS-PPO	0.011321	0.014809
OPTIDUS-DE1	0.002818	0.008319
OPTIDUS-DE6	0.069907	0.155670
OPTIDUS-EA	1.237943	1.453268

Table 7.1 Average best fitness values for the Himmelblau function for as found by various optimizers.

As expected, the SPSO-07⋆ and OPTIDUS-PSO implementations are quite close, but the SPSO-07 implementation scores much better. This seems to falsify the statement in [*Clerc*, 2008*a*] that the *information links topology* does not influence the outcome of the optimization. The OPTIDUS-PPO optimizer shows another interesting thing: it does not reach the optimum anytime, but all of its results are close to another.

The genetic algorithm implemented in OPTIDUS failed to converge for the used population size. Therefore, several larger population sizes were tested, also giving bad results.

The results given in [*Silva et al.*, 2003] are a lot better, so it might be worthwhile to investigate what caused that. Note that SAPPO also scores a lot worse than the normal PSO variant, just as in the OPTIDUS case. The Griewank function was the only function to exhibit this phenomenon in [*Silva et al.*, 2003].

Finally, OPTIDUS-DE6 managed to find a near-optimal solution in all 100 runs, doing even better than PSO (Silva), but OPTIDUS-DE1 is the worst performer for this function.

Implementation	$\bar{f}_{best}$	σ
Theoretical	0.000000	-
SPSO-07	0.013305	0.023714
SPSO-07⋆	0.146576	0.143635
PSO (Silva)	0.001115	n/a
SAPPO	0.004752	n/a
OPTIDUS-PSO	0.236207	0.413423
OPTIDUS-PPO	0.016518	0.019077
OPTIDUS-DE1	1.176597	1.091798
OPTIDUS-DE6	0.000000	0.000000
OPTIDUS-EA[$S = 30$]	142.371821	16.2646647996
OPTIDUS-EA[$S = 90$]	111.4192414	15.172419042
OPTIDUS-EA[$S = 300$]	89.3277521	11.3099994644
OPTIDUS-EA[$S = 3000$]	111.2387651	9.60405051626

Table 7.2 Average and standard deviation of the best fitness values for the 50-dimensional Griewank function as found by various optimizers.

7.2.4 Schaffer's f_6 function results

Just as with the Himmelblau function, no public test results were found for Schaffer's f_6 function, so they were added to SPSO-07 and OPTIDUS. For this test, a 5-dimensional variant was chosen, and according to (4.11), this problem should then be run with a swarm size of 14. Allowing a maximum of 1500 generations gives the results of table 7.3 and figure 7.8.

The results of SPSO-07 and OPTIDUS-DE6 stand out, with both good average values and a low standard deviation, with OPTIDUS-PPO in third place. SPSO-07★ and OPTIDUS-PSO have comparable results, and finally OPTIDUS-DE1 performed disappointing.

Implementation	$\bar{f}_{best}$	σ
Theoretical optimum	1.000000	-
SPSO-07	0.001496	0.003411
SPSO-07★	0.285879	0.644234
OPTIDUS-PSO	0.276903	0.606626
OPTIDUS-PPO	0.089222	0.442588
OPTIDUS-DE1	1.040333	1.691922
OPTIDUS-DE6	0.003274	0.003787
OPTIDUS-EA	3.317412	0.898857

Table 7.3 Average and standard deviation of the difference between the theoretical optimum and the best fitness values for Schaffer's f_6 function in 5 dimensions as found by various optimizers.

7.2.5 Deb-Tan function results

No public benchmarks were found for the Deb-Tan function, so it was added to SPSO-07 and OPTIDUS. In accordance with (4.11), the used swarm size was 13. Each run was allowed to run for 10000 generations or 130000 function evaluations.

Running these functions yielded the results summarized in Table 7.4 and Figure 7.9: Only the OPTIDUS-PPO implementation was able to reach the global optimum, and did so in all runs. The rest of the algorithms got stuck in the local optimum near $x = 0.9$ (in 1 or more dimensions). This is reflected by the big peak near $f = 0.8$ in the histograms. The fact that OPTIDUS-PPO did not get stuck can be attributed to the prey, which was introduced to reduce the occurrence of premature convergence.

As expected, the results of the reference implementation and OPTIDUS-PSO are comparable.

Implementation	$\bar{f}_{best}$	σ
Theoretical optimum	0.956683	-
SPSO-07	0.644944	0.050996
SPSO-07★	0.640366	0.065898
OPTIDUS-PSO	0.752771	0.073407
OPTIDUS-PPO	0.000000	0.000000
OPTIDUS-DE1	0.763800	0.086564
OPTIDUS-DE6	0.746423	0.087297
OPTIDUS-EA	0.499936	0.187404

Table 7.4 Average and standard deviation of the difference between the theoretical optimum and the best fitness values for the 3-dimensional Deb-Tan function as found by various implementations.

7.2.6 Conclusions for the verification

Looking at the results from these test-functions does not give a clear winner, as the best algorithm depends on the problem:

- OPTIDUS-PSO does not perform outstanding, but is in second place for the Himmelblau function. It also does not perform very badly.

- OPTIDUS-PPO is the clear winner in the Deb-Tan function, and also scores a second place in the Griewank optimization.
- OPTIDUS-DE1 is the winner for the Himmelblau function, but scores really badly for the Griewank and Schaffer's f_6 function.
- OPTIDUS-DE6 on the other hand did win the Griewank and had a shared victory on Schaffer's f_6 function with SPSO-07, and performs satisfactory for the other functions.
- SPSO-07 had a shared victory on Schaffer's f_6 function with OPTIDUS-DE6, and quite good results overall.
- SPSO-07* as expected had results similar to OPTIDUS-PSO.
- SPSO-07 and SPSO-07* do have different results, so maybe the use of a better form of *information link topology* might improve the results of the PSO even further.

Because no clear winner arises, all of DE1, DE6, PSO and PPO will be used to optimize the Solar Sailing Mission.

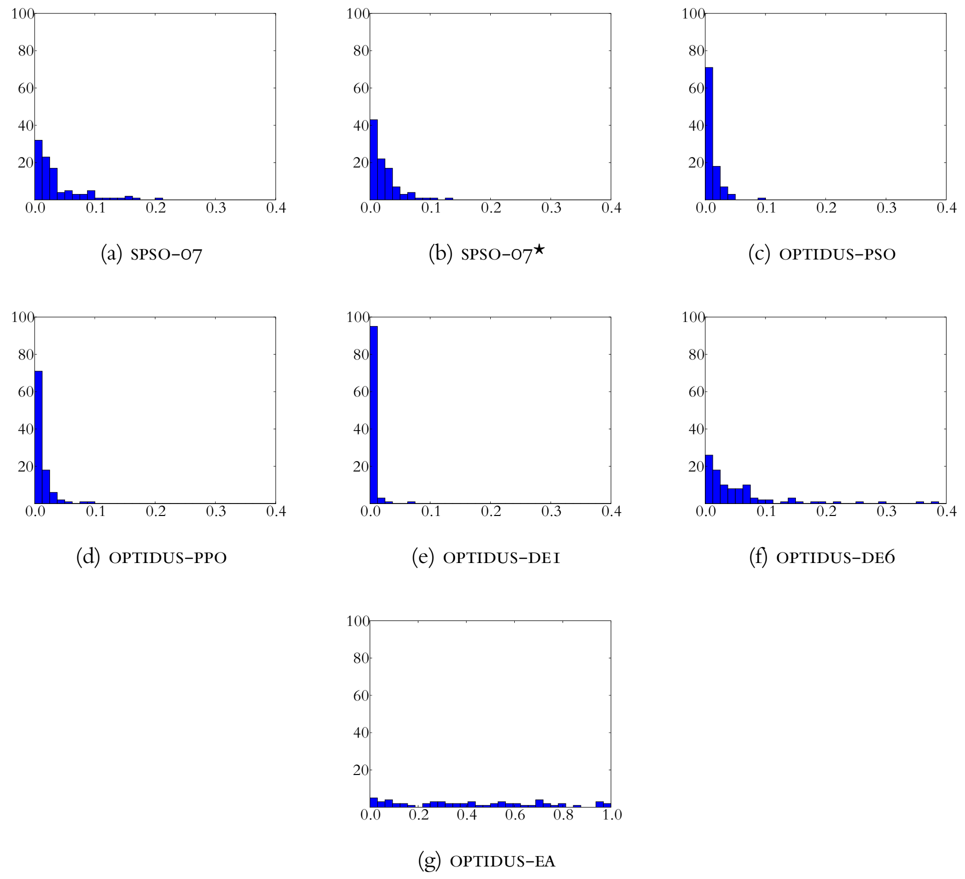


Figure 7.6 Histograms of the best fitness values of the Himmelblau function found by the various implementations.

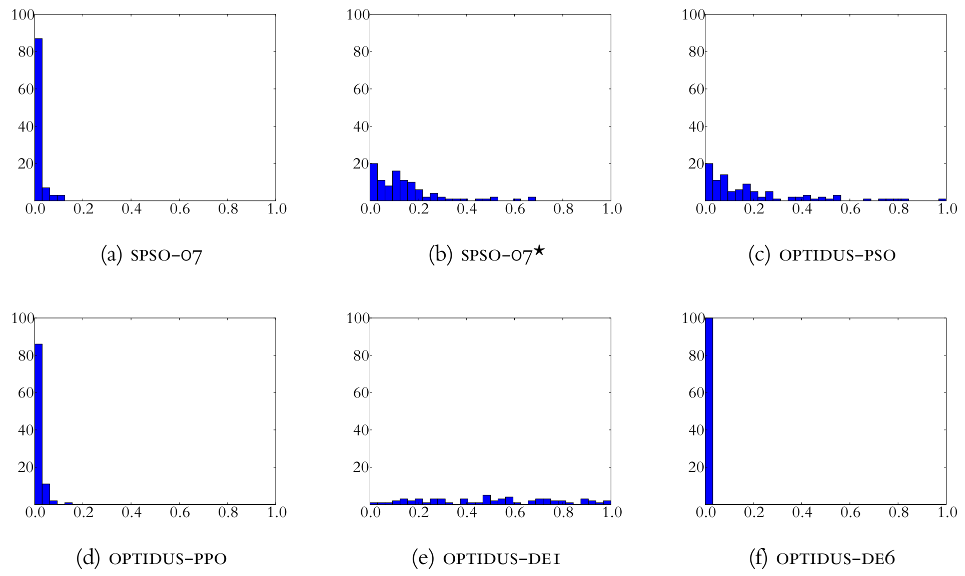
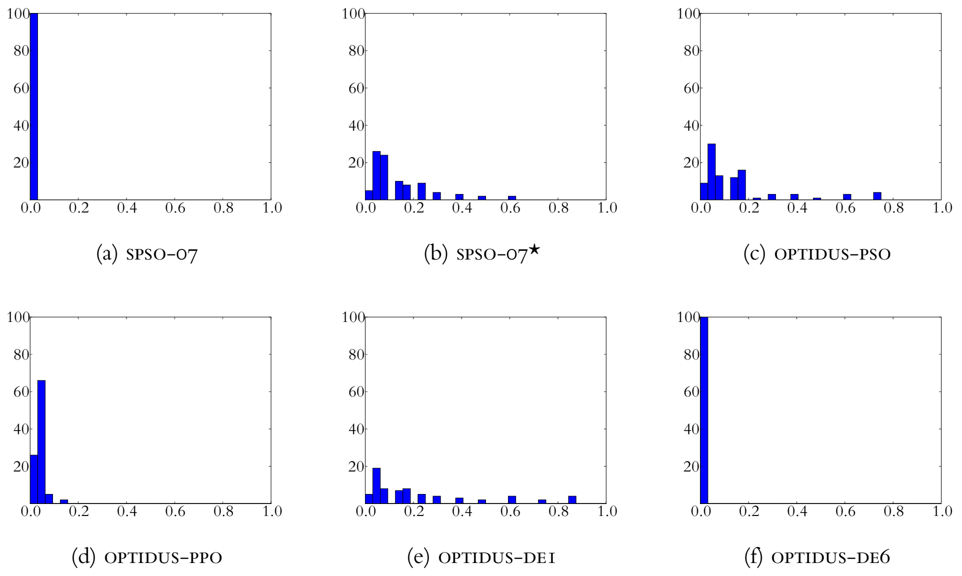


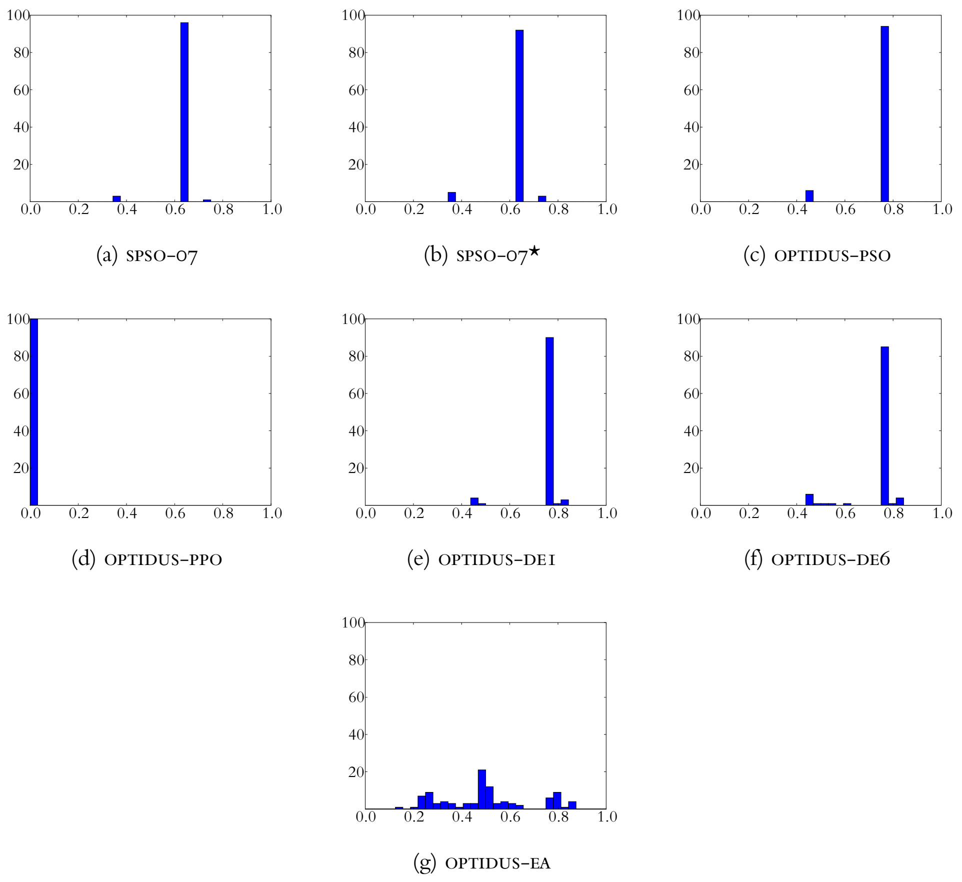
Figure 7.7 Histograms of the best fitness values of the 50-dimensional Griewank function found by the various implementations.



(a) SPSO-07 (b) SPSO-07⋆ (c) OPTIDUS-PSO

(d) OPTIDUS-PPO (e) OPTIDUS-DE1 (f) OPTIDUS-DE6

Figure 7.8 Histograms of the the difference between the theoretical optimum and best fitness values of Schaffer's f_6 function in 5 dimensions found by the various implementations.



(a) SPSO-07 (b) SPSO-07⋆ (c) OPTIDUS-PSO

(d) OPTIDUS-PPO (e) OPTIDUS-DE1 (f) OPTIDUS-DE6

(g) OPTIDUS-EA

Figure 7.9 Histograms of the difference between the theoretical optimum and the best fitness values of the 3-dimensional Deb-Tan function found by the various implementations.

Chapter *8*

Improving the Solar Polar Sail Mission Optimization

In [*Garot*, 2006] the optimal trajectory for a solar polar sail mission was determined using an old version of OPTIDUS. This version of OPTIDUS made use of a GAVaPS, which gave good results, but because it was expected that PSO and DE will perform better than regular GAs, it was proposed to redo the optimization of this problem using those algorithms. This chapter contains a description of the problem, and a comparison between the original results found using OPTIDUS, and the results found using differential evolution and PSO/PPO. Next, random sampling using both pseudo-random numbers and Sobol sequences were applied to improve the best solution found. Finally, Powell's quadratically convergent method was applied as a local optimizer.

8.1 Introduction to solar sailing

Solar sailing is a propulsion method which has some similarities to conventional sailing as done on the Earth. Just as with a ship, a spacecraft (making use of solar sailing) has a big sail attached to it, an example of which is shown in figure 8.1. Unlike a sail on a ship, it is not powered by aerodynamic forces from the wind blowing by, but instead works by reflecting photons colliding on its surface. Because photons have momentum, the change in their direction causes a small force to be exerted on the spacecraft. This force can be used to propel the spacecraft. The big advantage of this method of propulsion is that it does not consume fuel aboard the spacecraft, and can therefore be used for longer periods, as the spacecraft will not run out of fuel.

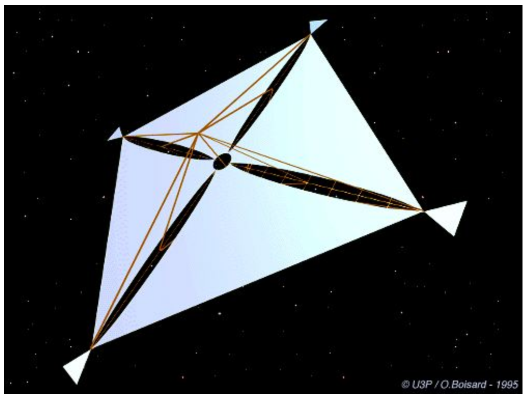


Figure 8.1 A proposed solar sail. (Source: `http://www.u3p.net`)

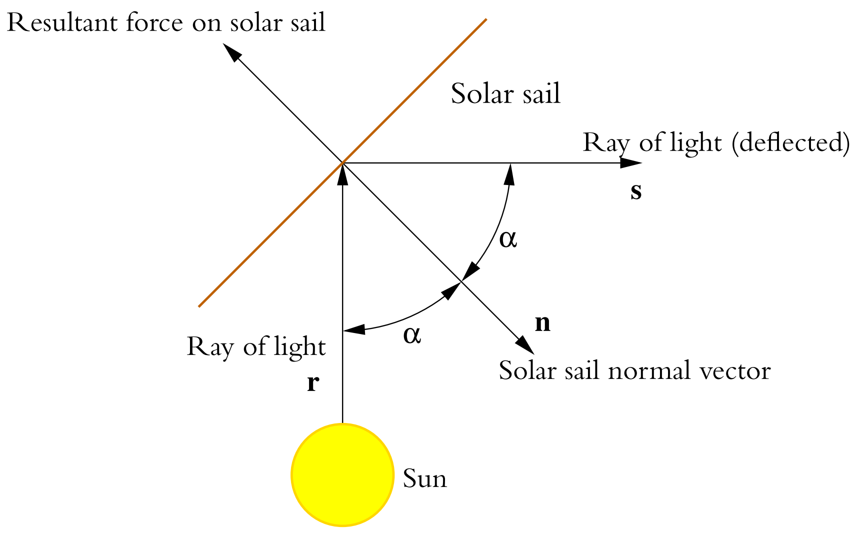


Figure 8.2 Forces acting on a solar sail. (2-dimensional and not to scale)

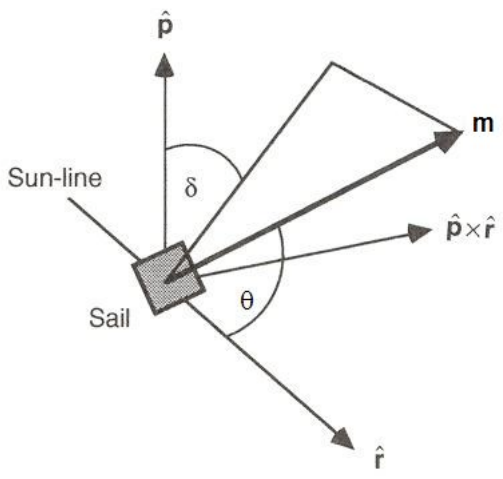


Figure 8.3 Solar sail cone and clock angle in the local sail reference frame. Source: [*McInnes*, 1999]

Just as with regular sailing, the magnitude and the direction of the force on the sail depend on the orientation of the sail. The basic geometry of the forces acting on a solar sail in 2D is depicted in figure 8.2. The pitch angle α is the angle between the incident ray of sunlight $\mathbf{r}$ and the vector perpendicular to the sail, $\mathbf{n}$. Figure 8.3 shows the definition of the clock angle δ: it is the angle between the vectors $\mathbf{p}$ and $\mathbf{n}$. This vector $\mathbf{p}$ is equal to the specific angular momentum of the spacecraft, and is given by $\mathbf{r} \times \mathbf{V}$. The vector θ is the cone angle, and determines the direction of the force vector $\mathbf{m}$. Note that there is a difference between $\mathbf{m}$ and $\mathbf{n}$, because the sail is not 100% efficient, because not all of the photons are reflected.

The vectors $\mathbf{m}$ and $\mathbf{n}$ can be expressed using the previous vectors and the angles:

$$\mathbf{m} = \cos\theta\ \hat{\mathbf{r}} + \sin\theta \sin\delta\ \hat{\mathbf{p}} \times \hat{\mathbf{r}} + \sin\theta\cos\delta\ \hat{\mathbf{p}} \tag{8.1}$$

$$\mathbf{n} = \cos\alpha\ \hat{\mathbf{r}} + \sin\alpha \sin\delta\ \hat{\mathbf{p}} \times \hat{\mathbf{r}} + \sin\alpha\cos\delta\ \hat{\mathbf{p}} \tag{8.2}$$

8.2 Model of the Solar Polar Sail Mission

The goal of the solar polar sail mission is to get a spacecraft into a polar orbit around the Sun, using only solar sailing for propulsion. This mission can be split into 2 distinct phases, the geocentric phase and the heliocentric phase. These are modelled separately.

8.2.1 Geocentric Phase: Departure from Earth

After launch, the spacecraft is put into a geostationary transfer orbit (GTO) with a perigee height of 185 km, an apogee height of 35885 km and zero inclination. From this orbit, the spacecraft spirals the Earth while accelerating towards escape velocity. An example of a spiraling trajectory is given in figure 8.6(a) on page 82.

Each orbit in this phase can be split into three parts, as shown in figure 8.4. These three parts are the acceleration part, feathering part, and the drag part, which occur all three during each revolution.

This separation is made to ease the control and modelling of the spacecraft. The equations of motion in this phase are given by:

$$\frac{\mathrm{d}^2\mathbf{r}}{\mathrm{d}t^2} = -\mu_E \frac{\mathbf{r}}{r^3} + \frac{f}{m}\left[\boldsymbol{\Phi}\cdot\mathbf{m}\right] + \mathbf{a}_{drag} + \mathbf{a}_{Sun} + \mathbf{a}_{Moon} \tag{8.3}$$

In this (and the following equations) $\boldsymbol{\Phi}$ is a 3x3 transformation matrix given by:

$$\boldsymbol{\Phi} = \begin{bmatrix}\hat{\mathbf{r}} & [\hat{\mathbf{r}}\times\hat{\mathbf{V}}]\times\hat{\mathbf{r}} & \hat{\mathbf{r}}\times\hat{\mathbf{V}}\end{bmatrix} \tag{8.4}$$

in which $\hat{\mathbf{r}}$ is the normalized Sun line, and $\hat{\mathbf{V}}$ the normalized velocity vector of the spacecraft.

