Gemfony Professional Series

Gemfony scientific

Parametric Optimization

with the

Geneva Library Collection

Version: 1.6 (Ivrea)

Dr. Rüdiger Berlich • Dr. Ariel García • Dr. Sven Gabriel

Parametric Optimization
The Geneva Library Collection

Authors:
Dr. Rüdiger Berlich
Dr. Sven Gabriel
Dr. Ariel García

Contact

You can contact us via E-Mail (contact@gemfony.eu), via letter post or through our Web page (http://www.gemfony.eu).

Address at the time of writing:
Gemfony scientific UG (haftungsbeschränkt)
Leopoldstr. 122
76344 Eggenstein-Leopoldshafen
Germany

Tel: +49 (0)7247 9342780
Fax: +49 (0)7247 9342781

Note that this contact information is subject to change without notice. Please see our web page for up-to-date contact information and registration information for Gemfony scientific UG (haftungsbeschränkt).

Trademarks

[This is No Scientific Document]

[Your Mileage Will Vary]

[Thank you for your Patience]

Contents

1. Introduction 1
1.1. Roadmap 2
1.2. Functionality 2
1.3. Scalability 2
1.4. Architecture 2
1.5. Platform 3
1.6. Licensing 3
1.7. Contact 3

I. Optimization Algorithms and Theory 5

2. General Considerations and Overview 7
2.1. Models and Reality 8
2.2. Choosing evaluation criteria 9
2.3. A Single Evaluation Criterion 10
2.4. Relying on quality surfaces 12
2.5. Multi-Criterion Optimization 13
2.6. Parameter Constraints 15
2.7. Definition 16

3. Gradient Descents 17
3.1. Mathematical Background 17
3.2. Application to Real-Life Problems 18
3.3. Inquest 20

4. Evolutionary Algorithms 21
4.1. Common Features 21
4.2. Evolution Strategies 27
4.3. Genetic Algorithms 30
4.4. Hybrid Feature Vectors 31
4.5. Multipopulations 32
4.6. Inquest 32

5. Simulated Annealing **33**
5.1. Nature as a Role Model . . . 33
5.2. The Algorithm in Pseudo-Code . . . 34
5.3. Means of Integration with Evolutionary Algorithms . . . 35
5.4. Inquest . . . 37

6. Swarm Intelligence **39**
6.1. Particle Swarm Optimization . . . 39
6.2. Ant Colony Optimization . . . 43
6.3. Inquest . . . 43

7. Parameter Scans **45**

8. Parallelization: General Considerations **47**
8.1. Application Types . . . 47
8.2. Data- and Task-based Parallelism . . . 48
8.3. Parallelizing Optimization Algorithms . . . 48
8.4. Characteristics of Parallel and Distributed Environments . . . 49
8.5. Constraints . . . 58

9. More Complex Demos and Use Cases **65**
9.1. Mapping Semi-Transparent Triangles to a Target Picture . . . 65
9.2. Protein Folding . . . 70
9.3. Training Feed Forward Neural Networks . . . 71

II. Using the Geneva Optimization Library **75**

10. Compilation and Installation **77**
10.1. Prerequisites . . . 77
10.2. Installation using binary packages . . . 80
10.3. Installation from source . . . 81

11. Defining a first Optimization Problem **87**
11.1. Outline . . . 88
11.2. Defining a paraboloid . . . 90
11.3. Class Declaration . . . 91
11.4. Member functions . . . 92
11.5. The `main()` function . . . 97
11.6. A note about performance . . . 99
11.7. What we didn't say 99

12. Class Hierarchies and Principles **101**
12.1. Core Optimization Classes . . . 101

12.2. Communication and Brokerage . . . 107
12.3. Random Number Creation . . . 107

13. Parameter Types 109
13.1. Overview . . . 109
13.2. Value Access . . . 111
13.3. Access to Value- and Initialization-Boundaries . . . 113
13.4. De-activation of Parameters . . . 114
13.5. Summary of Parameter Types . . . 114

14. Adaptors 133
14.1. General adaptor options . . . 133
14.2. `GDoubleGaussAdaptor` . . . 135
14.3. `GDoubleBiGaussAdaptor` . . . 136
14.4. `GInt32GaussAdaptor` . . . 138
14.5. `GInt32FlipAdaptor` . . . 140
14.6. `GBooleanAdaptor` . . . 141
14.7. Adaptors and Constrained Parameter Types . . . 142

15. Individuals and Parameters 145
15.1. General Principles . . . 145
15.2. `fitnessCalculation()`: Evaluating Individuals . . . 147
15.3. Serialization . . . 151
15.4. Further Interface Functions . . . 153
15.5. Personalities . . . 154

16. Advanced Constraint Handling 155
16.1. Visualization . . . 155
16.2. Problem Definition . . . 156
16.3. Identifying invalid candidate solutions with Geneva . . . 158
16.4. Transparent solution handling . . . 161
16.5. Constrained optimization with the USESIGMOID policy . . . 163
16.6. Other ways of identifying invalid solutions . . . 165
16.7. Accessing “true” and “transformed” fitness values . . . 165

17. Common Traits of Optimization Algorithms 167
17.1. Class Layout . . . 167
17.2. The Optimization Loop . . . 169
17.3. Geneva’s Halt Criteria . . . 170
17.4. The Population Interface . . . 171
17.5. Checkpointing . . . 172

18. Evolutionary Algorithms with Geneva 173
18.1. Looking Back at the Theory . . . 173

18.2. Construction of Evolutionary Algorithm Objects . . . 174
18.3. Specifying the Amount of Parents and Children . . . 176
18.4. Duplication Schemes . . . 178
18.5. Mutation . . . 179
18.6. Evaluation and Selection . . . 179
18.7. Mixing Parameter Types . . . 181

19. Simulated Annealing with Geneva 183

20. Particle Swarm Optimization with Geneva 185
20.1. Looking Back at the Theory . . . 185
20.2. Construction of PSO Objects . . . 186
20.3. Neighborhood-Sizes and Numbers of Neighborhoods . . . 187
20.4. Setting Progress Factors . . . 188
20.5. Constraints for Position Updates . . . 188
20.6. The Update Rule . . . 189

21. Gradient Methods with Geneva 191
21.1. Geneva's Steepest Descent Implementation . . . 191
21.2. Construction of Gradient Descent Objects . . . 192
21.3. Important Configuration Options . . . 194

22. Parameter Scans with Geneva 195
22.1. Construction of Parameter Scan Objects . . . 195
22.2. Random scan versus scan on a grid . . . 197
22.3. Specifying which parameters to scan . . . 197

23. Parallelization Modes 199
23.1. Serial Execution . . . 199
23.2. Multithreaded Execution . . . 200
23.3. Brokered Execution . . . 200
23.4. Direct Instantiation of Algorithms . . . 201

24. Unified Access to Optimization Algorithms 205
24.1. The `main()` function . . . 205
24.2. Adding further Algorithms . . . 209

25. Optimization Monitors 211
25.1. Internal Architecture . . . 211
25.2. Specifics for the Algorithms . . . 213
25.3. Pluggable Optimization Monitors . . . 213

26. A More Complex Example 215
26.1. Setting Up the Individual . . . 215
26.2. Creating a Factory . . . 218

26.3. Adding a Custom Optimization Monitor . . . 221
26.4. Setting up `main()` . . . 226

27. Caveats and Restrictions 229
27.1. Floating Point Accuracy . . . 229
27.2. Gradient Descent and Varying Parameter Value Ranges . . . 229
27.3. “Silent changes” to parameter values . . . 230
27.4. Individuals with a Variable Architecture . . . 230
27.5. The Effect of the Mutation Probability . . . 230
27.6. Value Range of Constrained Paramters . . . 231
27.7. Broker-flooding . . . 232
27.8. Assigning the worst possible evaluation . . . 232
27.9. Secure communication . . . 232

III. Details and Advanced Topics 235

28. Performing Meta-Optimization with Geneva 237
28.1. Multi-Populations . . . 237
28.2. Optimizing Configuration Parameters . . . 238
28.3. Letting different Algorithms Compete . . . 239

29. Coding Conventions 243
29.1. Code Documentation . . . 243
29.2. Coding Rules . . . 245
29.3. File naming schemes . . . 254

30. Helping Each other 257
30.1. Finding Help . . . 257
30.2. Suggesting Improvements . . . 258
30.3. Monetary donations . . . 259
30.4. Licensing . . . 259

IV. Independent Geneva Libraries 261

31. Creating Random Numbers with Hap 263
31.1. The Random Number Factory . . . 263
31.2. The Random Number Proxy `GRandom` . . . 264

32. Brokering with the Courtier Library 273
32.1. Architecture . . . 274
32.2. Requirements for Work Items . . . 276

32.3. Accessing the Broker . . . 276
32.4. Configuration Options of the Broker . . . 277
32.5. Submission of Work Items . . . 277

33. Common Functionality and Classes 281
33.1. A Thread-Safe Queue . . . 281
33.2. Raising Exceptions and Logging . . . 283
33.3. Parsing Configuration Files . . . 286
33.4. Creating Factories . . . 290
33.5. Singletons . . . 291
33.6. Global Options . . . 292
33.7. A Thread Group . . . 292
33.8. A Thread Pool . . . 292
33.9. The Plot Designer . . . 293
33.10. Parsing Formulas . . . 295

V. Appendix and Bibliography 299

A. Frequently Used Test Functions 301
A.1. Parabola . . . 301
A.2. Berlich Noisy Parabola . . . 301
A.3. Rosenbrock Function . . . 302
A.4. Ackley Function . . . 302
A.5. Rastrigin Function . . . 302
A.6. Schwefel Function . . . 302
A.7. Salomon Function . . . 303

B. The Boost Library Collection 309
B.1. Smart Pointers . . . 309
B.2. Serialization . . . 309
B.3. Threads . . . 310

C. The ROOT Analysis Framework 311

D. Important Open Source Licenses 315
D.1. The GNU Affero General Public License . . . 315
D.2. The Boost Software License, v 1.0 . . . 326

E. Glossary 329

F. References 331

Index 346

Chapter 1.

Introduction

This manual provides a thorough introduction into performing parametric optimization with the Geneva library collection. Geneva implements a collection of different optimization algorithms aimed at execution in distributed and parallel environments.

The software provides users with the means to tackle particularly large and complex optimization problems on devices ranging from multicore machines over compute clusters to Grids and Clouds. In addition, the broker-based architecture allows to define custom compute backends, such as a GPGPU provider[1]. The algorithms being used make but very few assumptions about the underlying optimization task, thus making them applicable to a very wide range of technical, scientific, and economic problem domains.

Geneva could be used for the improvement of a simulation of social behaviour just as easily as for the optimization of a combustion chamber, in order to engineer a more environmentally friendly car. The optional, fine-grained control over Geneva's inner workings – combined with sensible default values – as well as the type-agnostic implementation make it easy for users to define the parameter space underlying their optimization problem, and to model the associated evaluation function.

Geneva relieves users from the more daunting tasks usually associated with finding optimized solutions. In this sense, Geneva is a tool; and indeed it has been built as a toolkit rather than a stand-alone application.

The tell-tale acronym "Geneva" stands for "**G**rid-**en**abled **ev**olutionary **a**lgorithms" and also hints at the origin of this software in particle physics – the initial author of the code has worked for several international particle physics experiments at the European Organization for Nuclear Research CERN (Geneva, Switzerland) as well as the Stanford Linear Accelerator Center SLAC (Stanford, USA). All authors have been involved over several years in running, maintaining, and delivering services for one of the largest academic Grid installations world wide.

Geneva was developed with kind support from Karlsruhe Institute of Technology [101], Steinbuch Centre for Computing [103], as well as the Helmholtz Association of German Research Centres [115].

[1] . . . which has beed successfully implemented outside of the core Geneva codebase

1.1. Roadmap

The mentioning of evolutionary algorithms in the software's name only represents part of the truth nowadays, as Geneva now also implements several other optimization algorithms. At the time of writing, Geneva also covers particle swarm optimization, an implementation of gradient descents and a form of simulated annealing. There is also an implementation of parameter scans, covering all currently allowed parameter types.

All algorithms are based on the same data structures also used with evolutionary algorithms. Other optimization algorithms will follow and are indeed straight forwarwad to implement.

1.2. Functionality

The optional, fine-grained control over Geneva's inner workings – combined with sensible default values – as well as the type-agnostic implementation (floating point-, integer- and boolean-parameters are currently supported) make it easy for users to define the parameter space underlying their optimization problem, and to model the associated evaluation function.

In the Geneva code base, several libraries of more general applicability have been identified and separated from the optimization-centric code. This allows to use the Geneva library collection for more than "just" optimization. Among the features likely useful for other purposes is a broker infrastructure which handles communication between the server and consumers such as networked worker-nodes– and a random number factory – which fills random number buffers in several threads even when other parts of the application are idle. There is also a library that allows to easily create plots, which in the context of parametric optimization can be very useful to illustrate the progress over time.

Due to the variety of different algorithms, many essential parts of optimization algorithms have been implemented in base classes. Hence Geneva's components can also be used to implement new algorithms and to experiment with modifications of existing algorithms.

1.3. Scalability

From a user's perspective, distributed and multithreaded execution can be achieved just as easily as serial execution on a single CPU-core. The library has been tested with a hundred distributed nodes, each contributing its share to solving a large optimization problem in parallel. Performance, portability and extensibility are at the heart of the C++-based, purely object-oriented design.

1.4. Architecture

Geneva's architecture is based on the assumption that optimization problems are so complex that the evaluation of parameter sets will consume most of the processing time of the code, and that evaluation cycles will take minutes, hours or days, rather than just a fraction of a second. The efficiency of the application will thus usually be dominated by the implementation of the user-supplied and problem-specific evaluation function. Therefore, Geneva's core libraries can afford to put particular emphasis on stability.

1.5. Platform

The Geneva library collection has so far predominantly been developed on Linux, as this is the platform used by most Grid, Cloud, and Cluster installations.

At the time of writing, several different brands of Linux are supported, as well as versions of FreeBSD. An experimental port to MacOS exists, with a Microsoft Windows port envisaged for the future. Porting Geneva to other platforms than Linux will happen as the need arises.

The code base has minimal external dependencies. The only external components required come from the Boost library collection, which itself is designed to be highly portable, just like the CMake build environment.

Geneva compiles with different versions of the GNU Compiler Collection, as well as CLang. A test suite checks for errors during nightly builds.

1.6. Licensing

The code of the library collection, at the time of writing in version 1.6 (Ivrea), is available under a well known and established Open Source license – the Affero GPL v3 (see section D.1 for the full license).

1.7. Contact

Please contact us at `contact@gemfony.eu` if you would like to know more about the implications of using an Open Source product. Likewise, please do contact us if you are interested in customized and commercial licensing options or would like us to support you in assessing the viability of using Geneva for your optimization problems.

If there are more general questions you wish to ask we suggest that you post on one of the community fora available through the Gemfony web page (`http://www.gemfony.eu`). If you would like to help improve Geneva we kindly request that you submit feature requests and bugs through the Launchpad portal (see `http://www.launchpad.net/geneva`).

With this said, we hope that Geneva will increase your productivity and help you to solve the most daunting optimization problems. Please share your use-cases. We will consider the most interesting success stories on the Gemfony web page.

Enjoy! The Gemfony team.

Part I.

Optimization Algorithms and Theory

Chapter 2.

General Considerations and Overview

This chapter wants to define the term **optimization**, as used in this document, and to highlight some general considerations common to all types of optimization problems.

Key points: (1) All optimization problems need some sort of a metric, which can be used to evaluate and compare candidate solutions. (2) Computer-based automated parametric optimization can be defined as the search for the best *available* solution to a given problem under a number of practical constraints. (3) Noisy input data can be likened to fuzzy knowledge or deficiencies in a person's experience. (4) Finding the right balance between accuracy and efficiency can be difficult for models of reality. (5) Trying to find a good solution by evaluating several different values for each parameter will only work for very simple problem domains, as a very high multiplicity of candidate solutions will be created. (6) Optimization problems may feature different, possibly conflicting figures of merit. In this situation an optimization algorithm needs to find a suitable compromise or leave the ultimate choice to the user. (7) Optimization algorithms will always have to rely on the geometry of the quality surface, but might additionally use collaborative methods.

Literally translated, the Latin term *optimum* means *the best*. Following this translation, *optimization* would refer to the search for *the* best solution to a problem. Optimization is also part of daily life. Everyone strives to find optimal solutions to the problems faced in the course of carrying out ones duties, increasing the perceived benefit to oneself and any related entity.

The word *perceived* in the above sentence was used on purpose. Every individual will have a – possibly substantially – different understanding of what is, and what isn't, a suitable solution to a given problem. Reasons for this might include differences in each person's experience, or information that is available only to one person but not to the other. Personal taste and education will also play a significant role. Furthermore, where individual experience is involved, information might be used that is not consciously known or might not be explicable in easy terms to another person.

Finding good solutions will also often be an incremental procedure, where one starts from an acceptable starting point and, by changing individual aspects of a problem's solution, moves on to a better solution. Finding a *better* solution also implies some sort of a metric, as different candidate solutions need to be assessed and compared with each other. Ultimately, results obtained will differ from one person to the other, and different people will have varying success in their undertakings.

Some optimization tasks can be automated. However, from the discussion so far it should already have become clear that any sort of computer-based, automated optimization faces a number of difficulties.

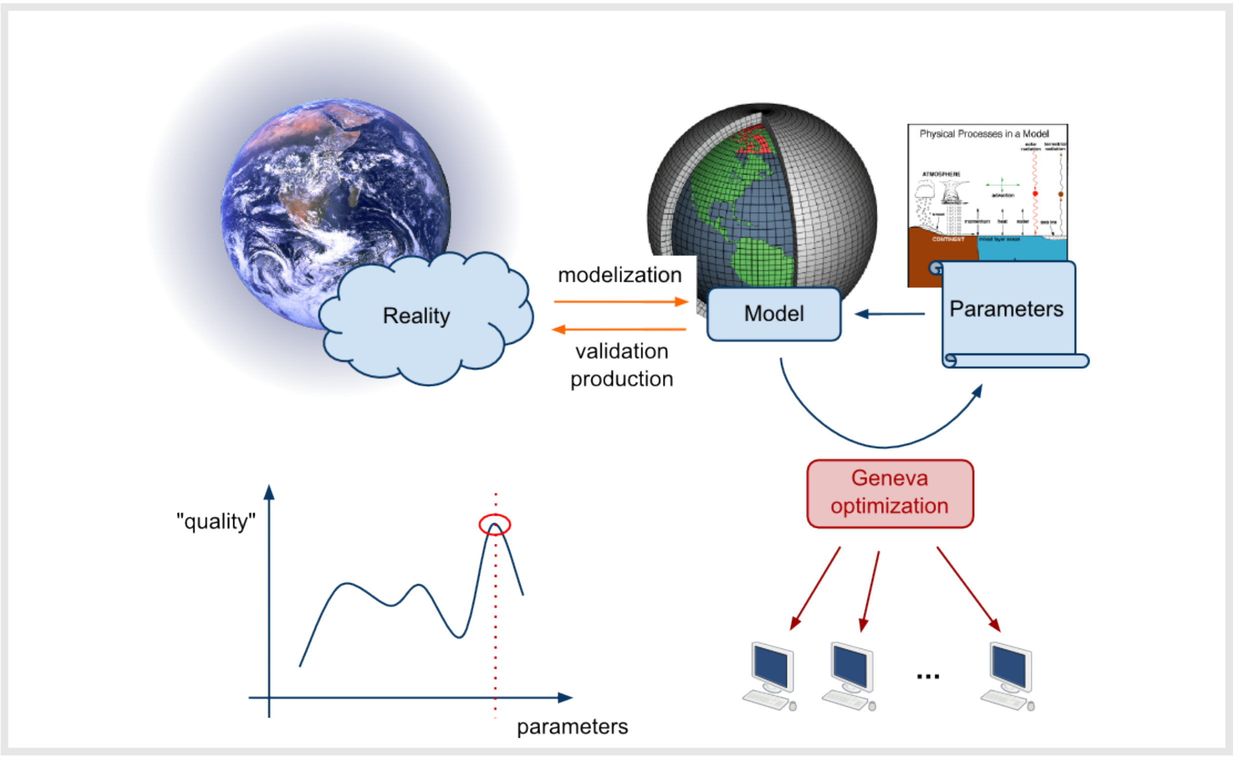


Figure 2.1.: *Optimizations often act on a model, whose resemblance to reality may vary.*

Most importantly, it codifies real-life problems and is thus subject to the same problems discussed above.

2.1. Models and Reality

It is also important to understand that an optimization procedure can never return better results than what is contained in the model of reality underlying the computation.

Figure 2.1 illustrates this fact. “Reality” is (thankfully) not by itself accessible to computer-based optimization. Rather, a digital model of reality is built inside of a computer. For example, in case of climate simulations one divides the atmosphere into volume elements. Calculations of the investigated parameters are then done individually for each volume element, but will take into account the interaction with its neighbors. Any optimization done on the basis of this simulation will have to live with its restrictions. And where the model in a subsection is wrong, the results of the optimization might differ substantially from what is “real”. E.g. it might assume that the vegetation consists of more trees than there are in reality, because someone cut them down in the meantime. This will have a significant impact on how close the simulation in an area comes to reality.

Creating very complex models might also yield bad results. One reason is that a complex model

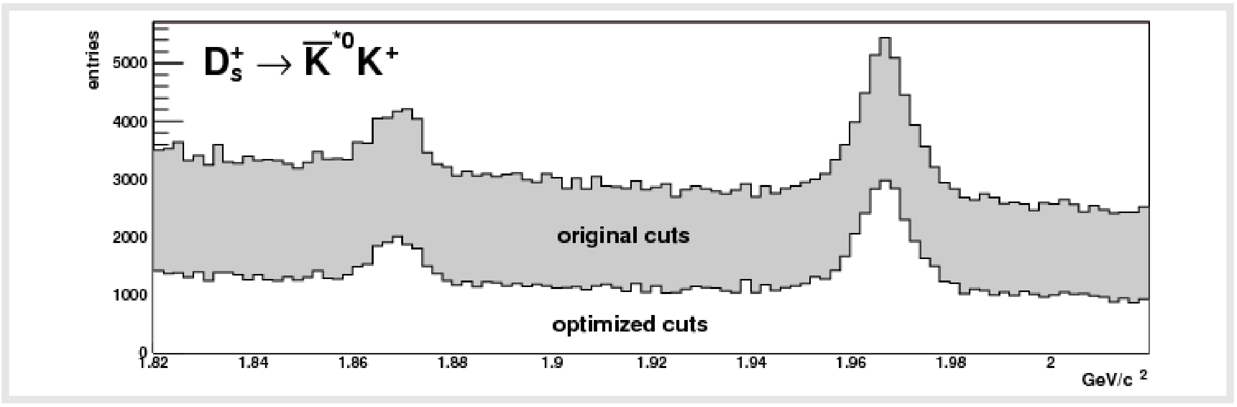


Figure 2.2.: *The EVA library (a predecessor of Geneva) was used in this example to optimize the selection of "events" coming from a particle physics experiment. A significant reduction of mis-reconstructed particles is visible (source: own pictures).*

is more likely to contain systematic errors that will likely go undetected. Secondly, complex models involve complex computations. And the limiting factor for many technical projects is the availability of resources, not a lack of knowledge Hence, only a smaller part of the parameter space will be explored, compared to a simpler model, as the computation would otherwise last too long. In the example shown in figure 2.1, the accuracy of the weather simulation will crucially depend on the granularity of the subsections, into which the (virtual) world has been divided. The smaller these sections, the better the simulation. On the other hand, the number of sections increases quadratically with the decrease of the sections' edge length. And the duration of the computation will be directly related to the number of sections. As there will be a limit to the amount of resources that can be assigned to the project, there will also be a lower limit to the size of the subsections, and hence to how close the simulation comes to reality. **Finding the right balance between accuracy and efficiency can be difficult.**

2.2. Choosing evaluation criteria

The success of automated parametric optimization crucially depends on the quality of the chosen evaluation criteria. Their formulation implies the codification of possibly implicit knowledge and will, for more complicated problems, likely be an iterative procedure[1]. Likewise, the success of computer-based optimization strongly depends on the experience of the engineer or expert in the chosen field of work. The result of the codification of a single evaluation criterion will be a mapping

[1] Automated parametric optimization can also help to make implicit knowledge explicit.

$$\vec{X} := \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \longrightarrow f(x_1,\ldots,x_n) =: Q \tag{2.1}$$

in the form of a *computer-implemented* procedure, which transforms a given set of input parameters into a single output value or quality Q[2]. $\vec{X}$ is called a *feature vector*. The fact that the mapping is computer-implemented means that it is expressed in terms of a programming language. It may thus imply any form of computer-based techniques, such as data-base look-ups, loops and logical branches (i.e. `if`-statements).

Thus, while $f(x_1,\ldots,x_n)$ can be likened to a mathematical function, it will often have additional, more unusual features. For example, it will very likely show discontinuities, and it will certainly not be differentiable[3].

2.3. A Single Evaluation Criterion

In the most simple case, parametric optimization means finding maxima or minima of a *single* evaluation criterion. Due to the special characteristics of computer-implemented evaluation criteria, though, standard mathematical procedures for finding the extreme values of Q can rarely be applied. And, as will be discussed in section 2.5, it is also possible to perform optimization in the presence of several, possibly conflicting evaluation criteria. We will first describe two examples for a *single* evaluation criterion in the next few paragraphs, though.

As a practical example, one might want to optimize the combustion in a car engine. The figure of merit that needs to be maximized might then be the torque for a given amount of fuel entering the combustion chamber. Parameters will certainly include the geometry of the combustion chamber, the amount, position and angle of fuel injectors, the pressure of the fuel entering the chamber and the timing of the ignition. Determining the torque for a given set of parameters will likely involve a computer-based simulation, as it will be virtually impossible to physically build a new engine for every set of parameters.

Hence the evaluation function is a computer-implemented procedure which maps the input parameters to the torque. Performing just a single evaluation of a given candidate solution could then take a significant amount of time – possibly minutes or even hours – as the chemical and physical processes inside of a combustion chamber are complex. It should also be very clear that standard mathematical algorithms for finding extreme values cannot be easily applied.