This transformation matrix is used to transform the vectors $\mathbf{m}$ and $\mathbf{n}$ (which are in a coordinate system attached to the orbit of the satellite) into a heliocentric coordinate system.

f is the value of the solar radiation pressure force given by

$$f = 2PA(\hat{\mathbf{r}}\cdot\hat{\mathbf{n}})^2 \tag{8.5}$$

where $\mathbf{n}$ is the unit vector normal to the sail given below, and $\hat{\mathbf{r}}$ is the normalized Sun line. A is the surface area of the sail, and P is the local solar radiation pressure given by

$$P = \frac{W}{c} = \frac{W_E\left(\frac{r_E}{r}\right)^2}{c} \tag{8.6}$$

in which W_E is the solar flux at the Earth, $W_E = 1367.6\ \mathrm{W/m^2}$, and r_E is the Sun-Earth distance (equal to 1 AU.)

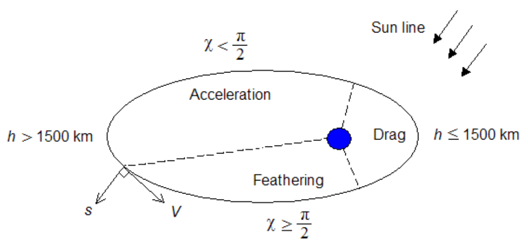


Figure 8.4 The sailcraft's orbit in the Earth escape phase is divided into an acceleration, feathering and drag part. Source: [*Garot*, 2006]

The accelerations caused by the gravitational pull from the Sun and the Moon (relative to the motion of the Earth) are given by:

$$\mathbf{a}_{Sun} = GM_S \left(\frac{\mathbf{r}_{sail-Sun}}{r^3_{sail-Sun}} - \frac{\mathbf{r}_{Sun}}{r^3_{Sun}} \right) \tag{8.7}$$

$$\mathbf{a}_{Moon} = GM_{Moon} \left(\frac{\mathbf{r}_{sail-Moon}}{r^3_{sail-Moon}} - \frac{\mathbf{r}_{Moon}}{r^3_{Moon}} \right) \tag{8.8}$$

Feathering phase

When the angle between the velocity vector and the Sun line, χ, is above $90°$, the solar pressure can only decelerate the spacecraft. For that reason, during this phase the sail is positioned such that it is parallel to the Sun angle and no force is exerted on the sail. Because the altitude is high enough, no drag force is present, so only the gravitational pulls by the Sun, Earth and Moon need to be taken into account.

Acceleration phase

When the angle between the velocity vector and the Sun line, χ, is below $90°$, the solar pressure can accelerate the spacecraft. This can be achieved by setting the clock angle $\delta = 90°$, and the pitch angle α can be chosen by the user: $0° \leq \alpha \leq 90°$. According to [*Garot*, 2006] splitting this acceleration phase into 6 subsections provides a good way to control the pitch angle. This division is shown in figure 8.5. This leads to a total of six pitch angles to be optimized, but this can be reduced to three by imposing a symmetric set of pitch-angles: $\alpha_1 = \alpha_6, \alpha_2 = \alpha_5$ and $\alpha_3 = \alpha_4$.

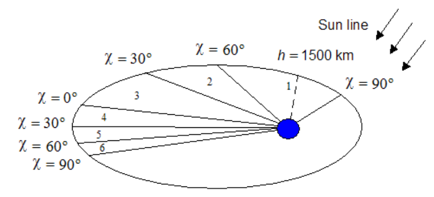


Figure 8.5 The acceleration phase is divided into six sections. Source: [*Garot*, 2006]

Drag phase

When the altitude is lower than 1500 km, atmospheric drag starts to influence the flight.. In that case, the sail is positioned such that it is parallel to the velocity, to minimize drag. The acceleration due to drag can be modelled as

$$\mathbf{a}_{drag} = \frac{N}{m} [\Phi \cdot \mathbf{n}] \tag{8.9}$$

In [*Garot*, 2006] the following expressions are derived for this force:

$$N = 2qA \sin^2(\alpha_{drag}) \tag{8.10}$$

in which A is the surface of the sail, α_{drag} the angle of attack, and $q = \frac{1}{2}\rho V^2$ is the dynamic pressure. For the atmospheric density ρ as a function of altitude h, the following relation is assumed:[1]

$$\rho(h) = 35 \left(243352 h^{-7.2305} + 4537152 h^{-5.6305(1.49 - \frac{h}{11000})} \right) \tag{8.11}$$

[1]This relation is a fit made in [*Foekema*, 2004] based on data given in [*Wertz and Larson*, 1999]

In the ideal case, α_{drag} would be zero, and no drag would be present. However, in reality some drag will occur. To add some drag to the model, the simulated angle of attack is set to 3 degrees.

8.2.2 Heliocentric phase

After the escape velocity has been reached, the heliocentric phase starts. During this phase, only the gravitational pull of the three inner planets of the solar system and the Sun and the force from the solar sail need to be taken into account so the equations of motion are given by:

$$\frac{\mathrm{d}^2\mathbf{r}}{\mathrm{d}t^2} = -\mu_S\frac{\mathbf{r}}{r^3} + \frac{f}{m}[\Phi\cdot\mathbf{m}] + \mathbf{a}_{Mercury} + \mathbf{a}_{Venus} + \mathbf{a}_{Earth} \tag{8.12}$$

with

$$\mathbf{a}_{Mercury} = GM_{Mercury}\left(\frac{\mathbf{r}_{sail-Mercury}}{r^3_{sail-Mercury}} - \frac{\mathbf{r}_{Mercury}}{r^3_{Mercury}}\right) \tag{8.13}$$

$$\mathbf{a}_{Venus} = GM_{Venus}\left(\frac{\mathbf{r}_{sail-Venus}}{r^3_{sail-Venus}} - \frac{\mathbf{r}_{Venus}}{r^3_{Venus}}\right) \tag{8.14}$$

$$\mathbf{a}_{Earth} = GM_{Earth}\left(\frac{\mathbf{r}_{sail-Earth}}{r^3_{sail-Earth}} - \frac{\mathbf{r}_{Earth}}{r^3_{Earth}}\right) \tag{8.15}$$

This phase is split into four phases, which will be given below. A complete trajectory comprising of these four phases is given in figure 8.6(b).

Inward spiral

After the spacecraft has left the Earth's sphere of influence, it needs to decrease its distance to the Sun. This is done by maximizing the component of the solar radiation pressure force in the negative direction of the velocity vector. The sail clock angle is equal to the velocity vector clock angle of 90 degrees and only the pitch angle needs to be optimized. For inward spiraling the pitch angle needs to be negative: $-90° \leq \alpha_S(1) < 0°$. This phase ends when the distance R_1 is reached. R_1 needs to be between 0.3AU and 0.7AU.

Orbit Circularization

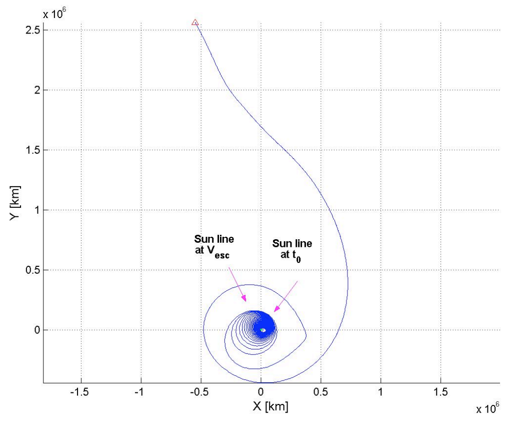
After reaching the distance R_1, the spacecraft continues to spiral towards the Sun, but with a different pitch angle, which is allowed to vary between $-90° \leq \alpha_S(2) \leq 90°$. The clock angle remains the same. This phase ends when the distance R_2 to the Sun is reached. R_2 needs to be between 0.26AU and $\min(R_1, 0.4AU)$.

Even though the distance to the Sun will be increased again in the final phase to 0.4AU, the spacecraft is forced to get closer to the Sun than that, as the orbit cranking phase takes much less time if the distance to the Sun is lower. [*Garot*, 2006] did an optimization in which cranking was done at 0.4AU, and flight time was almost one year longer.

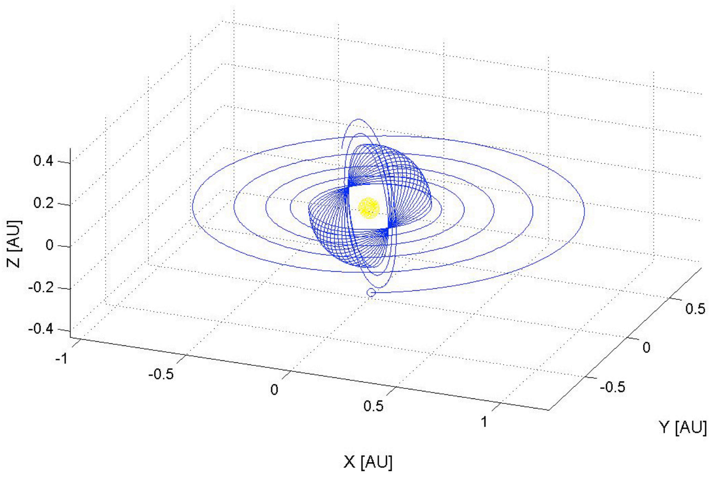
Orbit Cranking

After reaching the distance R_2, the cranking phase starts. During this phase, the sail is rotated over 180° every half orbit, to make sure that the force vector points above or below the orbital plane. This gives the following rule for the clock angle:

$$\delta = \begin{cases} 0° & \text{if } \dot{z} \geq 0 \\ 180° & \text{if } \dot{z} < 0 \end{cases} \tag{8.16}$$



(a) Geocentric



(b) Heliocentric

Figure 8.6 Trajectories for the best found solution for the solar polar sail mission in [*Garot*, 2006].

The pitch angle can be chosen freely between $0^\circ < \alpha_S(3) \leq 90^\circ$. This phase ends when the inclination reaches 90°.

Outward spiral

The last phase is used to increase the distance from the Sun again. This is done to decrease the heat-load on the spacecraft, so that it may operate longer.

Just as with the inward spiral and circularization phases, the clock angle is fixed to 90°, but now the pitch angle needs to be positive: $0^\circ \leq \alpha_S(4) \leq 90^\circ$. When the spacecraft reaches a distance of 0.4AU from the Sun, this phase ends.

8.3 Optimizing the Geocentric Phase

The goal of the first phase of the mission is to let the spacecraft reach escape velocity as quickly as possible. This trajectory is controlled by 5 variables: first of all, the launch date T_{launch}, the initial GTO-orientation ω and the pitch angles $\alpha_E(1)$, $\alpha_E(2)$ and $\alpha_E(3)$. The objective function outputs the time of flight until escape velocity is reached. The limits for these parameters are listed in table 8.1.

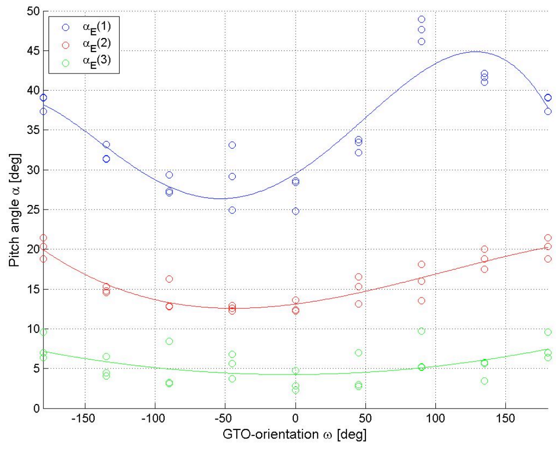


Figure 8.7 Polynomials through the pitch angles that correspond to the shortest flight times for the geocentric phase. Source: [*Garot*, 2006]

8.3.1 Original results from [*Garot*, 2006]

In [*Garot*, 2006], a model was created that allows one to calculate pitch angles α_E as a function of the GTO-orientation ω. This was done by running several optimizations for 8 values of ω. These sets of pitch angles where next fitted to three polynomials (of 5th, 4th and 3rd order for $\alpha_E(1)$, $\alpha_E(2)$, $\alpha_E(3)$ respectively), to get a continuous function. Figure 8.7 shows a plot of the pitch angles determined in those optimizations and the fitted functions. According to [*Garot*, 2006], better trajectories can be found by using other Earth escape pitch angles. For that reason, the first phase has been re-optimized.

8.3.2 Selecting an optimizer for the first phase

To determine the best optimizer to use for this mission phase, a set of preliminary runs was done using OPTIDUS-PSO, OPTIDUS-PPO, OPTIDUS-DE1 and OPTIDUS-DE2. These results, and the original results found by [*Garot*, 2006] are listed in table 8.7. Based on these results, OPTIDUS-PSO was chosen as the optimizer to use for the rest of this mission phase.

8.3.3 Improved results using optidus-pso

Using OPTIDUS-PSO, the same procedure was repeated, but for a much larger number of GTO-orientation values, 72 (spaced 5 degrees apart), and 10 separate runs per GTO-orientation.[2] Also another function was chosen to fit the data; because a GTO-orientation is an angle, a periodical function should be used to fit the data against. It was chosen to use

[2]This is a 30-fold increase of the number of optimizations, but the number of evaluations per run is on average a factor 6 less with PSO, when compared to the original runs, so only 5 times as many function evaluations were needed.

Symbol	[unit]	Lower limit	Upper limit
$\alpha_E(1)$	[°]	0	90
$\alpha_E(2)$	[°]	0	90
$\alpha_E(3)$	[°]	0	90
ω	[°]	0	360

Table 8.1 Limits of the optimization parameters for the geocentric phase

a 4th order sum of sines:

$$\alpha_E(i)(\omega) = a_{i,0} + \sum_{n=1}^{4} a_{i,n} \sin(n\omega + b_{i,n}) \tag{8.17}$$

The average pitch angles from the optimization runs were fitted to this function using the fit functionality of [*Gnuplot*, 2009], which does an iterative fit to reduce the residuals.

The α_E values found by the optimization runs, the mean values, standard deviation and the sine fit are shown in figure 8.8. The coefficients which correspond to the fit are listed in table 8.2.

Using the coefficients and the sine-function, pitch angles were calculated for the whole range of GTO-orientation angles ω. Next, the geocentric phase was simulated using these sets of pitch angles, and the resulting time of flight was determined. These values have been plotted in figure 8.9. In that figure, the original times of flight as found in [*Garot*, 2006] are also plotted.

Figure 8.9 shows that the pitch angles from the sinusoidal fit give better times of flight than the original polynomial fit; the biggest improvement was found at $\omega = 135^\circ$ where the time of flight decreased by 14.2% from 355.8 to 305.3 days, and the smallest improvement was found for $\omega = 0^\circ$ where the time of flight decreased by 3.8% from 278.4 to 267.8 days.

8.4 Optimizing the complete mission

The next step in the optimization process is to do a run of the complete mission. Table 8.3 lists the parameters which were optimized, and the limits used.

Because one of the goals of this thesis is to find out what the best optimizer is, and because those optimizers still need tuning, a batch of runs was done using 5 different optimizers and 7 population sizes. The population sizes initially used were $20 + 32i$, for $i \in \{0, 1, 2, 3\}$, but as the best and average fitness values as a function of population size still showed an upward trend, another set of optimizations was done for $i \in \{4, 5, 6\}$.

For each optimizer and population size, 5 runs were executed using 5 seed values for the random generator. This means $5 * 5 * 7 = 175$ runs were executed, resulting in a total of 3,288,206 function evaluations. The total (wall)time needed to run these two batches was 4.5 days (on a machine with two Intel Xeon (X5355) processors having 4 cores each, running at 2.66GHz).[3]

[3] Wall time is the amount of time which has elapsed on a clock on the wall – as opposed to cpu-time, which is the sum of the time spent by each processor core. For example, on an 8-core system, 8 seconds of cpu-time are available every wall-time second.

Coefficient	$\alpha_E(1)$	$\alpha_E(2)$	$\alpha_E(3)$
a_0	35.1325	15.4344	5.58173
a_1	9.49108	2.84364	0.795391
a_2	-4.28024	-0.573209	0.559949
a_3	-1.97354	-0.105089	-0.238481
a_4	0.97202	0.26879	-0.200426
b_1	-30.41	-44.55	-49.59
b_2	54.26	48.66	130.54
b_3	-36.30	127.64	89.82
b_4	93.59	139.02	68.04

Table 8.2 Values of the coefficients of the sine-fit of (8.17). All coefficients are in degrees.

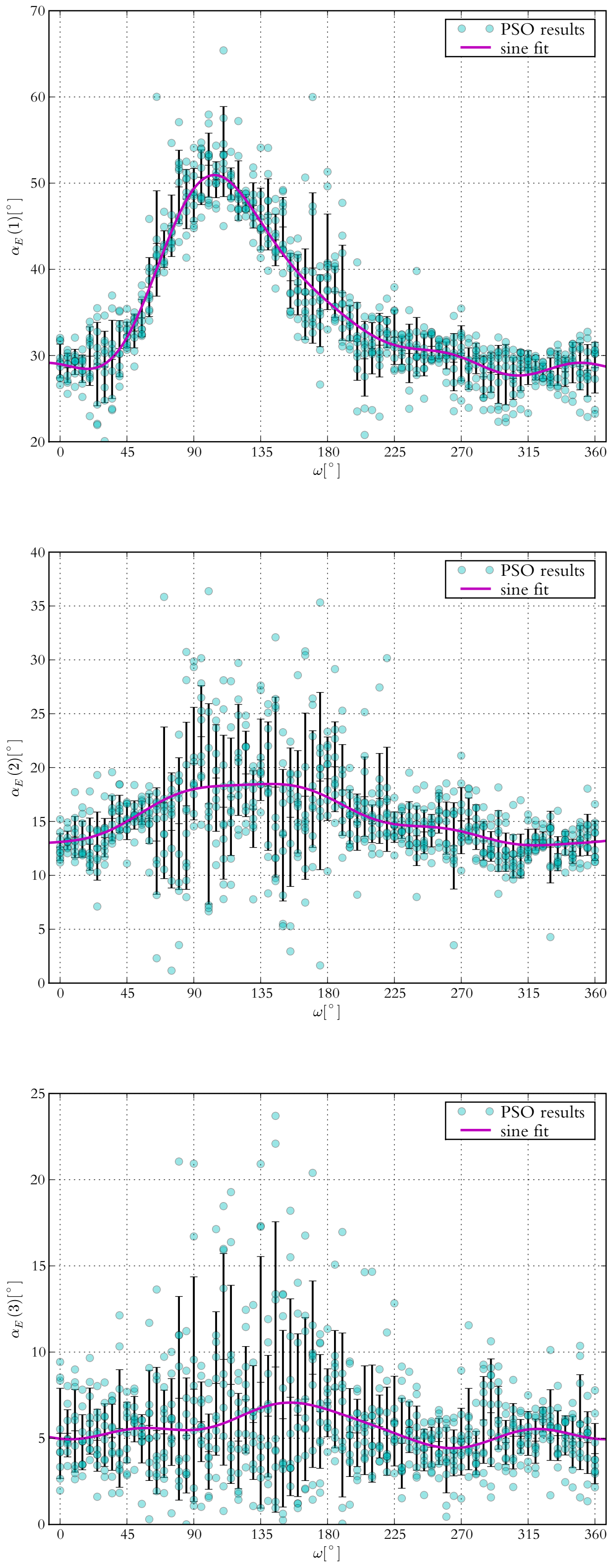


Figure 8.8 Optimal pitch angles α_E as a function of the GTO-orientation ω. The circles are the outcomes of optimization runs, the black bars denote their standard deviation and average, and the line is a the fit to the sinusoidal function (8.17).

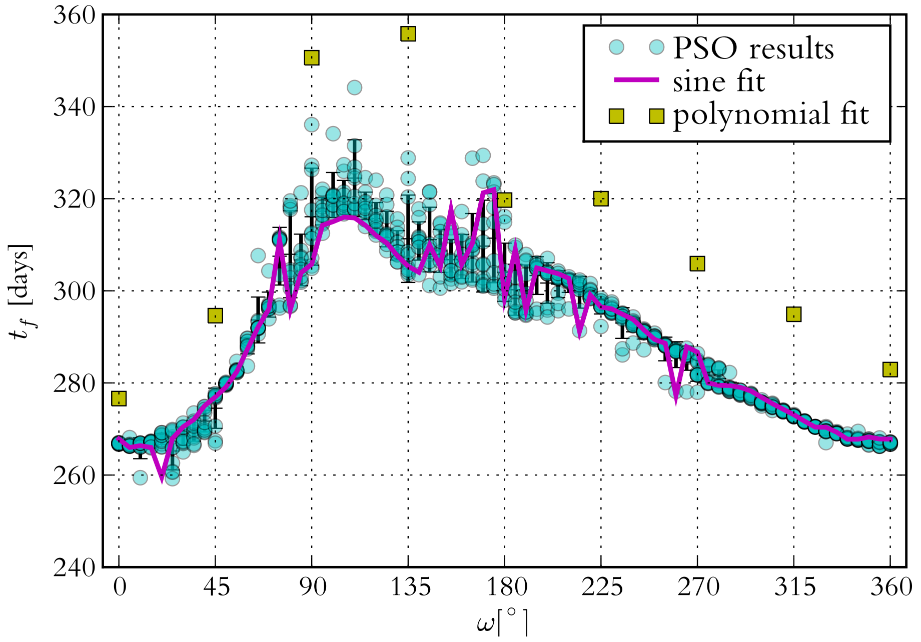


Figure 8.9 Comparison of the time of flight for the historical results (polynomial fit), free optimization (PSO) and optimization using the sine fit for the pitch angles (sine fit).

8.4.1 Settings for the optimization runs

The five optimizers used for these runs were the OPTIDUS implementations of DE1, DE6, PSO, PPO and a EA. For these optimizers, the parameters were set as follows:

DE1 and DE6 both used a static mutation constant F = 0.7, which is the only tunable parameter besides the population size. (See section 5.2 for how this constant is used.)

PSO and PPO both used the constricted formulation of [*Clerc and Kennedy*, 2002], with $\chi = 3.9$ and $\phi = 1$. No bounds were set on the velocity, but the position was limited using the method of section 4.5 to the bounds laid out in [*Garot*, 2006].

For the predator-prey optimizer (see section 4.4), c_4 from (4.13) (also known as f_pred) was set to 0.5, and the fear probability p_f or p_fear was set to 0.01.

EA is a evolutionary algorithm which makes use of a roulette selection based crossover operator, normal mutation, immigration and the stochastic universal sampling to weed out bad individuals during each generation. Additionally, elitism is applied to keep the best individual inside the population.

Two stop criteria were used: first of all, there was a limit on the total number of evaluations of 30000 per run. Secondly, if a population's best individual did not change over 90 generations, the optimization was stopped.

The outcome of the objective function for this optimization problem is a total mission cost estimate, which must be minimized.

8.4.2 Comparison of the results

The results of these runs are shown in figures 8.10 and 8.11, which show the average and best found fitness for each of the 5 runs which were done for each combination of optimizer and population size. From these figures, it is clear that for this problem, the differential evolution algorithms give the best average fitness for all population sizes, with the DE1 version just a bit better than the DE6 version. For small population sizes, PSO and PPO (on average) perform better than EA, but for large population sizes, the results are in the same league.

Looking at figure 8.11, the best fitness still seems to be rising with the population size. Due to lack of time, no bigger populations have been simulated; future research might check if this trend continues for even bigger populations.

The PSO, PPO and EA results are worse than the 3 results found by [*Garot*, 2006], which were 231.7, 228.3 and 232.8. For these results, 14266, 10047 and 14518 evaluations where needed respectively.

If the best fitness values are considered, the differential evolution algorithm again gives the best results, with the absolute best results found by DE1 for the population sizes 116 and 212, with fitness scores 220.91 and 220.92 respectively, which used 30048 and 30110 evaluations respectively.

The parameters found in those two runs are listed in table 8.3. One thing that raises attention is the difference in values for ω_0; the value of 115.9° found by [*Garot*, 2006] is very bad, according to figure 8.9 – this choice makes the geocentric phase more than 80 days longer than in the trajectories found using DE1. This accounts for the difference of almost half a year between the launch date found in [*Garot*, 2006] and the one from DE.[4]

Finally, a plot using Pareto front ranking was generated from the outcomes of these runs. This was possible by considering the process of optimizing optimization as a dual-objective problem, with the objective value of the solar polar sail mission as the first objective, and the amount of function evaluations as the other. Figure 8.12 shows a plot of the number of evaluations versus the fitness value reached for all optimization runs done, and in this plot, the Pareto fronts have also been drawn.

What is interesting is that in this plot, the first front contains four out of five optimization methods, so the Pareto fronts by themselves cannot tell which optimizer is the best. However, when looking at the reached fitness values, only the DE1 algorithm has reached fitness values above −225.

8.5 Local optimization using random sampling

Using the results from the previous section, a test was conducted to see if it is possible to improve the results by sampling the neighborhood (of the solution space) of the best solution. This was done using a normal pseudo-random generator and using numbers picked from a Sobol sequence.

[4]Table 8.6 shows that there is almost no difference in total time of flight for the heliocentric phase.

Symbol	DE1 best	DE1 (Powell)	DE1 runner up	Garot best	Limits
T_{launch}	2014-11-15	2014-11-15	2014-11-16	2014-05-17	[2010-01-01, 2018-07-01]
ω_0	1.44	9.98	303.17	115.9	$[0,360]$
$\alpha_S(1)$	−40.22	−40.25	−35.15	−46.0	$[-50,-20]$
$\alpha_S(2)$	−17.26	−17.25	−6.29	−12.3	$[-90,90]$
$\alpha_S(3)$	36.97	37.14	40.62	37.3	$[20,50]$
$\alpha_S(4)$	54.48	54.42	45.29	59.3	$[10,65]$
R_1	0.3030	0.2970	0.2982	0.29	$[0.26,0.50]$
R_2	0.2600	0.2600	0.2894	0.29	$[0.26,0.50]$
Fitness	−220.915	−220.671	−220.923	−228.3	–

Table 8.3 Optimal parameters for the solar sailing mission as found by the differential evolution runs.

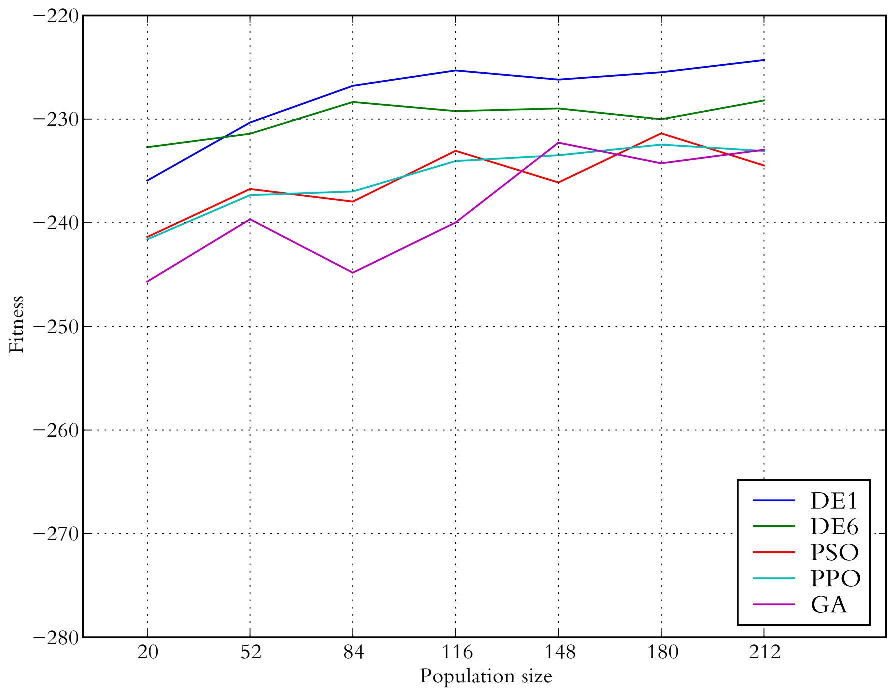


Figure 8.10 Average of the best fitness values for each optimizer/population size combination for the complete solar sailing mission.

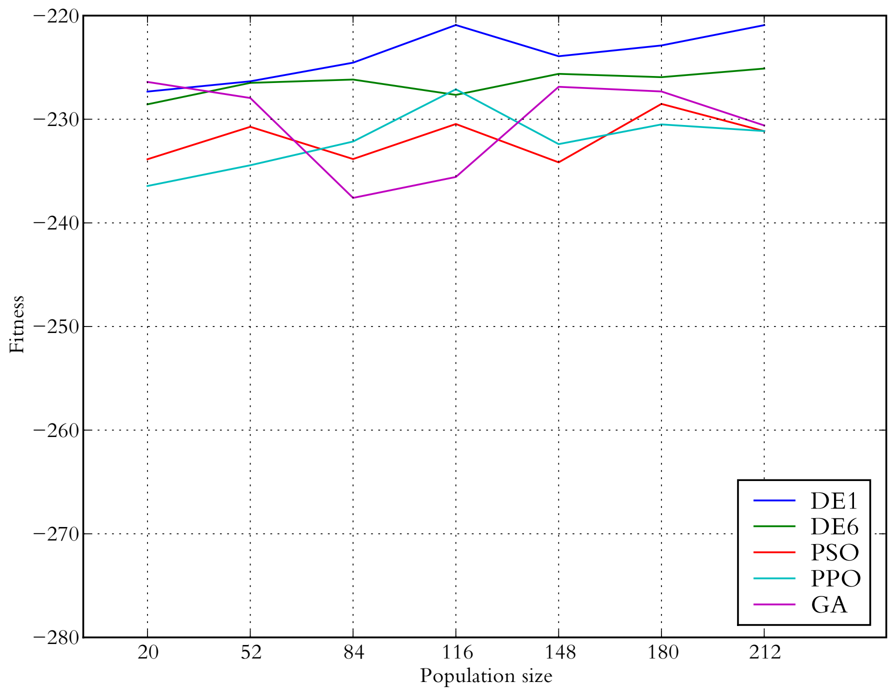


Figure 8.11 Best of the best fitness values for each optimizer/population size combination for the complete solar sailing mission.

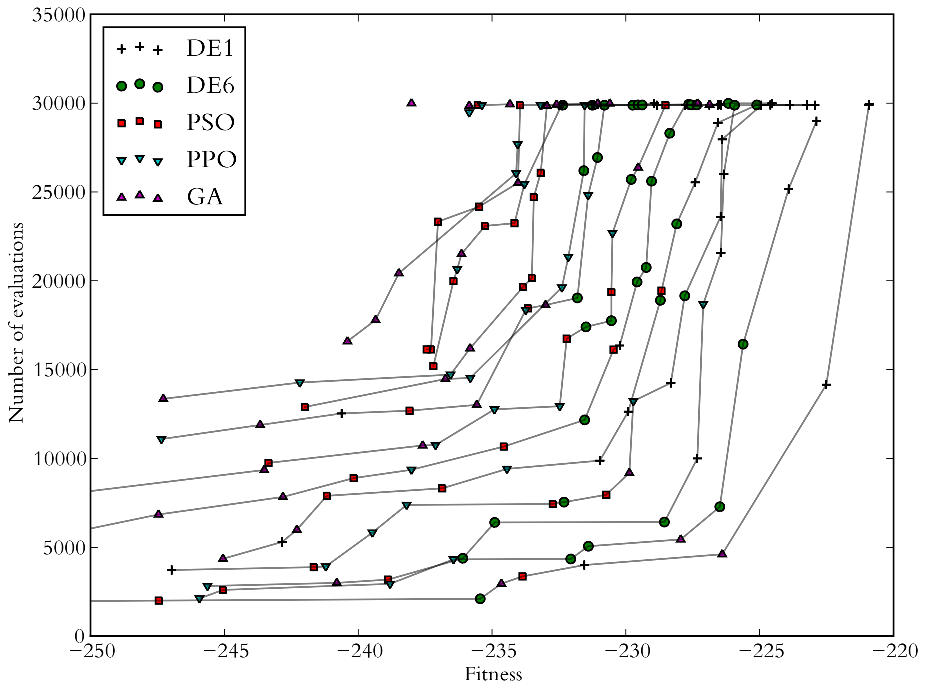


Figure 8.12 Fitness versus the number of evaluations for the different optimizers used, and the Pareto-fronts of those points.

run	*d*
1	0.02512
2	0.01585
3	0.01000
4	0.00631
5	0.00398

Table 8.4 Tabulated values of equation (8.18)

For this experiment, 5 runs were executed for both sequences, consisting of 5000 samples each. For each run, the range from which the samples were taken was controlled by the following formula:[5]

$$d = 0.1^{2+0.2(run-3)} \tag{8.18}$$

which is tabulated in table 8.4 Using the limits from table 8.3, the random points for each solution vector value i were picked uniformly from the interval

$$[x_i - 0.5 \cdot d \cdot (u_i - l_i), x_i + 0.5 \cdot d \cdot (u_i - l_i)]$$

so for example in run number 3, the value of ω_0 was allowed to vary between

$$(\omega_0)_{best} \pm 0.5 \cdot d_{run=3} \cdot (u_{\omega_0} - l_{\omega_0}) = 1.44 \pm 0.5 \cdot 0.01 \cdot 360 = [-2.16, 5.04]$$

Histograms of the change in objective value relative to the best solution found using a global optimizer have been plotted in figure 8.13. This shows that for $run = 1$ and $run = 2$, the

[5]An initial run was done to determine the range of d; first, $d = 0.1^{run}$ was used, which gave the best results for $run = 2$; using this formula, d runs from $0.1^{1.6}$ to $0.1^{2.4}$.

sampling area is too large, as almost no individuals were found which matched the original values.

The best fitness value found using the Sobol sequence was found for $run = 3$, having the value -220.5327, whereas the random sequence found its best value in run 5, with value -220.6796. This Sobol number seems to be an outlier though, as the runner-up value is -220.6904. The top-5 for each algorithm is listed in table 8.5, which shows that the 3rd-5th positions have better values for the pseudo-random sequence.

$run = 3$ means that the solutions are picked randomly from a box with a size of 1% of the solution space (in all directions).

For $run = 4$ and $run = 5$, the distribution of the fitness values becomes more concentrated around the original value, which is logical, as the size of the box from which solutions can be chosen decreases.

In figure 8.13, the shape of the histograms for both sequences is roughly the same. This, combined with the fact that the top-5 fitness values found by both number generators (except for 1 lucky outlier) have the same values makes it impossible to conclude that either one is better than the other, for this problem.

Position	Sobol	Random
1	−220.5327	−220.6796
2	−220.6904	−220.6942
3	−220.7325	−220.7012
4	−220.7709	−220.7117
5	−220.7833	−220.7569

Table 8.5 Top-5 of fitness values found by applying a random sampling local optimizer around the best solution of the solar polar sail mission.

8.6 Local optimization using Powell's method

Finally, the results found in section 8.4 where put into a local optimizer using Powell's method (as introduced in section 2.5).

This optimization needed 5840 evaluations, and resulted in an optimized fitness value of 220.6709. This is an absolute improvement of 0.24 (or 0.11%) relative to the best solution found using a global optimizer.

Compared to the solutions found using random sampling, this result reaches comparable fitness levels, but needs only a quarter of the number of evaluations. There is one issue though: Powell's method requires that all objective function evaluations are done sequentially, which means this algorithm cannot exploit multiple processor cores.[6] This means that on the system on which these optimizations were executed, the (wall)time needed for the Powell optimization was almost twice as long as for the random sampling case.

The geocentric trajectory found using Powell's method is shown in figures 8.14 and 8.15. A big difference is directly seen when comparing this trajectory with the one found in [*Garot*, 2006]: this trajectory does not make use of flybys near the Moon. The required time of flight in the new case is 277.5 days, which is a lot shorter than the 424 days reported in [*Garot*, 2006]. The reason for this extraordinary long time is that the escape velocity was reached while the spacecraft was travelling towards the Sun - so the sail was put in the feathering position, and could not be used to accelerate.

The heliocentric trajectory is plotted in figure 8.16. The times of flight for these two optimizations are summarized in table 8.6. Note that for the heliocentric phase, the differ-

[6]Unless single evaluations of the objective function can be written to make use of multiple cores.

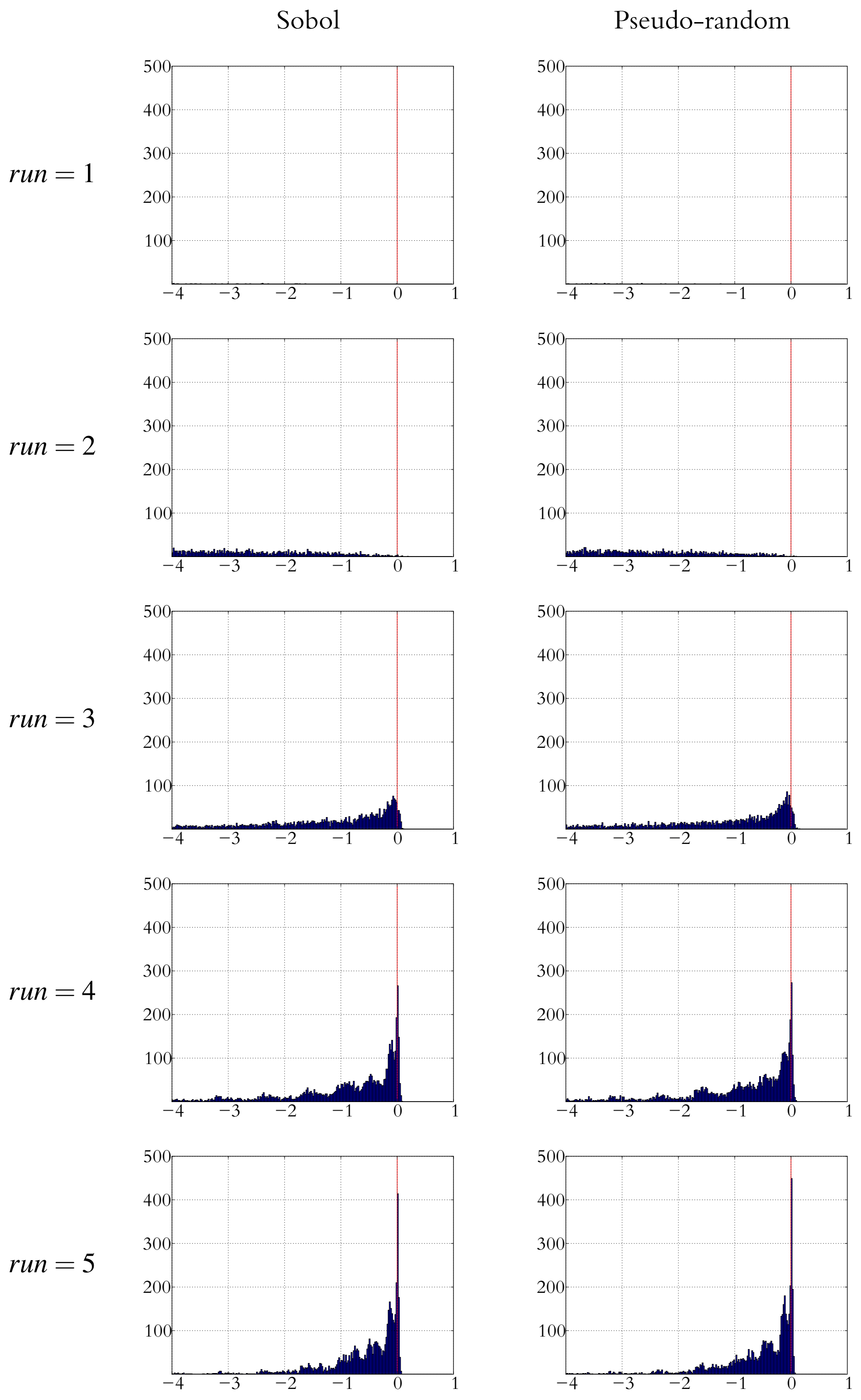


Figure 8.13 Histograms of the change in objective function value of the Solar polar sail mission, when applying a Monte Carlo sampling around the best found solution, using a Sobol random sequence (left) and a pseudo-random number generator (right) for different sampling area sizes (rows).

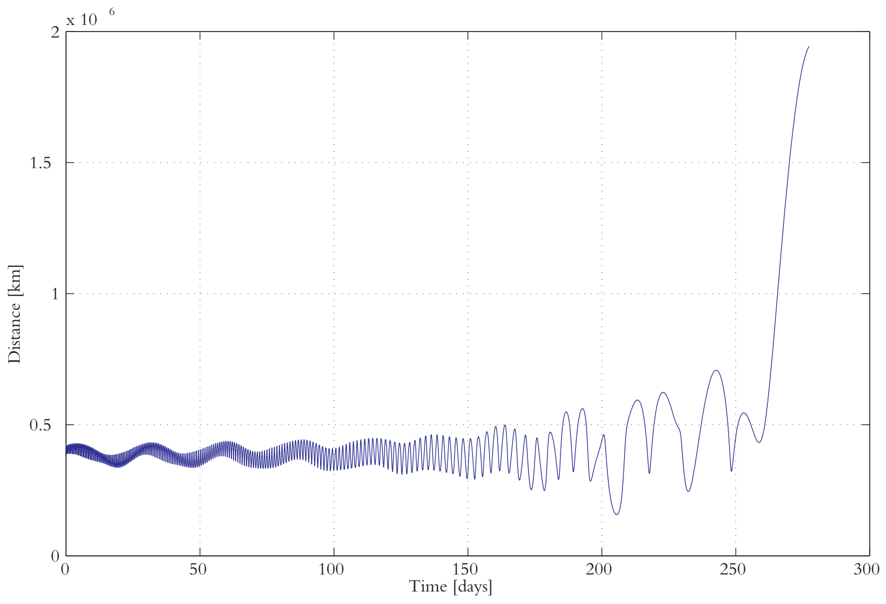


Figure 8.14 Distance from the spacecraft to the Moon during the geocentric phase as found using Powell's quadratically convergent method.

ence is only 8 days, on a total of almost 1800. This shows again that the total time of flight was primarily influenced by the geocentric phase.

Another noteworthy thing about the outcome of Powell's method is that the final conditions of the optimization better matched the requirements: the final eccentricity was 0.004, the inclination $90.0°$, and the semi-major axis 0.40AU. The values reported in [*Garot*, 2006] for the semi-major axis and the inclination are both of the same order, but the eccentricity was worse in both cases.

Phase	Garot	DEI Powell
Inward spiral + Circularization	768	826
Cranking	861	812
Outward spiral	166	149
Total	1795	1787

Table 8.6 Times of flight in days for the phases in the heliocentric phase.

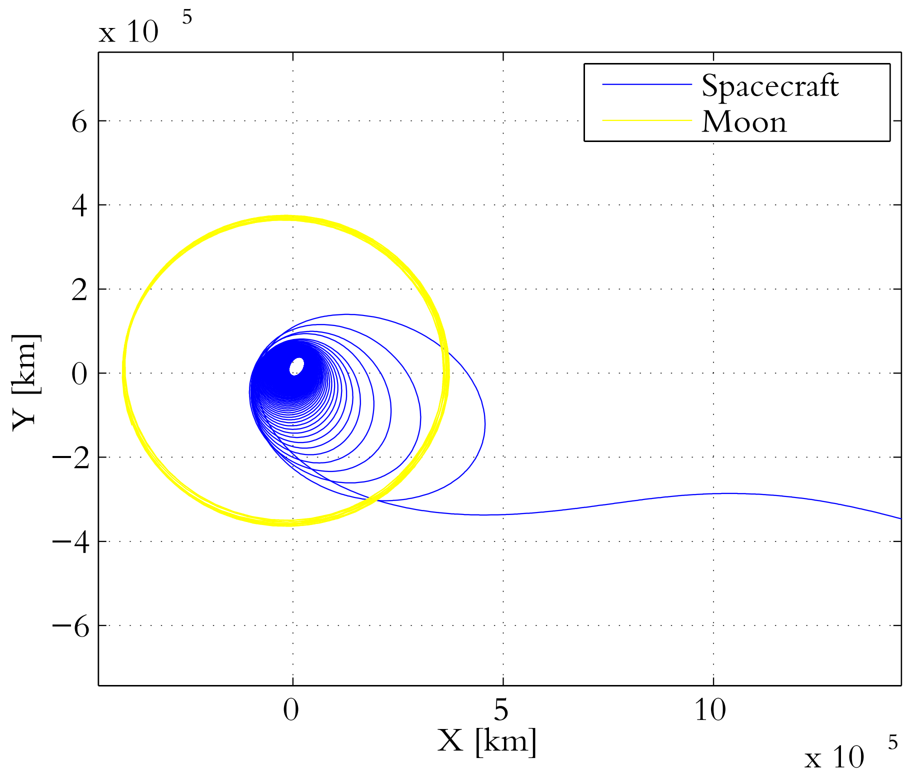


Figure 8.15 Trajectories of the spacecraft and the Moon during the geocentric phase as found using Powell's quadratically convergent method.

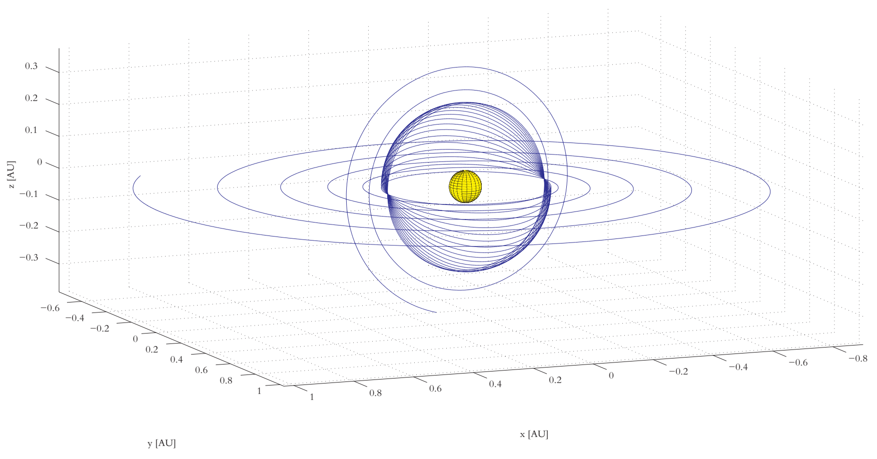


Figure 8.16 Trajectory of the spacecraft during the heliocentric phase as found using Powell's quadratically convergent method.