Another example stems from particle physics. Here, possibly unknown elementary particles are reconstructed from the tracks of other particles, as measured in particle physics experiments such as ATLAS or ALICE at LHC/CERN. The result will be a “peak” in a histogram.

[2]Depending on your problem, a high quality can also mean a low value of Q

[3]. . . although it might be possible to calculate approximate gradients – see chapter 3 on gradient descents.

The success of a scientific analysis will often depend on the quality of this peak. This quality in turn depends on a number of parameters ("cuts") of the (computer-implemented) analysis procedure, such as allowed momentum ranges and angles of particles emanating from a decay, or their so called invariant masses.

Running an analysis (as a means of determining the quality of the peak) can again take significant amounts of time, possibly hours. Finding suitable input parameters for the analysis can have a significant effect on scientific findings. Figure 2.2 shows a real-life example, achieved with a predecessor of the Geneva library[7].

Optimization problems with just one evaluation criterion are intuitive in the sense that their feature vectors $\vec{X_i}$ can be ordered (i.e. have a metric). In other words, it is possible to make a statement whether for two feature vectors $\vec{X_1}$ and $\vec{X_2}$

$$Q\left(\vec{X_1}\right) <= Q\left(\vec{X_2}\right) \quad \text{or} \quad Q\left(\vec{X_1}\right) > Q\left(\vec{X_2}\right) \tag{2.2}$$

is true. So the quality of both feature vectors can be directly compared. In two dimensions, it is even possible to visualize the quality surface, and analogons to the two-dimensional case help to also understand optimization problems with a higher number of parameters.

2.3.1. Why Is Brute Force Not Useful ?

If we assume for a moment that a single evaluation criterion $f(x_1 \ldots x_n)$ has been defined that depends on n floating parameters x_i, then one might reason that a good algorithm would be to try out a small number m of values for each parameter.

Let's assume that just $m = 5$ values should be tested per parameter (or dimension), and that $n = 10$. In order to find *the* best solution *in this set*, we would have to perform $m^n = 5^{10} = 9765625$ evaluations. Under the assumption that each evaluation takes a very short 1 millisecond[4], we would need just a little over two and a half hours to find the best candidate solution in the set on a single CPU core.

Particularly as the evaluation of different candidate solutions will usually be independent from each other, one could reason that this procedure can easily be parallelized. Indeed this is the case so that, ignoring the consequences of Amdahl's law[5], we might ideally be able to reduce this to less than 10 seconds on a massively parallel machine with 1024 CPU cores (or on a GPU).

Of course the globally best solution might be located somewhere in-between two tested parameter sets, which we would never find out about with the above procedure. So we might decide to look at 100 values per parameter instead (which would still not give us a guaranty for success). We would then have to perform $100^{10} \approx 10^{20}$ evaluations, taking a little over 3 million years on 1024 CPU cores. This would cost you 100 billion and produce over 100000 tons of CO_2[6].

It should thus be obvious that this procedure will only work for very simple problem domains, parti-

[4]...which will certainly not be sufficient for the combustion-chamber and particle physics examples in section 2.2

[5]See the discussion in section 8.5.2 for further information.

[6]We assume that each 8 cores are part of a single machine, which consumes 300 Watts, that each 1000 Watts cost you a very cheap 0.10 per hour, and that each 1000 Watts will result in the emission 1 kg of CO_2

cularly as many optimization problems feature a far higher amount of parameters than 10, and the evaluation of a single parameter set may be running for hours or even days at a time for complex optimization problems.

Likewise, finding good solutions "by hand" will generally be impossible for complex optimization problems, as we are usually dealing with very high-dimensional parameter spaces with many, possibly unknown, correlations between different parameters. Thus the effects of varying individual parameters will not be obvious even to the most experienced engineer.

2.4. Relying on quality surfaces

So, if brute force doesn't work, what information *can* an optimization algorithm rely on in order to find an acceptable solution to an optimization problem ?

It helps to remember that, in the case of optimization problems *with a single evaluation criterion*, we are dealing with a "simple" (from the mathematical point of view) transformation. If a problem description in pure mathematical terms was available, it would even be possible to search for the roots of the first derivative in order to identify the extreme values.

Since, in automated parametric optimization procedures, evaluation functions are usually expressed in some programming language, though, this option is not available. We can however calculate the value of the evaluation function at any point in the allowed parameter space we want. Hence, for continuous value ranges, we can still calculate the **difference** quotient (as opposed to the **differential** quotient), with a finite step width. This is essentially the basis of gradient descents, as discussed in detail in chapter 3. The simple idea here is that the next minimum (in case of minimization) lies down-hill. More specifically, the algorithm tries to walk in finite steps into the approximate direction of steepest descent.

Likewise, instead of trying to map large portions of the parameter space, most optimization algorithms (need to) rely on the shape of the quality surface. Note again, though, that this will work best in the presence of continuous surfaces and that the situation is more complicated if there are discrete parameters, such as booleans (compare section 4.3 on Genetic Algorithms).

Some algorithms, such as the Particle Swarm Optimization ("PSO") family (compare chapter 6), rely on collaborative information. Candidate solutions effectively follow a trail. They are "drawn" towards known good solutions, but also follow a path on their own, by adding a random element to their steps.

2.4.1. Local and Global Optima

If optimization (at least in the context of continuous parameter value ranges and with a single evaluation criterion) means finding the extreme values of the quality surface, then algorithms need to be able to cope with local optima in order to find the global best. Figure 2.3 illustrates the problem.

An algorithm that solely relies on the local geometry of the quality surface has no way of knowing, whether it is currently in a local optimum or not. There are even problem domains (such as functions

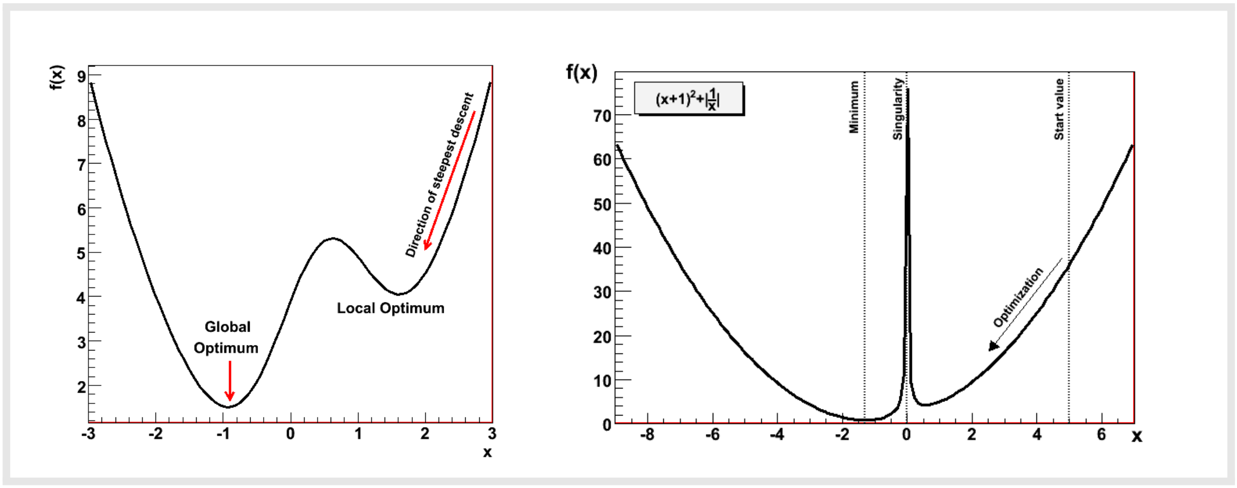


Figure 2.3.: *Local optima can prevent an optimization algorithm from finding the global optimum*

with singularities), where, from a theoretical perspective, there is "no way out" of a local optimum[7].

In such situations, and in the presence of discrete parameters, collaborative methods can help. They determine new candidate solutions on the basis of not one known solution (and its surrounding area), but two or more of them. Nevertheless this is no guaranty for success.

2.5. Multi-Criterion Optimization

The situation becomes even more complicated when more than one evaluation criterion is present. After all, a feature vector $\vec{X}$ that minimizes one evaluation criterion does not necessarily minimize another criterion – think back at the example with the car engine. Equation 2.1 then becomes

$$\vec{X} := \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \longrightarrow \begin{pmatrix} f_1(x_1,\ldots,x_n) \\ f_2(x_1,\ldots,x_n) \\ \vdots \\ f_m(x_1,\ldots,x_n) \end{pmatrix} =: \begin{pmatrix} Q_1 \\ Q_2 \\ \vdots \\ Q_m \end{pmatrix} =: \vec{Q} \qquad (2.3)$$

Obviously, it is then no longer possible to compare two feature vectors $\vec{X_1}$ and $\vec{X_2}$, as there is no defined metric for a vector of evaluation criteria $\vec{Q}$.

As an example, one might not only want to build a combustion chamber which produces an as high torque as possible, but might also be looking to minimize the emission of pollutants, and to simultaneously make the engine as quiet as possible. It is safe to assume that it is *impossible* to satisfy all three evaluation criteria with the same parameter set to the same extent.

[7]Of course, with finite steps in the parameter space, there is always a chance to step out of the trough . . .

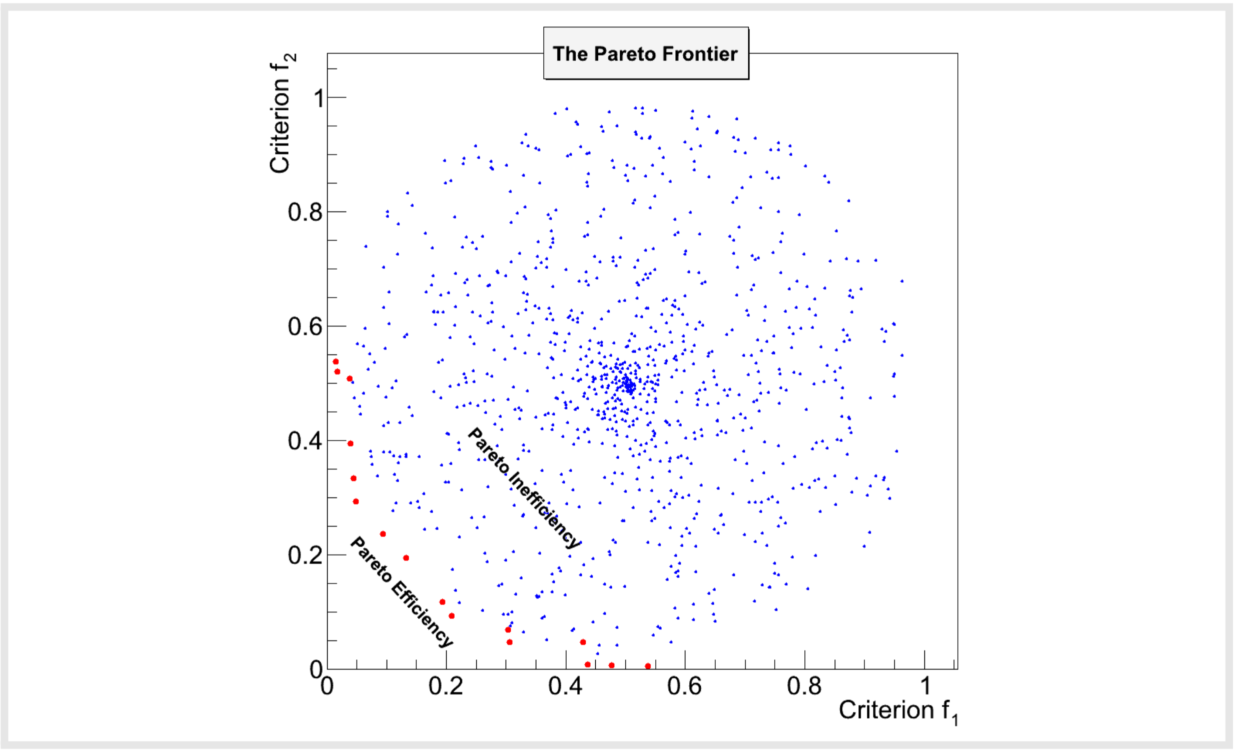


Figure 2.4.: *The Pareto Frontier is defined by data points that are not dominated by other solutions*

2.5.1. Amalgamating Evaluation Criteria

One way an engineer could take to resolve this situation is to specify, where an “ideal” quality for each criterion is located (which might not always be easy). A common goal would then be to minimize the deviation from the ideal qualities. Of course this implies replacing the three evaluation criteria with a single one, and the engineer needs to define an “ideal” solution for each sub-criterion. A new “global” evaluation criterion for m sub-criteria, depending on n parameters, could then become

$$\tilde{Q}=\sqrt{\left(f_1(x_1,\ldots,x_n)-Q_1^{ideal}\right)^2+\ \ldots\ +\left(f_m(x_1,\ldots,x_n)-Q_m^{ideal}\right)^2} \tag{2.4}$$

This brings us back to square one, though, as, from an optimization perspective, the new evaluation criterion $\tilde{Q}$ differs from Q in equation 2.1 by the fact that it is more complicated. In particular, we again have a metric that allows us to determine whether

$$\tilde{Q}\left(\vec{X_1}\right)<=\tilde{Q}\left(\vec{X_2}\right) \quad \text{or} \quad \tilde{Q}\left(\vec{X_1}\right)>\tilde{Q}\left(\vec{X_2}\right) \tag{2.5}$$

Geneva contains various methods for combining multiple evaluation criteria, including the one described in equation 2.4, and allows to define custom combination schemes.

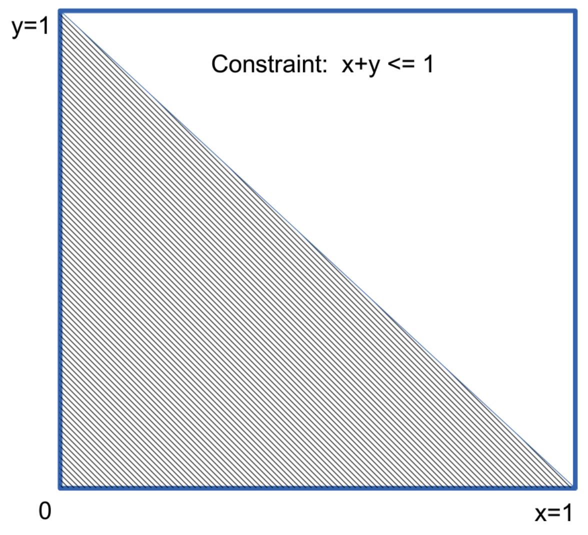


Figure 2.5.: *Boundary condition involving two parameters*

2.5.2. Pareto Optimization

There are other methods for performing multi-criterion optimization, which do not rely on a single target criterion. They do however often depend on the optimization algorithm. Here, we just want to hint at a method that is often used with evolutionary algorithms – after all the starting point of the Geneva library – and that is implemented in Geneva.

In a nutshell, in pareto optimization, one distinguishes between dominated ("pareto-inefficient") and dominating ("pareto-efficient") solutions. Being pareto-inefficient means that for a given feature vector $\vec{X}$ with a set of evaluation criteria f_i there is at least one other solution $\vec{X}^*$ for which at least one f_i is better and none of the other is worse than for $\vec{X}$.

In the end, the user gets a collection of solutions that fulfill the pareto condition and needs to decide himself which one is best suited for solving his optimization problem. In a way this also means that the computer-implemented evaluation criterion has been augmented by a "user-implemented" criterion.

Figure 2.4 illustrates this situation on the example of a two-dimensional data sample distributed randomly in a circle. Lower values are preferred. The red dots mark the pareto frontier.

2.6. Parameter Constraints

It may well be that not all combinations of the feature vector $\vec{X}$ are allowed. In the easiest case, the value range of individual parameters is constrained independently from other parameters. For example, it might be known that the solution to an optimization problem can be found inside of a "box", in the case of 3 floating point parameters[8]. In this case each parameter would be constrained to the lower and upper edge of each dimension.

However, there may also be cases where a constraint for one parameter depends on the current value of another parameter. This is illustrated on a simple example in figure 2.5. For a real-life example that would be described through the same equation $x+y \leq C$ (where C is a constant), think of two boxes that need to be packed side by side into a larger box. Only if the width of both boxes combined is not larger than the width of the surrounding box, will they fit. This is a problematic situation for optimization algorithms that will usually try to vary parameters freely in order to find the optimum.

And while this situation could still be resolved rather easily for the case presented in figure 2.5, finding solutions for the general case is far more tricky. Chapter 16 discusses a number of different approaches in the context of the Geneva library.

2.7. Definition

As a consequence of the variety of difficulties discussed in this chapter, “the best” solution[9] of an optimization problem will rarely be found. Rather, one will usually only be able to find “the best accessible” solution under a given number of constraints.

Among these, “logical constraints”, such as dependencies between parameter-boundaries, play a dominant role, as do “procedural constraints”, such as the available time and amount of computing resources. Other very important criteria include the amount and quality of the available information[10], and the quality of the chosen “model of reality” or, in more general terms, the quality of the evaluation criteria. Note that the latter is again closely linked to the experience of the engineer who has formulated the evaluation functions. We can conclude:

In the context of this document, automated parametric optimization is defined as the search for the best accessible solutions to a computable mapping $\vec{X} \rightarrow \vec{f}\left(\vec{X}\right)$, under a given set of logical or procedural constraints.

Here $\vec{X}$ is a set $(x_0, x_1, \ldots, x_n)$ of parameters, $\vec{f}\left(\vec{X}\right)$ may either be a single evaluation criterion, or a set of possibly conflicting criteria, amongst which further arbitration is needed to choose “the best available” solution.

[8]Geneva supports constrained value ranges of integer and floating point parameters. An example for a constrained parameter type would the `GConstrainedDoubleObject`, introduced in chapter 11.

[9]One might also call it the “ideal” solution ...

[10]In computer-based optimization, noisy input data can be likened to fuzzy knowledge, or possibly also deficiencies in a person’s recollection.

Chapter 3.

Gradient Descents

When water flows out of a spring, it will follow the path of steepest descent. On its way down-hill, it might temporarily end up in a local valley, forming a pond or little lake, until it flows over the valley's edge. Eventually it will end up in the ocean – in a manner of speaking the global optimum for water. These simple and easily understandable facts form the basis of a family of popular optimization algorithms, whose principles are discussed in this chapter.

Key points: (1) For a continuous mathematical transformation $\mathbb{R}^n \rightarrow \mathbb{R}^1$ it is possible to calculate the direction of steepest descent in a given position $\vec{X}^*$ by calculating the n partial derivatives $\frac{\delta f(\vec{X})}{\delta x_i}$ (2) Making a step in the direction of steepest descent will eventually leed to an optimum (albeit not necessarily the global optimum) (3) A similar method can be used to find optima of transformations represented by computer-implemented functions with floating point parameters. (4) As the most important difference, the partial derivatives need to be replaced by the difference quotient $\frac{\Delta f_i(\vec{X}^*)}{\Delta x_i^*}$ (5) Gradient descents are self-regulating and will efficiently find the *next* optimum (6) This algorithm type will easily get stuck in local optima and is best used for the "last mile", starting from a known good solution (7) Another disadvantage is a direct dependency of the number of necessary calculations and thus the algorithm's computational overhead on the number of parameters taking part in the optimization.

3.1. Mathematical Background

Imagine the $f:\mathbb{R}^1 \rightarrow \mathbb{R}^1$ transformation $f(x)=x^2$. In order to find f's root, one would calculate the function's first derivative, then find the value of x for which $\frac{df}{dx}$ becomes 0. $\frac{df}{dx}$ represents the gradient of f. In the case of a mapping $f:\mathbb{R}^n \rightarrow \mathbb{R}^1$ the gradient becomes a vector:

$$g(\vec{x}) = \nabla f = \begin{pmatrix} \frac{\delta f(\vec{x})}{\delta x_1} \\ \frac{\delta f(\vec{x})}{\delta x_2} \\ \vdots \\ \frac{\delta f(\vec{x})}{\delta x_n} \end{pmatrix} \tag{3.1}$$

The seemingly complex equation 3.1 actually expresses a very simple fact: You can find out the direction of steepest descent $g(\vec{x})$ of a function $f(\vec{x})$ in a position $\vec{x}^*$ by calculating the first partial derivative for each of its parameters x_i. $g(\vec{x}^*)$ will then point into the direction into which water would flow if the function were a mountain. If we follow this path, we will get to the next optimum.

Calculating the partial derivative $\frac{\delta f(\vec{x})}{\delta x_i}$ means: Assume all parameters of f (except for x_i) to be constant. Then make an infinitesimal step into the direction of x_i and calculate how f changes.

The term "*infinitesimal step*" means: "*a step of infinitely small size*". Infinitesimally small numbers are a mathematical abstraction, which of course cannot be expressed by "real" quantities.

3.2. Application to Real-Life Problems

The procedure shown in section 3.1 has made the implicit assumption that $f(x)$ is indeed differentiable, i.e. that it is possible to calculate $\frac{\delta f(\vec{x})}{\delta x_i}$ for every i. For real-life problems, such as the simulation of the combustion in a motor, this will rarely be the case.

The simple reason is that it is impossible to express the processes happening during a combustion through a mathematical function. Instead, what you get is a computer-implemented procedure, and you just cannot calculate the first derivative of an `if`-statement.

Gradient descents[1] only act on real values, which are usually represented by floating point types (i.e.. `float`, `double` and sometimes `long double`) in C++. Where a computer-based simulation can give you a single, numerical figure of merit rating the combustion, what you essentially get is a mapping $f:\mathbb{R}^n \rightarrow \mathbb{R}^1$ [2]. I.e., for every set of floating point parameters describing the combustion, you get a single floating point value back which describes whether the combustion was "good" or bad.

The associated function might not be differentiable. But nevertheless the procedure discussed in section 3.1 can be amended in such a way that it can also be applied to non-differentiable functions. The simple idea is to replace the infinitely small step with one of finite length. Hence, instead of $\frac{\delta f}{\delta x_i}$, in a given location $\vec{x}^*$, we now need to calculate

$$D_i\left(x_i^*\right) := \frac{\Delta f_i(\vec{x}^*)}{\Delta x_i^*} = \frac{f(\vec{x}^* + s * \vec{e}_i) - f(\vec{x}^*)}{s} \tag{3.2}$$

Here s represents a small variation of x_i, and $\vec{e}_i$ is the unit-vector of x_i, i.e. a vector of length 1, with 0s for all parameters except for x_i (compare equation 3.3). In mathematical terms we would speak about the difference quotient, as opposed to the differential quotient.

[1] Note that the term "gradient descent" will usually refer to the computer-implemented procedure involving **difference-quotients** in this chapter.

[2] This of course assumes that the simulation only depends on real values (see section 3.3 for a discussion). It also treats "floating point" values the same as the mathematical abstraction "real value". Also note that a simulation might produce additional figures of merit, such as information about the pollutants being produced. A gradient descent can only search for the minimum of one of them at a given time.

$$\vec{e}_i := \begin{pmatrix} e_1 \\ e_2 \\ \vdots \\ e_i \\ \vdots \\ e_n \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{pmatrix} \tag{3.3}$$

By making *finite* (as opposed to infinitesimal) variations of the x_i, we make a small error. The error's magnitude will depend on the value of s. s should thus be chosen as small as possible, but needs to take into account your system's limited floating point accuracy, particularly as equation 3.2 mandates that you need to devide a numberical quantity by $\mathtt{s}$.

Starting in a given location $\vec{X}^0$, we now need to calculate the (approximate) direction of steepest descent, then make a small step in this direction. This procedure needs to be repeated, until we reach a satisfactory optimum[3]. The position update in each iteration is calculated according to

$$\vec{X}^{k+1} = \vec{X}^k - \delta * D\left(\vec{X}^k\right) \tag{3.4}$$

if we are searching for a minimum, or

$$\vec{X}^{k+1} = \vec{X}^k + \delta * D\left(\vec{X}^k\right) \tag{3.5}$$

if we want to maximize the evaluation function. δ represents the size of the step made in each iteration. Suitable values need to be chosen individually for each optimization problem. Note that we have used the definition

$$D\left(\vec{x}^*\right) := \begin{pmatrix} D_1\left(x_1^*\right) \\ D_2\left(x_2^*\right) \\ \vdots \\ D_n\left(x_n^*\right) \end{pmatrix} \tag{3.6}$$

in equations 3.4 and 3.5.

Different variations of this basic algorithm exist. In particular, it is common to perform "line search" into the direction of steepest descent before calculating the next difference quotient.

Another important variation is represented by the *Conjugate Gradient* algorithm.

[3]See section 17.3 for a discussion of halt criteria

3.3. Inquest

The largest problem is the robustness of the algorithm in the presence of local optima. It is quite likely that a gradient descent, as defined in this chapter, does not get beyond the next local optimum. As we are making finite steps, we have a certain likelihood to step out of a valley[4]. However, it is far more likely that we will get ever closer to the valley floor instead and will never reach the global optimum. And unlike water, we cannot simply "fill" the valley and flow over its edge. **It is thus recommended to combine gradient descents with other optimization algorithms, letting them handle the "rough" work, and using the gradient descent to accurately pinpoint the current optimum.**

Then, gradient descents can only deal with real values (i.e. their approximation through floating point values in computers). You will have to resort to other algorithms if your parameters involve significant discrete values (such as integer- or boolean-values) that cannot be neglected during the optimization procedure. Note, though, that some other optimization algorithms also have this restriction.

Another potential disadvantage is the computational overhead. For every iteration, the evaluation function needs to be executed $n+1$ times, where n is the number of parameters. This can lead to very long computations for large values of n. Few other optimization algorithms exhibit this explicit dependency. For small numbers of parameters however, this can also turn out to your advantage.

On a side note, the fact that you will always have to run the evaluation function exactly $n+1$ times leads to some difficulties and inefficiencies when trying to parallelize gradient descents. Section 8.5.4 discusses this in more detail.

Advantages of gradient descents include their relative simplicity – they can be easily implemented and are thus certainly amongst the most used optimization algorithms in information technology.

Gradient descents are also self adjusting, as a steep descent corresponds to a large multiplicative factor $D\left(\vec{X}^k\right)$ to the constant step width δ (compare equations 3.4 and 3.5).

Likewise, the algorithm will make small adjustments in shallow areas. It is thus quite efficient in giving you a good approximation for the current (possibly local) optimum.

Another disadvantage lies in the fact that you will only be able to deal with problems that can be described in terms of a mapping $f:\mathbb{R}^n \rightarrow \mathbb{R}^1$. Or, in other words, gradient descents are unsuitable for multi-criterion optimization.

[4]here we assume that we are searching for a minimum

Chapter 4.

Evolutionary Algorithms

Charles Darwin has described the principles of evolution in the 19th century – the results of his research were published in *The Origin of Species*[16] in 1859. The history of *Evolutionary Algorithms* ("EA") of course doesn't go back quite as far, but still is impressive.

Already in the 1950s, first attempts were made to model evolution with computers. Artificial evolution as a means of solving parametric optimization problems became more widely known as a result of the work of Ingo Rechenberg and Hans-Paul Schwefel on *Evolution Strategies* ("ES")[63] [64] in the late 1960s and 1970s. John Holland introduced another variant – *Genetic Algorithms* ("GS")[36] in the early 1970s.

Evolution Strategies (and Genetic Algorithms), as two major representatives of Evolutionary Algorithms, will be discussed in more detail in this chapter. Other representatives, such as Genetic and Evolutionary Programming, will not be discussed here.

Key points: (1) ES and GA share a cycle of duplication/recombination , mutation and selection (2) ES act on floating point numbers (3) ES can cope well with local optima, but is less efficient than a Gradient Descent close to the global optimum (4) GA act on boolean values (5) Different parameter types are possible and may be mixed (6) EA suffers from a comparatively large number of configuration parameters, and finding suitable values for them can be an optimization task by itself

4.1. Common Features

Biological evolution follows a cycle of recombination and selection and also benefits from mutations in a species' gene pool. Like their archetype, Evolutionary Algorithms act on a population of *individuals* or *candidate solutions*. This section discusses this and other similarities between different types of Evolutionary Algorithms.

The discussion centers around the core algorithm, which is identical for Evolution Strategies and Genetic Algorithms. As shown in listing 4.1, it can be expressed in just a few lines of code.

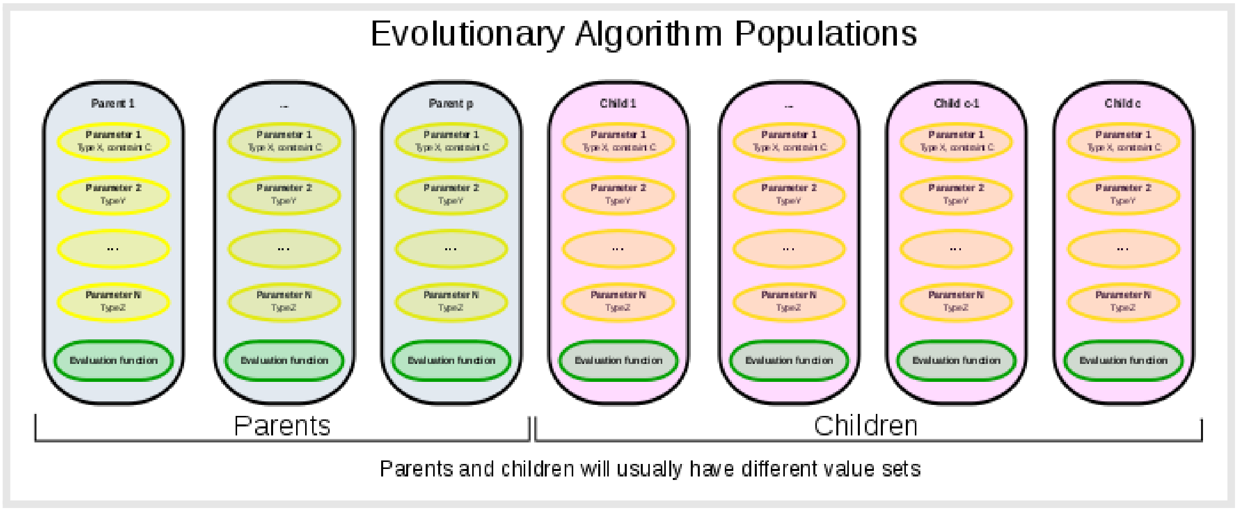

Evolutionary Algorithm Populations
Parent 1
...
Parent p
Child 1
...
Child c-1
Child c
Parameter 1 Type X, contain C
Parameter 2 Type Y
...
Parameter N Type Z
Evaluation function
Parents
Children
Parents and children will usually have different value sets

Figure 4.1.: *Evolutionary Algorithm populations consist of $p >= 1$ parents and $c >= p$ children*

Listing 4.1: Evolution Strategies and Genetic Algorithms share a common workflow

```
initPopulation();         // Create an initial set of parents
do {
        recombine();      // create copies of parents or recombine their features
        mutate();         // modify individual parameters
        select();         // evaluate candidate solutions and select the best

        generation++;     // Increment the generation counter
}
while(!halt());           // Terminate optimization when a halt criterion triggers
returnBestIndividual();   // Return the best individual found
```

4.1.1. Initialization

The algorithm starts with a predefined or randomly initialized selection of individuals specific to a given optimization problem. This initial set of candidate solutions is called *parents*. Predefined sets of individuals will often represent known good solutions. Randomly initialized candidate solutions will need to observe the boundary conditions for the optimization problem. In C++, candidate solutions will usually be represented as classes, whose most important component is the feature vector $\vec{X}$, i.e. a collection of parameters describing the current solution. There will also be an evaluation criterion Q associated with $\vec{X}$, which rates its *quality* or – using EA terminology – its *fitness*. Section 2.2 has

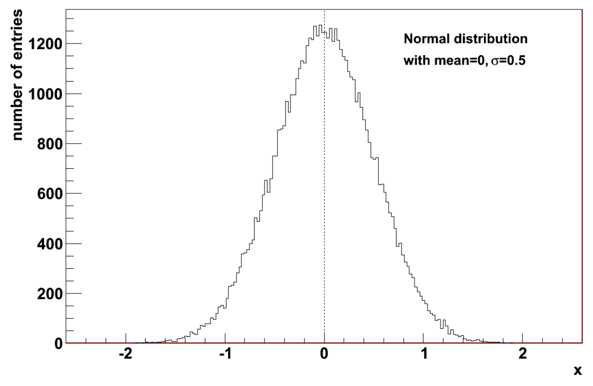


Figure 4.2.: *The most common mutation operator in Evolution Strategies adds gaussian-distributed random numbers to a given floating point parameter. The success of the optimization procedure crucially depends on the right choice for the width σ of the gaussian.*

discussed this topic in more detail[1].

4.1.2. Recombination

In the `recombine()` step, new candidate solutions are created from the best solutions known so far – the *parents* – using a multitude of possible methods. They will generally involve features of their parents and are thus usually referred to as *children*. Any number $p >= 1$ of parents may be involved in the creation of new child individuals, and the number c of children will usually be much larger than the number of parents. Together, parents and children form a population. Figure 4.1 illustrates this situation in more detail.

In the most trivial case, children will just be identical copies of a single parent, and will only gain features of their own in the `mutate()` step. The process could then be more accurately described as *duplication*. Even in this simple case, however, variations of the base algorithm exist. As just one example, there may be more than one parent, and the likelihood for one of them to be chosen as the origin for a new child might be chosen evenly or could involve the fitness of this particular parent, compared to the other parents.

There is a huge variety of recombination schemes involving more than one parent. Amongst the more usual ones is the *cross-over* scheme: In this context, a child could be created by adding the first half of the feature vector of parent A to the second half of parent B. There will also be a likelihood for such recombination schemes to be actually carried out.

Where the recombination step *does* involve more than just duplication, it could be likened to the biological recombination sequence, in which features of the female and male genome are amalgamated.

[1] Note again, though, that we might be looking to minimize Q – in this sense the terms “quality” and “fitness” are misnomers.

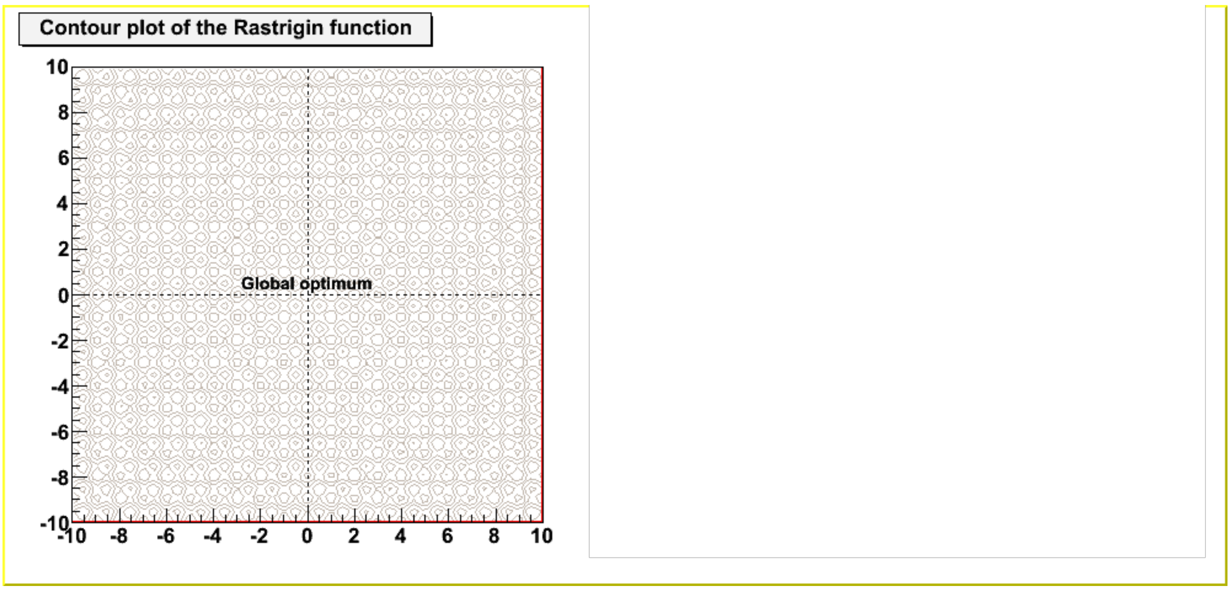


Figure 4.3.: *The Rastrigin function is a common test function for Evolution Strategies. Its major characteristic is a large number of local optima*

4.1.3. Mutation

In the `mutate()` step, random modifications will be applied to all or some parameters of the feature vector. Depending on the implementation, there will be parameters steering the characteristics and likelihood of the mutations being used. Section 4.2 will discuss some of the parameters for Evolutionary Strategies. Mutation also happens in natural evolution and might be triggered by radioactivity (such as cosmic radiation), as well as other causes.

4.1.4. Selection

Finally, in the `select()` step, child individuals will be rated. It is here that the evaluation function is required. It will translate the feature vector $\vec{X}$ into one or more numeric evaluations (see also section 2.2). In biological terms, the chance of survival scales with the fitness of a candidate solution. This creates an evolutionary pressure towards better (ideally optimal) solutions.

In programming terms, the chances of survival of an individual are simply determined by **sorting the population** according to a given selection criterion. It is here that new parents for the next generation are selected, and the cycle continues again with the duplication/recombination step, based on the new set of parents.

Note that in the selection step, again different modes are possible, depending on the chosen sorting criterion. In the case of a single evaluation criterion:

1. New parents could be chosen from the entire population, including parents. In this case, ex-

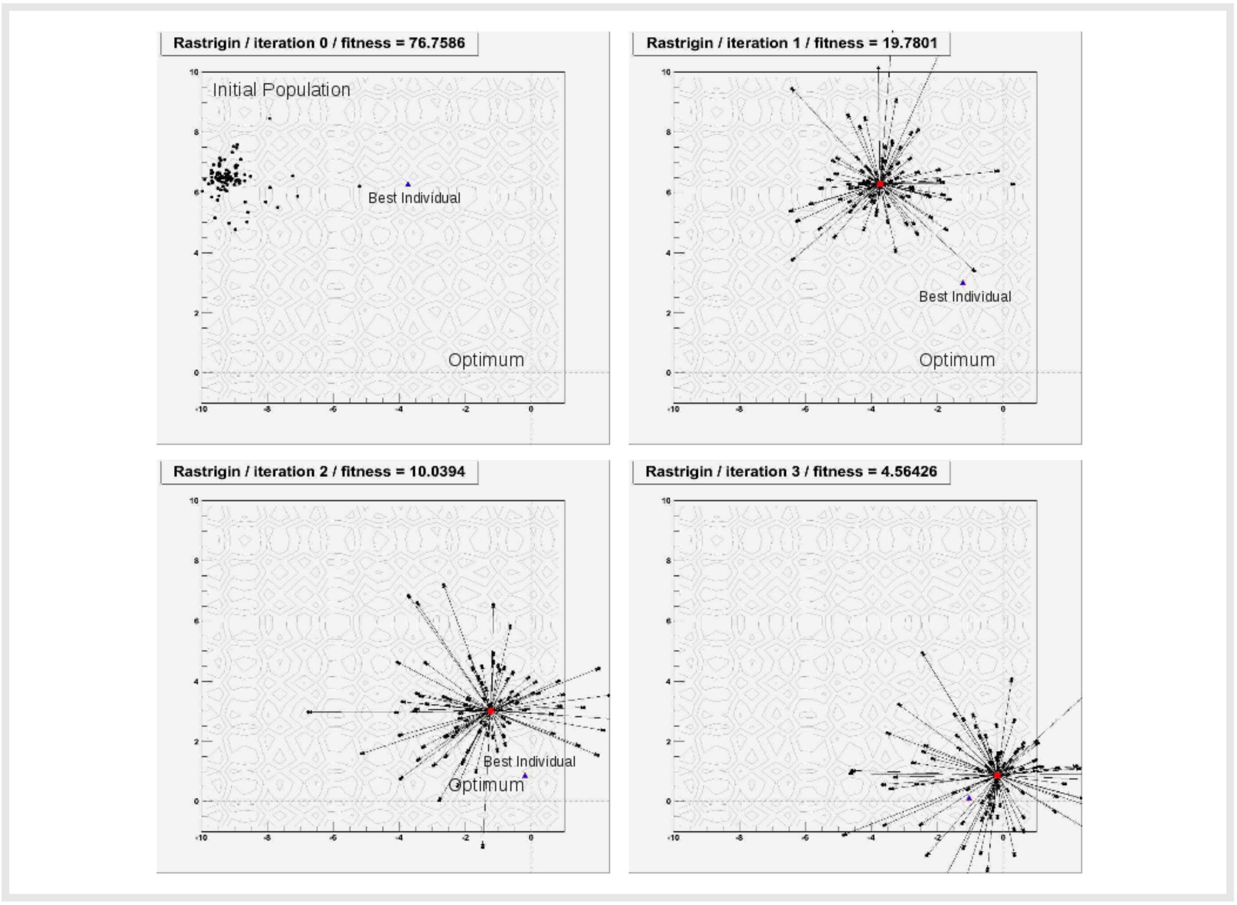


Figure 4.4.: *This picture demonstrates how an Evolution Strategy with a single parent searches for the minimum of the Rastrigin function in two dimensions. The Rastrigin function has a very large number of local optima (compare figure 4.3). Solely "Gauss" mutation is being applied to the parameters, and no recombination schemes are being used.*

isting parents could again become parents in the next generation, if no children with a higher fitness were found so far. One consequence of this selection scheme is that the overall fitness cannot get worse. Experience shows, however, that in this case the algorithm might converge prematurely. This mode of selection is often referred to as the $(\mu+\nu)$ selection scheme in this manual.

2. New parents could be chosen from the collection of children only. In this case, if no children with better quality than the parents were found, the overall fitness of the population (defined as the fitness of the best individual in it) can actually *decrease*. In theory, the fitness might even diverge. Experience shows, however, that one will almost always have a satisfying convergence behaviour, with only marginal decreases of the overall quality. The reason for this is that, through the selection procedure, one will always get the best *available* solution, given the chosen scheme, so that there is an evolutionary pressure towards the best solution, which usually prevents divergence. This mode of selection is often referred to as the (μ,ν) selection

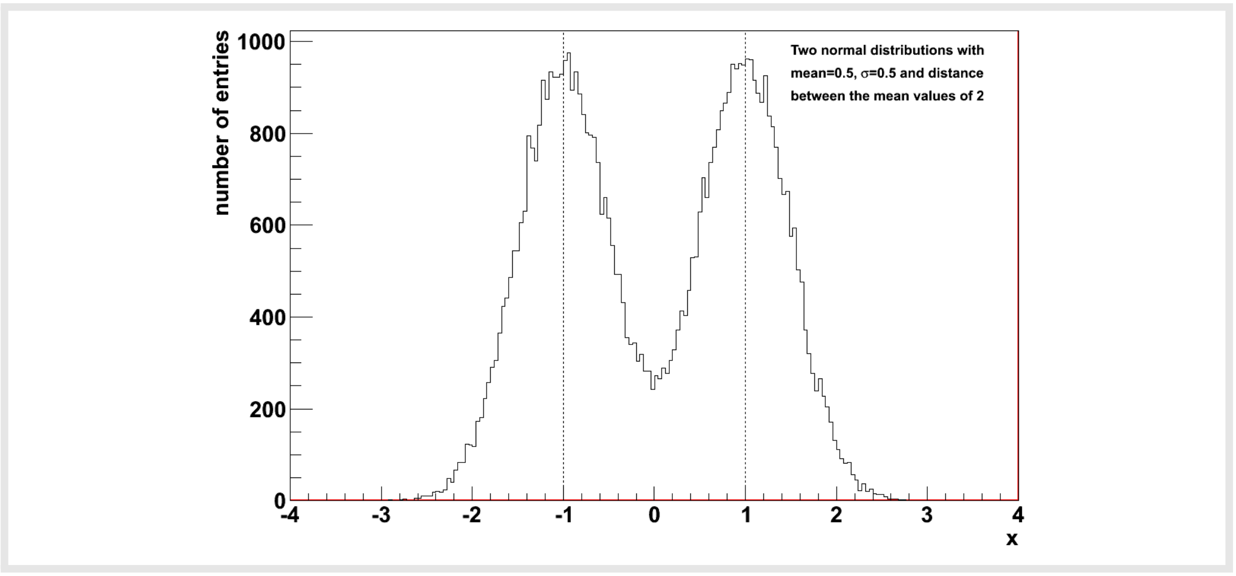


Figure 4.5.: *Using two gaussians instead of one for the mutation of floating point parameters in Evolution Strategies might boost performance, if the current best solution is still far away from the global optimum.*

scheme in this manual.

3. Hybrid modes are possible. E.g., Geneva implements a mode, where only one parent of the old population are retained (unless better children were found), and remaining parents are replaced by the best children found in the current generation. This mode of selection is often referred to as the hybrid selection scheme in this manual.

Selection criteria will become more complex and involve further, usually collaborative, information beyond the mere fitness of an individual, when using more than one evaluation criterion. One example would be a selection scheme, were parents are selected from the pareto front (compare section 2.5.2 and particularly figure 2.4).

Further selection schemes are possible. As an example, *Simulated Annealing*, as discussed in chapter 5, is implemented in Geneva as a special sorting mode in what must otherwise be considered to be evolutionary algorithm.

4.1.5. Halt Criteria

The optimization cycle will continue, until a halt criterion is reached (in which case the `halt()` function will return `true`). There is again an abundance of possible criteria, including:

- The optimization engineer might simply limit the maximum number of iterations or the amount of time an optimization may take. Such halt criteria will be chosen if – as will almost always be the case – the resources available for the optimization are limited.
- One might define a quality to be achieved by the optimization and let the optimization run until

this is the case. Note that this is dangerous as the optimization progress might stall, and the optimization process will carry on indefinitely

- It is possible to detect, however, if the optimization has stalled for a number of generations (i.e. no better solution was found for some time), in which case further optimization might not be useful and the halt() function could trigger (if desired by the user)
- One might want to combine several of these criteria, e.g. one might demand a quality to be achieved, but nevertheless terminate the optimization run after a given amount of time, if the desired quality hasn't been achieved yet.

Note: As stated in section 2.7, generally the best *possible* (or *ideal*) solution to an optimization problem will rarely be found, particularly for high-dimensional optimization problems. Hence it is usually impossible to know "how far" a given solution is from the ideal solution, and this distance can consequently not serve as a halt criterion.

4.1.6. Retrieval of the best available solution

When the optimization cycle is halted, there will be one or more "best" solutions. They need to be returned to the optimization engineer for further processing, or can serve as input for another optimization algorithm.

4.2. Evolution Strategies

Evolution Strategies apply the algorithm of listing 4.1 to optimization problems that solely use real numbers[2] for their parameter definitions. This implies a number of limitations, but still appears to be the natural way of describing many, particularly technical, optimization problems.

4.2.1. Specific Recombination Schemes

Apart from the "usual" cross-over scheme, real numbers allow for a number of recombination methods unique to real numbers. These might also involve more than just two individuals. E.g.:

- One might want to choose children only from the parameter space defined by a line stretching between two parents
- It is possible to define a circle using two parents, which are interpreted to be located at the opposing ends of this structure
- One might construct a triangle from three parents and then create children randomly on the edges of the triangle in hyperspace

On a side note, in conjunction with Evolution Strategies, one should carefully evaluate the usefulness of recombination schemes for a particular optimization problem. It is generally possible to use

[2]. . . which are usually represented by one of the system's native floating point types

Evolution Strategies without any "real" recombination schemes, using duplication only in the `recombine()` function.

When using cross-over in particular, one tendency observable in some two-dimensional toy problems is that, over the course of the optimization, parents will quickly move to neighboring regions of the parameter space. The recombination might then be unable to add significant new information (i.e. lead to the evaluation of new regions of the parameter space). However, one should investigate this for each optimization problem individually.

4.2.2. Specific Mutation Operators

The most common mutation operator in Evolution Strategies adds gaussian distributed random numbers to the floating point parameters of the feature vector $\vec{X}$.

Gauss Mutation

There is a number of methods available to create random numbers with a gaussian distribution. By default, the Geneva library uses the polar form of the Box-Müller transformation[141]. Figure 4.2 shows a histogram of random numbers created with this method, with a mean of 0 and $\sigma = 0.5$.

Random numbers with a gaussian distribution with mean value 0 will assume small values with a higher likelihood than larger values. As a result, when being added to the parameters of the feature vector, small changes of $\vec{X}$ will be more likely than large variations.

Figure 4.4 graphically demonstrates the effect of only using gauss mutation in an evolution strategy, using a single parent and 50 children. The example uses the Rastrigin function, whose minimum the Evolution Strategy should find. The Rastrigin function is defined by equation 4.1.

$$f\left(\vec{X}\right) = 10\,n + \sum_{i=1}^{n} \left[x_i^2 - 10\,cos\left(2\pi\,x_i\right)\right] \tag{4.1}$$

It is often used to test the performance of real-value based optimization algorithms, due to the very large number of local optima.

Children are scattered in a spherical cloud around the parent. The density of children decreases with increasing distance from the parent. This is the effect of using random numbers with a gaussian distribution for the modification of the feature vector.

As specified by the evolution cycle in listing 4.1, in the next step the best individual of the population – a child – is selected as the new parent, and the cycle starts afresh. It is clearly visible that the algorithm quickly approaches the global optimum at $(0,0)$.

The ability to cope with "noisy" quality surfaces is one of the major benefits of Evolution Algorithms. The reason for this ability becomes apparent when again taking a closer look at the Gauss Mutation in figure 4.4: While the close proximity of the parent is explored with many candidate solutions, there is always a small likelihood for children to be created in locations far away from the parent. Hence, even for larger local optima, the algorithm will eventually "jump" out of a valley.

Of course, recombination schemes can help here as well. E.g., in the case of cross-over, if the parents are not located too close to each other, the resulting individual might not be located in the same local optimum as its parents.

And last, but certainly not least, the (μ,ν) selection scheme (compare section 4.1.4) provides a certain protection against getting stuck in local optima, albeit at the price of having to accept a temporary decrease in the quality of parents. Note that the $(\mu+\nu)$ selection scheme doesn't give you this level of protection.

Evolution Strategies also share some features with Gradient Descents. When using Gauss Mutation, and given a large enough number of children, the direction of the next step will be roughly the direction of steepest descent (albeit not measured over longer distances than in the case of Gradient Descents).

There is also an inherent problem to using Gauss Mutation, which is not immediately visible. The gaussian will always have a mean of 0. However, suitable σ values (i.e. the "width" of the bell curve) are problem dependent and, what is more, the ideal value will change during the course of the optimization. The value of σ also has a huge influence on how quickly the Evolution Strategy can find the global optimum, or whether it can find it at all[3].

The reason is simple: The geometry of the quality surface, whose minima or maxima the optimization algorithm needs to find, will differ between different regions. Smooth areas will certainly require a different step width than "rugged terrain". Likewise, steep and flat areas of the quality surface will require different σ values. Comparing the paraboloid in figure 11.3 with the Rastrigin function in figure 4.3 should make this clear. Choosing a suboptimal value for σ will slow down the progress of the Evolution Strategy, or can even halt it completely.

As the geometry of the quality surface is not generally accessible in its entirety (or else one would already know the location of the global optimum), the only solution thus seems to be to adapt σ as part of the algorithm, or live with a possibly unsuitable, constant σ.

There are again a number of possible adaption strategies, of which some "traditional" strategies are listed below:

1. The $1/5$ rule: The algorithm checks the number of children of a population, whose mutation resulted in an improved result. If more than 20% of the individuals show an improved quality, then σ is increased, otherwise decreased.
2. A random, yet directed adaption of sigma: Here an adaption rate ω determines, to what extent σ should be adapted. The adaption follows the scheme $\sigma_{new} = \sigma_{old}\, e^{G(\omega)}$, where $G(\omega)$ represents gaussian-distributed random numbers with a standard deviation of ω.
3. Another method would be to decrease σ by a constant factor for half of the population in each generation, and to increase it for the other half (with random selection of individuals for increase and decrease of sigma, so that sigmas do not always get increased or decreased.)