8.7 Conclusions and recommendations for the Solar polar sail mission

From the optimizations done for this problem, the following can be stated:

- Using particle swarm optimization to optimize the geocentric phase yielded the best results of all optimizers.
- A 4th order sine polynomial can be used as a fitting function for the pitch angles as a function of the GTO-orientation, and gives a much better fit than the original polynomial function.
- The total mission was best optimized using differential evolution. This way, an improvement of 7.38 [no unit] of fitness value was made possible, which is an improvement of 3.24% over the best result found previously. There is one drawback: this result required almost thrice as many function evaluations as the original optimization. Furthermore, most of the reduction in fitness value was caused by a better timed geocentric phase.
- Increasing population sizes while keeping the maximum number of iterations the same tends to improve the results of all optimizers.
- Local optimization using random sampling is not sensitive to the random generator used: the samples generated using a pseudo-random number generator and the samples generated using a Sobol-sequence gave comparable results.
- Local optimization using Powell's method gives (for this problem) results which are comparable to those found using local optimization by random sampling. It does however need only a quarter of the number of evaluations.
- Local optimization (using either method) improves the fitness values found.
- Local optimization is a good tool to reduce the violation of constraints; all final conditions were better met after using Powell's method than in any of the other runs.

Also, some possible questions for future research:

- Right now, the maximum population size used was 212 individuals. What happens if this number is increased even further?
- Can the solution be improved by not using a fit for the pitch angles in the geocentric phase, and instead letting them be optimized as independent variables?
- Can the use of hybrid algorithms further improve the convergence speed of the optimization?

Optimizer	ω [°]	t_f [days]	n_{gen}	n_{eval}	$\alpha_E(1)$ [°]	$\alpha_E(2)$ [°]	$\alpha_E(3)$ [°]
OPTIDUS	0	276.2	36	2209	28.4	13.6	2.3
OPTIDUS-DE1		270.5	48	615	28.53	11.52	4.50
OPTIDUS-DE2		271.0	36	615	27.57	14.86	2.18
OPTIDUS-PSO		270.9	47	615	24.04	12.80	4.21
OPTIDUS-PPO		270.9	30	615	32.19	13.33	5.21
OPTIDUS	45	289.9	44	5926	32.2	15.3	2.8
OPTIDUS-DE1		273.0	16	618	33.58	19.89	2.32
OPTIDUS-DE2		273.0	32	618	35.46	12.50	7.45
OPTIDUS-PSO		271.5	39	618	25.28	18.57	7.93
OPTIDUS-PPO		271.9	49	615	34.00	13.71	3.16
OPTIDUS	90	338.7	20	4949	47.7	16.0	5.2
OPTIDUS-DE1		328.5	2	688	59.32	3.52	0.00
OPTIDUS-DE2		328.3	48	688	57.40	0.00	0.00
OPTIDUS-PSO		328.7	39	656	53.08	27.87	−11.51
OPTIDUS-PPO		344.5	42	656	47.76	24.21	8.57
OPTIDUS	135	336.5	32	2993	41.6	18.8	5.6
OPTIDUS-DE1		329.7	0	676	43.60	38.48	11.48
OPTIDUS-DE2		328.2	30	676	56.25	6.04	0.00
OPTIDUS-PSO		329.2	28	640	44.97	10.47	23.92
OPTIDUS-PPO		344.7	48	640	47.83	18.35	3.47
OPTIDUS	180	319.4	36	5632	39.0	20.3	9.6
OPTIDUS-DE1		325.0	15	638	34.15	17.44	8.59
OPTIDUS-DE2		324.6	40	638	38.56	19.94	4.93
OPTIDUS-PSO		324.3	46	635	37.05	19.75	5.76
OPTIDUS-PPO		324.5	50	635	37.23	20.23	5.32
OPTIDUS	225	306.0	50	4329	31.3	14.8	4.4
OPTIDUS-DE1		301.6	11	623	28.96	13.87	0.00
OPTIDUS-DE2		301.5	30	623	36.31	11.04	3.08
OPTIDUS-PSO		301.5	49	616	26.99	13.64	3.24
OPTIDUS-PPO		300.4	11	616	42.45	10.95	−4.69
OPTIDUS	270	293.7	34	3039	27.1	12.8	3.3
OPTIDUS-DE1		284.9	5	615	29.00	12.60	11.02
OPTIDUS-DE2		284.0	38	615	36.71	12.36	4.55
OPTIDUS-PSO		284.7	50	614	22.31	19.33	2.30
OPTIDUS-PPO		284.2	43	614	20.99	12.34	2.45
OPTIDUS	315	281.5	50	5713	29.2	12.9	3.7
OPTIDUS-DE1		279.1	6	615	27.65	18.61	0.00
OPTIDUS-DE2		277.9	44	615	23.88	13.36	3.57
OPTIDUS-PSO		277.3	45	614	30.84	9.68	2.77
OPTIDUS-PPO		277.4	50	614	32.08	11.09	2.01

Table 8.7 Results of the optimization of the geocentric phase of the solar polar sail mission using several optimizers. OPTIDUS is the original version used by [*Garot*, 2006].

Chapter 9
Third Global Trajectory Optimization Competition

This chapter contains an overview of the third Global Trajectory Optimization Competition (GTOC3) which was held at the end of 2007. A team from Delft also participated, but was not able to come up with a satisfactory result. This chapter uses some of the results which the other teams submitted, and tries to find a good solution using DE and Powell's local optimizer.

9.1 Introduction to GTOC

The Global Trajectory Optimization Competition (GTOC) is a series of worldwide competitions in which the goal is to find the best trajectory for a given problem. The first competition was organized in 2005 by the Advanced Concepts Team at ESA, and has since then been held annually by the winners of the previous edition [*GTOC1*, 2005]. The first edition was won by the Outer Planets Mission Analysis Group of the Jet Propulsion Laboratory, who organized the second contest in 2006 [*GTOC2*, 2006]. In December 2007, the third edition was organized by the Dipartimento di Energetica of the Politecnico di Torino [*Casalino*, 2007]. This competition was won by CNES, who have organized the fourth edition, which took place in March 2009, and which was won by a team from Moscow State University [*GTOC4*, 2009].

In each of the 4 GTOC contests, the goal is to optimize a proposed space mission, while adhering to the given boundary conditions. For example in GTOC2, the goal was to optimize a multiple asteroid rendezvous mission. The participants were given four groups of asteroids, and the trajectory had to visit one asteroid from each of these four groups. The spacecraft to visit these asteroids can only use a low-thrust engine for its propulsion. This problem has been analyzed in detail in [*Evertsz*, 2008].

9.2 The GTOC3 problem

In GTOC3, a mission similar to the one from GTOC2 has to be optimized: this time, three asteroids from one group of 140 asteroids have to be visited, and the spacecraft must return to Earth at the end of the mission. At each of these three asteroids, the spacecraft should stay for a period of at least 60 days, but preferably longer: the performance metric of each solution is a weighted sum of the final mass of the spacecraft and the minimum stay time at the three asteroids (see (9.1)). Again, this mission must make use of a low-thrust engine, but it may also use gravity assists from the Earth. [*Casalino et al.*, 2007] is the official problem description for GTOC3, and some parts will be repeated here for clarity.

Symbol	Description	[unit]
m_i	Initial mass	2000 [kg]
I_{sp}	Specific impulse	3000 [s]
T_{max}	Maximum thrust	0.15 [N]

Table 9.1 GTOC3 Spacecraft properties

The launch is at the Earth in the MJD interval [57388, 61041].[1] After launch, the spacecraft can be given a hyperbolic excess velocity of up to 0.5 km/s, in any direction. From then on, only electronic propulsion and gravity assists can be used to influence the orbit of the spacecraft.

The spacecraft has to rendezvous with 3 asteroids, while staying at each asteroid for a minimum duration of 60 days. Finally, the spacecraft must return to Earth within 10 years after departing. During this flight, the spacecraft is allowed to perform Earth flybys, when it is at a minimal distance of 6871 km from the center of the Earth.

The properties of the spacecraft and its engine are given in table 9.1.

The objective function of the optimization is given by the nondimensional quantity J:

$$J = \frac{m_f}{m_i} + K\frac{\min_{j=1,3}(\tau_j)}{\tau_{max}} \tag{9.1}$$

where m_i and m_f are the spacecraft initial and final mass, respectively; τ_j, with $j = 1, 3$, represents the stay-time at the j-th asteroid in the rendezvous sequence and $\min_{j=1,3}(\tau_j)$ is the shortest asteroid stay-time; $\tau_{max} = 10$ years is the available trip time, and K is a weighting factor set to $K = 0.2$.

For the Earth and all asteroids, a set of Kepler elements are given, and these bodies are assumed to follow elliptical orbits without any perturbations.

Finally, the conditions for rendezvous are also prescribed: the spacecraft is considered to be in the same orbit as the asteroid when the position difference between the spacecraft and the asteroid is less than 1000 km, and when the speed difference is less than 1 m/s.

The orbits of these asteroids and that of the Earth are shown in figure 9.1; the semi-major axis of these asteroids vary from 0.90 to 1.10 AU, the inclination from 0 to 10° and the eccentricity from 0.0 to 0.9 (with 133 out of 140 asteroids having an eccentricity below 0.6).

9.3 Solutions of the GTOC3 problem

Almost all teams attacked this problem by splitting the problem in two subtasks: the first task was to identify good sequence(s) of asteroids to visit. This was generally done by determining the required ΔV for trajectories using high thrust manoeuvres between using Lambert targeting. Some teams used deep space manoeuvres and reduced the number of possible combinations by only considering a subset of the asteroids, based of their orbital elements.

The second step then was to find the best low-thrust trajectory for a given sequence, mostly done using a local optimizer.

In the end, the best solution was found by the team from CNES. Their solution used the sequence E E E 49 E 37 85 E E - their spacecraft was launched from Earth, then passed it two times for gravity assists, visited asteroid number 49, once again did a flyby past Earth, then visited asteroids 37 and 85, did another flyby, and returned home. This resulted in a final mass $m_f = 1733$[kg] with a minimum stay-time $\tau_{min} = 60$[days].

[1] This MJD interval corresponds to the interval 2016-01-01 and 2026-01-01.

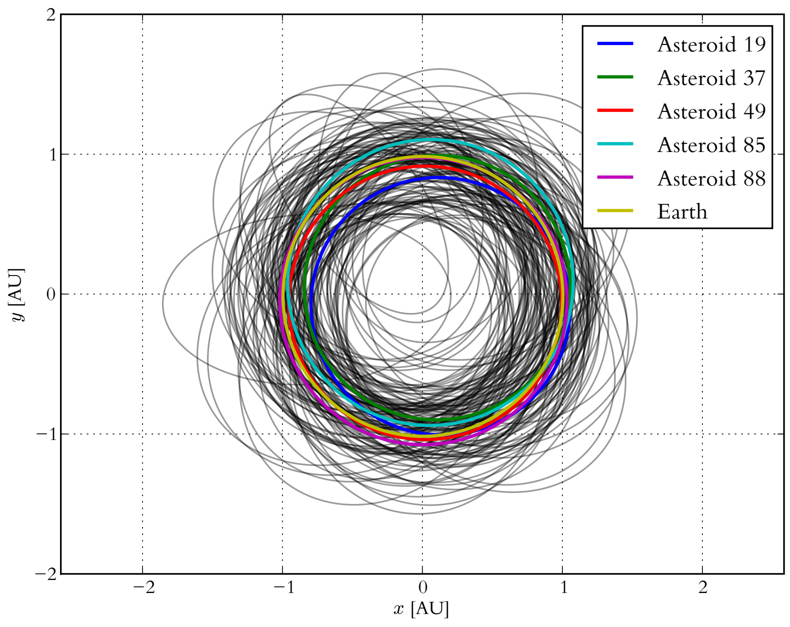


Figure 9.1 Orbits of the asteroids and Earth around the Sun, as given in the GTOC3 problem. The asteroids chosen by the winning team (CNES, using 49, 47, 85) and the asteroids used in the best solution without flybys (88, 19, 49) have been highlighted.

The first three runners-up visited the same sequence of asteroids, but had variations in the gravity assists. There were only small differences in the final mass, with #4 having a final mass $m_f = 1717$[kg], which is less than 1% difference. The top four all had $\tau_{min} = 60$[days].

The first team not making use of gravity assists was lead by the University of Glasgow, and ranked 10th. They used the sequence E 88 19 49 E. Their final objective results were $m_f = 1606$[kg] and $\tau_{min} = 62$[days].

The Delft University's team submitted a solution using the sequence E 96 122 85 E, with a final mass $m_f = 1130$kg stay time $\tau_{min} = 94$d.

This solution was produced in a two step-process: first, the feasible asteroids to visit where selected on the basis of their orbital elements. Next, the ΔV needed for impulsive transfers were calculated using GALOMUSIT, and a further reduction in the set of feasible asteroids was made. Finally, low-thrust orbits between these asteroids where determined using a DE algorithm and a (rather slow) local (steepest descent) optimizer.

The resulting solution was not ranked because it violated the rendezvous constraints. For example, the arrival at asteroid 122 in leg 2 had a large velocity error, as can be seen in figure 9.3. Figures 9.2-9.5 contain plots of the 4 legs of the solution which was submitted.

A complete overview of the results submitted for GTOC3 can be found at the competition's website [*Casalino*, 2007].

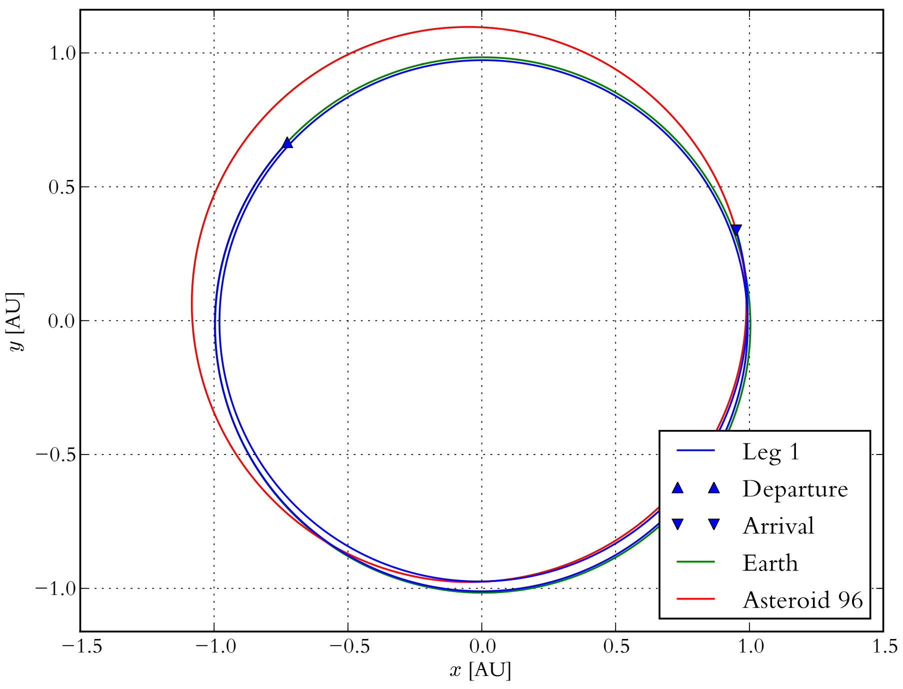


Figure 9.2 Leg 1 of the original TUDelft GTOC3 solution (Earth to asteroid 96).

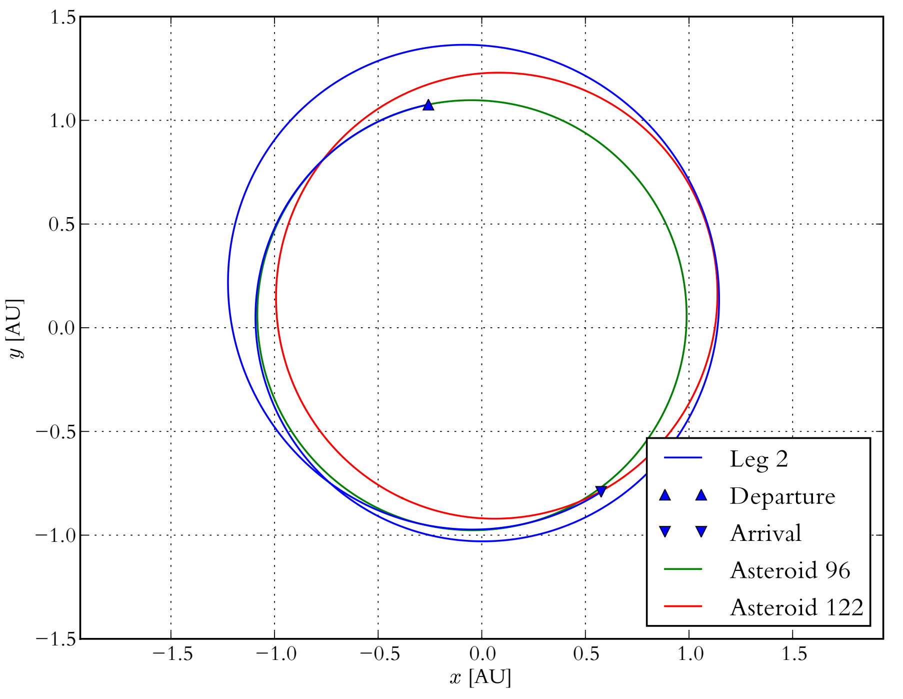


Figure 9.3 Leg 2 of the original TUDelft GTOC3 solution (asteroid 96 to 122).

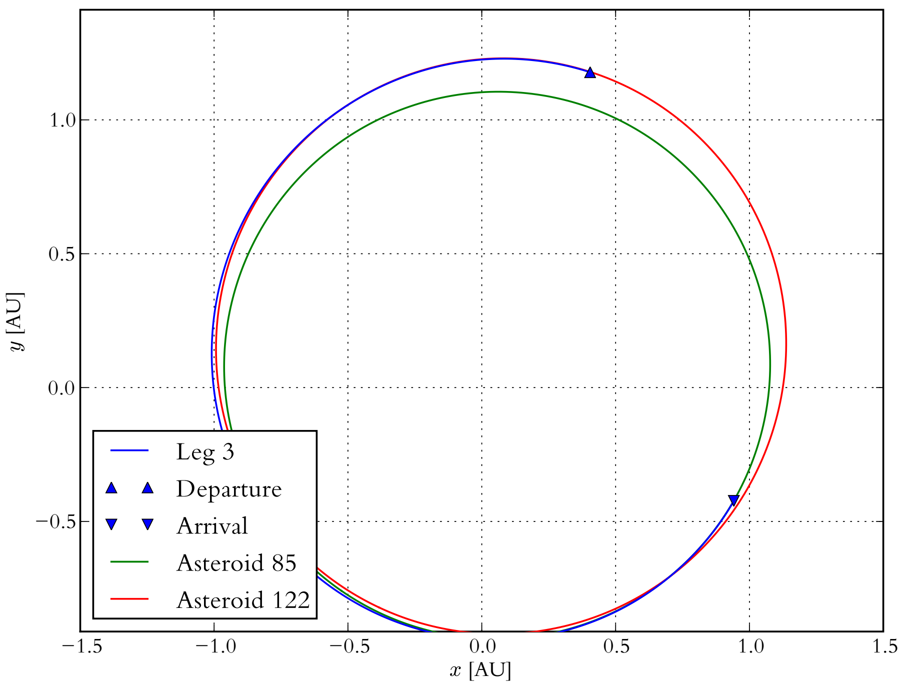


Figure 9.4 Leg 3 of the original TUDelft GTOC3 solution (asteroid 122 to 85).

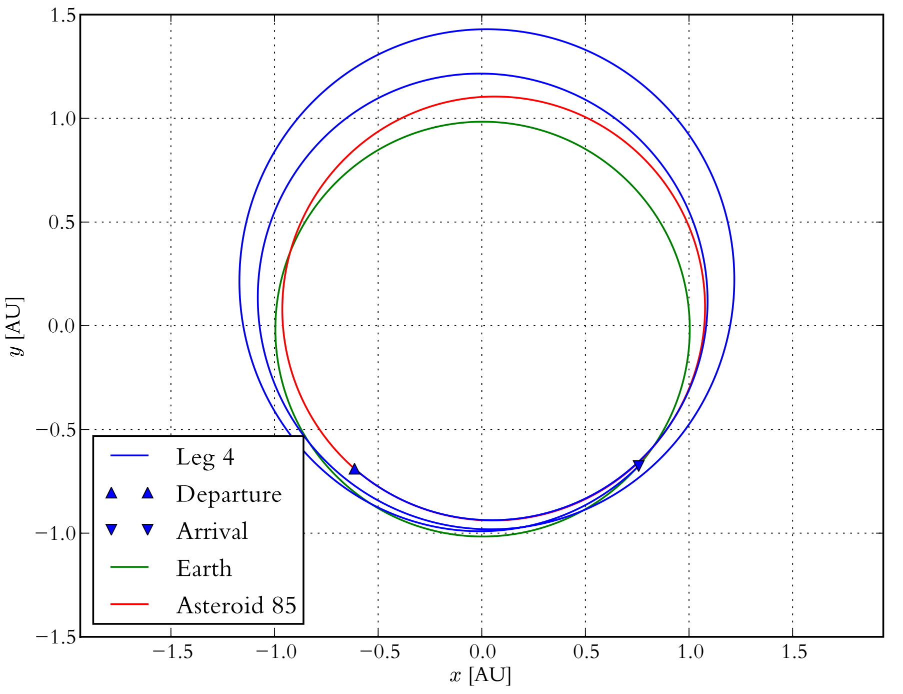


Figure 9.5 Leg 4 of the original TUDelft GTOC3 solution (asteroid 85 to Earth).

9.4 Simulation Model

9.4.1 First order differential equations

The spacecraft is only subjected to two forces: the gravitational pull by the Sun and the thrust $\mathbf{T}$ generated by the engine. Thus, the second derivative of the position $\mathbf{r}$ of the spacecraft is given by the following equation:

$$\ddot{\mathbf{r}} = -\mu_S \frac{\mathbf{r}}{|\mathbf{r}|^3} + \frac{\mathbf{T}}{m} \tag{9.2}$$

In this equation, m is the instantaneous mass of the spacecraft, and $\mu_S \approx 1.3271 \cdot 10^{11} \mathrm{kg}^3\ \mathrm{s}^{-2}$ the Sun's gravitational parameter.

The spacecraft's mass decreases as fuel is consumed, and the fuel flow depends on the magnitude of the thrust:

$$\dot{m} = \frac{|\mathbf{T}|}{g_0 I_{sp}} \tag{9.3}$$

In this equation, $g_0 = 9.80665\mathrm{ms}^{-2}$ is the standard gravity acceleration and I_{sp} the specific impulse of the spacecraft's engine, having a value of 3000s.

To be able to generate a trajectory, the state variables must be integrated. In this case, this is done using a first order differential equation using the state derivatives in the form

$$\dot{\mathbf{x}} = \mathbf{f}(t, \mathbf{x})$$

The state vector $\mathbf{x}$ can be defined as

$$\mathbf{x} = (x\ y\ z\ \dot{x}\ \dot{y}\ \dot{z}\ m)^T$$

. The coordinates and velocities are given in the J2000 heliocentric reference system.