The process of adapting σ as part of the Evolution Strategy is also sometimes called *self-adaption*.

[3] σ can thus also be compared to the step width of the gradient descent (compare e.g. equations 3.4 and 3.5). Indeed, in this document, we will often refer to σ as "step width".

Suitable start values for σ again depend on your optimization problem, particularly the allowed value range of the floating point parameters and the “ruggedness” and general slope of the quality surface.

The situation gets even more complicated when there is more than one quality surface, in the case of multi-criterion optimization. σ will quite likely be sub-optimal for some of the criteria.

Other Mutation Operators

Gauss Mutation forms the core mutation operator in Evolution Strategies. Given the versatility and wide value range of floating point variables, however, many other mutation schemes are possible.

As just one example, one might argue that adding gaussian distributed random numbers to a variable will put too much emphasis on the close proximity of an already known good solution, as the maximum of the gaussian is at 0. This might be a good strategy if you need to find the exact location of an optimum, but not, if you suspect that you are still far away from the global optimum.

Figure 4.5 demonstrates a random number distribution that could be used instead of a simple gaussian. The algorithm, as implemented in the Geneva library, can also mutate the distance between the two peaks, as well as the width of the gaussians, alongside the feature vector. It is even possible to allow for different widths and adaption rates of both gaussians. This is a more versatile, but also far more complex alternative to the “simple” Gauss Mutation.

4.3. Genetic Algorithms

Just like Evolution Strategies, *Genetic Algorithms* are based on the algorithm shown in listing 4.1. Instead of real numbers, however, their core parameter type is boolean. Consequently, of the recombination schemes discussed in section 4.2.1, only cross-over can be applied to Genetic algorithms. Likewise, “Gauss Mutation” is not applicable to Genetic Algorithms. The only possible mutation operation for boolean values is to flip a `true` value to `false` and vice versa. What can be varied, however, is the probability of flips.

Indeed, just like the ω parameter used to adapt the σ values in Gauss Mutation (compare section 4.2.2), it can be treated as part of the feature vector. It is then adapted as part of the usual optimization cycle. Other means of adapting the mutation probability p exist. Similar to *Simulated Annealing* (cmp. chapter 5), p could also be a function of the current generation g and decrease with increasing g.

One could argue that a vector of boolean values, with cross-over as the dominant recombination scheme, comes much closer to the natural example than a feature vector consisting of real values. However, for technical problems involving integral or real values, some severe difficulties are involved.

At first sight, there is *no* big difficulty encoding integers and real values through boolean values. In a computer, integers and floating point values are coded as binary arrays already, which are equivalent to arrays of boolean values. However, when bit-flipping is the only accessible mutation, the effect of this procedure can be almost negligible or extremely big, depending on which bits are flipped. Hence, in order to avoid completely random mutations, jumping from one end of the allowed value range to the other, mutations need to be aware of which bits are flipped. Note that another way of encoding numerals is the *Gray Code*[136]. However, it doesn't change the general problem that mutation operators need to be aware of the entity being mutated. Taking all this into consideration it seems easier and more natural to use floating point variables to encode real values in optimization algorithms. This becomes particularly important if you want to allow interaction of different optimization algorithms, such as using the result of an optimization using algorithm $\mathbb{A}$ as input for algorithm $\mathbb{B}$.

4.4. Hybrid Feature Vectors

Real-life optimization problems can rarely involve just one parameter type alone. As an example, feed-forward neural networks can be described in terms of the number of layers, the number of nodes in each layer and the weights between nodes of two layers. Hence a full description of a neural network requires both integer and floating point variables. Optimizing both the network's architecture and weights seems to be a worthwhile goal[8] (albeit not as easy to accomplish, as one might think).

It thus makes sense to generalize the concept of Evolutionary Algorithms and to allow not just one parameter type, but many of them simultaneously[4]. *Boolean*, *integral* and *real* values will usually be sufficient to describe any problem. E.g., integers could be used to describe characters or could serve as a key for any general sort of object. Yes/no decisions can be described in terms of boolean values. And the majority of parameters in a technical context can well be described with floating point variables (which will usually only be allowed to assume a limited value range).

In pseudo-mathematical terms, an evaluation criterion mixing parameter types could be described as a transformation

$$f : \mathbb{P}^n \rightarrow \mathbb{R}^1 \tag{4.2}$$

Here $\mathbb{P}^n$ means *n parameters of arbitrary type*[5]. The goings-on in an optimization involving only real-valued feature vectors are naturally the easiest to understand, particularly for problems involving but two parameters (compare figure 4.4). Mixing different parameter types, particularly in high-dimensional feature vectors, seems to be a little nondescript in comparison.

Using the picture of a quality surface again[6], for hybrid feature vectors to make sense, **it seems necessary to require that small changes of $\vec{X}$ lead to small changes of the figure of merit (or fitness) Q**. In other words, we need to be able to determine, whether two feature vectors $\vec{X}_1$ and $\vec{X}_2$ are "close". Where a single bit-flip changes Q by several orders of magnitude, it seems unlikely that an Evolutionary Algorithm will be able to find good optima, particularly if this situation prevails for many parameters in $\vec{X}$.

In such situations it can make sense to separate the optimization for different parameter types. I.e., one could first optimize all floating point parameters, then all boolean values, and then start again with

[4]It was one of the core design decisions of the Geneva library to make it possible to mix different parameter types.
[5]On a side note, what you get in the case of multi-criterion optimization (compare section 2.5) is a mapping $f : \mathbb{P}^n \rightarrow \mathbb{R}^m$
[6]. . . which is not entirely valid in this case, as we allow integral and boolean values for the parameters

the floating point values. However, this is certainly not guaranteed to succeed.

Nonetheless the flexibility mixed feature vectors provide make it worthwhile to follow this path.

4.5. Multipopulations

One of the unique features of Evolutionary Algorithms is the ability to make entire populations subject to evolutionary optimization. In other words, entire Evolutionary Algorithm populations act as individuals.

Recombination can simply mean duplication, but can in addition also involve shifts. I.e., entire populations are offset by a constant amount (in the case of real value or integral parameters). Mutation in this context means an entire optimization cycle of a “sub-population”. Selection has the same meaning as in conventional Evolutionary Algorithms. The figure of merit for a population is usually the quality of the best individual.

The thought can be carried further – i.e., one could create populations of populations of populations (and so on), although more than 2–3 levels will hardly be useful (and highly computationally expensive).

Using multiple populations, it is possible to explore the parameter space from more than one starting point. In theory this could also be achieved with multiple parents. However, experience shows that different parents of the same population tend to quickly move to adjacent areas in the parameter space, unless there is a means of keeping them apart (which would severely restrict the selection of new parents).

4.6. Inquest

What remains to be done in this chapter is to summarize the major advantages and disadvantages of Evolutionary Algorithms. On the positive side, Evolutionary Algorithms offer great flexibility and perform very well in the presence of local optima. This algorithm type is also quite easy to implement and, as will be discussed in chapter 8, can be parallelized in such a way that even missing responses from networked clients do not matter. The biggest drawback seems to be the huge number of configuration options inherent to any Evolutionary Algorithm (compare section 4.1). Also, particularly Evolution Strategies perform worse than some other algorithms close to the global optimum. The reason seems to involve difficulties in adjusting the step width quickly enough, so that the algorithm tends to “jump” around the optimum. Hence the combination with other algorithms such as Gradient Descents “for the last mile” makes sense.

Chapter 5.

Simulated Annealing

Simulated Annealing is another member of the family of stochastic optimization algorithms. This chapter gives a short introduction and discusses the variant that has been integrated into the Geneva library collection.

Key points: (1) Simulated Annealing mimics the behaviour of molten metal when cooling down (2) As the most important ingredient, the user needs to specify a suitable cooling schedule (3) Finding a suitable cooling schedule can be difficult, as premature convergence needs to be avoided (4) It is possible to extend the standard Simulated Annealing algorithm so that it fits almost seamlessly into the Evolutionary Algorithm workflow (5) In this case, an SA specific selection scheme needs to be implemented (6) Simulated Annealing is also frequently used for combinatorial problems (7) Like Evolutionary Algorithms, Simulated Annealing can be used both for floating point parameters and for integral and boolean paramaters.

5.1. Nature as a Role Model

Just like the other algorithms introduced in this document, Simulated Annealing is inspired by processes occurring in nature. When molten metal cools down, crystals start to progressively develop in the liquid before, ultimately, a solid state is reached. The controlled process of cooling down the liquid (or annealing) determines, how strong the material will be. Cooling the metal down too quickly will lead to defects, as the atoms can reach a lower energetic state when slowing this procedure down.

While the temperature is high and the atoms moving around in the matter have a high kinetic energy, the liquid metal can move strough states, whose potential energy is higher than the one before. This means that the matter is less stable. As the temperature falls, such increases in the potential energy become less and less likely. In “Simulated Annealing”, candidate solutions can be compared to a single state of this liquid. And just like the atoms moving aimlessly and unpredictably through the molten matter, changes to the candidate solution can involve a highly random component.

Major components of the algorithm are:

- A fitness or quality measure Q (equivalent to the potential energy of a state)
- A probability P for a given state (or candidate solution) s to be replaced by a new state s'.

- A "temperature" T, which will usually be a function of the current iteration. With increasing iteration, the temperature will converge towards 0. A high energy leads to a high probability P for a change of state, a low temperature will decrease P.
- A means of choosing new candidate solutions, based on the current state (or current candidate solutions).

Simulated Annealing was first introduced in the mid-1980s.

5.2. The Algorithm in Pseudo-Code

Instead of speaking of states, we will from now on use the terms "feature vector" or "candidate solution", as we have done for the other algorithms before. As before, it will be referred to by $\vec{X}$, or *individual* in listings. Note that $\vec{X}$ may contain different parameter types, not only those based on floating point variables.

Simulated Annealing can be expressed by a simple workflow:

Listing 5.1: The basic workflow of simulated annealing in pseudo code

```
currentCopy = getStartIndividual(); // Retrieve a starting point

do {
    // Update the temperature
    T = getNewTemperature(iteration, T);

    // Create and modify a clone of the currentCopy individual
    workingCopy = adapt(currentCopy);
    // Calculate a probability for currentCopy to be replaced by currentCopy
    pPass = P(workingCopy, currentCopy, T);
    // Get a uniform random numbe in [0,1[ and replace currentCopy if required
    if((rand = get01RNR()) < pPass) currentCopy = workingCopy;

    iteration++;          // Increment the iteration counter
} while(!halt());
```

In each iteration, the energy is updated. Then a working copy of the current solution is created and small modifications are applied to it. Based on the current temperature and the fitness of the current- and the working-copy, a probability is calculated for the current solution to be replaced; the replacement is carried out only with this probability.

"Good" definitions of the "temperature" T depend on the underlying optimization problem. If i signifies the current iteration, then a possible definition could simply be

$$T_{i+1} = \alpha T_i \tag{5.1}$$

where $\alpha < 1$ is a constant. The result will be an exponential decrease of T that will be the steeper the smaller α is. Other possibilities for what is frequently called the *cooling schedule* might depend directly on the current iteration i; e.g. a linear decrease:

$$T_{i+1}(i) = T_i - \beta\, i \tag{5.2}$$

where β is a small constant, or

$$T_{i+1}(i) = \frac{\gamma}{log(i+\delta)} \tag{5.3}$$

Again, γ and δ are constants.

In accordance with the original proposal for the simulated annealing algorithm, the probability for a feature vector $\vec{X}$ to be replaced by a new vector $\vec{X}^*$ can be defined as

$$P\left(\vec{X},\vec{X}^*,T\right) = \begin{cases} 1 & \text{if } Q\left(\vec{X}^*\right) > Q\left(\vec{X}\right) \\ e^{-\frac{Q\left(\vec{X}^*\right)-Q\left(\vec{X}\right)}{T}} & \text{otherwise} \end{cases} \tag{5.4}$$

$Q\left(\vec{X}\right)$ represents the evaluation function of the optimization problem. The term $Q\left(\vec{X}^*\right) > Q\left(\vec{X}\right)$ means $Q\left(\vec{X}^*\right)$ *is better than* $Q\left(\vec{X}\right)$. It does *not* make the statement, that Q should be maximized.

5.3. Means of Integration with Evolutionary Algorithms

Listing 5.1 already looks surprisingly similar to an evolutionary algorithm (compare listing 4.1). In contrast to it, though, only one "parent" and one "child" are used in each iteration. Among other consequences, this makes it very difficult to parallelize the algorithm. This document wants to propose an extension to the basic workflow, so that it fits more easily into the Evolutionary Algorithm workflow. Indeed, it then becomes possible to express it as an Evolutionary Algorithm with a special selection scheme. With these modifications, Simulated Annealing can also be parallelized seamlessly, on the basis of the evaluation of candidate solutions. Listing 5.2 shows the proposed workflow.

The algorithm starts with the creation of a set of feature vectors, sorted according to their fitness. We will call them *parents*, in order to achieve a consistent naming scheme. Note that, in contrast to listing 5.1, there may now be more than one `currentCopy` or starting point.

The optimization cycle commences with the calculation of a new temperature, based on the current iteration and the last known temperature. Some possible cooling schemes have already been described in section 5.2.

In the `recombine()` step, a number of "*child*" individuals are created, either – in the simplest case – by copying parent individuals, or by recombining two or more of them. This can be done in a similar way as described in section 4.1.2. The population layout will be the same as in figure 4.1.

The `mutate()` step is the equivalent of the `mutate()` call in evolutionary algorithms (compare section 4.1.3). Small modifications are applied to children here[1].

[1]. . . or indeed any kind of modification defined by the user or the provider of an optimization library

If the `recombine()` step only involves the creation of exact copies of parent individuals (as would be common in "traditional" Simulated Annealing implementations), then this is the step where children gain a unique identity.

Listing 5.2: The basic workflow of simulated annealing, as it will be used in the Geneva library

```
initPopulation();                // Initialize the starting point(s)

do {
    T = getNewTemperature(iteration, T);        // Update the temperature

    recombine();        // Create new candidate solutions
    mutate();    // Make small random changes
    saSelect(T, iteration);      // Selection scheme specific to simulated annealing

    iteration++;        // Increment the iteration counter
} while(!halt());

returnBestIndividual(); // Return the best individual found
```

The selection scheme `saSelect(T, iteration)` requires a more thorough explanation, as it is this part of listing 5.2 that is most specific to Simulated Annealing. Its main purpose is to take a decision, whether a given parent individual needs to be replaced by a particular child. Listing 5.3 shows a possible implementation in pseudo code:

Listing 5.3: A possible selection scheme for Geneva's simulated annealing implementation

```
void saSelect(T, iteration) {
  sortChildren(); // Evaluate and sort the children

  for(np=0; np<nParents; np++) {
    // Calculate a probability that the child replaces "its" parent
    pPass = P(population[np], population[nParents+np], T);
    // Get a uniform random number in [0,1[ and replace parent, if required
    if((rand = get01RNR()) < prob) population[np] = population[nParents+np];
  }

  sortParents(); // Sort new parents according to their fitness
}
```

In a nutshell, instead of (potentially) replacing just one current working copy with a new candidate solution, the first $nParents$ children are used to (potentially) replace their corresponding parents. In the listing, `population[nParents+np]` refers to the child corresponding to the parent in position `np`. Note that after sorting, when arranged in an array together with the parents, the `nParents` best children are located directly after the parents.

The fact that we are now dealing with an entire population of candidate solutions opens up new possibilities:

- Parallelization can be done in the same way as for "standard" Evolutionary Algorithms. Indeed we can use the same code.

- If the evaluation of children is done in parallel (and the evaluation function lasts sufficiently long), not more time will be needed for each iterations than if only one working copy would have to be evaluated
- There is a higher chance that a better solution is found within one iteration, as many new individuals are created and evaluated in parallel
- It becomes possible to mix different optimization procedures. I.e. one might want to use an SA selection scheme with “standard” Evolutionary Strategy Gauss mutation.

Note that, when only one parent and one child exist, the entire procedure will be identical to the original algorithm, as shown in listing 5.1.

In summary we believe that, through our proposal, Simulated Annealing can become even more powerful.

5.4. Inquest

Just like in the case of “standard” Evolutionary Algorithms, Simulated Annealing seems to suffer from the many problem-specific parameters and settings (e.g. the selection of a suitable cooling schedule) the user needs to be aware of.

The algorithm is also frequently used for combinatorial problems, such as the proverbial “travelling salesman” problem. Here, the `mutate()` call will rearrange the routes the salesman needs to take, so that the entire length of his trip gets minimized.

Performing such optimizations will be more difficult with standard Evolutionary Algorithms.

Chapter 6.

Swarm Intelligence

This chapter introduces the topic of *Swarm Intelligence*[10], with a particular focus on *Particle Swarm Optimization* ("PSO"). Just like Evolution Strategies, PSO acts on populations (or swarms) of candidate solutions (or individuals), and is used mostly for problem domains, whose parameters can be expressed by floating point numbers. And just like ES, PSO tries to mimic a natural example. The means of updating the individuals' positions, however, are entirely different, and based on the interaction of the entire population, rather than isolated action.

Key points: (1) Swarm Intelligence is based on "social" behaviour rather than "merely" local information (2) Major representatives are *Particle Swarm Optimization* and *Ant Colony Optimization* (3) PSO is mostly used for problem domains, whose parameters can be expressed as floating point numbers (4) ACO is often applied to combinatorial prolems (5) Both algorithms follow the natural example of different swarm types, as found for insects and animals (6) Particle Swarm Optimization dates back to the mid nineties and is thus newer than Evolutionary Algorithms (7) PSO can be represented with just a few lines of code, yet the results are impressive (8) Nevertheless PSO is no "silver bullet" for all types of optimization problems.

6.1. Particle Swarm Optimization

Historically, *Particle Swarm Optimization* is far newer than Evolution Strategies. PSO was proposed in 1995 by J.Kennedy and C.Eberhart [41]. It is modelled after the behaviour of different swarm-building animal- and insect-species. PSOs do not use local gradient information, but rather depend on the interaction of candidate solutions. Hence their features – and indeed the reason why this algorithm type *does* perform optimization (and does so well) – is still subject to an ongoing discussion. Rather than mathematical descriptions, such research is often based on simulations and models that help to describe the properties of PSO in a given context.

6.1.1. The Core Algorithm

Listing 6.1 shows a possible implementation of a PSO algorithm. A population of individuals is divided into neighborhoods, to which a "neighborhood best" individual is assigned. It represents the best solution that was found for this neighborhood during the course of the optimization. A globally best individual stores the information about the best solution found for the entire swarm so far. Each individual is also assigned a personal best, reminiscent of a personal recollection of worthwhile locations in the parameter space.

Listing 6.1: The basic workflow of particle swarm optimization

```
initPopulation();                // Initialize the population

do {
        evaluateIndividuals();   // Find out about the fitness of all individuals

        updatePersonalBests();   // Initialize/update personal, local and global bests
        updateNeighborhoodBests();
        updateGlobalBest();

        r0 = getUniform01RNR(); // Calculate three random numbers uniform in [0,1[
        r1 = getUniform01RNR();
        r2 = getUniform01RNR();

        // Update the individuals' positions
        for(ind=0; ind<getNumberOfIndividuals(); ind++) {
          for(dim=0; dim<getNumberOfParmatersInIndividual(); dim++) {
            Delta[ind][dim][iteration] = w * Delta[ind][dim][iteration-1]
             + c0 * r0 * (x_personalbest[ind][dim] -x[ind][dim])
             + c1 * r1 * (x_nbhbest[neighborhood(ind)][dim] - x[ind][dim])
             + c2 * r2 * (x_globalbest[dim] - x[ind][dim]);
            x[ind][dim] +=  Delta[ind][dim][iteration];
          }
        }

        iteration++;
} while(!halt());

returnBestIndividual(); // Return the best individual found
```

In each iteration, the position of each particle is updated in such a way that it is drawn towards the global best, as well as the best individual of its neighborhood. The Geneva library also adds a "personal best" component for each member of the swarm.

In programming terms, a neighborhood is simply defined as a "slice" of the population. E.g., if the population is implemented as a `std::vector<individual>` of 100 candidate solutions, individuals $0-9$ could be the first neighborhood, $10-19$ would be the second and so on (for a total of 10 neighborhoods).

Random numbers r_0, r_1 and r_2, distributed evenly in the range $[0,1[$ and multiplied by constants c_0, c_1 and c_2 are multiplied by the difference-vector from each individual to the personal, local and global bests. This ensures that a wide area of the parameter space is searched. A "velocity vector" (which in this document will be referred to as `Delta`) is calculated from these values. It also includes the last iteration's `Delta`. This way, the recollection of past successes features in the optimization process.

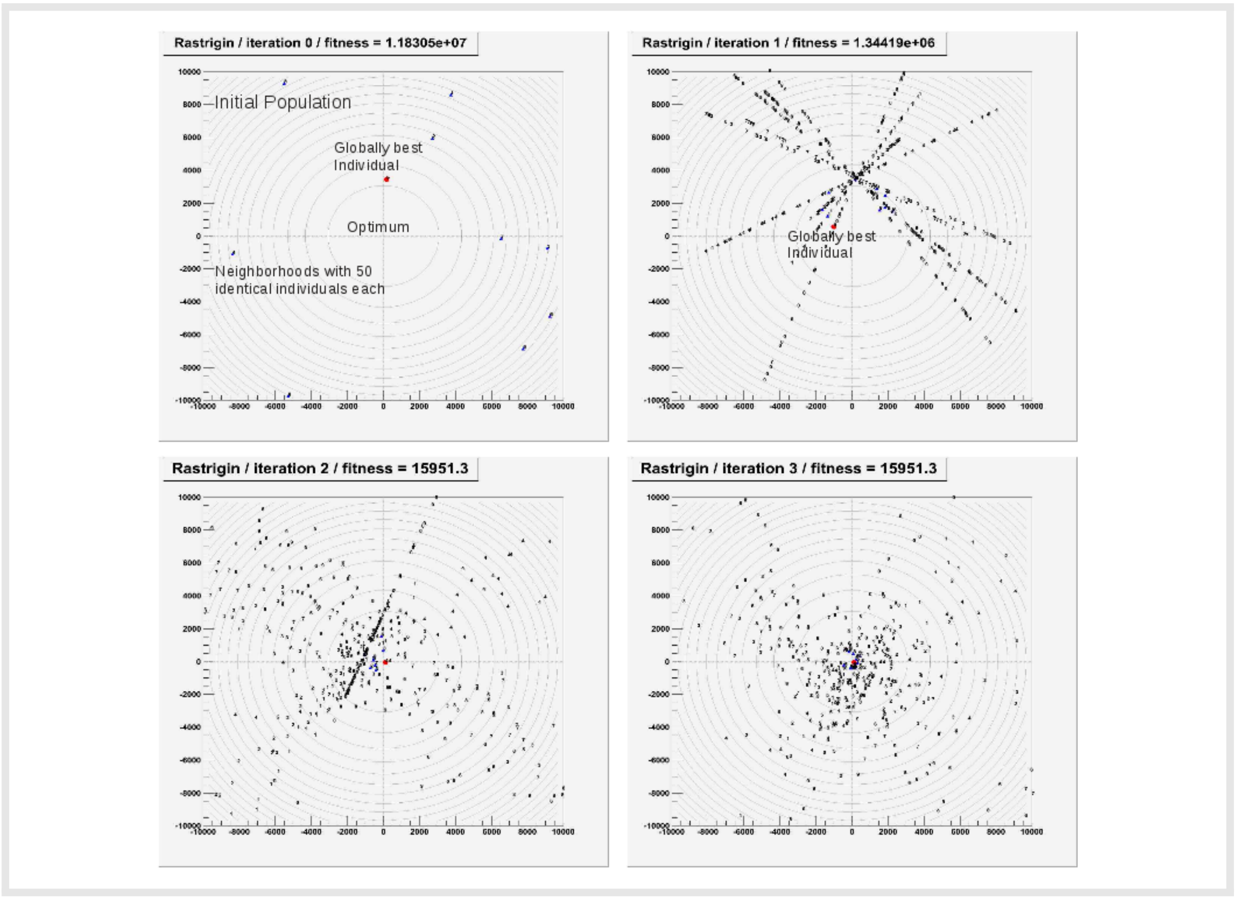


Figure 6.1.: *This picture demonstrates how a PSO algorithm searches for the minimum of the Rastrigin function in two dimensions. Note that, in comparison to figure 4.4, the algorithm needs to search a far bigger area for the optimum, as the allowed value range has been increased.*

Once calculated, the position of each individual is updated by adding the corresponding `Delta` to it. A possible variant of this algorithm reverts the sign of `Delta` in case a user-defined number of stalls has been exceeded. The algorithm then has a chance to excape from local optima.

Just like in the case of the other optimization algorithms discussed in this document, the optimization continues until a custom halt criterion is reached.

6.1.2. PSO in Action

Figure 6.1 visualizes, how a swarm modelled according to listing 6.1 searches for the minimum of the Rastrigin function (compare figure 4.3 and appendix A). Note that, in comparison to figure 4.4, the algorithm needs to search a far bigger area for the optimum, as the allowed value range has been increased.

The algorithm starts with 10 neighborhoods of 20 individuals each. Every individual belonging to the same neighborhood is initialized with the same values. A global best is selected, whose value is visible as a small red triangle, identical to the position of one of the neighborhoods. Local bests for each neighborhood will be identical to that neighborhoods individuals, hence position updates in the first iteration will only involve a component for the globally best individual.

This changes in the following iterations, as all candidate solutions have now assumed a unique position in the parameter space. Local bests (visible as small blue triangles) and the global best quickly start to gather around the global optimum at $(0,0)$. It is visible (and impressive) that the algorithm has only needed 4 iterations to reduce the overall evaluation from $1.2*10^7$ to 15951.

6.1.3. Variants

Many different variants of PSO exist. An article by *Christian Blum* and *Xiaodong Li* in Blum's and Merkle's book *Swarm Intelligence*[10, pp. 43-85] lists a number of possibilities. A variant, which replaces the particle update of lines 15 – 23 in listing 6.1 is shown in listing 6.2. Gaussian-distributed random numbers are calculated on the basis of the global and local bests, as shown in equation 6.1, and are added to each individual. Incidentally, this variant was also proposed by *Kennedy*.

$$\vec{\Delta}_i := \vec{G}\left(\frac{\vec{X}_g + \vec{X}^i_l}{2},\ ||\vec{X}_g - \vec{X}^i_l||\right) \qquad \vec{X}_i = \vec{X}_i + \vec{\Delta}_i \tag{6.1}$$

A vector $\vec{\Delta}_i$ of gaussian-distributed random numbers is calculated for each individual. The mean of this "gaussian cloud" is located at the average of the positions of the global best $\vec{X}_g$ and each individual's local best $\vec{X}^i_l$, the width of the gaussian is the distance from global to local best.

Listing 6.2: Many different variants of PSO exist. In the following code, gaussian-distributed random numbers are calculated based on the global and local bests, following equation6.1.

```
do {
        evaluateIndividuals();   // Find out about the fitness of all individuals

        updateLocalBests(); // Initialize/update local and global bests
        updateGlobalBest();

        // Update the individuals' positions
        for(ind=0; ind<getNumberOfIndividuals(); ind++) {
          x[ind] += Delta(getLocalBest(ind), getGlobalBest(ind));
        }
} while(!halt());
```

In the context of the Geneva library, this variant seems to be particularly appealing, as it combines some of the features of *Evolutionary Strategies* with particle swarm optimization. The algorithm also becomes far simpler than the original[1].

[1]Note that, at the time of writing, this PSO-variant hasn't been implemented in Geneva yet.

6.2. Ant Colony Optimization

Another major representative of Swarm Intelligence is the “Ant Colony Optimization” (ACO), which is particularly well suited for combinatorial optimization problems. As the name suggests, the algorithm is modelled after the foraging behaviour of ant colonies.

While searching for food, ants leave a pheromone trail. The pheromone level along their track depends on the type and quality of the food found by an individual, as well as the time since the ant has passed. Other ants will tend to follow an existing pheromone trail, and will thus be able to partake in another ant’s success. By following the same trail, they will also add to the already existing pheromone trail, so that other ants are more likely to follow the same route. Ants will also leave an existing trail with a certain likelihood, so that they may discover other, possibly better food sources. If this would not happen, all ants would ultimately travel to the same food source.

The algorithm will not be covered here in more detail, as it has not been implemented in Geneva (yet).

6.3. Inquest

Swarm Algorithms feature fewer steering parameters than Evolutionary Algorithms and are thus easier to handle. Tests show that they perform well even for very “noisy” evaluation functions. They seem to be more difficult to implement than EA, and there is less potential for parallelization. Effectively only the evaluation stage can be performed on another host. The position update involves interaction with other individuals and can thus not easily be parallelized. The algorithm is also more difficult to implement, particularly when the definition of individuals may involve different parameter types. Thus, in networked execution, the server might have to sustain a higher load, compared to Evolutionary Algorithms. Last, but not least, the inner workings are not as well understood as those of Evolutionary Algorithms. This can make it difficult to predict, whether this algorithm type is the most suitable for a given problem domain.

Chapter 7.

Parameter Scans

Parameter scans can help to gain valuable information about an optimization problem, whose "structure" in terms of the quality surface is otherwise mostly unknown.

Key points: (1) Parameter scans explore the parameter space either on a grid or through random sampling. (2) They may be used either to explore a small subset of the parameter space, where one might e.g. want to get a visual impression of the surface geometry in a 2- or 3-dimensional case. (3) They might also help to find a good starting point for another optimization algorithm (4) They may serve as an optimization algorithm in their own right in very simple cases.

Section 2.3.1 has discussed the question, why "brute force" is a bad tactics to find the optima particularly of high-dimensional optimization problems. "Brute force", in this context, referred to attempts to sample the entire parameter space. The major difficulty resulted from the high multiplicity of candidate solutions that needed to be evaluated even for very moderate numbers of sampling points per parameter.

There are some situations, however, where this approach may still be useful. Situations, where parameter scans can help include:

- Sampling of only part of the parameter set. In 2- and 3-dimensional cases this may help to understand the structure of the parameter space, e.g. whether it contains a large number of local optima, or whether it is rather "smooth"; whether it contains "steep valleys" or is predominantly "flat". This can affect the choice of configuration parameters for a given optimization algorithm, or the choice of the algorithm in general.
- When a solution was found using automated parametric optimization, it may be interesting to explore the immediate vicinity of the optimum
- When no good starting point is known, it may help to perform a short parameter scan throughout the allowed parameter space in order to find a good starting point.

Two approaches exist for the sampling of the parameter space:

- One may define a grid, i.e. a number of sampling points per parameter, so that for `n` sampling points of `m` floating point parameters n^m candidate solutions would need to be evaluated. This

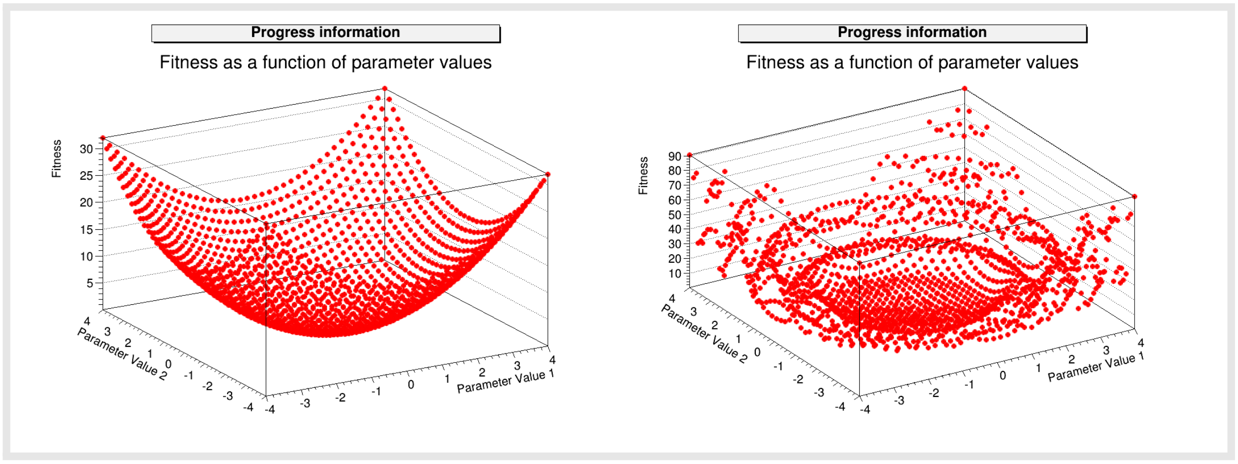


Figure 7.1.: *Parameter scans on a regular grid, for a simple parabola (left) and a parabola with overlaid local optima (right). The grid comprises 40 measurements in each direction, for a total of 1600 measurements. The plots were created with the help of pluggable optimization monitors (compare section 25.3).*

approach is suitable only when sampling a small number of parameters, e.g. when dealing with a sub-set of the parameter space, or if the sampling range and number of division points is very limited. The number of sampling points is fixed in this case. It is suitable for exploring an evaluation function in a 2- or 3-dimensional case, where the results may be visualized.

- It is possible to explore the parameter space with an evenly distributed, random set of sampling points. In this case one can directly limit the amount of evaluations. Large parameter spaces, however, will be sampled only very sparsely. This approach is suitable for finding a good starting point for another optimization algorithm.

Some minor difficulty may arise from the fact that we might be dealing with different parameter types. This may result in a situation, where different parameters will be explored with a different number of sampling points. E.g., a boolean parameter can only assume two states, whereas the number of states a floating point parameter may assume is unlimited for all practical purposes.

Figure 7.1 shows two parameter scans. The left plot shows a scan on a 40x40 grid of a simple parabole, amounting to 1600 measurements. The right plot shows a similar scan, this time for the "noisy parabola" (compare appendix A.2 and figure A.2), which contains many local optima. The plots were created with the help of pluggable optimization monitors (compare section 25.3).

Chapter 8.

Parallelization: General Considerations

This chapter discusses means of parallelizing different optimization algorithms, and the requirements and boundary conditions for running parallelized algorithms in various environments, such as Clusters, Grids, Clouds and with GPGPUs.

Key points: (1) Applications of the “nicely parallel type” are suited best for execution in parallel and distributed environments (2) Parallelization of optimization algorithms happens best on the level of evaluation functions (3) Optimization algorithms come close to the “nicely parallel” ideal, if in each iteration many candidate solutions need to be evaluated, and each evaluation takes a long time compared to the overhead of parallelization (4) Task-based parallelism can be exploited on the level of the optimization algorithm itself, while data-based parallelism might contribute to the overall speedup mostly on the level of the evaluation function (5) For distributed execution, both for performance and security reasons, local clusters are best suited (6) Where these are not sufficient, e.g. due to a lack of resources, it is recommended to use Cloud resources (7) Multi- or many-core machines might be a good alternative, if the computational demands of the evaluation function are low (8) GPGPU is an alternative for problems that can be expressed in OpenCL and “fit” into the graphics card’s memory (9) Individuals, whose evaluation function is expressed in OpenCL, cannot be ported easily to other parallel environments (10) A number of constraints exist particularly for distributed optimization (11) Among them, the restrictions expressed through Amdahl’s law seem to be the most severe.

8.1. Application Types

In parallel and distributed environments, several distinct application types can be identified, for which parallelization yields different results. Two of them are particularly suitable for distributed computing environments, due to their tolerance to high latencies:

- “Tightly coupled” applications exchange data frequently between sub-applications. An example would be fine-grained weather simulations where neighboring cells need to exchange data, e.g. to take into account information on an incoming storm.
- “Nicely parallel” tasks (often also called “loosely coupled”)[1] can be subdivided into mostly independent sub-tasks whose execution takes a long time. Data exchange is only needed in order

[1]These used to be referred to as “embarrassingly parallel”, but we live in more optimistic times now . . .

to send applications to remote sites and in order to retrieve the results of a calculation back. Many Grid applications are of the nicely parallel type. Examples include simulations in particle physics or data mining in very large data sets.

- “Streaming” applications produce data on one end of a network and send a continuous stream of data to a remote site. Little or no data exchange is needed in the opposite direction. Hence it does not matter, how long the data has travelled through the network.

8.2. Data- and Task-based Parallelism

Parallelization of computing-intensive applications usually happens in one of two ways:

- There might be means of identifying tasks in an application that do not depend on each other and can thus be executed in parallel. This is called task-based parallelism
- There are also often situations where large data sets need to be processed. It can then make sense to let identical applications work on subsets of the overall data and to consolidate the results later.

8.3. Parallelizing Optimization Algorithms

Optimization algorithms act in iterations, and each iteration needs the result from the preceding cycle. **Hence it should be clear that any means of parallelization is restricted to a single iteration.**

Figure 4.4 has demonstrated the search for the global minimum of the Rastrigin function. It should be obvious from this picture that the largest potential for parallelization lies in the evaluation of each iteration’s candidate solutions.

There is usually no dependency between different individuals. Hence the evaluation of candidate solutions can be individually delegated to another thread of execution or even “shipped off” to another machine altogether. Note that the latter will usually involve the (de-)serialization of data structures on the server and the worker node, which introduces a non-negligible overhead and can be computationally demanding. As a side note, serialization can be done in a portable way in C++ using the Boost.Serialization library [61].

If the number of candidate solutions is large enough and the evaluation takes long enough, the parallelization of the optimization algorithms introduced in this document can be regarded as being of the “nicely parallel” type. Small numbers of candidate solutions with short compute times tend more to the tightly coupled type.

Parellization of the optimization algorithms introduced in this document can be done in the same manner, albeit to different extents:

- *Gradient Descents* (see chapter 3) have a fixed number of parameter sets to be evaluated in each iteration. Unlike all other algorithms discussed in this document, evaluations for all parameter sets need to be available in order to move on to the next iteration. As a consequence, in an error-prone environment such as a Wide Area Network, when a response from a worker node is missing, it is necessary to resubmit the parameter set whose evaluation isn't available yet. This will at least double the execution time for that iteration. Also, for a small number of parameters p, the maximum theoretic speedup is low, as the number of evaluations equals $p+1$. Compare also the discussion of Amdahl's law in section 8.5.2.

- *Evolutionary Algorithms* (see chapter 4), *Particle Swarm Optimization* (chapter 6) and *Simulated Annealing* (chapter 5) can be easily fixed, if a response is missing, by just copying one of the other solutions. It is even possible to take into account "late returns" (i.e. evaluations returning after a timeout for a particular iteration), so that no compute time is wasted. Another advantage of these algorithm types with respect to parallelization is that there is no requirement for a fixed number of individuals. It is thus possible to adapt the number of individuals to the available resources.
- *Parameter Scans* (see chapter 7) may depend on the return of all individuals, if the grid-scan type has been chosen. Missing returns do not matter for random scans.

Parallelization of other parts of the optimization than "just" evaluation may be possible, but will be highly dependent on the specific algorithm. A trivial example is the mutation of individuals in Evolutionary Algorithms. It can be done with negligible overhead in parallel. The recombination step, in comparison, cannot be performed (easily) simultaneously, if the parallelization uses distributed resources[2].

The optimization algorithms themselves will usually benefit most from task-based parallelism. However, data-based parallelism can often contribute to the overall speedup on the level of the evaluation function. I.e., if the evaluation step requires the processing of large data sets, it can make sense to explore means of parallelising the evaluation step itself, through concurrent processing of small portions of the overall data set.

8.4. Characteristics of Parallel and Distributed Environments

This section discusses some of the characteristics of distributed and parallel environments that are relevant for parametric optimization with the Geneva library.

8.4.1. Compute Clusters

Clusters are collections of fast computers, usually in a single location, and connected through fast networks (>= Gigabit Ethernet) or specialized interconnects such as Infiniband$^{\text{TM}}$. They are managed by a batch submission system, such as LSF[102]. Its purpose is to assign incoming, new jobs to free resources, and enforce limits for their maximum runtime.

[2] It would of course be possible to ship the parent individuals that form the basis for e.g. a cross-over operation to another computer, which then also handles the evaluation of the newly formed individual. However, the overhead of the serialization of the parents will outweigh the benefit of performing the recombination in parallel.

Figure 8.1.: *The GridKa compute cluster is a Tier-1 centre in the academic EGI grid, located at Steinbuch Centre for Computing of Karlsruhe Institute of Technology. At the time of writing it comprised over 10000 CPU cores. (Picture used with permission from Steinbuch Centre for Computing)*

Today's dominant cluster type became mainstream in the early 1990s, then under the name of "*Beowulf Cluster*". At the time of writing, of the 500 fastest computer systems in the world (of which the majority is of the cluster type), over 90% were using a form of Linux[82].

Typical characteristics of Clusters include: (1) Traditional clusters will run a single operating system type, whose properties (installed libraries, version of the kernel) are not accessible to the user (2) The compute nodes' hardware will be heterogeneous, as the infrastructure has grown over time. They will almost always have the x86 (32 or 64 bit) architecture (3) Compute nodes may have local storage (such as internal hard drives), but will have access to large network storage devices, possibly connected through a Storage Area Network. (4) Network storage may be shared amongst compute nodes, so that the same programs are accessible to all nodes (5) There will be a single IP space (6) Communication between compute nodes will rarely be restricted by firewalls (7) Today, each worker node will comprise 4 or more cores, and have 4 or more Gigabytes of main memory (8) Users can choose how many nodes they want to allocate (9) Compute nodes will be shared between different jobs (although the system administrator may limit access to single jobs) (10) Unless the job itself crashes, the server will almost always get a response. Fault tolerance with respect to missing answers from clients is thus of minor importance in a local cluster (11) Network latency in a compute cluster will be low, compared

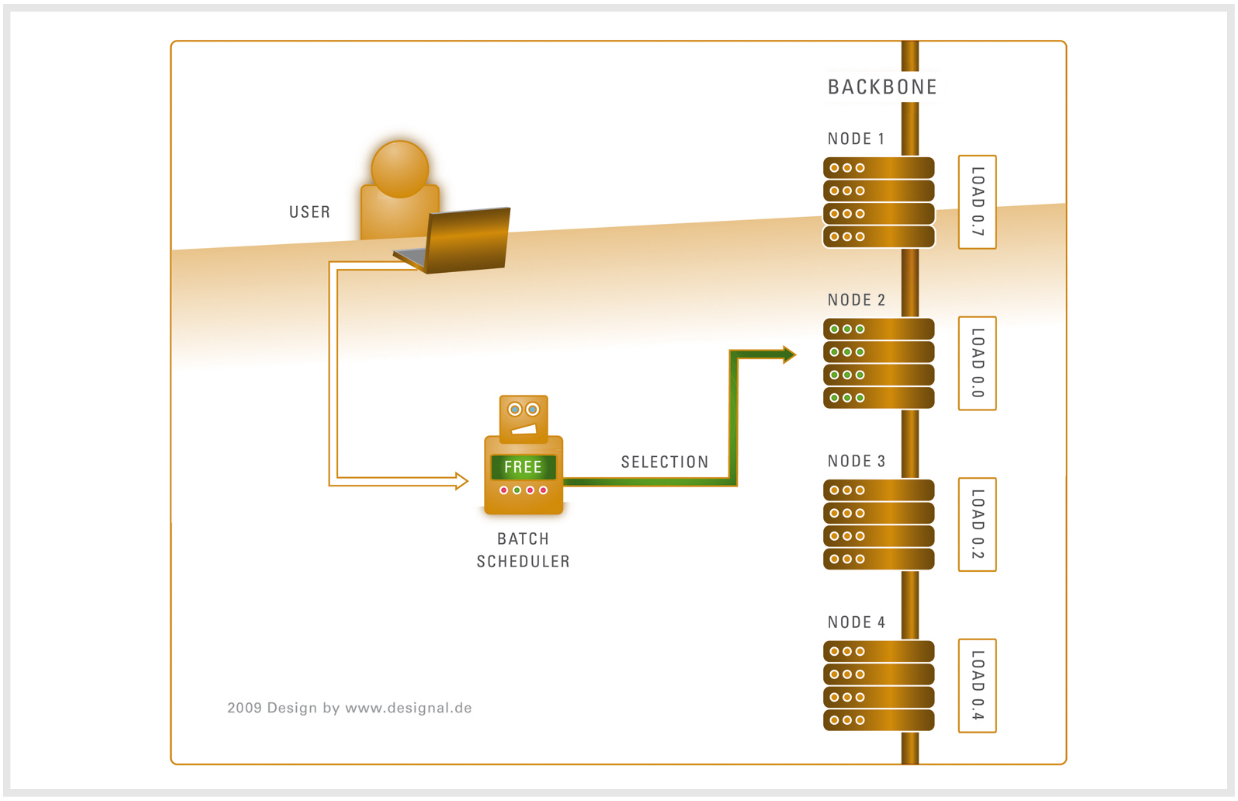


Figure 8.2.: *Batch submission systems assign incoming processing requests from users to the most suitable resource. Picture courtesy of Martina Hardt (designal • conceptional work by hardt – www.designal.de)*

to wide area installations. (12) If a fast interconnect, such as Infiniband, is installed, it is possible to use "distributed shared memory" via specialized libraries.

Communication between (sub-)applications running on a cluster can happen via the *Message Passing Interface* protocol ("MPI"), or can happen on a lower level, through network sockets. Special MPI implementations would be able to take advantage of Infiniband connects. Sometimes no communication takes place at all at run-time. Clients in a network just start execution, based on the information in some configuration file or relayed through their command line parameters. At the end of the execution, the result of the computation will be collected by the server, usually by accessing a corresponding output file.

Amongst all means of distributed computation, the performance of a parallel application running in networked mode will likely be highest in a local cluster. However, unless the cluster offers virtual compute nodes and allows users to provide their own disk images, a user will rarely be able to influence the run-time environment of the worker nodes.

Figure 8.1 shows the GridKa Compute Cluster, among other uses a Tier-1 centre in the Worldwide LHC Computing Grid, "WLCG".

Figure 8.3.: *Ian Foster gives a presentation at the Sun booth during Supercomputing 2001 in Denver/USA. It is worth noting the slogan "Sun Powers The Grid" (source: own pictures)*

8.4.2. Grids

The term "Grid" describes the vision of making "digital resources" available to users as easily as electricity. Just like the latter is produced in central power stations and exchanged via the electrical power grid, today's network infrastructure can give users access to resources from the computing world that were formerly unavailable to them. Grid Computing is a catch-all term for the technical and management infrastructure needed to facilitate this access.

The September 2008 start-up of the Large Hadron Collider (LHC) and the associated LHC Computing Grid[117] (LCG) at CERN near Geneva, Switzerland, made it obvious that today, Grid Computing has long moved past the stage of a mere *vision*. As of 2009, the four particle physics experiments of LHC – ATLAS, , LHCb and ALICE – have started producing in the range of 10 petabyte[3] of data per year[4].

Multiple copies of these data sets are stored in compute centres around the world, together with simulated data, again amounting to several tens of petabytes each year. Many thousand physicists

[3]1 petabyte $= 10^{15}$ bytes
[4]A failure in the cooling system of LHC prevented data-taking in 2008

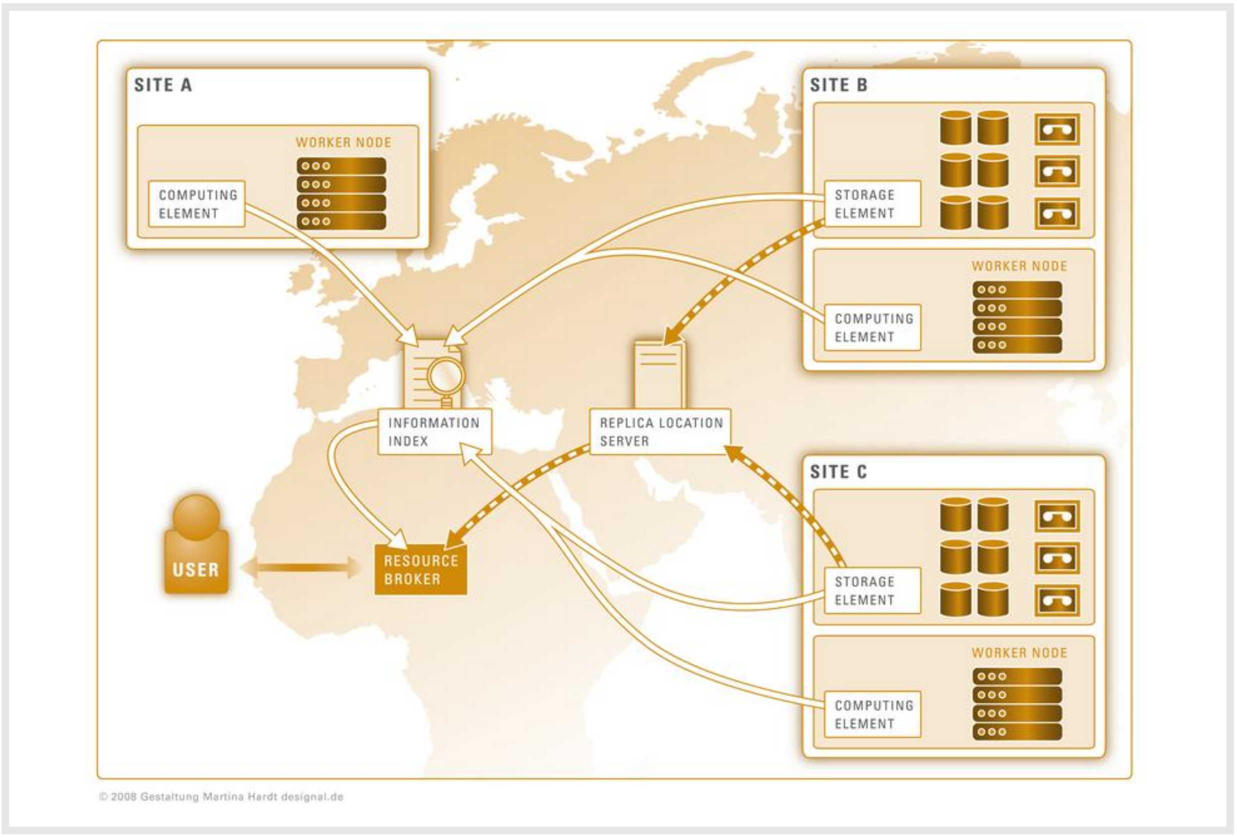


Figure 8.4.: *Schematic architecture of a global Grid infrastructure. Picture courtesy of Martina Hardt (designal • conceptional work by hardt – www.designal.de)*

world-wide aim at making new physics discoveries, based on the collected data. This search is conducted by repeatedly running analysis programs over large parts of the measured and simulated particle collisions.

A distributed computing infrastructure – namely a set of interconnected data centres providing users with a single point of contact – has turned out to be the most suitable set-up to handle this immense work load. However, when the Large Hadron Collider entered the planning stage in the mid 1990s, no software infrastructure, let alone an organizational framework, was available that could help orchestrate this massive demand for computing power.

Subsequently, numerous research projects were started that should help find answers to these questions. Among the most prominent European ones are the "European Data Grid" (EDG)[109] and its successors "Enabling Grids for E-SciencE" (EGEE)[106] and "European Grid Infrastructure" (EGI)[110]. All of them have been co-funded by the European Union. With over 300000 CPUs world-wide and running hundreds of thousands of programs each day, the EGI Grid (which is in large parts also responsible for processing the LHC data) can be considered to have reached production mode. Figure 8.4 shows a schematic view of the architecture of a Grid of the EGI type. In a sense, the EGI Grid can be described as a "cluster of clusters".