The derivatives of this state vector are given by a set of first-order differential equations following from (9.2) and (9.3):

$$\dot{\mathbf{x}} = \begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ \ddot{x} \\ \ddot{y} \\ \ddot{z} \\ \dot{m} \end{pmatrix} = \begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \\ -\mu_S \frac{x}{r^3} + \frac{T_x}{m} \\ -\mu_S \frac{x}{r^3} + \frac{T_y}{m} \\ -\mu_S \frac{x}{r^3} + \frac{T_z}{m} \\ \frac{|\mathbf{T}|}{g_0 I_{sp}} \end{pmatrix} \tag{9.4}$$

in which x, y, z are the components of $\mathbf{r}$, T_x, T_y and T_z are the components of the thrust vector $\mathbf{T}$.

9.4.2 Integrator

The used integrator is a fifth-order Runge-Kutta integrator with adaptive stepsize h. The general form of a fifth-order Runge Kutta formula is: [*Press et al.*, 2007]

$$y_{n+1} = y_n + c_1 k_1 + c_2 k_2 + c_3 k_3 + c_4 k_4 + c_5 k_5 + c_6 k_6 + O(h^6) \tag{9.5}$$

i	a_i	b_{ij}						c_i	c_i^*
		j	1	2	3	4	5		
1								$\frac{37}{378}$	$\frac{2825}{27648}$
2	$\frac{1}{5}$		$\frac{1}{5}$					0	0
3	$\frac{3}{10}$		$\frac{3}{40}$	$\frac{9}{40}$				$\frac{250}{621}$	$\frac{18575}{48384}$
4	$\frac{3}{5}$		$\frac{3}{10}$	$-\frac{9}{10}$	$\frac{6}{5}$			$\frac{125}{595}$	$\frac{13525}{55296}$
5	1		$-\frac{11}{54}$	$\frac{5}{2}$	$-\frac{70}{27}$	$\frac{35}{27}$		0	$\frac{277}{14336}$
6	$\frac{7}{8}$		$\frac{1631}{55296}$	$\frac{175}{512}$	$\frac{575}{13824}$	$\frac{44275}{110592}$	$\frac{253}{4096}$	$\frac{512}{1771}$	$\frac{1}{4}$

Table 9.2 Cash-Karp parameters for embedded Runge Kutta method, [*Press et al.*, 2007]

with k_1 to k_6 are given by

$$\begin{aligned}
k_1 &= hf(t, y_n) \\
k_2 &= hf(t + a_2 h, y_n + b_{21} k_1) \\
k_3 &= hf(t + a_3 h, y_n + b_{31} k_1 + b_{32} k_2) \\
&\vdots \\
k_6 &= hf(t + a_6 h, y_n + b_{61} k_1 + \ldots + b_{65} k_5)
\end{aligned} \tag{9.6}$$

The adaptive stepsize is determined by estimating the error of each integration step using an embedded fourth-order formula:

$$y_{n+1}^* = y_n + c_1^* k_1 + c_2^* k_2 + c_3^* k_3 + c_4^* k_4 + c_5^* k_5 + c_6^* k_6 + O(h^5) \tag{9.7}$$

In these equations, a_i, b_{ij}, c_i and c_i^* are constants. [*Press et al.*, 2007] recommends the use of Cash-Karp parameters, which are repeated in table 9.2. These values have been used.

Now, the difference between these to integrations is an estimate of the error:

$$\Delta = y_{n+1} - y_{n+1}^* = \sum_{i=1}^{6} (c_i - c_i^*) k_i + (h-1) O(h^5) \tag{9.8}$$

This term can be used to estimate the required stepsize to integrate with a certain error. This desired error is the scale vector $\mathbf{x}_{scale}$, and is calculated as the product of a tolerance ε (which can be chosen by the user) and the absolute values of the terms in the state vector:

$$\mathbf{x}_{scale} = |\mathbf{x}| \varepsilon$$

To prevent problems with near-zero values of terms in $\mathbf{x}$, which would cause the stepsize to become too small, [*Press et al.*, 2007] recommends adding the product of $h\mathbf{f}$ to this term, giving the following expression for the scale vector:

$$\mathbf{x}_{scale} = |\mathbf{x}| \varepsilon + h|\mathbf{f}|$$

The choice of the value of ε was based on the work done in [*Garot*, 2006]. Even though this is a problem-dependent choice, the orbits used are quite similar: the orbit used for determining the optimal value of ε was an elliptical orbit with perihelion at 0.4AU and aphelion at 1.0AU, whereas the asteroids to be visited by the GTOC3 spacecraft orbit the Sun in (elliptic) orbits with semi-major axes between 0.8 and 1.2AU.

In [*Garot*, 2006] it was found that $\varepsilon = 10^{-9}$ resulted in a maximum error of 3.74 km for the semi-major axis, when integrating the motion of the spacecraft over a period of 10 years. 3.74km might seem to be a big error, but the constraints allow for an error of 1000 km, so this is an acceptable compromise between error and computation time. Note that for other values of ε, the magnitude of the error scales approximately linear with this local tolerance.

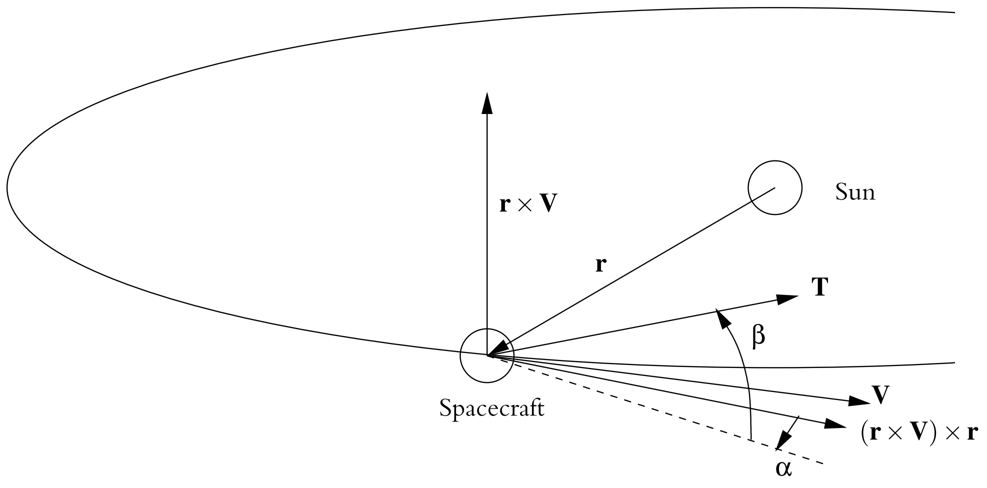

$\mathbf{r}\times\mathbf{V}$
Sun
$\mathbf{r}$
$\mathbf{T}$
β
$\mathbf{V}$
Spacecraft
$(\mathbf{r}\times\mathbf{V})\times\mathbf{r}$
α

Figure 9.6 Definition of the thrust angles and transformation matrix for the thrust vector from the spacecraft oriented frame to the heliocentric frame.

9.4.3 Thrust direction parametrization

The thrust vector $\mathbf{T}$ used in the previous section is written in the inertial reference frame, but according to [*Yam and Longuski*, 2006] it is desired to specify the thrust using angles in reference frame coupled to the spacecraft. Figure 9.6 shows how this coupling is made.

In this case, a reference frame coupled to the position and velocity vectors of the spacecraft relative to the Sun is used, of which the axes are defined by:

$$\begin{aligned}\mathbf{e}_1 &= \hat{\mathbf{r}} \\ \mathbf{e}_3 &= \hat{\mathbf{r}} \times \hat{\dot{\mathbf{r}}} \\ \mathbf{e}_2 &= \mathbf{e}_3 \times \mathbf{e}_1\end{aligned} \tag{9.9}$$

The thrust vector in this reference frame $\mathbf{T}_{sc}$ is determined by two angles α and β which determine the direction, and the magnitude T_{mag}:

$$\mathbf{T}_{sc} = T_{mag}\begin{pmatrix}\cos\beta\cos\alpha \\ \cos\beta\sin\alpha \\ \sin\beta\end{pmatrix} \tag{9.10}$$

Now the thrust vector in the inertial reference frame is calculated with:

$$\mathbf{T} = [\mathbf{e}_1\mathbf{e}_2\mathbf{e}_3]\,\mathbf{T}_{sc} \tag{9.11}$$

9.4.4 Thrust magnitude parametrization using Chebyshev polynomials

In the Chebyshev formulation, both the magnitude and direction angles of the thrust are modelled using Chebyshev polynomials. This is done to have a smooth, continuous thrust profile with a low number of variables.

This, then, gives the following formula for the magnitude of the thrust over time:

$$T_{mag}(x) = T_{max}\sum_{i=0}^{k} c_i T_i(x) \tag{9.12}$$

where T_i is the ith order Chebyshev polynomial, which can be defined using:

$$T_0(x) = 1 \tag{9.13}$$

$$T_1(x) = x \tag{9.14}$$

$$T_n(x) = 2xT_{n-1}(x) - T_{n-2}(x) \tag{9.15}$$

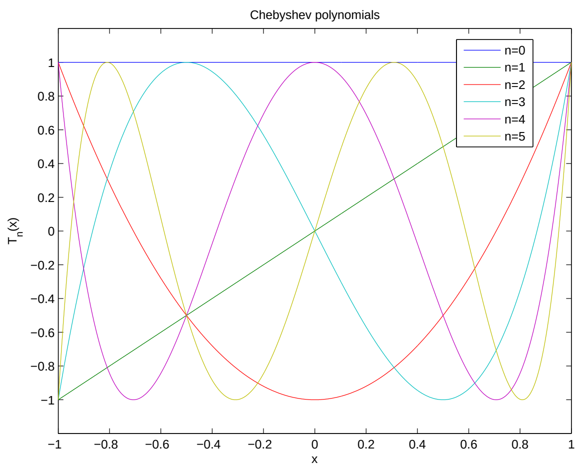


Figure 9.7 Chebyshev polynomials

The formulas for the thrust angles are similar:

$$\alpha(x) = \pi \sum_{i=0}^{k} a_i T_i(x) \tag{9.16}$$

$$\beta(x) = \pi/2 \sum_{i=0}^{k} b_i T_i(x) \tag{9.17}$$

More information on Chebyshev polynomials can be found in [*Demeyer*, 2007, Appendix A] and [*Press et al.*, 2007]. The first 6 Chebyshev polynomials are plotted in figure 9.7.

Using these polynomials in the thrust model is done by putting the c_k terms in the solution vector, instead of putting in the raw values of the thrust magnitude and direction at certain moments in time.

A set of 5th-order Chebyshev polynomials have been used in the current implementation; no research has been done to identify the optimal number of parameters.

Of course, care has to be taken that the magnitude of the thrust does not exceed the maximum allowed thrust. This is done by clipping the thrust to the maximum allowed value if it is too high.

9.4.5 On/off switching

[*Yam and Longuski*, 2006] suggests using on/off switching for the thrust arcs, to be able to coast. This has been implemented by adding a list of switching moments to the solution vector. During the execution of the objective function, at each integration step it is checked if the thrust is in the on state, and only then is it allowed to thrust. Five switching moments are used in this case.

Because the thrust must be non-zero when the spacecraft leaves the Earth or an asteroid, the thrust is put into the on state at the start of a leg. After the first time interval, it is the deactivated. At the start of the third interval, it is reactivated again, and so on, until after the fifth the engine is shut down again.

At this moment, the spacecraft should be near the target asteroid: because the rendezvous conditions prescribe that the position and velocity of both the spacecraft and the target body should match, the spacecraft will only stay in the same orbit when no thrust is applied.

9.5 Launch conditions

9.5.1 First leg

The spacecraft has a launch date t_0 which lies between MJDs 57388 and 61041; at some moment between these dates, it is launched with a hyperbolic excess velocity V_0 with a maximum of 0.5 [km/s].

Because the direction of this heliocentric velocity can be chosen by the optimizer, two angles α_0 and β_0 are introduced to control this direction. By applying the same transformation as for the thrust (see figure 9.6, but this time using the velocity and position of the Earth as the basic velocity, the direction of the hyperbolic excess velocity can be determined in the heliocentric reference system.

9.5.2 Other legs

For the remaining legs, instead of an absolute date, a stay-time is used. After the spacecraft has been observing the asteroid for some time, the engine is started again, and the next leg begins. Because no excess velocity can be given at this stage, the solution vector does not contain the three terms determining the excess velocity.

9.6 Solution vector

The complete solution vector for a leg has been described in the previous sections; it contains the following 24 terms:

$$\Big[t_0 \quad \underbrace{\alpha_0\ \beta_0\ V_0}_{\text{Initial velocity}} \quad \underbrace{a_1\ a_2\ a_3\ a_4\ a_5}_{\text{Chebyshev coefficients } \alpha} \quad \underbrace{b_1\ b_2\ b_3\ b_4\ b_5}_{\text{Chebyshev coefficients } \beta}$$
$$\underbrace{c_1\ c_2\ c_3\ c_4\ c_5}_{\text{Chebyshev coefficients } T} \quad \underbrace{t_1\ t_2\ t_3\ t_4\ t_5}_{\text{Switching moments}} \Big]$$

For the subsequent legs, the parameters describing the initial velocity can be left out, so only 21 terms are left to optimize in that case.

9.7 Constraint handling

GTOC3 was quite difficult to optimize; during the first attempts in 2007 the big issue was in satisfying the constraints for rendezvous. Because it was impossible to find solutions satisfying these constraints, use was made of augmented objective functions. This way, non-satisfying solutions are allowed to exist, but selective pressure tries to improve the solutions so that the violations are reduced from the solution.

The violations of the individuals are given as Δr and Δv, which are the position and velocity of the spacecraft relative to the target body, at the end of the leg. These are satisfied if $\Delta r < 1000$ [km] and $\Delta v < 0.001$ [km s^{-1}].

Several methods to augment the fitness function with these violations have been tested; the simplest one is a weighted sum of the deviations (with static weights):

$$\mathcal{F}(i) = f(i) + w_1 \max(\Delta r - 1000, 0) + w_2 \max(\Delta v - 0.001, 0) \tag{9.18}$$

The second one used the Euclidian distance of the (scaled) deviations:

$$\mathcal{F}(i) = f(i) + \sqrt{(w_1 \max(\Delta r - 1000, 0))^2 + (w_2 \max(\Delta v - 0.001, 0))^2} \tag{9.19}$$

Also, a product-form was used:

$$\mathcal{F}(i) = f(i) + (1 + w_1 \max(\Delta r - 1000, 0)) * (1 + w_2 \max(\Delta v - 0.001, 0)) \tag{9.20}$$

These three forms did not lead to satisfying results: in all three cases, it was impossible to find a solution satisfying either of these constraints.

For that reason, the optimization was also done using multi-objective optimization (MOO). Both violations were considered as objectives, and the optimizer was allowed to run with this, so the augmented objective functions are given by:

$$\vec{\mathcal{F}}(i) = \begin{pmatrix} f(i) \\ \max(\Delta r - 1000, 0) \\ \max(\Delta v - 0.001, 0) \end{pmatrix} \tag{9.21}$$

This also did not result in satisfying solutions, but the results where found using less function evaluations.

Note that this MOO variant cannot be used for local optimization using Powell's method, because an actual fitness value is used. In this case, the third formulation was used, with $w_1 = 1$ and $w_2 = 1000000$. This resulted in a solutions which satisfied the velocity constraint, and still had a position error of approximately $160 \cdot 10^3$[km]. As an experiment, $w_2 = 0$ was also optimized, and this did yield a solution which satisfied the position constraint, but the velocity error increased from 40.1 to 46.4ms^{-1}. For that reason, consecutive local optimization runs for each constraint where not considered.

9.8 Optimization

9.8.1 Global optimization

The global optimization was done using the OPTIDUS implementation of DE, using the scheme Tasoulis6. As a first attempt, only the first leg of the proposed trajectory was done, from the Earth to asteroid 88, the first asteroid targeted by the highest ranking solution not making use of gravity assists.

The population size was set to 100 individuals, and the population was initialized with 100 randomly chosen solutions, allowed to violate the rendezvous distance constraint by $3 \cdot 10^6$.

However, after numerous attempts, no solution matching the constraints were found using DE. The best solutions that were found violated one or both constraints. The best solution (in terms of augmented fitness value) is given in table 9.3.

9.8.2 Local optimization

Only at this moment, a problem in the code was found; because an adaptive stepsize was used, the simulation flew past the asteroid at the end of the run. This was fixed by testing if the end of the simulation was closer than the previous stepsize, and if so, manually calling a non-adaptive version of the integrator.

After applying this fix, no time was left for doing another DE run. However, the best global solution was optimized using Powell's method. This was done three times: once for only satisfying the positional constraints, once for satisfying the velocity constraint, and once

Parameter	Value found by DE	Powell 1	Powell2	Powell 3
t_0 [mjd]	59456.68	59455.72	59456.83	59456.82
α_0 [rad]	-0.31336	-0.22437	-0.32780	-0.32402
β_0 [rad]	1.39964	1.39942	1.39957	1.39961
V_0 [km/s]	0.366	0.36465	0.36627	0.36627
a_1 [-]	0.79943	0.79613	0.79868	0.79919
a_2 [-]	-0.58037	-0.58037	-0.58039	-0.58038
a_3 [-]	-0.41105	-0.41104	-0.41155	-0.41111
a_4 [-]	0.60793	0.60787	0.60793	0.60793
a_5 [-]	0.52894	0.52587	0.52594	0.52594
b_1 [-]	-0.57154	-0.56802	-0.57161	-0.57154
b_2 [-]	0.74460	0.75138	0.74343	0.74350
b_3 [-]	-0.23721	-0.23266	-0.23810	-0.23753
b_4 [-]	-0.86592	-0.86839	-0.86597	-0.86592
b_5 [-]	-0.85234	-0.87357	-0.85282	-0.85234
th_1 [-]	0.54810	0.54859	0.54795	0.54810
th_2 [-]	0.86279	0.85368	0.86149	0.85698
th_3 [-]	0.12228	0.11981	0.12215	0.12228
th_4 [-]	-0.07070	-0.06932	-0.07108	-0.07068
th_5 [-]	0.02496	0.02658	0.02458	0.02496
t_1 [days]	192.89	193.23	192.82	192.89
t_2 [days]	19.16	19.12	19.18	19.17
t_3 [days]	177.63	177.63	177.63	177.63
t_4 [days]	21.28	21.28	21.28	21.28
t_5 [days]	27.94	27.94	27.94	27.94
n_{eval}	-	7867	3031	12980
$m_i - m_f$ [kg]	69.915	69.649	69.881	69.712
Δr [km]	157598	< 1000	169131	159577
Δv [m/s]	40.1	46.4	< 1	< 1

Table 9.3 Solution vector, number of evaluations, fuel consumption and constraint violation of the best solution from the global and local optimizations for GTOC3.

for satisfying both constraints. (This was done by ignoring the constraint violations not to be satisfied in the augmented objective function.)

The result of these optimizations can also be found in table 9.3. As can be seen, optimizing for one of both constraints was successful, yet trying to match both constraints did not succeed.

The trajectory of the solution found using the global optimizer and the solution from Powell 3 are plotted in figures 9.8 and 9.9.

Note that the local optimization only considered 1 individual. The fuel consumption (which is the term influencing the fitness function) is not changed considerably by the local optimization. For that reason, doing a global optimization with loosened constraints and assuming that a local optimizer will find a solution which does satisfy the original constraints seems to be a working method.

Unfortunately, only the final mass (after the complete mission has ran) has been published, so it is not possible to compare the found result to the competition.

9.9 Conclusions and recommendations for gtoc3

Conclusions:

- A global optimizer using DE is unable to satisfy the constraints for the GTOC3 mission when modelling the thrust profile using Chebyshev polynomials and on/off switching.

- Using MOO to optimize the satisfaction of the constraints for the GTOC3 mission yields better results than augmented fitness functions which are weighted sums.

- A local optimizer improves the violations of the constraints, but still was unable to satisfy both constraints of the first leg of the GTOC3 mission.

- For the GTOC3 mission, running a local optimizer does not have a significant impact on the original fitness value, or the values inside the solution vector. Therefore, doing a global optimization with loosened constraints to explore the solution space is a good strategy, as the found solutions can be patched to satisfy the constraints better.

Recommendations:

- Investigate whether better solutions can be acquired by having the legs consist of multiple thrusting arcs, each with separated Chebyshev polynomials.

- Investigate other thrust modeling options; Chebyshev polynomials of 5th order might not give enough control. Also, increasing the number of on/off switching moments might improve the optimization.

- Investigate if alternative thrust modelling options give better results.

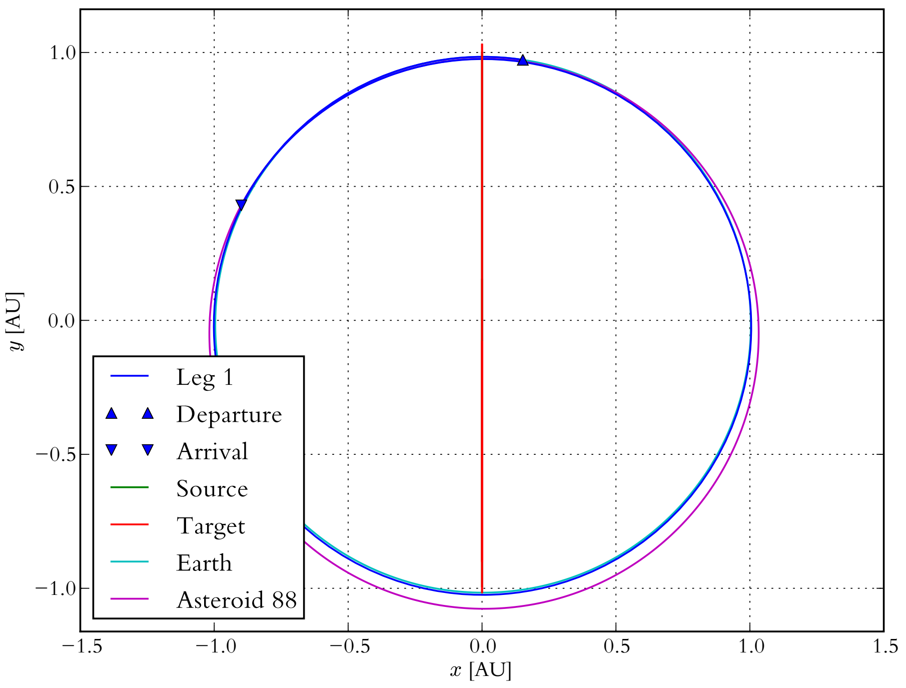


Figure 9.8 Optimal trajectory of the first leg for GTOC3 found using DE.