Typical characteristics of Grids include: (1) Users of Grids of the EGI type today almost entirely come from academia (2) These Grids are often monolithic in the sense that they run a monoculture of operating systems (The EGI Grid uses almost entirely Scientific Linux on its compute nodes) (3) No single IP space exists (4) Worker nodes in a local cluster will usually have a private IP, so that programs running on nodes in different locations will not be able to communicate directly (5) Even if they can, communication will be restricted by firewalls (6) Security is a moot point in Grid Computing, as academic data often does not need the same level of protection as data from the commercial regime (7) Turnaround times even for trivial "Hello World" jobs can be in the range of minutes, as jobs first need to pass the workload management system and will then enter the queue of a local batch submission system, from where they are finally assigned to a worker node (compare figure 8.4) (8) Network latencies are large, as Grids act in a wide-area context (9) Sub-applications will start running at different times, as they might be running in different geographical locations (10) Running distributed applications in a Grid environment requires a fault-tolerant software architecture, as (sub-)applications might not return results (in time) (11) An immense amount of compute power is accessible through Grids. However, access-rights are tightly regulated, as one needs to become a member of a virtual organization first. (12) The control over Grid infrastructures and the responsibility for their long-term availability is being taken over by National Grid Initiatives.

Due to their static nature and the long turn-around times of job submissions, Grids seem to be mostly suited for batch-type jobs, as they are common e.g. in particle physics. Where Grids are the only means of computation available, however, they might well be used for parametric optimization. If a local cluster is available, though, it should be preferred over distributed execution spread over different geographic locations.

While MPI implementations exist for Grid environments, they are usually maintained by research initiatives. Their future might then depend on further (government-)grants being available and their destiny – just like that of Grid Computing – is not clear. Communication through network sockets (rather than usage of Grid-specific means of communication) should thus be preferred in Grid-environments, as it makes optimization algorithms independent from the means of distributed execution and works well both for clusters, Grids and is also suitable for Cloud installations, as discussed below.

8.4.3. Clouds

Something has clearly changed in the IT landscape, particularly as far as business interest in Grids is concerned. At Supercomputing 2001 in Denver, USA, Sun (today Oracle) had presented itself with the slogan "Sun Powers The Grid" (see figure 8.3). It should be noted, though, that the company's definition of Grid Computing might be different from today's understanding of this term, as Sun's Grid Engine can indeed be likened to a batch submission system. It is hard to think of a stronger statement of interest in Grid Computing, though. And even Ian Foster, who had crafted the term "Grid"[5] in his book "The Grid: Blueprint for a New Computing Infrastructure"[23], presented his ideas at the Sun booth, thus paying respect to Sun's goals. Likewise, other heavy-weights of the IT sector voiced their support for the new development. Some companies also contributed heavily to the Grid.

[5]together with Carl Kesselman

Occasionally one could even hear Grids being touted as the successor of the World Wide Web. Today, however, it is the authors' distinct impression that the obvious expectation for the Grid to enter the main stream business world has *not* been fulfilled. This is visible in the lack of start-up companies in the field as much as in less-than-encouraging responses to some EU Grid project's business initiatives. Grid deployments, it appears, seem to be mostly limited to scientific applications.

In a 2008 article[28] for the former online-publication GRIDtoday, Prof. Wolfgang Gentzsch, formerly a Director at Sun Microsystems and later head of the German national Grid initiative D-Grid, even posed the question: "Grids are dead! Or are they?"[6].

In parallel to these developments, over the past few years, ample activity could be seen in a new field, called *Cloud Computing*, with contributions coming predominantly from larger enterprises. Just like in the case of Grid Computing, though, it is difficult to clearly define this term, as it grew out of the activities of few individuals and organizations. The ultimate, application-defined meaning can thus hardly be derived from today's situation. The attention (read: "hype") around Cloud Computing does not help to clarify the situation[7].

This initial hype is also a striking similarity between Clouds and Grids. Another joint characteristic is the attempt to leverage the ever increasing speed of today's network infrastructure. Wide-area connections today often achieve bandwidths of 10 Gigabit / second, with even faster connections on the horizon. If this bandwidth could be fully utilized[8], the content of (almost) two CDs could be transferred over a network every second. It then starts to make sense to outsource specific work items to a remote location. After all, it is not important to a user, where the execution of his program took place. **What *is* important, though, is the speed of execution, reproducible results and the prevention of unauthorized access to data and programs.**

So it appears as if Clouds and Grids both cater for similar needs. At the very least they share a common heritage, and it remains to be seen, to what extent synergies between both will lead to a cooperation rather than competition of ideas and developments.

In the context of this document, Cloud Computing is understood as a means of getting on-demand access to virtualized or physical compute nodes, possibly running in different geographical locations[9]. While the hardware (and possibly the virtualization environment) of such a compute node is defined by the provider, the Operating System and installed software is defined by the user, who makes a virtual machine image available to the provider.

Typical characteristics of Clouds include: (1) Contrary to Grid Computing, offers in Cloud Computing are dominated by commercial providers (2) Access to computing resources happens "on demand". (3) Payment for the services depends on the type of resources used and the amount of time they have been used (4) The actual run-time environment inside of a virtual machine in the cloud is defined by the user (5) Thus interaction with other users of the Cloud infrastructure is minimized, and the

[6]Note that, contrary to perception, Prof. Gentzsch's article does *not* predict the imminent death of Grid Computing. Instead it represents an (albeit strong) criticism of the current state of the art: "*I'm sorry to say it, but, so far, grids have not kept their promise.*"

[7]It is suggested to search for the terms "Grid Computing" and "Cloud Computing" on Google Trends `http://www.google.de/trends` to get an impression of the scale of the change.

[8]which is usually not possible

[9]Other uses of "The Cloud", such as online-storage, are not considered here.

user can define freely, which libraries and programs need to be installed on the resource (6) Like Grids, Clouds act in a wide-area context. Hence network latencies are high (7) As clients are started in virtual machines that are already running, there is no time overhead from workload management systems and local queues, like it is found in Grids (8) Missing responses from worker nodes running in a Cloud are arguably less likely than in Grids.

Clouds offer great flexibility, and due to their dynamic nature, most communication restrictions (such as firewalls) are under the control of the user. There is also no dependency on the local installation, as it is found in Grids and Clusters. Hence, for doing parametric optimization, they seem to be a more suitable choice than Grids. Compute nodes can be set up only for the time they are needed for the optimization, and most large Cloud providers will not restrict you in the amount of resources you need. Hence you can also tackle very complex and large optimization problems, requiring many individuals.

Just like in the case of Grids and Clusters, and as both server and clients are under the control of the user, the best means of communication in a wide area context seems to be network sockets. Another possibility would be web services.

As Clouds act in a Wide-Area context, **this means of computation should only be used if network latencies do not matter much**. In all other cases, execution on multi-/manycore-systems or Clusters should be preferred. However, as Clusters cannot be arbitrarily scaled on demand, Clouds might well be a good choice to satisfy a short-term demand for large amounts of computing power.

8.4.4. Multithreading

It is not uncommon to find 16 or even 32 CPU cores in today's worker nodes[10]. The number of cores seen by applications can be even higher, if Intels hyperthreading or related technoligies are enabled. Applications can take direct advantage of this abundance of computing power by starting separate threads of execution on each core. For optimization algorithms and their inherent parallelizability, this is an ideal situation. Compared to networked execution, there is practically no overhead involved in evaluating candidate solutions in parallel, nor is it necessary to serialize candidate solutions to send them to a remote location. Multithreaded environments make it possible to also parallelize some of the more obscure parts of optimization algorithms (such as performing cross-over for different children in parallel in Evolutionary Algorithms).

The biggest drawback of multithreading is the missing scalability of the underlying hardware architecture. In a cluster-environment, if another 100 evaluation units are needed in an optimization algorithm, one would simply start the required number of jobs. On a single worker node, however, the number of cores is constant.

Another possible disadvantage, particularly for evaluation functions needing a lot of external input, is the access to data stored on disks. Then either the network (in case of network storage being used) or access to the local disk becomes the bottleneck. Remember that, in a multithreaded environment (and assuming that evaluation functions run for equal amounts of time), all threads might try to access the external data at the same time. There are also some rules that need to be followed in a

[10]The largest system Geneva has been tested on today in multi threaded mode comprised 48 cores

multi-threaded environment. There are not many inherent locking issues in optimization algorithms. However, evaluation functions need to be re-entrant (i.e. it must be possible for multiple threads to run the same function in parallel). One general advice that can be given here is to avoid global variables in evaluation functions. Where these are used, one needs to synchronize access to the variables, which will slow down the execution.

On a side-note, a very portable and well documented means of creating multithreaded programs in C++ is the Boost.Thread library[145]. The thread API used there has also mostly become part of the new C++ standard, C++11 .

8.4.5. GPGPU

GPGPU stands for *General Purpose Computation on Graphics Processing Units*.

State-of-the-art *Graphics Processing Units* consist of possibly thousands of specialized processors. Jointly, they can deliver far beyond a *Teraflop*[11] of computing power. As an example, AMDs Radeon HD6990 can deliver over 4 Teraflops. This is over an order of magnitude faster than the CPU power available from the latest processor generations at the time of writing.

As graphics processing units are very specialized, though, access to this resource is not as easy as it is for general purpose CPUs. GPUs are operated according to the *Single Instruction, Multiple Data* model. Here, a single instruction set is executed for many different data sets. Programming graphics cards was thus for a long time a difficult operation, and a unified access model across different graphics cards was only available for graphics operations (OpenGL, DirectX).

Access models changed over time, as the usefulness of graphics hardware for computationally expensive general purpose calculations became apparent. As this opened up a new market for vendors of graphics hardware as well, they gradually started to improve the accessibility of their devices. Today, two general purpose programming models prevail: CUDA[15] and OpenCL[51]. CUDA seems to be dominated by NVidia, while OpenCL is an open standard of the Khronos Group. CUDA-based OpenCL implementations exist. OpenCL is not limited to graphics hardware, but is meant to be a general programming model for heterogeneous parallel devices. OpenCL is very similar to the C programming language. A program running on a general purpose CPU can start an OpenCL kernel on the graphics hardware by specifying the path to the OpenCL code. It will then be compiled and loaded into the graphics hardware. The next call to the OpenCL function can then just use the pre-compiled code. Parameters can be passed to the OpenCL function, and results returned to the caller. Effectively, the GPU then acts as a co-processor.

GPGPU would be an almost ideal environment for performing parametric optimization in parallel, if it were not for the inherent limitations of GPUs. The parallelization of the algorithms discussed in this document (compare chapters 3, 4, 5) and 6 can be done using the SIMD model. It is then the evaluation function (plus possibly other code, like the mutation of individuals in EA) that would be executed for all candidate solutions in parallel on the GPU. And due to the large number of processing units, a far larger number of individuals could be used, compared to typical cluster installations.

[11]A Teraflop is the equivalent of 10^{12} single precision floating point operations per second

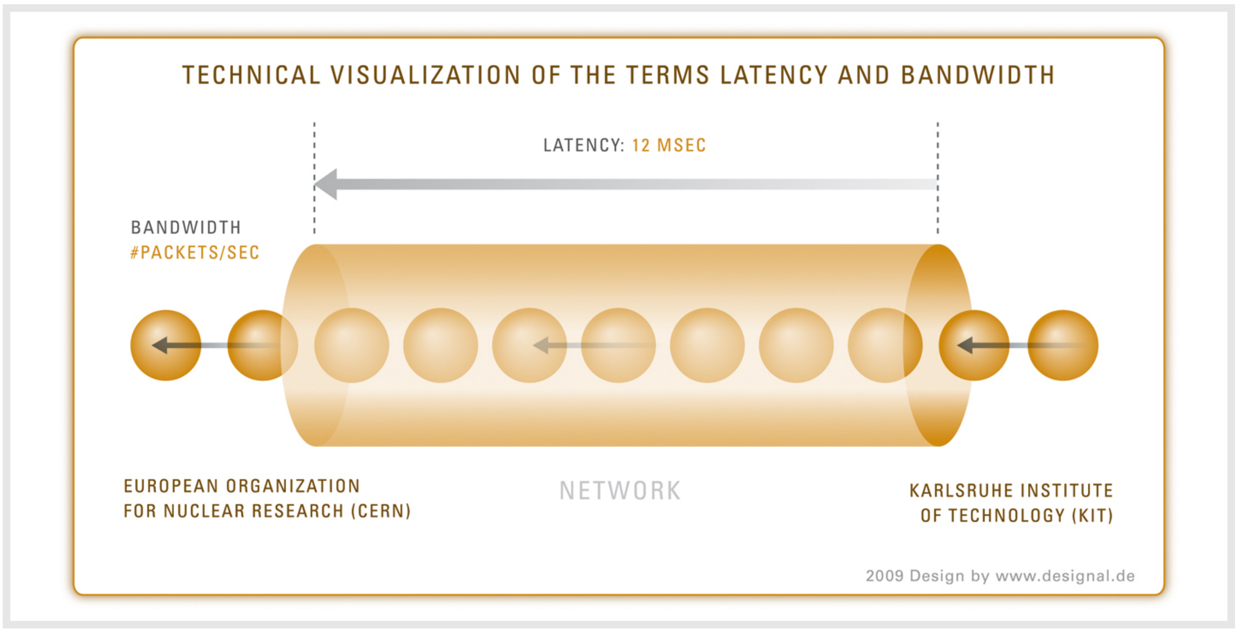

TECHNICAL VISUALIZATION OF THE TERMS LATENCY AND BANDWIDTH
LATENCY: 12 MSEC
BANDWIDTH
#PACKETS/SEC
EUROPEAN ORGANIZATION FOR NUCLEAR RESEARCH (CERN)
NETWORK
KARLSRUHE INSTITUTE OF TECHNOLOGY (KIT)
2009 Design by www.designal.de

Figure 8.5.: *The "speed" of a network consists of two components – latency and bandwidth. Picture courtesy of Martina Hardt (designal • conceptional work by hardt – www.designal.de)*

On the downside, the memory available to each of a GPUs processing units is very limited and, taken individually, they do not have a particularly high performance. Also, the fact that evaluation functions need to be implemented in special languages, such as OpenCL, makes it difficult to write general purpose code. Thus there are also strong limits to what can be done using GPUs in the context of parametric optimization.

Integration in Geneva may happen either through specialized individuals or through Geneva's broker. At the time of writing, an experimental (albeit not publicly available) GPGPU-consumer exists for Geneva.

8.5. Constraints

This section lists some of the boundary conditions and constraints that govern parallel execution of optimization algorithms.

8.5.1. Network Speed: Bandwidth versus Latency

In a distributed system, all communication has to pass through a computer network. It is immediately obvious that the speed of this network forms a limiting factor for any application wishing to access or

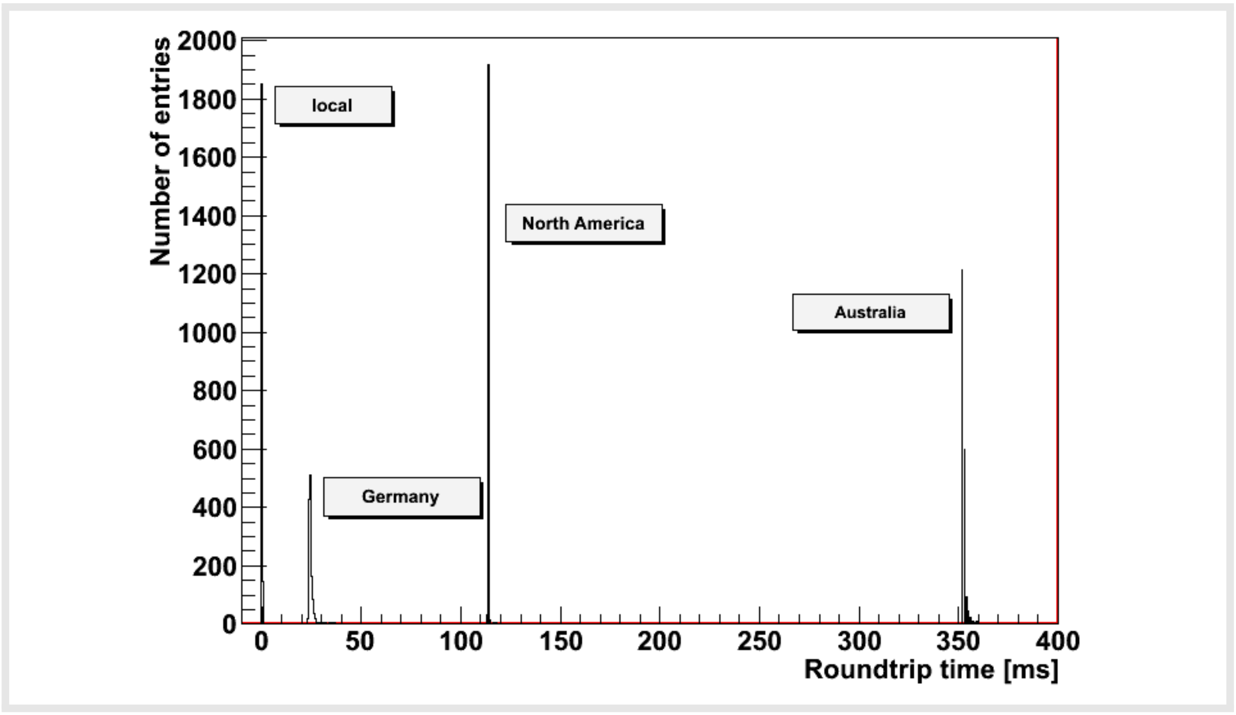


Figure 8.6.: *Response times from servers running in different geographic locations show large variations. Parallel execution in a wide area setting needs to take this into account.*

use a remote resource.

In the context of computer networking, however, the term “speed” needs further explanation. Figure 8.5 shows a schematic view of a network. Data is fed into it on one side and is received on the other end. Two very distinct quantities can be identified that govern how fast a network appears to a user. The *bandwidth* of the network describes the amount of data received each second. The *latency* refers to the amount of time a single data package needs to pass through the network.

The impact of the network’s speed will vary, depending on the application. One important quantity here is the amount of information that needs to be exchanged between remote sites. The frequency of the exchanges is another important factor. Distributed applications featuring frequent exchanges of small amounts of data will be limited by the network’s latency, while applications with rare exchanges of large amounts of data will be limited by the bandwidth.

But while the bandwidth could theoretically be scaled to virtually any desired level[12], latency does not scale well. First and foremost, data can not travel faster than the speed of light ($c \approx 3*10^8 m/s$). A data package en route from Hamburg / Germany to Adelaide in Australia and back again will travel at least 30806 kilometers[19], which at the speed of light would take roughly 0.1 seconds.

The authors have performed a simple measurement of the roundtrip-time of a network signal, using the UNIX `ping` command. `ping` sends a network package to a remote site and measures the time

[12]Given sufficient funding, one could just buy or even build additional network connections to increase the bandwidth.

until the signal returns. 3000 signals each were sent from Karlsruhe Institute of Technology (KIT) to different sites. Figure 8.6 shows a histogram with the measured roundtrip times from KIT to a local site, to a system at Steinbeis University (SHB) in Germany, to another computer in North America and to a fourth system in Australia.

While there is only a negligible amount of time needed to reach a local system, sending a signal to Australia and back again takes over 0.3 seconds. It should be obvious that a distributed application with sub-processes in Australia and Germany would be severely slowed down if it needed to exchange data frequently and wait for the results of a remote calculation. As a comparison, a modern processor running at a clock frequency of 3 GHz would be capable of performing almost a billion operations within 0.3 seconds in a single core (not taking into account any SIMD units in the CPU). **This should make it clear that it strongly depends on the application type whether distributed or local execution yields better results.**

The fact that a data transfer to Australia takes 0.3 seconds rather than the 0.1 seconds (the “natural” time needed for the transfer when only taking into account the speed of light) can be easily explained. Data signals pass different technical components on their way, such as repeaters and routers. Each of them adds additional delays to the data transfer, as the signal might be buffered until there is free capacity on a network, or the correct routing has been calculated. Given the large distance to Australia and the variety and amount of infrastructure a signal needs to pass on its way, it is indeed surprising that the roundtrip time measures “only” 0.3 seconds.

Putting Latency into Context

It has been said that latency forms a limiting factor for distributed applications. But the problem is not as bad as it sounds. Average access times of modern hard drives (not of the solid state disk type . . .) range from 2–15 milliseconds[132]. A data transfer between KIT and Steinbeis Business Academy takes in comparison around 25 milliseconds. This makes it feasible to create “virtual harddrives”[13] using geographically distributed storage devices. Indeed such efforts exist. One example would be the Andrew Filesystem[105]. And, more recently, many Cloud-based offers have appeared that allow a user to store data in a remote location and treat it like data on a local storage device.

8.5.2. Amdahl’s Law and its Consequences

Amdahl’s law [130] gives an estimate of the maximum achievable speedup of a parallel or distributed application. It is shown in equation 8.1.

$$S = \frac{1}{(1-P)+o(N)+\frac{P}{N}} \leq \frac{1}{1-P} \tag{8.1}$$

Here, S means *Speedup*, P is the percentage of the overall execution time consumed by execution

[13]A better, but less intuitive way of referring to this technical development would be a “distributed filesystem”

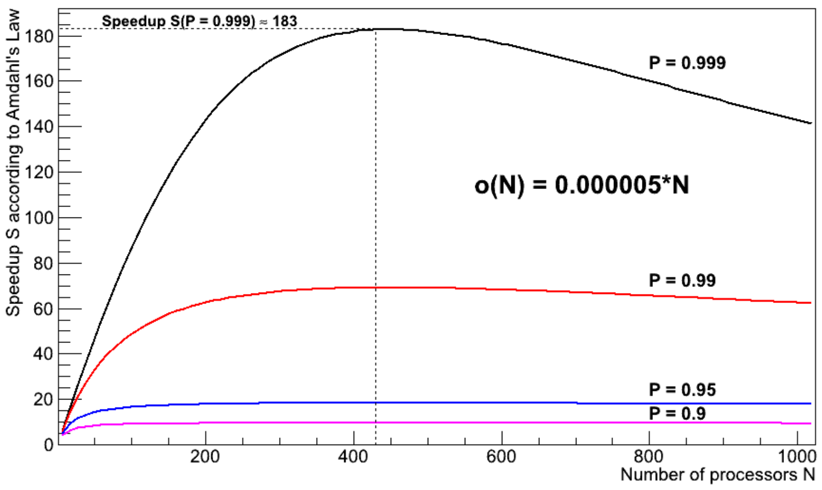


Figure 8.7.: *Plot of Amdahl's law:* $S = \frac{1}{(1-P)+o(N)+\frac{P}{N}}$ *as a function of* N*, for a fixed value of* $o(N)/N$ *and different values of* P

units running in parallel, N is the number of processing units contributing to the parallel execution, and $o(N)$ is the "cost" of communication and synchronization between processes[14].

In a nutshell, Amdahl's law states that the maximum speedup depends on the amount of time spent in parallel execution, compared to the overall runtime, and that an additional overhead from communication and synchronization has to be taken into account. Figure 8.7 shows S as a function of N, for a fixed value of $o(N)$ and different values of P.

While the law in itself may not be very surprising, the magnitude of the effects should very well come as a surprise: Even if the percentage of parallel execution is as hight as 99.9% and the overhead $o(N)$ has only a very weak dependence on N, the maximum speedup S achieved with up to 1024 processors reaches only a disappointing ≈ 183. What is even more surprising is that the maximum speedup is not achieved for 1024 processing units, but for $N \approx 430$. This is the effect of the communication and synchronization overhead[15].

Thus, as a general advice, don't be overambitious with the amount of resources you put into solving an optimization problem. There is maximum for the useful number of processing units[16].

[14]An example would be the time needed to serialize individuals

[15]Even without it, there would be a rather disappointing limit to the achievable speedup according to Amdahl's law.

[16]Geneva has, for networked execution, chosen to target optimization problems with particularly long running evaluation functions. This, in turn, has triggered all sorts of other decisions, such as to rate stability of the core library higher than efficiency. After all, the bulk of the runtime will be consumed by the evaluation function.

8.5.3. The Dynamic Nature of Distributed Infrastructures

Distributed computing infrastructures are dynamic in nature. This simple statement should not come as a surprise – where execution and submission of applications do not happen in the same location, users will usually have no direct control over the remote resources.

As a consequence, computing resources and networks may fail at unpredictable times, data may not be found in the location it is supposed to be and the people responsible for the maintenance of the systems may change. **In other words, nothing can be taken for granted in a distributed computing infrastructure.**

The only real remedy for such failures is fault tolerance on the application side[17]. As an example, if an "unprotected" application[18] uses 100 processors in different locations, and each sub-process has a 1% risk of failure while the application is running, then the entire application will fail with a probability higher than 60% ($\approx 1-0.99^{100}$). **The application thus needs to find ways to stay alive if sub-processes fail.**

8.5.4. Fault Tolerance and Timeouts

We have already mentioned that it is easy for Evolutionary Algorithms, Particle Swarm Optimization and Simulated Annealing to recover from missing responses, and that this is more difficult for Gradient Descents.

But responses may not simply be missing. In real-life environments, particularly when performing optimization in distributed environments, response times from worker units can vary. As an example, a client might have to share a system with another program that consumes all of the CPU time. The network might temporarily be clogged, or there can be large differences in the network latency, particularly in a wide-area context.

If the server does not maintain a permanent connection to the client (which is not useful, if it doesn't have to communicate with it for a long time), it cannot know what the status of the client is. The client might simply have crashed, or a response might come too late for the optimization procedure to be efficient.

In this situation a timeout needs to be set, after which optimization continues with the next iteration (or some parameter sets are resubmitted in the case of Gradient Descents).

An additional problem arises here, as optimization algorithms are per se generic, i.e. the algorithm does not have much information that tells it how long the evaluation function will run.

One possible solution is to record the time passed until the first response from a worker node arrives. The timeout can then be a multiple of this time frame, and might in addition take into account the overall number of workernodes.

Note that there is a danger here, though, as this solution implicitly assumes that the runtime for all evaluation functions is approximately the same. Where some functions run for seconds and

[17]. . . and patience on the side of their users

[18]in the sense of an application that fails if at least one of its sub-processes fails

some for hours, one might have to enact a manual timeout, based on the knowledge of the maximum runtime. If this is not done, the optimization algorithm will favour solutions with a short runtime of the evaluation function, which can lead to a big bias in the result of the optimization.

One might in this situation also require that all evaluations return a response, in which case one looses the benefits of fault tolerance. Or it would be possible to resubmit missing jobs after the timeout, which can however also cause problems, if the resubmitted parameter sets are always those with a long runtime.