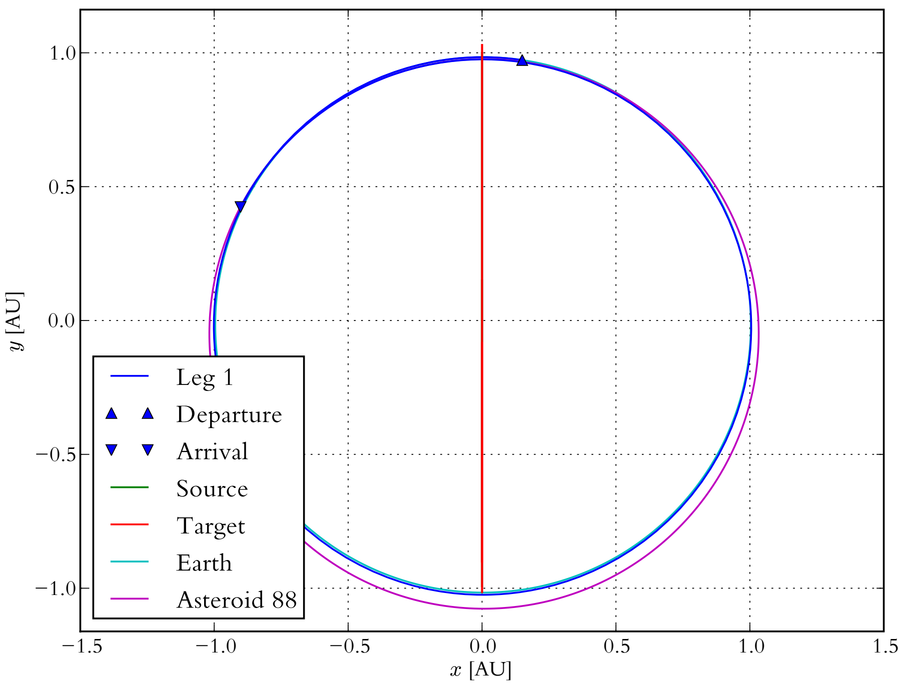


Figure 9.9 Optimal trajectory of the first leg for GTOC3 found using DE and a successive local optimization using Powell's method.

Chapter 10

Conclusions and recommendations

10.1 Fulfillment of objectives

As stated in the introduction, the objectives of this thesis work are:

1. This thesis work will try to improve the performance of OPTIDUS by implementing PSO and DE.

2. By making use of parallel computation, calculation time on systems with multiple cores can be reduced.

3. This improvement will be benchmarked against historical results of the Solar polar sailing mission from [*Garot*, 2006].

4. Also, an attempt will be made to find a solution to the third GTOC problem which does satisfy the constraints.

The first three objectives have been fulfilled; the final objective has been fulfilled partially.

10.2 Conclusions

Regarding the first objective, a new version of OPTIDUS has been developed, which is capable of doing PSO and DE optimizations, both using single as well as multi objective functions. There is also support to do local optimizations using Powell's quadratically convergent method.

Regarding the second objective, the use of parallel programming vastly improves the performance of the optimizer; by using the OpenMP library, it was possible to write an optimizer in which the number of objective function evaluations per second scales almost linearly with the amount of cores allocated. This does come at a cost: the objective function must be written in the form of *pure* functions, which do not have global state. This might pose a strain on the programmer.

Regarding the third objective, improving the optimization of the solar polar sailing mission, the following can be concluded:

- Using particle swarm optimization to optimize the geocentric phase yielded the best results of all optimizers.

- A 4^{th} order sine polynomial can be used as a fitting function.

- The heliocentric phase was better optimized using differential evolution.
- Increasing population sizes while keeping the maximum number of iterations the same tends to improve the results of all optimizers.
- Local optimization using random sampling is not sensitive to the random generator used: solutions sampled using numbers from a uniform random number generators and solutions sampled using numbers from Sobol sequences where rather similar.

And for GTOC3, the following:

- A global optimizer using DE is unable to satisfy the constraints for the GTOC3 mission when modelling the thrust profile using Chebyshev polynomials and on/off switching.
- Using MOO to optimize the satisfaction of the constraints for the GTOC3 mission yields better results than augmented fitness functions which are weighted sums.
- A local optimizer improves the violations of the constraints, but still was unable to satisfy both constraints of the first leg of the GTOC3 mission.
- For the GTOC3 mission, running a local optimizer does not have a significant impact on the original fitness value, or the values inside the solution vector. Therefore, doing a global optimization with loosened constraints to explore the solution space is a good strategy, as the found solutions can be patched to satisfy the constraints better.

10.3 Recommendations

Regarding the first objective, implementing SA and hybrid optimizations might improve the software package even further. Making these methods available in the form of a C++ library will also prove helpful.

Regarding the second objective, it is interesting to investigate if the use of the FORALL statement available in Fortran95 instead of OpenMP is easier on the programmer; by having the compiler check for purity of functions, potential bugs can be identified. Support in the GCC Fortran compiler is bad, but other compilers might give much better results.

Regarding the Solar polar sailing mission:

- Right now, the maximum population size used was 212 individuals. What happens if this number is increased even further?
- Can applying a local optimizer like Powell's method be applied as a final step?
- Can the solution be improved by not using a polynomial fit for the pitch angles in the geocentric phase, and instead letting them be optimized as independent variables?

Recommendations regarding GTOC3:

- Investigate whether better solutions can be acquired by having the legs consist of multiple thrusting arcs, each with separated Chebyshev polynomials.
- Investigate other thrust modeling options; Chebyshev polynomials of 5th order might not give enough control. Also, increasing the number of on/off switching moments might improve the optimization.
- Investigate if alternative thrust modelling options give better results.

And a last but not least, a final word of advice: do not start working at your job before all thesis work has been done, as this has a really bad impact on the final convergence speed of your thesis.

Bibliography

Advanced Concepts Team (2005), GTOC1: Competition Announcement, Available from World Wide Web: http://www.esa.int/gsp/ACT/mad/pp/GTOC1/gtoc1ann.htm [cited 2009-01-06].

Almering, J.H.J. (1993), *Analyse*, Delftse Uitgevers Maatschappij b.v., 6th edition.

Bäck, Thomas (2000), *Binary strings*, chapter 15, Volume 1 : Basic Algorithms and Operators of Bäck et al. [2000*a*].

Bäck, Thomas, David B. Fogel, and Zibgniew Michalewicz (eds.) (1997), *Handbook of Evolutionary Computation*, IOP Publishing Ltd. Bristol, UK.

Bäck, Thomas, David B. Fogel, and Zibgniew Michalewicz (eds.) (2000*a*), *Evolutionary Computation*, vol. 1 : Basic Algorithms and Operators, IOP Publishing, Bristol, UK.

Bäck, Thomas, David B. Fogel, and Zibgniew Michalewicz (eds.) (2000*b*), *Evolutionary Computation*, vol. 2 : Advanced Algorithms and Operators, IOP Publishing, Bristol, UK.

Baker, James E. (1985), Adaptive selection methods for genetic algorithms, in *Proceedings of the 1st International Conference on Genetic Algorithms*, pp. 101–111, Lawrence Erlbaum Associates, Inc., Mahwah, NJ, USA.

Baker, James E. (1987), Reducing bias and inefficiency in the selection algorithm, in *Proceedings of the Second International Conference on Genetic Algorithms on Genetic algorithms and their application*, pp. 14–21, L. Erlbaum Associates Inc., Hillsdale, NJ, USA.

Baluja, Shumeet, and Rich Caruana (1995), Removing the genetics from the standard genetic algorithm, in *The Int. Conf. on Machine Learning 1995*, edited by A. Prieditis and S. Russel, pp. 38–46, Morgan Kaufmann Publishers, San Mateo, CA, USA.

Becerra, V.M., D.R. Myatt, S.J. Nasuto, J.M. Bishop, and D. Izzo (2005), An Efficient Pruning Technique for the Global Optimisation of Multiple Gravity Assist Trajectories, in *Proceedings of GO 2005*, pp. 1–7.

Bertrand, Régis (2009), Available from World Wide Web: http://cct.cnes.fr/cct02/gtoc4/index.htm [cited 2009-07-29].

Burkardt, John (2006), SOBOL, The Sobol Quasirandom Sequence, Available from World Wide Web: https://people.sc.fsu.edu/~burkardt/f_src/sobol/sobol.html [cited 2009-03-20].

Caprani, Ole, Kaj Madsen, and Hans Bruun Nielsen (2002), Introduction to interval analysis, Found online at http://www2.imm.dtu.dk/pubdb/views/edoc_download.php/1462/pdf/imm1462.pdf.

Casalino, Lorenzo (2007), 3rd Global Trajectory Optimisation Competition, Available from World Wide Web: `http://www2.polito.it/eventi/gtoc3/` [cited 2009-05-22].

Casalino, Lorenzo, Guido Colasurdo, and Matteo Rosa Sentinella (2007), Problem Description for the 3rd Global Trajectory Optimisation Competition, Available from World Wide Web: `http://www2.polito.it/eventi/gtoc3/gtoc3_problem.pdf` [cited 2009-01-06].

Chapman, B., G. Jost, R. Van Der Pas, and D.J. Kuck (2007), *Using OpenMP: portable shared memory parallel programming*, The MIT Press.

Chu, Weiwei (2007), Interval Analysis Applied to Re-Entry Flight Trajctory Optimization, Master's thesis, Delft University of Technology, Delft, NL.

Clerc, Maurice (2008*a*), Standard PSO 2007, Available from World Wide Web: `http://particleswarm.info/standard_pso_2007.c`.

Clerc, Maurice (2008*b*), Why does it work?, *International Journal of Computational Intelligence Research*, *4*(2), 79–91.

Clerc, Maurice, and James Kennedy (2002), The particle swarm - explosion, stability, and convergence in a multidimensional complex space, *IEEE Trans. Evolutionary Computation*, *6*(1), 58–73.

De Jong, K.A. (1975), *An Analysis of the Behavior of a Class of Genetic Adaptive Systems*, Ph.D. dissertation, University of Michigan, MI, USA.

Deb, K., and R.B. Agrawal (1995), Simulated binary crossover for continuous search space, *Complex Systems*, *9*(2), 115–148.

Deb, Kalyanmoy (1995), *Optimization for Engineering Design, Algorithms and Examples*, Prentice Hall of India, New Delhi, India.

Deb, Kalyanmoy (1999), Multi-objective genetic algorithms: Problem difficulties and construction of test problems, *Evolutionary Computation*, *7*(3), 205–230.

Deb, Kalyanmoy (2001), Genetic algorithms for optimization, *2001002*, Kanpur Genetic Algorithms Laboratory, Kampur, India.

Deb, Kalyanmoy, Samir Agrawal, Amrit Pratap, and T. Meyarivan (2001), A Fast Elitist Non-Dominated Sorting Genetic Algorithm for Multi-Objective Optimization: NSGA-II, *2000001*, Kanpur Genetic Algorithms Laboratory, Kampur, India.

Demeyer, Jacob (2007), Orbital Design for a Formation Flying Mission in the Distant Retrograde Orbits, Master's thesis, Delft University of Technology, Delft, NL.

Eiben, A.E., C.H.M. van Kemenade, and J.N. Kok (1995), *Orgy in the Computer: Multi-parent Reproduction in Genetic Algorithms*, Centrum voor Wiskunde en Informatica, Amsterdam, Netherlands.

Elvik, Sander (2004), Optimization of a low-energy transfer to Mars using Dynamical Systems Theory and low-thrust propulsion, Master's thesis, Delft University of Technology, Delft, NL.

ESA (2004), Electric spacecraft propulsion, Available from World Wide Web: `http://sci.esa.int/science-e/www/object/index.cfm?fobjectid=34201` [cited 2009-05-13].

Evertsz, Chantale (2008), GTOC2: Multiple asteroid rendezvous, Master's thesis, Delft University of Technology, Delft, NL.

Fan, H.Y., and J. Lampinen (2003), A trigonometric mutation operation to differential evolution, *Journal of Global Optimization*, *27*(1), 105–129.

Foekema, R.T. (2004), Solar sail spacecraft to Pluto, Master's thesis, Delft University of Technology, Delft, NL.

Fogel, David B., and Peter J. Angeline (2000), *Guidelines for a suitable encoding*, chapter 20, Volume 1 : Basic Algorithms and Operators of Bäck et al. [2000*a*].

Fogel, L.J., A.J. Owens, and M.J. Walsh (1966), *Artificial intelligence through simulated evolution*, John Wiley & Sons Inc.

Garot, Danielle (2006), Trajectory optimisation of a solar polar sail mission, Master's thesis, Delft University of Technology, Delft, NL.

Gnuplot (2009), *Gnuplot homepage*, Available from World Wide Web: `http://www.gnuplot.info/` [cited 2009-07-16].

Goldberg, David E. (1989), *Genetic algorithms in search, optimization, and machine learning*, Addison Wesley, Reading, MA, USA.

Grefenstette, JJ (1986), Optimization of control parameters for genetic algorithms, *IEEE Transactions on Systems, Man and Cybernetics*, *16*(1), 122–128.

Griewank, Andreas O. (1981), Generalized descent for global optimization, *Journal of Optimization Theory and Applications*, *34*(1), 11–39.

Heitkoetter, Joerg, and David Beasley (2001), The Hitch-Hiker's Guide to Evolutionary Computation: A list of Frequently Asked Questions (FAQ), *USENET: comp.ai.genetic*, Retrieved on 2007-09-17 from `http://faqs.cs.uu.nl/na-dir/ai-faq/genetic/part1.html`.

Holland, J.H. (1967), Nonlinear environments permitting efficient adaptation, in *Computer and Information Sciences, II: Proceedings*, p. 147, Academy Press.

Joines, JA, and CR Houck (1994), On the use of non-stationary penalty functions to solve nonlinearconstrained optimization problems with GA's, in *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on*, pp. 579–584.

JPL, NASA (2006), Nasa - searching for the grandest asteroid tour, Available from World Wide Web: `http://www.nasa.gov/vision/universe/solarsystem/asteroidf-20070404.html` [cited 2009-05-11].

Kennedy, James, and Russell C. Eberhart (1995), Particle swarm optimization, in *Proc. IEEE International Conference on Neural Networks*, pp. 1942–1948.

Kirkpatrick, S., C.D. Gelatt Jr., and M.P. Vecchi (1983), Optimization by simulated annealing, *Science*, *220*(4598), 671–680.

Lu, Ping, and M. Asif Khan (1994), Nonsmooth Trajectory Optimization: An Approach Using Continuous Simulated Annealing, *Journal of Guidance, Control and Dynamics*, *17*(4), 685–691.

McInnes, C.R. (1999), *Solar Sailing: Technology, Dynamics and Mission Applications*, Springer-Praxis Publishing.

Melman, Jeroen (2007), Trajectory Optimization for a Mission to Neptune and Triton, Master's thesis, Delft University of Technology, Delft, NL.

Michalewicz, Z. (1995), A Survey of Constraint Handling Techniques in Evolutionary Computation Methods, *Evolutionary Programming IV: Proceedings of the Fourth Annual Conference on Evolutionary Programming*.

Michalewicz, Zbigniew (1996), *Genetic algorithms + data structures = evolution programs*, Springer-Verlag, 2nd edition.

Nam, Dongkyung, and Cheol Hoon Park (2000), Multiobjective Simulated Annealing: A Comparative Study to Evolutionary Algorithms, *International Journal of Fuzzy Systems*, *2*(2), 87–97.

Nam, Dongkyung, and Cheol Hoon Park (2002), Pareto-Based Cost Simulated Annealing for Multiobjective Optimization, in *Proceedings of the 4th Asia-Pacific Conference on Simulated Evolution and Learning (SEAL'02)*, vol. 2, edited by Lipo Wang *et al.*, pp. 522–526, Nanyang Technical University, Orchid Country Club, Singapore, November 2002.

Noomen, Ron (2007), Space mission design: Optimization, Slides from the course AE4-878 at Delft University of Technology.

Orlando, G., C. Evertsz, J. Heiligers, K. Kumar, J. Melman, R. Oldenhuis, T.Paulino, R. Prakash, and C.J. Spaans (2007), GTOC3 Solution by Delft University of Technology - Space Trajectories Advanced Research by Students (STARS) Team , -.

O'Sullivan, Bryan, Donald Bruce Stewart, and John Goerzen (2008), Real World Haskell, Available from World Wide Web: `http://book.realworldhaskell.org/`.

Patterson, David A. (2006), Computer science education in the 21$^{\text{st}}$ century, *Commun. ACM*, *49*(3), 27–30.

Pols, C.L. van der (2006), Halo Orbit Station Keeping, Master's thesis, Delft University of Technology, Delft, NL.

Press, William H., Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery (2007), *Numerical Recipes: The Art of Scientific Computing*, Cambridge University Press, Cambridge, UK, third edition, eBook edition.

Radcliffe, N.J. (1991), Equivalence class analysis of genetic algorithms, *Complex Systems*, 5(2), 183–205.

Safipour, Ebrahim (2007), Trajectory Optimization for a Mission to Neptune and Triton, Master's thesis, Delft University of Technology, Delft, NL.

Schaffer, J. David, Rich Caruana, Larry J. Eshelman, and Rajarshi Das (1989), A study of control parameters affecting online performance of genetic algorithms for function optimization, in *ICGA*, edited by J. David Schaffer, pp. 51–60, Morgan Kaufmann.

Schneier, B., *et al.* (1996), *Applied cryptography: protocols, algorithms, and source code in C*, Wiley New York.

Schwefel, H.P. (1977), Numerische Optimierung von Computer-Modellen mittels der Evolutionsstrategie, volume 26 of Interdisciplinary systems research.

Shi, Yuhui, and Russell C. Eberhart (1998*a*), Parameter selection in particle swarm optimization, in *Evolutionary Programming*, vol. 1447 of *Lecture Notes in Computer Science*, edited by V. William Porto *et al.*, pp. 591–600, Springer.

Shi, Yuhui, and Russell C. Eberhart (1998*b*), Modified particle swarm optimizer, *The 1998 IEEE International Conference on Evolutionary Computation, ICEC'98*, pp. 69–73.

Silva, Arlindo, Ana Neves, and Ernesto Costa (2002), Chasing the Swarm: a Predator-Prey Approach to Function Optimisation, *Proceedings of the MENDEL2002—-8th International Conference on Soft Computing. Brno, Czech Republic.*

Silva, Arlindo, Ana Neves, and Ernesto Costa (2003), Sappo: A simple, adaptable, predator prey optimiser, in *EPIA*, vol. 2902 of *Lecture Notes in Computer Science*, edited by Fernando Moura-Pires and Salvador Abreu, pp. 59–73, Springer.

Spaans, Jasper (2008), Measuring rotary position using compact disc technology, Available from World Wide Web: `https://secure.jasper.es/svn/studie/Completed/ae4-s02/essay/essay.pdf`, Essay for the course AE4-S02 Spacecraft Mechatronics at Delft University of Technology.

Storn, Rainer, and Kenneth Price (1997), Differential evolution–a simple and efficient heuristic for global optimization over continuous spaces, *Journal of Global Optimization*, *11*(4), 341–359.

Syswerda, G. (1989), Uniform crossover in genetic algorithms, *Proceedings of the third international conference on Genetic algorithms*, pp. 2–9.

Tan, Kay Chen, Eik Fun Khor, Tong Heng Lee, and Y. J. Yang (2003), A tabu-based exploratory evolutionary algorithm for multiobjective optimization, *Artif. Intell. Rev.*, *19*(3), 231–260.

Tasoulis, D.K., N.G. Pavlidis, V.P. Plagianakos, and M.N. Vrahatis (2004), Parallel differential evolution, *Evolutionary Computation, 2004. CEC2004. Congress on*, *2*.

Tweakers.net (2008), Tweakers.net Best Buy Guide: editie juli 2008, Available from World Wide Web: `http://tweakers.net/reviews/917/tweakers-punt-net-best-buy-guide-editie-juli-2008.html` [cited 2008-08-05].

Vinkó, T., D. Izzo, and C. Bombardelli (2007), Benchmarking different global optimisation techniques for preliminary space trajectory design, in *58th International Astronautical Congress, International Astronautical Federation (IAF).*

Vuik, C., P. van Beek, F. Vermolen, and J. van Kan (2006), *Numerieke Methoden voor Differentiaalvergelijkingen*, VSSD, Delft, NL.

Wertz, J.R., and W.J. Larson (1999), Space mission analysis and design, *Space mission analysis and design, by Wertz, James Richard.; Larson, Wiley J. El Segundo, Calif.: Microcosm Press; Dordrecht; Boston: Kluwer, c1999. Space technology library, 8.*

Whitley, D., *et al.* (1989), The GENITOR algorithm and selection pressure: Why rank-based allocation of reproductive trials is best, *Proceedings of the Third International Conference on Genetic Algorithms*, *1*, 116–121.

Whitley, Darrell (1997), *Permutations*, chapter C1.7, in Bäck et al. [1997].

Whitley, Darrell, V. Scott Gordon, and A. P. Willem Böhm (1997), *Knapsack problems*, chapter G9.7, in Bäck et al. [1997].

Wikipedia (2008*a*), CPU power dissipation — Wikipedia, The Free Encyclopedia, Available from World Wide Web: `http://en.wikipedia.org/w/index.php?title=CPU_power_dissipation&oldid=256265466` [cited 2008-12-10].

Wikipedia (2008*b*), Himmelblau's function — Wikipedia, The Free Encyclopedia, Available from World Wide Web: `http://en.wikipedia.org/w/index.php?title=Himmelblau%27s_function&oldid=233230231` [cited 2008-10-07].

Yam, Chit Hong, and James M. Longuski (2006), Reduced Parameterization for Optimization of Low-Thrust Gravity-Assist Trajectories: Case Studies, *AIAA/AAS Astrodynamics Specialist Conference and Exhibit*, AIAA 2006-6744.

Zandbergen, B.T.C. (1997), Inleiding ruimtevaarttechniek, Lecture notes for the course LR2-6 at Delft University of Technology.

Appendix A
Test functions

This chapter lists some of the test functions encountered in other literature, but not used directly in this thesis work.

A.1 De Jong's functions

The De Jong functions are given in [*De Jong*, 1975, appendix A] and form a set of 5 functions. They are considered a standard set of test functions, and are used throughout the literature as such. An examples of usage includes [*Storn and Price*, 1997]. They are defined as follows:

A.1.1 De Jong F1

The F1 function is a 3-dimensional parabola, and as such continuous, convex, unimodal and low-dimensional. It has a mininum of 0 at the origin.

$$f_1 = \sum_{i=1}^{3} x_i^2 \tag{A.1}$$

with boundaries $-5.12 \leq x_i \leq 5.12$ and resolution $\Delta x_i = 0.01$.

A.1.2 De Jong F2, aka Rosenbrock

The F2 function is a 2-dimensional function that is continuous, non-convex, unimodal. It has a minimum of 0 at (1,1). This function is also appears as the Rosenbrock function in the literature.

$$f_2 = 100(x_1^2 - x_2)^2 + (1 - x_1)^2 \tag{A.2}$$

with boundaries $-2.048 \leq x_i \leq 2.048$ and resolution $\Delta x_i = 0.001$.

A.1.3 De Jong F3

The F3 function is a 5-dimensional, piece-wise constant, discontinuous unimodal defined as follows:[1]

$$f_3 = \sum_{i=1}^{5} \text{floor}(x_i) \tag{A.3}$$

with boundaries $-5.12 \leq x_i \leq 5.12$ and resolution $\Delta x_i = 0.01$.

[1] [*De Jong*, 1975] uses the notation $[x_i]$ to denote the operator that represents the greatest integer less than or equal to x_i. In modern programming languages, this function is named floor, which is the notation used in this report.

A.1.4 De Jong F4

The F4 function is a 30-dimensional continuous, convex, unimodal function with added Gaussian noise, defined as:

$$f_4 = \sum_{i=1}^{30} i x_i^4 + \text{gauss}(0,1) \tag{A.4}$$

with boundaries $-1.28 \leq x_i \leq 1.28$ and resolution $\Delta x_i = 0.01$.

A.1.5 De Jong F4, modified

[*Storn and Price*, 1997] gives a modified version of this function, in which the Gaussian random noise is replaced by a random variable from a uniform function, and this noise is then placed inside the summation loop, so the modified f_4 then looks like:

$$f_{4m} = \sum_{i=1}^{30} (i x_i^4 + \text{uniform}(0,1)) \tag{A.5}$$

A.1.6 De Jong F5

The F5 function is a 2-dimensional continuous function with 25 local minima. The global minimum is found at coordinates (a_{11}, a_{21}). The function is defined by:

$$\begin{aligned} \frac{1}{f_5} &= \tfrac{1}{K} + \textstyle\sum_{j=1}^{25} \tfrac{1}{f_j(x)} \\ \text{where} \quad f_j(x) &= c_j + \textstyle\sum_{i=1}^{2} (x_i - a_{ij})^6 \end{aligned} \tag{A.6}$$

where the constants a_{ij} are given by the elements of matrix $\mathbf{A}$:

$$\mathbf{A} = \begin{bmatrix} -32 & -16 & 0 & 16 & 32 & -32 & -16 \ldots & 0 & 16 & 32 \\ -32 & -32 & -32 & -32 & -32 & -16 & -16 \ldots & 32 & 32 & 32 \end{bmatrix}$$

and $c_j = j$ and $K = 500$.
The boundaries are given by $-65.536 \leq x_i \leq 65.536$ and the suggested resolution is given as $\Delta x_i = 0.001$.

A.2 Corana's parabola

This function is a bimodal function with 4 inputs. It is defined as:

$$f = \sum_{j=0}^{3} \begin{cases} 0.15(z_j - 0.05\,\text{sgn}(z_j))^2 d_j & \text{if } |x_j - z_j| < 0.05 \\ d_j x_j^2 & \text{otherwise} \end{cases} \tag{A.7}$$

where sgn is the sign function,[2] and

$$z_j = 0.2\,\text{sgn}(x_j) \left[\left| \frac{x_j}{0.2} \right| + 0.49999 \right]$$

and

$$d_j = 1, 1000, 10, 100$$

and boundaries $x_j \in [-1000, 1000]$ for $j = 0,1,2,3$. It has a lot of local minima near the global minimum, so downhill optimizers will get captured in those holes.

The minimum $f_6 = 0$, when $|x_j| < 0.05$. This function is mentioned in [*Storn and Price*, 1997].

[2] The sign function maps negative values to -1, is 0 for 0 and maps positive numbers to 1

A.3 Zimmermann's problem

Zimmermann's problem is defined in [*Storn and Price*, 1997] as finding the minimum for the function:

$$f(x) = 9 - x_0 - x_1 \tag{A.8}$$

constrained by

$$(x_0 - 3)^2 + (x_1 - 2)^2 \leq 16 \tag{A.9}$$

$$x_0 x_1 \leq 14 \tag{A.10}$$

$$x_0, x_1 > 0 \tag{A.11}$$

The optimum is located at a corner of the (constrained) search space, namely $f(7,2) = 0$. The fact that this is at the corner of a search space, makes it difficult.

Appendix *B*

Numerical approximation of derivatives

The calculation of numerical derivatives is a delicate problem, since (machine) rounding errors occur if the stepsize is too small, and no real derivatives are calculated if the stepsize is too big.

According to the listing in [*Press et al.*, 2007, section 10.9.1], the stepsize should be equal to the square of the machine precision. Then, the gradient of a function can be calculated as in listing B.1.

```
EPS = 1e-8  # sqrt(machine precision), for IEEE-754 doubles
def num_deriv(x, func):
    # calculate the locations of the points x+h and x-h explicitly ,
    # so we can divide by their difference

    h = max(EPS * abs(temp), EPS)
    xplus = x + h
    xminus = x - h

    fplus = func(xplus)
    fminus = func(xminus)

    return (fplus - fminus) / (xplus - xminus)
```

Listing B.1 Calculating the gradient numerically in Python

Appendix C
Parallel Processing in Fortran using OpenMP

The frequency at which people predict the end of Moore's law doubles every 18 months. – Lerc[1]

This chapter contains some theoretical backgrounds of parallel programming, and practical guidelines for writing Fortran algorithms that can be ran on more than one processor. Making use of more than one processor can give a large performance boost for population-based evolutionary optimization.

C.1 What is parallel processing, and why is it needed?

Parallel processing is programming technique to make use of more than one processor at the same time. In general, this is done by splitting the workload into several work units, and then distributing them over the available processors.

Recent processors come in the *dual core* variant: they contain more than one *core*. Each core in a processor is able to process instructions independently of the other core, so a processor with more 2 cores can run 2 programs at the same time.[2]

If 2 cores is not enough, *quad core* chips are also available. In these chips, 2 processors are combined on one piece of silicon, giving a total of 4 cores.[3]

Finally, if that still is not enough power, it is possible to put more than 1 chip inside a computer, or to build a *cluster*: a network of several computers which are used to perform calculations. No use of clusters was made for this thesis work.

[*Patterson*, 2006] lists some of the reasons why parallel processing is necessary. The most important one is that the limits for power dissipation per chip have been reached.

Because power dissipation is roughly proportional to the square of the operating frequency of a processor, it is desirable to lower this frequency. However, this also lowers the amount of operations which can be performed. If one wants to keep the same amount or grow the number of operations, this means more processors are needed.

In numbers: the number of operations per second (OPS) by a processor with c cores running at a frequency f roughly equals

$$\text{OPS} = cf/k_1 \tag{C.1}$$

[1]Found on `http://www.reddit.com/r/programming/comments/8i9h7/someone_has_told_me_that_this_may_take_several/c09e9z0`

[2]The two cores do share cache memory, and memory bandwidth, so for memory intensive workloads (like video encoding or encryption) the performance gain will probably be less than a factor two.

[3]These quad core chips really contain two processors, each with its own cache memory. Memory bandwidth is shared between the processors though.

in which k_1 is a parameter which represents the number of clock cycles used per operation. Advances in chip technology such as increasing the cache size will make this number smaller.

One of the metrics of processor power usage is the "speed-power ratio", $r_{s/p}$, which is usually given in [MHz/W]. The total dissipated power (TDP) for a processor operating at a frequency f is given by:

$$\text{TDP} = cfr_{s/p} \tag{C.2}$$

[*Wikipedia*, 2008*a*] lists values for $r_{s/p}$ for a wide range of processors. As can be seen, on that page, a modern chip of the Core 2 Duo variant has a $r_{s/p}$ of approximately 100, whereas an older chip, the classical Pentium, only reaches 10. This increase can be attributed to improvements in chip design and manufacturing.

However, a Core 2 Quad chip running at the same frequency as the Core 2 Duo can reach up to 130.[4] This means that it can execute 30% more instructions at the same heat cost. This shows that the best way to improve processing power is by going the multi-core way.

At the time of writing (Q3 2008), according to [*Tweakers.net*, 2008], a standard system has a single processor with 2 cores, and systems with a single processor with 4 cores are becoming common. In high end workstations, two processors with 4 cores each can be fitted, giving a total of 8 cores available for performing calculations. Thus, the trend is clearly to increase the number of cores avaible, instead of increasing the processing speed of individual cores. Not making use of these multi-core systems would be a big waste of resources.

C.2 Multithreaded Fortran using OpenMP

One programming interface to parallel processing in Fortran is available in most compilers, namely OpenMP. OpenMP is an open industry standard and is widely supported: it can be used with (amongst others) gfortran[5], IBM's XL Fortran compiler[6] and the Intel Fortran compiler[7]. The homepage for OpenMP is the website `http://openmp.org/`, where the standard and reference documentation can be found.

When a problem is to be run in parallel, it is necessary to split the problem into several independent sub-problems, which can then be solved in parallel. An example can be found in the PSO implementation: adjusting the speed and position of each particle and the subsequent calculation of the fitness is only dependent on the particle itself and the position of the best particle in the swarm ever, which does not change during a generation.

This means that each particle forms its an independent work packet. By distributing these work packets over the cores, the work can be executed in parallel, leading to substantial reduction of the time needed to run an optimization.

A trivial example of a loop which can be run in parallel is given here:

```
PROGRAM demo_multithreading
  !$OMP PARALLEL DO
  DO i = 1, 4
    WRITE (*,"(I2)"), i
  END DO
  !$OMP END PARALLEL DO
```

[4]The cited values for the C2D/C2Q chip are from the Wolfdale-6M(E0) chip family; this means they share the same chip design and manufacturing process.

[5]`http://gcc.gnu.org/fortran/`

[6]`http://www.ibm.com/software/awdtools/fortran/`

[7]`http://www.intel.com/cd/software/products/asmo-na/eng/compilers/282048.htm`

```
END PROGRAM
```

The lines starting with !$OMP are a signal to the compiler that OpenMP code is to be found here. However, if OpenMP support is absent, the lines starting with !$OMP are seen as a comment. In this case, the compiler is told that the DO loop may be run in parallel. It might split the problem in two parts, such that on one core, the loop is running for `i = 1,2` and on another core, the loop is running for `i=3,4`.

For a complete introduction to OpenMP, please see [*Chapman et al.*, 2007].

C.3 Performance caveats in a multithreaded environment

Under some circumstances, running a multithreaded version of an optimization can be slower than the singlethreaded version. An example of this is the small optimization run done to determine the optimal swarm size (see 4.3.3). Running this simulation with only one thread took 0.57[s] of CPU-time, using one CPU completely. The eight-thread version took 13.78[s] of CPU-time, but what was even worse was that it took 6.67[s] of wall time, meaning it was only able to make use of two out of eight cores. [8]

The explanation for this is straightforward: for a simple problem such as the Himmelblau function, only a very small amount of time is spent doing the calculation of the fitness. After this is done, the outcome of this calculation has to be reported to the other processor cores which are also doing calculations. This reporting takes a (relatively) large amount of time as data exchange between different cores is slow. This is the main cause of the increase of CPU-time.

Another consequence of this reporting of outcomes is that after a unit of work has been completed by a thread, it has to wait until the rest of the threads also have finished their work. This leads to the low saturation of the cores.

A third effect, which is closely tied to the reporting problem is *lock contention*. This happens when two threads try to access a shared resource which is guarded by locks, as in the following Fortran fragment:

```
!$OMP CRITICAL
ind%race%n_eval = ind%race%n_eval + 1
!$OMP END CRITICAL
```

The CRITICAL pieces are locks, and are needed to prevent so-called concurrent writes to the same variable. An example of what is meant by that is given in table C.1. In that example, two cores try to increment the value of a value somewhere in memory at the same time. However, because they need to load the data to the core first, the increment operation will be on data inside that core only. If the data is stored in main memory again directly afterwards, both cores will store the same value again, and its value will only have been incremented once instead of twice.

This can be prevented by adding locks to the problem; in that case, the flow over the two cores is more like that shown in table C.2. Because (by definition) only one core can own a lock at a time, this problem is prevented. If a core tries to acquire a lock which another core already owns, that core will have to wait until it is released before it can continue.

By placing code inside a critical block, the compiler adds code that acquires and releases locks at the start and end of such a block respectively.

This locking thing does make the program slower though, as the second thread has to wait for the first one to complete before it can enter the critical block. This can be clearly

[8]However, at the time this benchmark was run, the system was busy running another simulation, so only four out of eight cores were available – meaning that on an unloaded system, wall time would have been half of the current one.

seen, as the non-locked version only takes only $4+4$ clock cycles, and the version with locks takes $6+11$ for the "same" operation. (This can be reduced to $6+6$ cycles if the second core is allowed to do other work while it is waiting for a lock.)

Note however that when doing optimizations with complex objective functions, the overhead cost is negligible, and the optimization will benefit from having the possibility to use more than one core.

C.4 Pure functions

As outlined in the previous section, special care has to be taken when there data which is shared and modified between multiple threads. This can be done by adding locks around code which modifies shared data, but there is another way called *pure functions*.

This is a term coming from *functional programming*. Functional programming is a concept in which the goal is to make everything a function (hence the name), and getting rid of shared state. This might sound a bit abstract however it is not; in Fortran it can be done by abandoning the use of shared, save and intent(inout) variables. The reason for getting rid of shared state is that a function call then becomes a transformation, and a function that obeys these rules is called a pure function.[9]

Several languages exist which offer good support for functional programming, and some well-known examples are Haskell, Erlang and F#. Interested readers are referred to [*O'Sullivan et al.*, 2008] for a good introduction to functional programming in general and Haskell in particular.

So, why is the concept of pure functions interesting for this solving optimization problems? For one good reason: Because pure functions do not depend on modifying any shared

[9]Note that a pure function may not call impure functions, as that would compromise the purity of the caller.

Clock	Shared	core 1		core 2	
cycle	data	instruction	data	instruction	data
1	3	start	-	start	-
2	3	load data	3	load data	3
3	3	add 1 to data	4	add 1 to data	4
4	4	store data	4	store data	4

Table C.1 Simultaneous addition in a multiprocessor environment without locks.

Clock	Shared	core 1		core 2	
cycle	data	instruction	data	instruction	data
1	3	start	-	start	-
2	3	acquire lock	-	(acquire lock)	-
3	3	load data	3	wait for lock	-
4	3	add 1 to data	4	wait for lock	-
5	4	store data	4	wait for lock	-
6	4	release lock	4	wait for lock	-
7	4	...	-	acquire lock	-
8	4	...	-	load data	4
9	4	...	-	add 1 to data	5
10	5	...	-	store data	5
11	5	...	-	release lock	5

Table C.2 Simultaneous addition in a multiprocessor environment with locks.

data, they can be executed in parallel without locks. Thus to be sure that an optimization can be done in parallel, it is necessary to implement the routine that calculates the objective value as a pure function – thus it may not write to variables defined at the module scope, or make use of save and intent(inout) variables, and it may only call pure functions.

Therefore, the programmer has to take special care that the functions called are all pure. Not doing this may give rise to really subtle problems, which are difficult to debug. Starting with Fortran95, purity is a thing that should be checked by the compiler, but not all compilers support this out of the box. Therefore, it is recommended that the programmer does this check.

Appendix *D*

API documentation OPTIDUS-2009

This appendix contains an overview of the design behind OPTIDUS-2009, and the data structures and functions provided by it.

D.1 Design

OPTIDUS-2009 is a complete overhaul of the original OPTIDUS optimizer. A fundamental change is that the original OPTIDUS was written originally as a monolithic program to do optimizations, in which problem-specific routines have to be added.

The 2009 version is a library of functions, which are to be used in problem-specific programs. This allows the independent development of multiple problem-solvers at the same time.

Another big change is the use of the object-oriented features available since Fortran90: most data is stored in special data types.

Finally, the methods operating on the population may make use of parallelism. If this is used, the objective function implementation must be able to be run in parallel.

D.2 Evolutionary Algorithms

The module `genes` contains several data structures and functions which help in implementing EAs.

D.2.1 RaceType

`RaceType`

First of all, the format of the solution vector need to be declared. As this format is shared between all members of the population, this data structure is called the `RaceType`, and is defined in the module `genes` as follows:

```
TYPE RaceType
   REAL, KIND=(KPR) , DIMENSION(GeneLength)        :: Minimum , Maximum
   CHARACTER(LEN=GeneDescriptionLength) ,DIMENSION(GeneLength) :: description
   CHARACTER(LEN=RaceLength)                       :: id
   INTEGER                                         :: igen
   INTEGER                                         :: n_par
   INTEGER                                         :: n_eval
   REAL(KIND=KPR) , DIMENSION(10)                  :: extra_state
END TYPE RaceType
```

The attribute `extra_state` is provided to pass extra state data to the objective function. For example, in the GTOC3 optimization, this was used to pass the origin and target bodies, which were parameters not to be touched by the optimizer.

`NewRace` An instance of this type can be instantiated easily by calling the function `NewRace`, which accepts the name and the number of elements in the solution vector as parameters. After calling this function, a new race will be returned. The calling signature is as follows:

```
TYPE(RaceType) FUNCTION NewRace(id, n_par_arg) RESULT(r)
  CHARACTER(*), INTENT(IN)                    :: id
  INTEGER, INTENT(IN), OPTIONAL               :: n_par_arg
  INTEGER                                     :: n_par
```

`RaceSetGene` The next thing to do is to set the names and the bounds on the elements of the solution vector. This is done using the subroutine `RaceSetGene`.

```
SUBROUTINE RaceSetGene(race, n, min, max, desc)
  TYPE(RaceType), INTENT(INOUT) :: race
  INTEGER, INTENT(IN) :: n
  REAL(KIND=KPR), INTENT(IN) :: min, max
  CHARACTER(*), INTENT(IN) :: desc
```

`RacePrint` Finally, the definition of a race can be confirmed by printing it after running. This can be done by calling the subroutine `RacePrint`, which has the following calling signature:

```
SUBROUTINE RacePrint(race)
  TYPE(RaceType), INTENT(IN) :: race
```

A complete example of a program defining a race and printing it is shown here:

```
PROGRAM test_race
  USE genes
  USE precision
  IMPLICIT NONE
  TYPE(RaceType) :: myrace
  myrace = NewRace('Himmelblau', 2)
  CALL RaceSetGene(myrace, 1, 0._KPR, 5._KPR, 'x1')
  CALL RaceSetGene(myrace, 2, 0._KPR, 5._KPR, 'x2')
  CALL RacePrint(myrace)
END PROGRAM

SUBROUTINE EvaluateFitness(ind)
  USE genes
  IMPLICIT NONE
  TYPE(Individual), INTENT(INOUT) :: ind
END SUBROUTINE
```

Compiling and running this program gives the following output, which is what was expected:

```
[optidus-tng spaans] ./test_race
 Himmelblau
```

n	Description	Mininum	Maximum
1	x1	0.00	5.00
2	x2	0.00	5.00

D.2.2 Individual

Individual

The second type to be defined is the type `Individual`. This type can be used to store the data of individuals, and is heavily used in OPTIDUS. This datatype is composed as follows:

```
TYPE Individual
   REAL(KIND=KPR), DIMENSION(GeneLength)    :: values, &
                                               velocities, bestval
   TYPE (RaceType), POINTER                 :: race
   REAL, DIMENSION(FitnessLength)           :: fitness, bestfit
   INTEGER                                  :: front, frontrank
   LOGICAL                                  :: feasible=.FALSE., &
                                               fitnessvalid =.FALSE., &
                                               velocitiesvalid = .FALSE.
   INTEGER                                  :: age, lifetime
END TYPE Individual
```

NewIndividual

The simplest way to create a new individual is by calling the function `NewIndividual`, which has the following signature:

```
TYPE(Individual) FUNCTION NewIndividual(race) RESULT(i)
  TYPE(RaceType), INTENT(IN), TARGET :: race
```

When this function is called, a new individual is created by picking one randomly from the solution space given by the race definition. Because this function can also be used to initialize PSO and PPO individuals, not only the values of the solution vector but also the values of the velocity vector are initialized.

Destroy-Individual

If an individual needs to be terminated (for example because its age has exceeded its lifetime), this can be done by calling the `DestroyIndividual` function. This resets the `race` property. It has the following signature:

```
SUBROUTINE DestroyIndividual(ind)
  TYPE(Individual), INTENT(INOUT) :: ind
```

Individual-Print

Individuals can be printed using the function `IndividualPrint`. It has the following signature:

```
SUBROUTINE IndividualPrint(ind)
  TYPE(Individual), INTENT(IN) :: ind
```

IndividualAdd

To make the implementation of DE and PSO-like algorithms easier, three operators have been defined which can be applied to individuals. The first one is the `IndividualAdd` function, which takes two individuals and adds their values. This is not a standard addition, as it does take the situation into account when the lower boundary of the element to be added is nonzero: if the lower boundary is l, adding the two values x_1 and x_2 means the two offsets $x_1 - l$ and $x_2 - l$ are added, which then gives the offset from the lower boundary. This means the sum, taking this lower boundary into account equals:

$$s = (x_1 - l) + (x_2 - l) + l = x_1 + x_2 - l \tag{D.1}$$

The call signature is the following:

```
TYPE(Individual) FUNCTION IndividualAdd(ind1 , ind2) RESULT (nind)
  TYPE(Individual), INTENT(IN) :: ind1 , ind2
```

Individual-Substract

In a similar fashion, the function `IndividualSubstract` is defined, which subtracts the second from the first individual. This also takes a nonzero lower bound into account, using the following formula:

$$s = (x_1 - l) - (x_2 - l) + l = x_1 - x_2 + l \tag{D.2}$$

The call signature is the following:

```
TYPE(Individual) FUNCTION IndividualSubstract(ind1 , ind2) RESULT (nind)
  TYPE(Individual), INTENT(IN) :: ind1 , ind2
```

IndividualMult

Thirdly, a special method to scale an individual is defined, which can be used to multiply all the values in a vector by a given scalar. It is called `IndividualMult`. This function also needs to take the nonzero lower bound into account, and uses the following formula for that:

$$s = (x - l) * f + l = x * f + (1 - f) * l \tag{D.3}$$

The call signature is the following:

```
TYPE(Individual) FUNCTION IndividualMult(ind , factor) RESULT (nind)
  TYPE(Individual), INTENT(IN) :: ind
  REAL, INTENT(IN) :: factor
```

Note that these last three functions are also accessible using the standard notation, so the following works as expected:

```
 ind1 + ind2   ! Adds the two individuals
 ind1 - ind2   ! Subtracts one individual from the other
 ind1 * 0.5    ! Multiplies the values of an individual by 0.5
```

The following listing shows how these functions can be used together:

```
PROGRAM test_race
  USE genes
  USE precision
  IMPLICIT NONE
  TYPE(RaceType) :: myrace
  TYPE(Individual) :: myinds(3)
  myrace = NewRace('Himmelblau', 2)
  CALL RaceSetGene(myrace, 1, 0._KPR, 5._KPR, 'x1')
  CALL RaceSetGene(myrace, 2, 0._KPR, 5._KPR, 'x2')
  CALL RacePrint(myrace)

  myinds(1) = NewIndividual(myrace)
  myinds(2) = NewIndividual(myrace)
  myinds(3) = (myinds(1) + myinds(2)) * 0.5 + &
              (myinds(1) - myinds(2)) * 0.5
  CALL IndividualPrint(myinds(1))
  CALL IndividualPrint(myinds(2))
  CALL IndividualPrint(myinds(3))
END PROGRAM

SUBROUTINE EvaluateFitness(ind)
  USE genes
  IMPLICIT NONE
  TYPE(Individual), INTENT(INOUT) :: ind
END SUBROUTINE
```

Running this program gives the following output, and it can be seen that individual 1 and 3 both have the same values for the coordinates, which is as expected.

```
Himmelblau
 n  Description                          Mininum        Maximum
 1  x1                                        0.00           5.00
 2  x2                                        0.00           5.00
 1  x1                                        1.63
 2  x2                                        2.74
Lifetime:            500  Age:           0  Fitness:   ...
 1  x1                                        4.39
 2  x2                                        0.66
Lifetime:            500  Age:           0  Fitness:   ...
 1  x1                                        1.63
 2  x2                                        2.74
Lifetime:              0  Age:           0  Fitness:   ...
```

D.2.3 Population

`Population`

Finally, a type to hold a population of individuals is defined. This type is called `Population`, and is defined as follows:

```
 TYPE Population
    TYPE (Individual), DIMENSION(MaxPopSize)    :: members
    TYPE (Individual)                           :: best
    INTEGER                                     :: size = 0
 END TYPE Population
```

No constructors for this type exist.