In summary, if we want to maintain the generic nature of optimization algorithms, unfortunately there might be no silver bullet, when it comes to suitable timeouts. The best solution can be problem-dependent.

8.5.5. Quantization Effects

In an ideal environment[19], if n worker nodes are available for the processing of m tasks, you may observe what is commonly called a *quantization effect*. As an example, if $n == m - 1$, then there will still be one remaining task when all worker nodes have completed their initial assignments. A single worker node then needs to process this item, and all others wait until it is finished. This way, processing will take twice as long as if $n == m$. Likewise, adding more nodes, so that $n > m$ will not help you, as there are no tasks available for the additional worker nodes. Similarly, if $n < m$, there will be thresholds at which an increase of n will lead to added performance. Of course, quantization effects will be altered by non-ideal environments, where response times of worker nodes will vary. If your parallel optimization algorithm is fault tolerant and occasionally a response from a worker node is missing (or the server runs into a timeout), then suitable values for n can be different than in an ideal environment.

8.5.6. Security

The implications of security on performing parametric optimizations in distributed environments are virtually endless and cannot be discussed here in detail. E.g., Grids are mostly built for academic purposes and will consequently have lower security standards than, say, a bank would require.

But even in Cloud Computing, where users have total control over the content of their own virtual machines and can enact all sorts of security measures (e.g. encrypted disk images, stringent firewall settings, VPNs between worker node and server, ...), security will be (by far) lower than it is for local clusters and particularly single, isolated machines. Hence, before using geographically distributed worker nodes, one should think twice about the security concept and possible dangers. Ideally, one should have full control over who has physical or network access to the worker nodes and the server, and all communication between worker nodes and server should be encrypted[20]. VPNs between

[19]An environment where the processing of all work items takes equally long and it does not matter if the worker nodes start communicating with the server all at once. Also, it is assumed that there will always be answers from all worker nodes.

[20]...which adds a non-negligible overhead particularly for frequent, small data exchanges.

Clients and Server are an attractive option.

Chapter 9.

More Complex Demos and Use Cases

This chapter introduces a few more complicated demos, meant to illustrate some particular technical (and administrative) features of optimization problems. They can be used to benchmark the Geneva library in a more real-life context than is possible with the test functions of appendix A.

9.1. Mapping Semi-Transparent Triangles to a Target Picture

The following toy example illustrates a number of important aspects of optimization algorithms. The goal of this demo is to replicate a given target picture with semi-transparent triangles. They are allowed to overlap and can be freely positioned on a canvas with the same geometry as the target picture. Each triangle has 6 coordinates (three corners), 3 colors and an opaqueness, amounting to a total of 10 parameters per triangle.

Colors can be represented either as integers in the range $[0,256[$ or floating point values in the range $[0,1[$. Likewise the coordinates can either be floating point or integer values. In the latter case they would represent the x- or y-id of a pixel, in the case of floating point values they could again be varied in the range $[0,1[$. Pixel-ids can then be calculated by multiplying the coordinate with the number of pixels in x- or y-direction. The opaqueness will be a floating point value, variable in the range $[0,1[$. Hence this is a prime example for a problem involving different parameter types.

Note that it makes sense to put the triangles with the lowest opaqueness on top, so that triangles with high opaqueness do not obscure others with a low opaqueness. This will involve sorting the array of triangles of a candidate picture according to its opaqueness.

When all triangles have been positioned on the canvas, a color can be calculated for each pixel, and the deviation between this "candidate" and the target picture can be calculated. One simple possibility is to check for the difference between the colors of each pixel. The quality Q of a candidate solution can then be represented as shown in equation 9.1.

$$Q = \sum_{\text{Pixel } p} \left(\sqrt{\left(r_{\text{t}}^{p} - r_{\text{c}}^{p}\right)^2} + \sqrt{\left(g_{\text{t}}^{p} - g_{\text{c}}^{p}\right)^2} + \sqrt{\left(b_{\text{t}}^{p} - b_{\text{c}}^{p}\right)^2} \right) \tag{9.1}$$

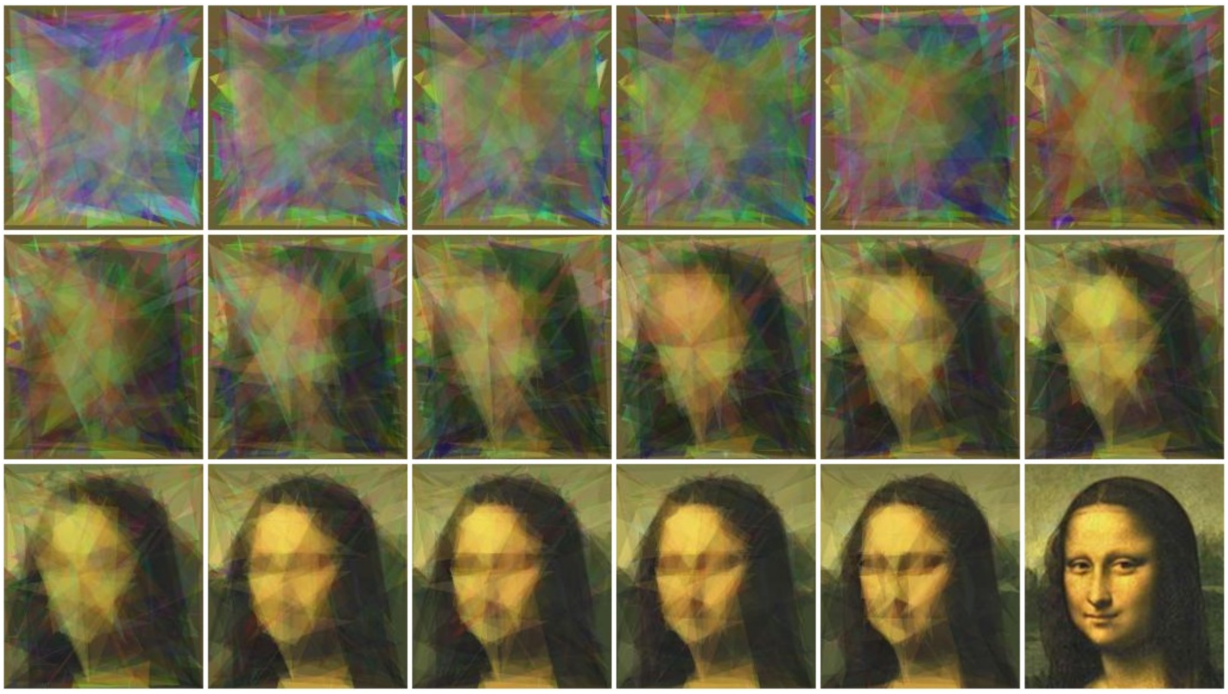


Figure 9.1.: *Semi-transparent triangles can be super-imposed in such a way by optimization algorithms that the amalgamation starts to look like a given target picture. In this example, a clipping from* da Vinci*'s Mona Lisa has been used. The algorithm starts with a random collection of 300 triangles (equivalent to 3000 parameters to be optimized). The target picture is shown in the lower right-hand corner of this figure.*

Here `t` and `c` refer to a pixel of the `target` and `candidate` picture respectively. `r`, `g` and `b` stand for the `red`, `green` and `blue` value of a pixel.

9.1.1. Mapping the Mona Lisa

The optimization algorithm then needs to minimize Q. Figure 9.1 shows the result of applying this procedure to a clipping of *da Vinci*'s Mona Lisa. 300 triangles were used, equivalent to 3000 free parameters. Note that one will rarely perform optimization runs with such a high number of variables, as the parameter space becomes very big and it becomes increasingly unlikely to find the global optimum.

In our example, an Evolution Strategy was used, and all parameters were treated as floating point values. Tests with Swarm Algorithms didn't yield satisfactory results, and Gradient Descents failed completely. This reinforces again the statement that one needs to carefully decide which optimization algorithm should be used for a given problem. Where no existing experience with this problem domain exists, it might be useful to try out several algorithms on a simplified version of the problem, before taking a decision.

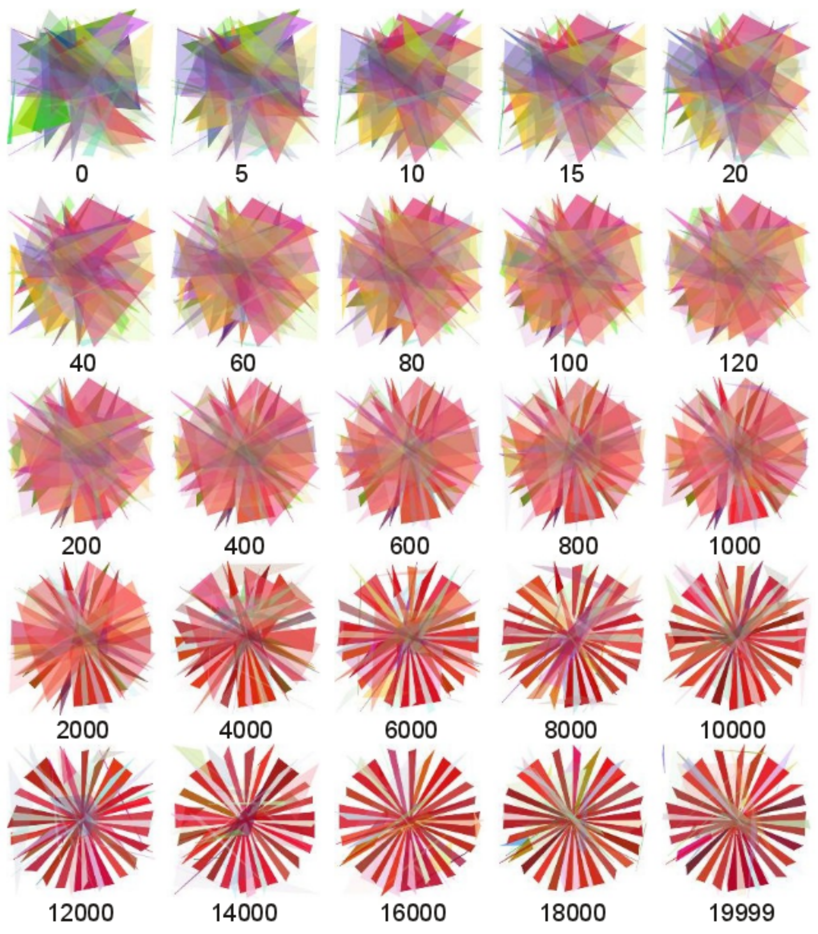


Figure 9.2.: *Mapping a star-like structure using the method described in section 9.1. The target image is shown in figure 9.3 .*

A very appealing feature of the “Mona Lisa example” is that, despite the very high number of parameters, it is indeed possible to *see*, how close the optimization algorithm has come to the global optimum. If a perfect mapping is reached (which is very unlikely, as long as the number of triangles is smaller than the number of pixels), Q becomes 0. The candidate picture will increasingly look like the target picture, the smaller 0 gets. This is in stark contrast to almost any other high-dimensional optimization problem, where one will rarely know how close one has come to the global optimum.

Note that equation 9.1 does not use any information whatsoever about the target picture, other then the color of each pixel[1]. Also, each triangle serves more than one purpose, as it will usually span many pixels. As triangles may overlap, different pixels of the same triangle may lead to different colors

[1]It would be possible to design an evaluation criterion that would increase the similarity between the candidate solution and the target image more quickly. However, the purpose of this example is *not* to design the perfect evaluation criterion, but rather to present optimization algorithms with a difficult problem, on whose example some important characteristics of real-life optimization problems can be demonstrated.

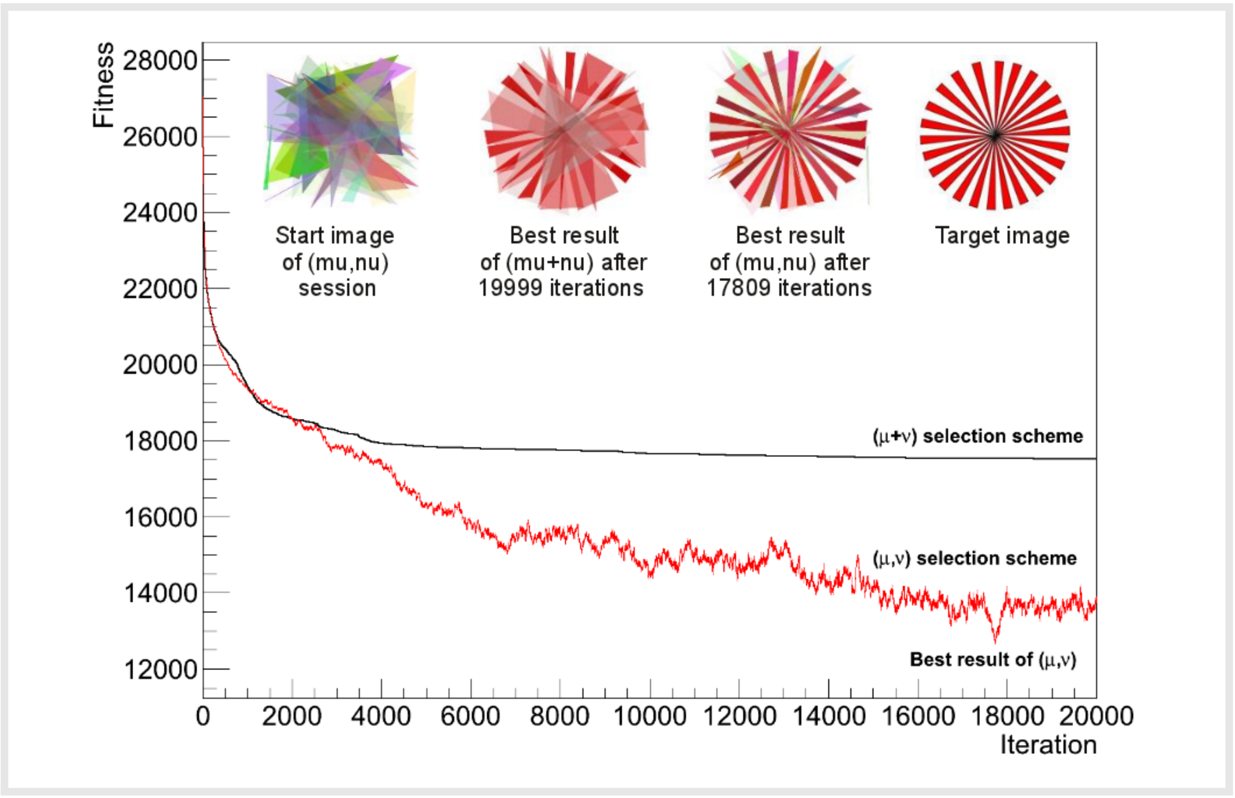


Figure 9.3.: *Our experience shows that, in Evolutionary Algorithms, a* (μ,ν) *selection scheme often performs better than a* $(\mu+\nu)$ *scheme. In both selection schemes, most of the progress during the mapping of the star-like structure is achieved before the first 20% of the optimization run.*

on the candidate image, even though all pixels of the triangle have the same color.

This is also the source of a strong and unpredictable correlation between the parameters of different triangles, with respect to the quality criterion Q. During the course of the optimization, triangles change position and size, so that in each iteration different triangles overlap and jointly form the pixels on the candidate picture. Hence small changes in the parameters can have large effects on the evaluation function.

If it could be visualized (which is impossible), it would likely show a huge amount of local optima and many discontinuous regions[2]. There is also a very large number of global optima[3], as the parameters of each two triangles could be exchanged without changing the value of the evaluation criterion (equation 9.1).

[2]In the sense that adjacent values can show abrupt changes – we are dealing with an optimization problem here that can be expressed by discrete rather than continuous variables.

[3]. . . with identical values, by definition.

9.1.2. Mapping a Star-Like Structure

The mapping of the Mona Lisa does show the ability of optimization algorithms to perform optimization even for very large parameter spaces. However, the target image is quite complex and forces the algorithm to make use of overlapping triangles to model the details of the target image sufficiently well. We will now use a much simpler image, consisting of triangles arranged in a circle (compare figure 9.3) in order to demonstrate some characteristics of optimization algorithms.

Rather than 300 triangles, we can restrict the optimization to only 50 triangles (which still amounts to 500 parameters to be optimized). Figure 9.2 shows several candidate solutions during the course of the optimization, which was limited to a maximum of 20000 iterations ($0--19999$). Small numbers below each picture indicate, to which iteration the solution belongs. The optimization has again been performed with an Evolution Strategy[4], and took about 12 hours in multithreaded mode on an Intel Core-2 Quad processor.

As a special characteristic, the population has grown from 100 individuals at iteration 0 to 400 individuals over the course of the optimization. Better solutions are relatively easy to find at the beginning of the optimization. Hence a speedup can be achieved by reducing the size of the population in the beginning and then letting it grow, as it becomes more and more difficult to find better solutions.

The optimization was done in two modes – $(\mu+\nu)$ and (μ,ν). The example demonstrates that often the (μ,ν) scheme performs better that the $(\mu+\nu)$ scheme. In (μ,ν), new parents are chosen from the collection of children only. However, a visible feature in figure 9.3 is that the fitness can worsen in (μ,ν) mode (albeit usually not dramatically). Compare section 4.1.4 for a more detailed explanation.

One of the more dominant features of this optimization – even more visible than in the case of figure 9.1 – is that the target image becomes recognizable very early during the optimization. Figure 9.3 shows the fitness according to equation 9.1, as a function of the iteration. It is visible that the optimization very quickly reaches a quality close to the final result, and that, after about 20-30% of the optimization, the speed of improvements slows down significantly in both modes – $(\mu+\nu)$ and (μ,ν). **This is a very common behaviour in many types of optimization algorithms and makes it likely, that for many, even highly computationally expensive, technical optimization problems, satisfactory results can be achieved within short time**[5].

Iterations $400--2000$ can be interpreted as local optima, some large, albeit translucent, triangles can be seen overlapping the whole picture. The other, circular triangles are still rather far away from the desired target color. Hence moving the large, overlapping triangles will result in a decrease of quality. Nevertheless the optimization left this particular local optimum before 20% of the iterations had passed.

The optimization has taken care of the most visible features of the logo first, namely the color and the circular triangle structure.

[4]. . . and the Geneva library, of course

[5]. . . at least if it is not a strict requirement that the global optimum is found instead of an optimum representing a significant improvement of the figure of merit.

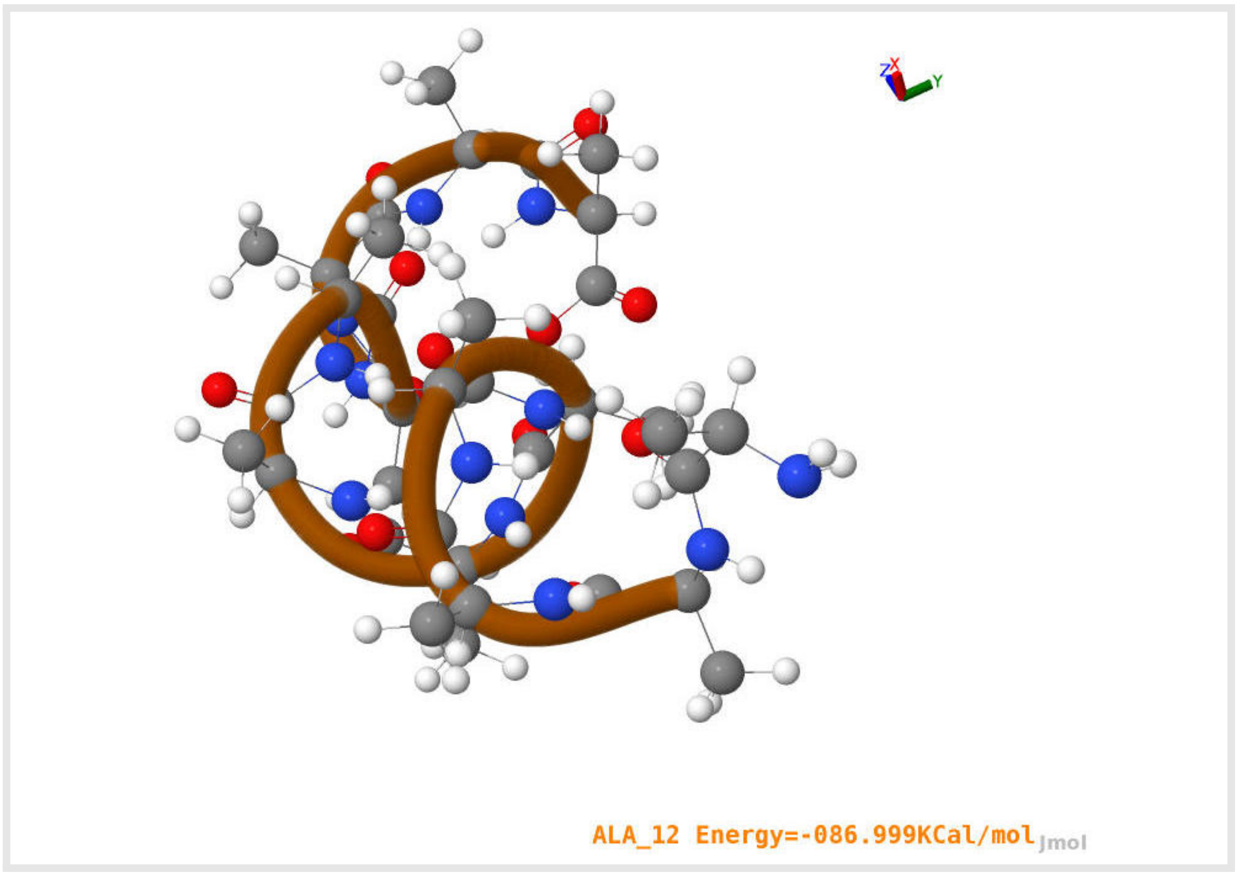


Figure 9.4.: *This form of the protein ALA12H was obtained by minimizing the potential energy of the molecule through variations of the geometry. The picture was created with the molecular viewer JMol.*

9.2. Protein Folding

Proteins are built from chains of amino acids. Due to the intra-molecular forces, proteins never appear as a linear chain in nature, but usually appear in a “globular” or “fibrous” form. In biology and particular in the pharmaceutical industry, it is of the utmost importance to understand and predict, in what shape a protein of a given atomic structure will appear in nature. An exact understanding of this structure allows for example the *design* of substances, which will dock to active centers exposed by the protein. This allows for example to create new drugs to help cure diseases.

Protein folding means the minimization of the potential energy of a molecule by varying its geometric parameters (e.g. the angles between adjacent atoms, with respect to the backbone). This in itself is not a complicated procedure. The difficulty, however, lies in the exact calculation of the force field for a given geometry, which serves as this optimization problem’s evaluation function. In the most simple case, forces are modelled as “springs”. However, even though the calculation of the potential energy for a given geometry is simple in this case, there are strong limits to the validity of such models. More realistic models of the force field need to use quantum-mechanical methods, which in turn results in computationally very expensive evaluation functions.

Hence, for the engineer designing the evaluation function and performing the actual optimization, it is important to find the right balance between correctness (i.e. the quality of approximations) and speed. It is also very important to use optimization algorithms that converge quickly towards a satisfactory optimum, so that as few as possible evaluations of candidate solutions are necessary. Protein folding thus is an art in itself.

Figure 9.4 shows the result of such an optimization, in the case of Ala12H. A simple force field, calculated using OpenBabel [50], was used. The picture was created using the JMol [42] molecular viewer.

One of the difficulties encountered while dealing with this optimization problem was the license of Open Babel. Both Open Babel and Geneva are Open Source. However, Open Babel uses the GNU General Public License "GPL" in version 2, while Geneva uses the newer (Affero) GPL version 3. Even though both have originated from the same source, they are not compatible with each other [26]. As a direct consequence, the only way to call Open Babel's functions for the determination of a protein's energy in the evaluation function was to use an external program for the evaluation.

Another experience made was that the best results could be obtained when using different optimization algorithms alternatingly for the optimization of different regions of the protein.

9.3. Training Feed Forward Neural Networks

Training a Feed Forward Neural Network ("FFNN") means adapting the weights associated to each node in such a way that, during supervised learning, for a given input sample the deviation of the actual from the desired output becomes minimal. In other words, training a FFNN means minimizing equation 9.2,

$$E(\vec{w}) = \frac{1}{2}\sum_{\nu=1}^{p}\sum_{k}(y_k^{\nu} - s_k(x^{\nu}))^2 \tag{9.2}$$

where $\vec{w}$ are the weights, p is the total number of input/output patterns used in supervised training, k are the output nodes, y_k^{ν} is the actual output of each output node as a result of input pattern x^{ν}, and $s_k(x^{\nu})$ is the desired output associated to x^{ν}.

Traditionally, the training was done by using the error-backpropagation algorithm. It was introduced not the least because at the time of its invention, computing resources were scarce and expensive. However, with today's powerful computers, it is well possible to use standard parametric optimization algorithms, such as Evolution Strategies, for the minimization of equation 9.2, though [9, 8].