D.2.4 Managing the population

`FillPopulation`

At the start of an optimization run, the population will need to be built up from individuals. To do this efficiently, a routine called `FillPopulation` is provided, which has the following signature:

```
 SUBROUTINE FillPopulation(pop, race, n_new)
   TYPE(Population), INTENT(INOUT) :: pop
   TYPE(RaceType), INTENT(IN) :: race
   INTEGER, INTENT(IN),OPTIONAL :: n_new
```

Calling this function adds n_new individuals of the given race to the population. This is done by randomly creating individuals using the `NewIndividual` function, and the evaluating those functions. If a suggested individual does not meet the constraints (it is marked as infeasible), it is discarded, and a new one is constructed. This function can make use of parallelism.

Import-Individual

To add an individual to the population, the function ImportIndividual is supplied, with the following call signature:

```
SUBROUTINE ImportIndividual(pop, par, ind)
  TYPE(Individual), INTENT(INOUT) :: ind
  TYPE(EvolutionaryParameters), INTENT(IN) :: par
  TYPE(Population), INTENT(INOUT) :: pop
```

This function does two things: it adds the individual to the list of individuals in the population, and sets its velocity vector to the average velocity of the swarm.

Export-Individual

The function which does the opposite of this function is named ExportIndividual, and has the following call signature:

```
TYPE(Individual) FUNCTION ExportIndividual(pop, i)
  TYPE(Population), INTENT(INOUT) :: pop
  INTEGER, INTENT(IN) :: i
```

This routine accepts a population and the number of the individual inside the population to be exported. After running, it returns this individual, which has been removed from the population.

These two functions can be used to "teleport" individuals between multiple populations.

D.2.5 Mutation

DoSingle-Mutation

To mutate a single individual, the function DoSingleMutation is provided, with the following call signature:

```
SUBROUTINE DoSingleMutation(ind, par)
  TYPE(Individual), INTENT(INOUT)              :: ind
  TYPE(EvolutionaryParameters), INTENT(IN)     :: par
```

This function iterates over the elements of the solution vector, and for each element, there is a chance (defined in par) that it will mutate. If it mutates, the element is subjected to non-uniform mutation (see section 3.4.3). Next, the objective function is called, to see if the mutated individual is feasible. If it is not, the process is repeated.

Mutate-Population

To mutate a whole population, the function MutatePopulation is provided, with the following call signature:

```
SUBROUTINE MutatePopulation(pop, par)
  TYPE(Population), INTENT(INOUT)              ::  pop
  TYPE(EvolutionaryParameters), INTENT(IN)     ::  par
```

This function calls DoSingleMutation for each member of the population pop. This function can make use of parallelism.

D.2.6 Crossover

DoSingle-Crossover

To do single-point crossover with two individuals, the routine DoSingleCrossover is supplied, which has the following call signature:

```
SUBROUTINE DoSingleCrossover(ind1_in, ind2_in, ind1_out, ind2_out)
  TYPE(Individual), INTENT(IN)   :: ind1_in,  ind2_in
  TYPE(Individual), INTENT(OUT)  :: ind1_out, ind2_out
```

This routine chooses a point to execute crossover randomly, and then combines the two input individuals, applying crossover at the right place, to form the two output individuals. No check is done to make sure the resulting individuals are feasible.

`Crossover-Population`

To crossover all individuals in a population, two functions are provided. The first one is `CrossoverPopulation`, which has the following call signature:

```
SUBROUTINE CrossoverPopulation(pop, par, xovers_numb)
  TYPE(Population), INTENT(INOUT)  ::  pop
  TYPE(EvolutionaryParameters), INTENT(IN)  :: par
  INTEGER,INTENT(IN),OPTIONAL  :: xovers_numb
```

The optional parameter `xovers_numb` can be used to control the amount of individual pairs that will be sent to the `DoSingleCrosssover` function. During execution of this function, parent pairs are selected at random, and crossover is applied. If the resulting children are feasible, they are appended to the population. If not, the children are discarded. This is repeated until the desired amount of children is generated. This function can make use of parallelism.

`Roulette-Crossover-Population`

The second function provided for doing crossover is the function `RouletteCrossover-Population`, which has a similar call signature:

```
SUBROUTINE RouletteCrossoverPopulation(pop, par, xovers_numb)
  TYPE(Population), INTENT(INOUT)  ::  pop
  TYPE(EvolutionaryParameters), INTENT(IN)  :: par
  INTEGER, INTENT(IN), OPTIONAL  :: xovers_numb
```

This function works the same as the previous one, but does not pick the parents at random; it uses roulette wheel selection (see section 3.5) to do a weighted selection of the parents, based on their fitness. This function can make use of parallelism.

D.2.7 Selection

`SetupRoulette-Wheel`

For doing roulette wheel selection (see section 3.5), some functions are provided. First of all, the roulette wheel needs to be initialized. This is done using the function `Setup-RouletteWheel`. It has the following call signature:

```
SUBROUTINE SetupRouletteWheel(pop, par, wheel)
  TYPE(Population), INTENT(IN)                    :: pop
  TYPE(EvolutionaryParameters), INTENT(IN)        :: par
  REAL(KIND=KPR), DIMENSION(:), INTENT(OUT)       :: wheel
```

This function sets up the roulette wheel using the fitness values (which need to be precalculated); first, it determines the smallest fitness value f_{min}, and then scales the roulette wheel slots by subtracting this minimum value from the fitness values of the individuals.

`RouletteWheel`

Once the wheel has been set up, individuals can be picked using the function `Roulette-Wheel`. This function has the following call signature:

```
INTEGER FUNCTION RouletteWheel(pop, par, wheel) RESULT(position)
  TYPE(Population), INTENT(INOUT)                 :: pop
  TYPE(EvolutionaryParameters), INTENT(IN)        :: par
  REAL(KIND=KPR), DIMENSION(:), INTENT(IN)        :: wheel
```

This function simulates throwing a roulette ball using the biased wheel, and returns the index of selected individual in the population.

`Roulette-Selection`

To reduce the size of a population using selection, some methods are provided. The first one uses the aforementioned roulette wheel selection, and is called `RouletteSelection`. This method has the following call signature:

```
SUBROUTINE RouletteSelection(pop, par, popsize)
  TYPE(Population), INTENT(INOUT) :: pop
  TYPE(EvolutionaryParameters), INTENT(IN) :: par
  INTEGER, INTENT(IN) :: popsize
```

This creates a new population by sampling the old population (with replacement) `popsize` times. This method takes care of setting up and destroying a roulette wheel by itself.

`Tournament-Selection`

Another method to select individuals from the population can be done using `Tournament-Selection`. This function has the following call signature:

```
SUBROUTINE TournamentSelection(pop, par, popsize)
  TYPE(Population), INTENT(INOUT) :: pop
  TYPE(EvolutionaryParameters), INTENT(IN) :: par
  INTEGER, INTENT(IN) :: popsize
```

This method picks two individuals from the input population (with replacement), and puts the best of the two into the new population. It repeats this procedure until the new population reaches size `popsize`.

`Stochastic-Universal-Sampling`

The third method to select a given number of individuals from a population can be done using the routine `StochasticUniversalSampling`. It has the following call signature:

```
SUBROUTINE StochasticUniversalSampling(pop, par, popsize)
  TYPE(Population), INTENT(INOUT) :: pop
  TYPE(EvolutionaryParameters), INTENT(IN) :: par
  INTEGER, INTENT(IN) :: popsize
```

This method is described in section 3.5. It does a variant of roulette wheel selection, with the guarantee that each individual is selected either $\text{floor}(c_i)$ or $\text{ceiling}(c_i)$ times exactly, where c_i is the relative size of the roulette wheel slot of individual i compared to the average size of the slots. This means that a good individual (which will have $c_i > 1$) will not disappear from the population due to bad luck.

D.2.8 Routines for GAVaPS

The genes module also contains some building blocks that can be used to implement a GAVaPS. These routines deal with setting the lifetime of individuals, and removing individuals which are too old from the population.

`Assign-LifetimeStatic`

The simplest way to assign lifetimes to individuals is to give them all the maximum allowed lifetime. This is done by the routine `AssignLifetimeStatic`, which has the following call signature:

```
SUBROUTINE AssignLifetimeStatic(pop, par)
  TYPE(Population), INTENT(INOUT)           :: pop
  TYPE(EvolutionaryParameters), INTENT(IN)  :: par
```

`Assign-LifetimeRanked`

Another method to assign lifetimes to individuals in the population is by sorting the new individuals in order of fitness value, and assigning lifetimes according to their rank. This is done using the routine `AssignLifetimeRanked`, which has the following call signature:

```
SUBROUTINE AssignLifetimeRanked(pop, par)
  TYPE(Population), INTENT(INOUT)              :: pop
  TYPE(EvolutionaryParameters), INTENT(IN)     :: par
```

After the new individuals have been ranked from 1 to n_{new}, the lifetimes are assigned according to the following formula:[1]

$$L_i = \text{round}(\frac{r_i - 1}{n_{new} - 1}(L_{min} - L_{max}) + L_{max}) \tag{D.4}$$

The individual with $r_i = 1$ will be assigned the maximum lifetime L_{max}, whereas the individual with rank $r_i = n_{new}$ is assigned the minimum lifetime L_{min}, and individuals in between are given lifetimes which follow from linear interpolation between those two points.

`AssignLifetime`

Finally, the method `AssignLifetime` is provided which assigns lifetimes that are proportional to the fitness of an individual. This function has the following call signature:

```
SUBROUTINE AssignLifetime(pop, par)
  TYPE(Population), INTENT(INOUT)              :: pop
  TYPE(EvolutionaryParameters), INTENT(IN)   :: par
```

This routine uses the following fitness scaling algorithm:

$$L_i = \text{round}(L_{min} + \frac{f_i - f_{min}}{f_{max} - f_{min}}(L_{max} - L_{min})) \tag{D.5}$$

In this equation, f_{min} is the worst and f_{max} is the best fitness value in the population.

`DebugLifetime`

For debugging the assigned lifetimes, the function `DebugLifetime` is provided, with the following call signature:

```
SUBROUTINE DebugLifetime(pop, par)
  TYPE(Population), INTENT(IN)  :: pop
  TYPE(EvolutionaryParameters), INTENT(IN)  :: par
```

This routine prints the distribution of the assigned lifetimes of the fresh individuals in the population.

`BumpAge`

During each generation, the age of all individuals needs to be increased once. This can be done using the routine `BumpAge`, which has the following call signature:

```
SUBROUTINE BumpAge(pop)
  TYPE(Population), INTENT(INOUT)  :: pop
```

After calling this procudure, the age of all individuals will have been increased by 1.

`KillOverAge-Individuals`

Finally, the individuals that are too old need to be removed from the population; this is done using the routine `KillOverAgeIndividuals`, which has the following call signature:

[1] **round** is the operator which rounds a real to the nearest integer

```
SUBROUTINE KillOverAgeIndividuals(pop)
  ! Soylent Green is made of Genepool !
  TYPE(Population), INTENT(INOUT)              :: pop
```

This routine removes all individuals from the population whose age is bigger than their assigned lifetime, and compacts the population so that no holes remain between the individuals.

D.2.9 Other routines

Wrapped-Evaluate-Fitness

One of the procedures which is called quite often by the other procedures is `WrappedEvaluateFitness`. This routine checks if the current fitness value of an individual is still valid, and if that is the case, returns that value. If not, it will call the (user supplied) `EvaluateFitness`, and stores that value. It has the following call signature:

```
SUBROUTINE WrappedEvaluateFitness(ind)
  TYPE(Individual), INTENT(INOUT) :: ind
  EXTERNAL EvaluateFitness
```

BindValues

For some algorithms like PSO and PPO, an individual might move outside of the problem space. To make sure this does not happen, those individuals can be kept inside the solution space using the algorithm from section 4.5. This algorithm is implemented in the routine `BindValues`, which has the following call signature:

```
SUBROUTINE BindValues(ind, par)
  TYPE(Individual), INTENT(INOUT) :: ind
  TYPE(EvolutionaryParameters), INTENT(IN) :: par
```

FindBest

To find the best individual in a given population, the function `FindBest` is supplied, which iterates over the members of the population and sets the `best` member to point to the current best individual. Its call signature is as follows:

```
SUBROUTINE FindBest(pop, par)
  TYPE(Population), INTENT(INOUT)              :: pop
  TYPE(EvolutionaryParameters), INTENT(IN)     :: par
```

Elitism

A related routine is the `Elitism` routine. This can be called to make sure that the best individual from the population does not disappear. This is useful in case the best individual is mutated, and becomes a new individual which has a worse fitness value. The call signature is as follows:

```
SUBROUTINE FindBest(pop, par)
  TYPE(Population), INTENT(INOUT)              :: pop
  TYPE(EvolutionaryParameters), INTENT(IN)     :: par
```

BumpGen

For keeping track of the number of generations which has passed, another method is supplied which increases the generation counter for `RaceTypes`. This method is called `BumpGen`, and has the following call signature:

```
SUBROUTINE BumpGen(r)
  TYPE(RaceType), INTENT(INOUT) :: r
```

D.3 PSO/PPO

D.3.1 Predator type

Predator

The module genes defines the type Predator. This can be used to store the position and velocity vectors of the predator, in case of PPO. It is defined as follows:

```
TYPE Predator
   REAL(KIND=KPR), DIMENSION(GeneLength)          :: position, velocities
END TYPE Predator
```

No methods are provided to directly modify instances of this type; this is handled by the PSO code itself.

D.3.2 PSO/PPO routine

PSO

The module ppso contains the subroutine PSO. It has the following call signature:

```
SUBROUTINE PSO(pop, par, pred)
  TYPE(Population), INTENT(INOUT) :: pop
  TYPE(EvolutionaryParameters), INTENT(IN) :: par
  TYPE(Predator), INTENT(INOUT), OPTIONAL :: pred
```

This method implements one step of a PSO (and PPO) algorithm. It starts out by identifying the members of the 1st Pareto front, using the NSGA-II algorithm.

Next, for all members in the population, the components of the solution vectors are iterated over, and the particle velocities are adjusted according to (4.7):

$$\mathbf{v}_{i,new} = w \cdot \mathbf{v}_i + c_1 \cdot u() \cdot (\mathbf{x}_i^* - \mathbf{x}_i) + c_2 \cdot u() \cdot (\mathbf{x}_G^* - \mathbf{x}_i) \tag{D.6}$$

If the optional predator argument is given, each component has a chance of being affected by the predator. If the predator affects the particle, the velocity is updated using a variant of (4.15):

$$v_{i,k,new} = wv_{i,k} + c_1 \cdot u() \cdot (x_{i,k}^* - x_{i,j}) + c_2 \cdot u() \cdot (x_{G,k}^* - x_{i,j}) + \mathrm{sgn}(x_i - x_{pred})D(d) \tag{D.7}$$

in which

$$D(d) = ae^{-bd} \tag{D.8}$$

and

$$d = |x_i - x_{pred}| \tag{D.9}$$

and sgn is the sign function[2].

After updating all the components of the velocity vector, the position vectors are updated, and the fitness (and feasibility) of the individuals are determined by calling the objective function. If the new individual is valid, it is accepted, if not, the procedure is repeated.

After all individuals have been updated, the speed and position of the predator are also updated (if it is given as an argument).

D.4 Differential Evolution methods

Tasoulis1

For doing differential evolution, 6 functions are supplied, in the module diffevo, which are called Tasoulis1 to Tasoulis6. All these six functions share the same call signature, being:

[2] The sign function maps negative values to -1, is 0 for 0 and maps positive numbers to 1

```
SUBROUTINE Tasoulis1(pop, par)
  TYPE(Population), INTENT(INOUT) :: pop
  TYPE(EvolutionaryParameters), INTENT(IN) :: par
```

These functions generate mutation vectors using the schemes given in section 5.2. This mutation vector is combined with the original vector using a two-point crossover, and the fitness is evaluated. If it has improved, the new solution is saved, or else the old one is kept. This is done S (the population size) times

Note that `test_gtoc3.f90` contains an ad hoc implementation of a `Tasoulis1` function which can also be applied to MOO problems. In the future, this should be implemented for all **6** schemes and backported to the `diffevo` module.

D.5 Multi Objective Routines

The module `nsga2` and `constant_parameters` contain some routines which are helpful for dealing with multi-objective optimization problems (MOO).

`InitEvolDir`

The first function of use is the function `InitEvolDir`. This function allows the direction of the optimization to be set separately for each objective function. This allows for problems in which the first objective value should be maximized and the second one to be minimized to be written down in their natural form. This function has the following call signature:

```
SUBROUTINE InitEvolDir(n, dir, dirs)
  INTEGER, INTENT(IN), OPTIONAL :: n
  REAL, INTENT(IN), OPTIONAL :: dir
  REAL, INTENT(IN), DIMENSION(:), OPTIONAL :: dirs
```

Despite this seemingly chaotic definition (as everything is defined optional), only two forms should be used. If all directions are the same, a shortcut can be used, by only supplying `n` and `dir`, for example by calling `InitEvolDir (3, -1)` to set up the system to have 3 objective functions which should be minimized.

To support the scenario written above, it is also possible to give an array as an argument. This then takes the form `InitEvolDir ((/ 1, -1 /))`.

Note that these directions are stored in an array in the global namespace, which means only one set of directions can be used at the same time.

`dominates`

Once these optimization directions have been set, it is possible to test if one solution is better than the other one using the `dominates` function, which has the following call signature:

```
  LOGICAL FUNCTION dominates(f1, f2, altevoldir)
    REAL, INTENT(IN), DIMENSION(:) :: f1, f2
    REAL, INTENT(IN), OPTIONAL :: altevoldir
```

The use of the argument `altevoldir` is deprecated. This function returns `.TRUE.` if the solution `f1` dominates solution `f2`, as described in chapter 6, or `.FALSE.` otherwise.

`DoMOO-Tournament`

To support MOO variants of DE, the function `DoMOOTournament` is supplied. This function has the following call signature:

```
  TYPE(Individual) FUNCTION DoMOOTournament(pop, p1, trial)
    TYPE(Population), INTENT(IN) :: pop
    TYPE(Individual), INTENT(IN) :: trial
    INTEGER, INTENT(IN) :: p1
```

This function checks if the individual `trial` dominates the `p1`th individual in the population. If it does, it returns the `trial` solution, if it does not, it returns the original solution.

This can be used as follows in code:

```
newpop(i) = DoMOOTournament(pop, i, trial)
```

`fastNondomSort` `fastNondomSort` is a function to rank all individuals in a population. It has the following call signature:

```
SUBROUTINE fastNondomSort(pop, n)
  TYPE(Population), INTENT(INOUT) :: pop
  INTEGER, INTENT(in), OPTIONAL :: n
```

This function ranks the individuals in the first `n` fronts, using the algorithm NSGA-II.

Once all individuals have been ranked, the order inside the ranks can be determined using the routine fnnrankFronts. This function calls the routine `rankFrontByLCD` for all fronts in the population. It has the following call signature:

```
SUBROUTINE rankFronts(pop)
  TYPE(Population), INTENT(inout) :: pop
```

`rankFrontByLCD` The function `rankFrontByLCD` can be used to rank the individuals inside one front, using the local crowding distance as defined in chapter 6.

```
SUBROUTINE rankFrontByLCD(pop, front)
  TYPE(Population), INTENT(INOUT) :: pop
  INTEGER, INTENT(IN) :: front
```

Note that this function can also be called by the end-user code.

`PickFromFront` Finally, `PickFromFront` is a function for picking a random individual from a front. It is provided to help in the implementation of PSO, and has the following call signature:

```
TYPE(Individual) FUNCTION PickFromFront(pop, f)
  TYPE(Population), INTENT(IN) :: pop
  INTEGER, INTENT(IN) :: f
```

It returns a random member from the population `pop` which is in front `f`.

D.6 Random numbers

Generating random numbers is treated extensively in [*Press et al.*, 2007]. However, no random number generator (rng) which yields consistent results in a multi-threading environment is given.

For that reason, a rng which does give reproducible results, even in a multi-threading environment has been implemented for OPTIDUS-2009.

Normally, rngs are initialized by setting a seed, and subsequent calls then yield a sequence of numbers with a random-like distribution. In between calls, the state of the rng is saved.

This might lead to problems when executing this program on a multiple processor machine: the order of execution can vary when multiple threads are running in parallel. A program to demonstrate this behavior is shown in below, with the output in table D.1.

```
PROGRAM demo_multithreading
  !$OMP PARALLEL DO
  DO i = 1, 4
    WRITE (*,"(I2)"), i
  END DO
  !$OMP END PARALLEL DO
END PROGRAM
```

The solution to this problem is to give each work unit its own random generator. In the random_mp module, this is done using the following functions:

---

random_init

When the number of work units is known, the random generators can be initialized with a call to the function random_init, which has the following call signature:

```
  SUBROUTINE random_init(n, s)
    INTEGER, INTENT(IN) :: n
    INTEGER, INTENT(IN), OPTIONAL :: s
```

The parameter n is the amount of random generators to initialize. These random generators are seeded with a value from the *master* PRNG, which in turn can be seeded using the optional argument s.

---

random_set_-block

Then, when a certain random generator is needed, a thread can select one by calling random_set_block. It has the following call signature:

```
  SUBROUTINE random_set_block(i)
    INTEGER,INTENT(IN) :: i
```

The parameter i should be set to the number of the work unit. Then, subsequent calls to the random() function from within the same thread will return numbers from that rng.

---

random

Finally, random numbers can be generated by calling the function random, which has the following call signature:

```
  REAL FUNCTION random(idum)
    INTEGER(KIND=K4B), INTENT(IN), OPTIONAL :: idum
```

Note that the argument idum should only be set by the code in this module.

A routine demonstrating the proper use of these functions is given below:

```
SUBROUTINE PopFitness(pop, fit)
  TYPE(Population), INTENT(IN)  :: pop(:)
  TYPE(Fitness), INTENT(OUT)    :: fit(SIZE(pop))
```

1 thread	2 threads
1	1
2	3
3	2
4	4

Table D.1 Execution order for different numbers of threads in a multi-core system..

```
  INTEGER :: i
  CALL random_init(SIZE(pop))   ! First, initialize the rng
  !$OMP PARALLEL DO SHARED(pop, fit)
  DO i = 1, SIZE(pop)
    CALL random_set_block(i)    ! Select the sub-rng according to the block
    CALL EvaluateFitness(pop(i), fit(i))
  END DO
  !$OMP END PARALLEL DO
END SUBROUTINE
```

API index

AssignLifetime, 139
AssignLifetimeRanked, 139
AssignLifetimeStatic, 138

BindValues, 140
BumpAge, 139
BumpGen, 140

CrossoverPopulation, 137

DebugLifetime, 139
DestroyIndividual, 133
dominates, 142
DoMOOTournament, 142
DoSingleCrossover, 136
DoSingleMutation, 136

Elitism, 140
ExportIndividual, 136

fastNondomSort, 143
FillPopulation, 135
FindBest, 140

ImportIndividual, 136
Individual, 133
IndividualAdd, 133
IndividualPrint, 133
IndividualSubstract, 134
IndividualMult, 134
InitEvolDir, 142

KillOverAgeIndividuals, 139

MutatePopulation, 136

NewIndividual, 133
NewRace, 132

PickFromFront, 143
Population, 135
Predator, 141
PSO, 141

RacePrint, 132
RaceSetGene, 132
RaceType, 131
random, 144
random_ init, 144
random_ set_ block, 144
rankFrontByLCD, 143
RouletteCrossoverPopulation, 137
RouletteSelection, 138
RouletteWheel, 137

SetupRouletteWheel, 137
StochasticUniversalSampling, 138

Tasoulis1, 141
TournamentSelection, 138

WrappedEvaluateFitness, 140