Figure 9.6 illustrates this on the example of two value distributions, for a FFNN with the geometry 2-2-1

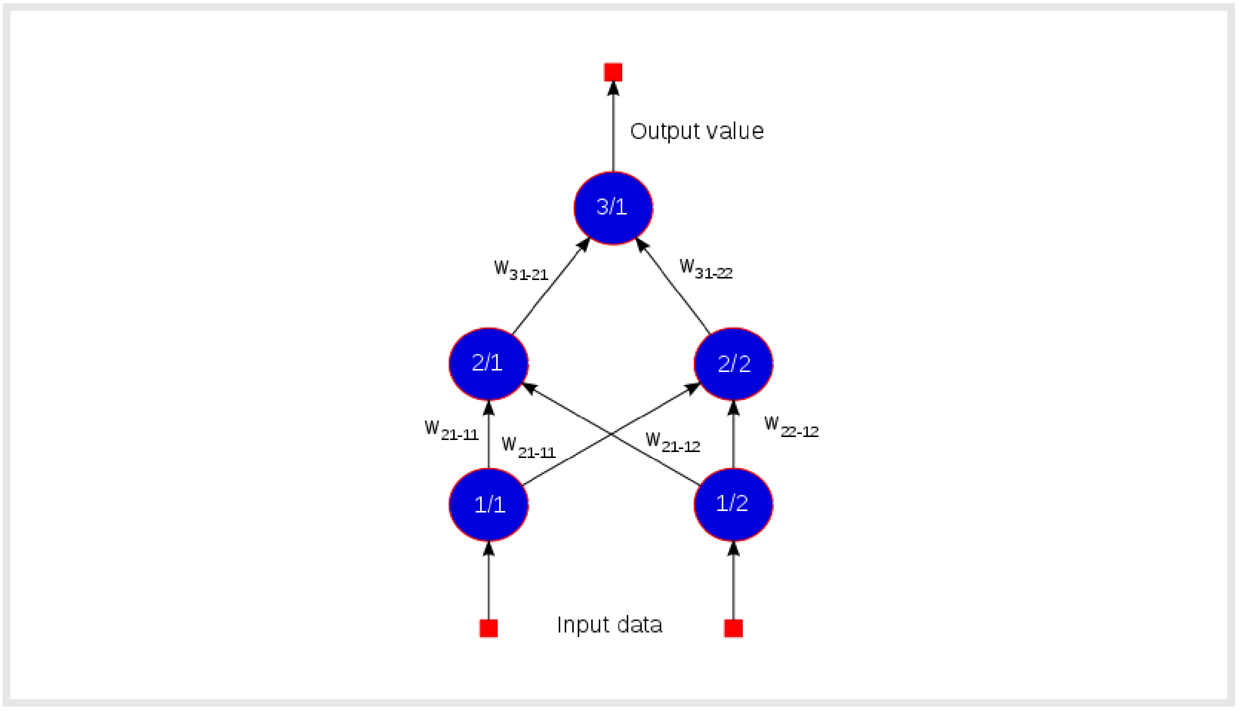


Figure 9.5.: *A feed-forward neural network calculates the output of a node by applying a sigmoid function to the weighted sums of the output of preceding nodes (minus a threshold)*

(2 input nodes, 2 nodes in one hidden layer and one output node). One data set is evenly distributed across the entire value range, the network's desired output for this pattern is 0. The other is confined to the proximity of the x- and y-axis. The desired output value of the network is 1. It is impossible to exactly distinguish between both data sets, as they overlap, and the only defining characteristic is the location on the canvas. However, a neural network must be able to find suitable cuts that help to make the optimal distinction, with only a minimal error.

In figure 9.6, the output of the trained network is superimposed to the picture (multiplied by 10 and rounded). It is clearly visible that, by requiring the output of the network to be lower than 0.4, one would be able to retrieve an almost clean sample of the evenly distributed data set.

Note that it would also be possible to train the architecture of the network, albeit with more difficulty.

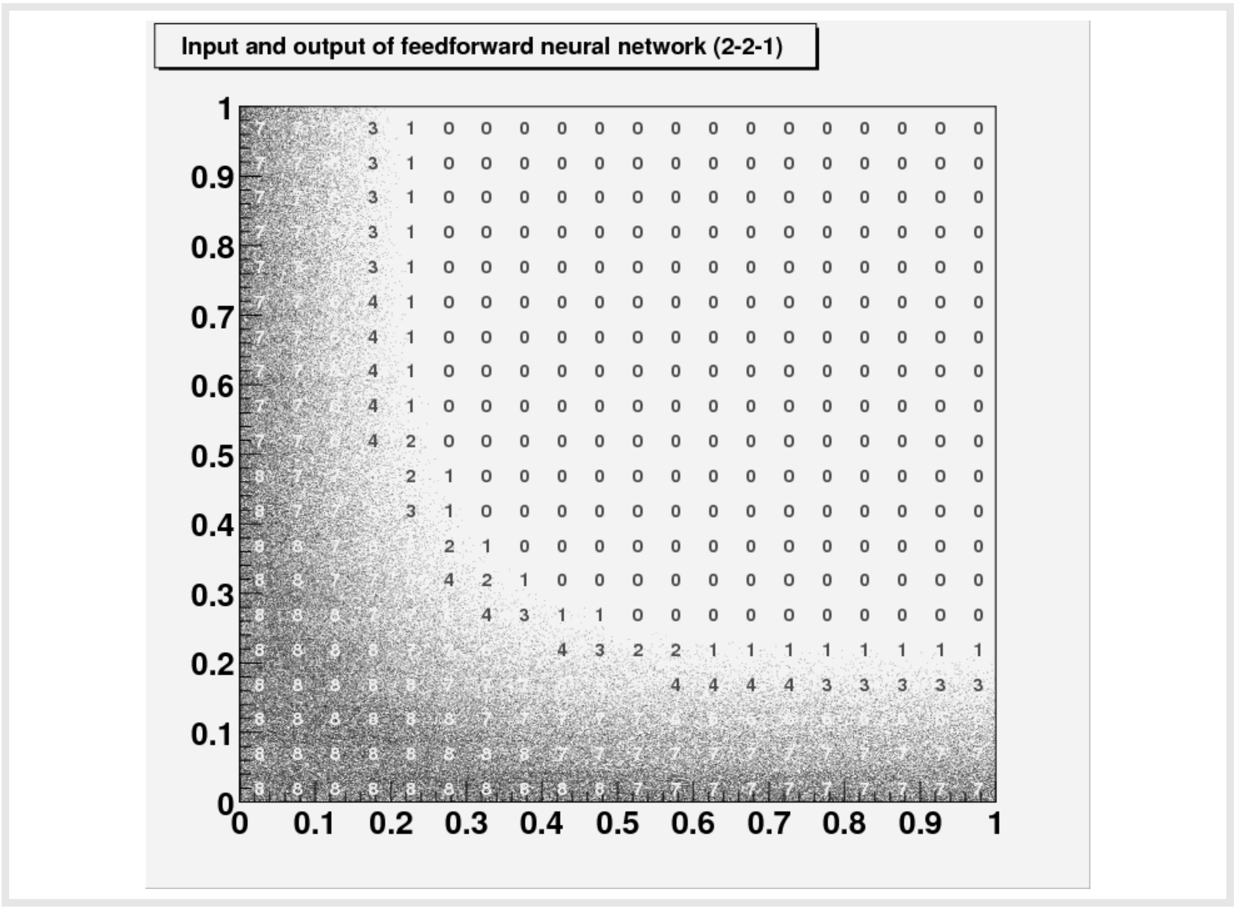


Figure 9.6.: *Input pattern used to train a 2-2-1 network (an even distribution of input values and a distribution confined to the x- and y-axis) and the output of the network after the training in different areas of the canvas (multiplied by 10 and rounded).*

Part II.

Using the Geneva Optimization Library

Chapter 10.

Compilation and Installation

This chapter provides an overview of the steps you need to take in order to build the Geneva library run and run one of the examples. In particular, you will learn here how to compile and install Geneva.

10.1. Prerequisites

A few pre-conditions need to be met in order to use the Geneva library.

Operating System

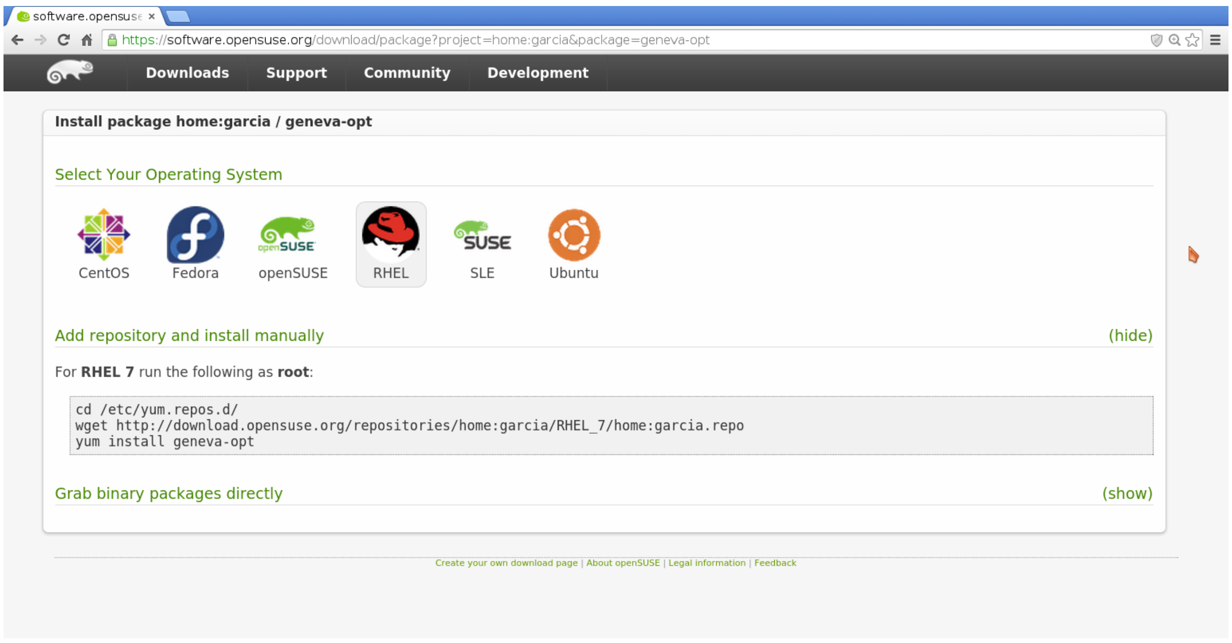
Geneva is being developed on the Linux platform, specifically under Ubuntu Linux. You should be able to install it on just about any other recent Linux system and might be able to do the same on other Unix(-like) systems.

In particular, at the time of writing, we do nightly builds and test-runs on **Ubuntu 14.04** and **12.04**. **CentOS 7.0**, **CentOS 6.6** and **FreeBSD 9.0** are also among the test-platforms. Geneva is known to work under various versions of **Scientific Linux**, although this is not regularly tested by the project team. Frequent builds also happen on **Debian** (testing branch), and experimental support exists for **MacOS 10.10**. Moreover, Geneva builds and works on MS Windows under the **Cygwin** environment, and it is known to compile natively on MS Windows with very few changes, while full support is being worked on.

Build Environment

Geneva uses the free CMake build environment on all platforms. As this is the same environment that is used e.g. for Linux's larger GUI systems[1], a recent version should already be available on your system. If not, then try installing it from the software repositories of your Linux distribution.

[1] such as KDE


software.opensuse
https://software.opensuse.org/download/package?project=home:garcia&package=geneva-opt
Downloads
Support
Community
Development
CentOS
Fedora
openSUSE
RHEL
SLE
Ubuntu
(hide)
(show)
Create your own download page | About openSUSE | Legal information | Feedback

Install package home:garcia / geneva-opt

Select Your Operating System

Add repository and install manually

For **RHEL 7** run the following as **root**:

```
cd /etc/yum.repos.d/
wget http://download.opensuse.org/repositories/home:garcia/RHEL_7/home:garcia.repo
yum install geneva-opt
```

Grab binary packages directly

Figure 10.1.: *Binary packages of the Geneva library are available for some Linux flavours in the dedicated download area of the OpenSUSE Build Service:* `https://software.opensuse.org/download/package?project=home:garcia&package=geneva-opt`

Enter the command "`cmake --version`" on the command line in order to check if this build environment is installed on your system. Geneva requires **CMake 2.8** or newer. Most Linux systems will provide a pre-compiled CMake package.

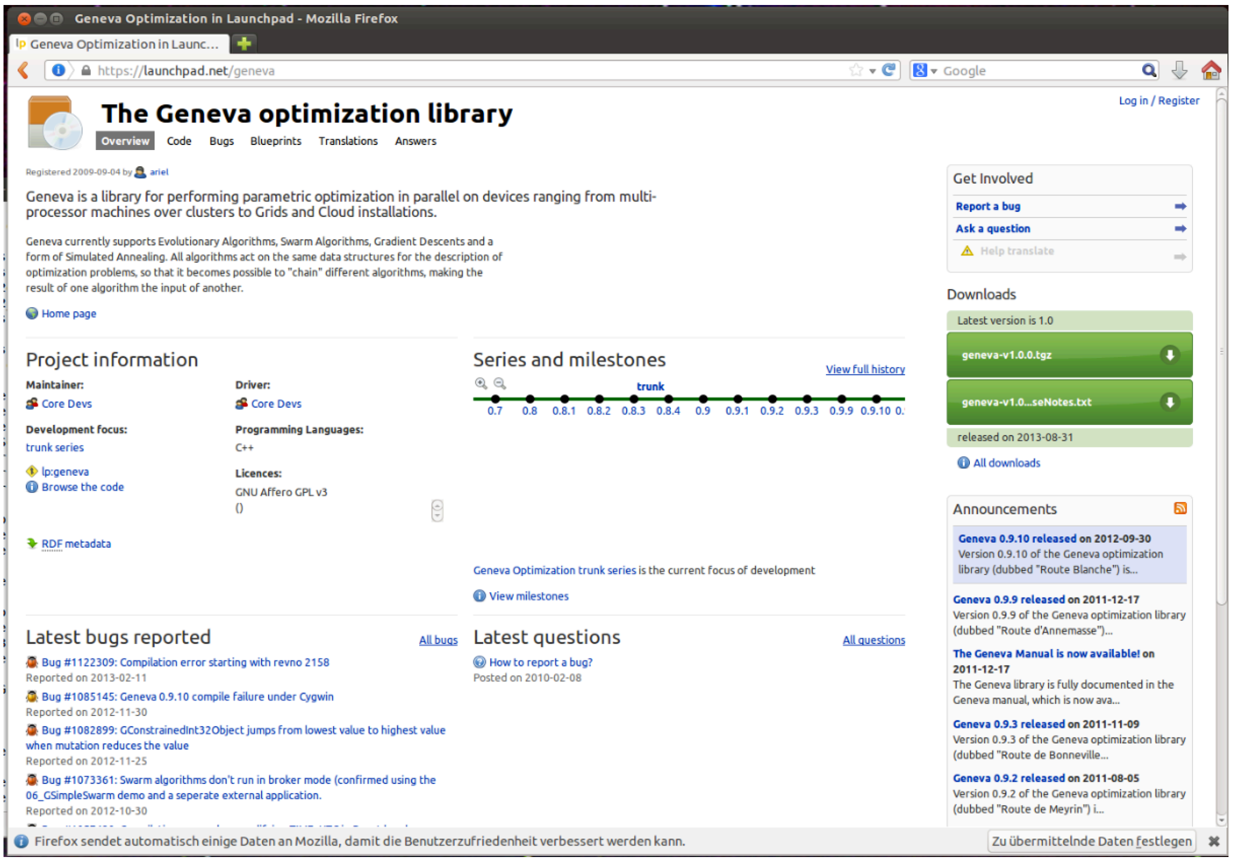
You do not need CMake if you install a pre-compiled Geneva package and you use another build environment (like Autotools, etc.) for your own software.

Compiler

Geneva is developed and tested with different versions of the GNU C++ Compiler **`g++`** and with the LLVM compiler frontend **`clang`**. Any reasonably recent Linux distribution will provide a compatible GCC version, so you shouldn't need to worry about this.

Enter the command `g++ -v` or `clang -v` at the command prompt in order to check whether the compiler is already installed and its version. In case the compiler was not installed yet, use your distribution's package manager to install it first: the packages are called `g++` on Ubuntu and Debian, and `gcc-c++` on Red Hat/Fedora and SUSE variants, `clang` will be named similarly.

At the time of writing, the most current and recommended version of `g++` is 4.9. For `clang` we recommend to use the latest version that is available at the time of your tests.



Geneva Optimization in Launchpad - Mozilla Firefox

Geneva Optimization in Launc...

https://launchpad.net/geneva

Google

Log in / Register

The Geneva optimization library

Overview Code Bugs Blueprints Translations Answers

Registered 2009-09-04 by ariel

Geneva is a library for performing parametric optimization in parallel on devices ranging from multi-processor machines over clusters to Grids and Cloud installations.

Geneva currently supports Evolutionary Algorithms, Swarm Algorithms, Gradient Descents and a form of Simulated Annealing. All algorithms act on the same data structures for the description of optimization problems, so that it becomes possible to "chain" different algorithms, making the result of one algorithm the input of another.

Home page

Project information

Maintainer: Core Devs

Driver: Core Devs

Development focus: trunk series

Programming Languages: C++

lp:geneva

Browse the code

Licences: GNU Affero GPL v3 ()

RDF metadata

Series and milestones

View full history

trunk

0.7 0.8 0.8.1 0.8.2 0.8.3 0.8.4 0.9 0.9.1 0.9.2 0.9.3 0.9.9 0.9.10 0.

Geneva Optimization trunk series is the current focus of development

View milestones

Latest bugs reported

All bugs

Bug #1122309: Compilation error starting with revno 2158
Reported on 2013-02-11

Bug #1085145: Geneva 0.9.10 compile failure under Cygwin
Reported on 2012-11-30

Bug #1082899: GConstrainedInt32Object jumps from lowest value to highest value when mutation reduces the value
Reported on 2012-11-25

Bug #1073361: Swarm algorithms don't run in broker mode (confirmed using the 06_GSimpleSwarm demo and a seperate external application.
Reported on 2012-10-30

Latest questions

All questions

How to report a bug?
Posted on 2010-02-08

Get Involved

Report a bug

Ask a question

Help translate

Downloads

Latest version is 1.0

geneva-v1.0.0.tgz

geneva-v1.0...seNotes.txt

released on 2013-08-31

All downloads

Announcements

Geneva 0.9.10 released on 2012-09-30
Version 0.9.10 of the Geneva optimization library (dubbed "Route Blanche") is...

Geneva 0.9.9 released on 2011-12-17
Version 0.9.9 of the Geneva optimization library (dubbed "Route d'Annemasse")...

The Geneva Manual is now available! on 2011-12-17
The Geneva library is fully documented in the Geneva manual, which is now ava...

Geneva 0.9.3 released on 2011-11-09
Version 0.9.3 of the Geneva optimization library (dubbed "Route de Bonneville...

Geneva 0.9.2 released on 2011-08-05
Version 0.9.2 of the Geneva optimization library (dubbed "Route de Meyrin") i...

Firefox sendet automatisch einige Daten an Mozilla, damit die Benutzerzufriedenheit verbessert werden kann.

Zu übermittelnde Daten festlegen

Figure 10.2.: *In order to download the Geneva library collection, visit* `https://launchpad.net/geneva` *and click on the green download link on the right side of the page*

The build system will try to enable some `C++11` constructs, if available `g++` version is modern enough, but it will always enable those (and therefore require `C++11` support) with `clang`.

The Boost library collection

The only external code dependency of Geneva is on the Boost collection of C++ libraries [72]. Thus a recent version of Boost needs to be installed on your system for both compiling and running Geneva code. Boost is a peer-reviewed, very portable collection of high-quality, open source components. Its license (compare appendix D.2) allows you to use Boost libraries with only very few restrictions in a commercial context. Many of Boost's libraries constitute a reference implementation for new features of the new C++ standard C++11, but can be used with older compilers as well.

We currently support **Boost version 1.48** or newer, with a **recommendation for Boost 1.55**. Recent releases of most Linux distributions seem to ship a modern enough version, so chances are you do not need to install Boost separately, or can rely on the mechanisms provided by your distribution of choice to install it. Note that you will need both the binaries of the libraries and the header files. The latter are contained in packages whose names usually end on `-dev` or `-devel` (for instance, `libboost-dev` in Debian).

If you install a Geneva binary package, then your system's package manager will take care of installing the required Boost dependencies, as these are declared in the Geneva packages.

Listing 10.1 shows two commands that you can use to check which components of Boost are installed on your system. The first command is valid for systems that are based on the `dpkg` packaging system (such as Debian and Ubuntu). The second one can be used on systems using the `rpm` package manager (such as Fedora-derivatives or OpenSUSE).

Listing 10.1: Checking installed Boost version on dpkg- or rpm-based systems

```
Debian> dpkg -l | grep -i boost
Fedora> rpm -qa | grep -i boost
```

If the Boost version on your system is older than expected, then you should install a newer version from source and compile it yourself: instructions can be found on the Boost web page. In this case, we recommend *not* to install Boost in a standard system location like `/usr`, but to use a custom location instead[2], such as `/opt/boost`.

You will come across two Boost components particularly often when using Geneva:

- Boost's reference-counted smart pointer `shared_ptr` [11]. For stability reasons, Geneva uses these wherever possible.
- The Boost.Serialization library [61] enables Geneva's objects to be passed over a network.

The appendix gives a short introduction into both (see B.1 and B.2). A very good online description of many of Boost's libraries is provided by Boris Schäling [70]. It is also available in book form [71] and as an e-book. An older, albeit still good introduction in book form was written by Björn Karlsson [43]. For more detailed information we recommend to have a look at Boost's thorough documentation [73].

10.2. Installation using binary packages

Once the prerequisites are fulfilled, you need to install Geneva. Two alternatives may be available in your case: either installing from binary packages or building and installing from source.

Starting from version `1.4.1`, binary Geneva packages are available for several relevant Linux distributions, including RedHat Enterprise Linux 7, SUSE Linux Enterprise 12, Ubuntu, Fedora, and a few others. If you run one of these Linux distributions you may spare yourself some work –and some CPU-cycles– by installing the corresponding pre-compiled package. As mentioned in the previous section, not all prerequisites are strictly necessary in this case (but they might be however needed for your own application linking with Geneva)... If you are out of luck and no binary packages are available

[2]Reason: As many Linux applications rely on the version already available on your system, you should keep additional Boost versions separate.

for your OS, then you will need to build and install Geneva from source: please proceed to the next section for more details.

The Geneva binary packages are built by the OpenSUSE Build Service, and are made available at the corresponding download area: `https://software.opensuse.org/download/package?project=home:garcia&package=geneva-opt`. Just select your distribution and follow the (very simple) installation instructions provided there. Figure 10.1 shows that download page.

Beware, these packages should still be considered "preliminary" and your mileage may vary with them. Please report any issues or suggestions via Geneva's bugtracker: `https://bugs.launchpad.net/geneva`.

10.3. Installation from source

Building and installing Geneva from source is your only alternative if binary packages are not available for your operating system, if you want to take a deeper look at the source code, or even to investigate some issue that you might have. The source code is available from the Launchpad software collaboration portal at `https://launchpad.net/geneva`. Download the current stable distribution[3]. Figure 10.2 shows Launchpad's download page.

10.3.1. Unpacking

Next, unpack the sources (shown in listing 10.2 on the example of Geneva 1.6 (Ivrea)). `/home/developer/>` represents the Unix prompt.

Listing 10.2: Unpacking the Geneva sources

```
/home/developer/> tar -xvzf geneva-v*.tgz
```

There should now be a directory `/home/developer/geneva-1.6 (Ivrea)`. You may now create a build directory anywhere on your system and change to that directory. Thanks to `CMake`, Geneva is capable of doing "out of source builds", meaning that all object files and libraries resulting from the compilation of Geneva will end up in the build directory instead of cluttering the source tree. Listing 10.3 shows the necessary commands.

Listing 10.3: Creating a build directory

```
/home/developer/> mkdir build
/home/developer/> cd build
```

[3]If you would like to use the current development version, you can use the command "`bzr lp:geneva .`" to download the code. "`bzr`" is the command line client for the Bazaar version control system. Note, though, that it is quite possible that you encounter problems with the development release of Geneva.

There is also a `build` directory in the source tree that you can use. This is advisable if you intend to use `eclipse` for the modification of the Geneva sources rather than "just" writing applications based on Geneva.

10.3.2. Building

To build Geneva you could just call `cmake` with the appropriate parameters. However, `CMake` has a large amount of options, and the Geneva compilation may be controlled with its own options, so this can be cumbersome.

Thus, in order to simplify the procedure and make the most useful build options easily accessible, a configurable build script `scripts/prepareBuild.sh` is provided in the source tree. This will take care of running `CMake` with all the necessary parameters, while keeping the desired options in a small configuration file, `genevaConfig.gcfg`. Actually, running this configuration script without any configuration file is possible, and it will set sensible default values: that will be in many cases enough for successfully compiling Geneva.

However, if the default values set by `prepareBuild.sh` are not enough for successfully configuring Geneva, or you just want to fine-tune some parameter values, then you will need to provide it with a configuration file. A sample configuration file with detailed comments is provided in the Geneva source tree: `scripts/genevaConfig.gcfg`. You may copy that file, edit its contents, and subsequently call the build script. **Using the configuration script is the recommended way of building the Geneva library.** Listing 10.4 shows the required commands, again on the example of Geneva 1.6 (Ivrea).

Listing 10.4: Preparing the Geneva build configuration

```
~developer> cd build
~developer/build> cp ../geneva-1.4.1/scripts/genevaConfig.gcfg .
~developer/build> your-editor-of-choice ./genevaConfig.gcfg
~developer/build> ../geneva-1.4.1/scripts/prepareBuild.sh ./genevaConfig.gcfg
```

The example assumes that the `build`-directory as well as the Geneva source tree are located in the developer's main directory.

The build configuration

The file `genevaConfig.gcfg` will look similar to listing 10.5. Under normal circumstances you will only have to edit the variables `BOOSTROOT` and `INSTALLDIR`:

- `BOOSTROOT` points to the root of your boost installation. Usually this should only be needed if you are not using your operating system's version of Boost, but your own manually compiled copy. The variable's value in Listing 10.5 assumes that you have installed your own version of Boost in `/opt/boost`.
- `INSTALLDIR` tells the build system whereto the compiled libraries should be copied.

If you run into problems while using the Geneva library, three more settings might be useful:

- BUILDMODE lets you build the Geneva library in Debug mode.
- BUILDTESTCODE specifies whether the build system should also build Geneva's tests. Note that the build procedure will last substantially longer in this case.
- VERBOSEMAKEFILE, when set to "1", results in verbose output during the compilation procedure. This may help to detect erronous compiler- or linker flags applied on your system.

We suggest to read chapter 30.1 in case of problems with Geneva, and particularly to report any problem you might encounter back to gemfony http://www.gemfony.eu/.

Listing 10.5: A sample genevaConfig.gcfg file

```
CMAKE=/usr/bin/cmake                 # Where the cmake executable is located
BOOSTROOT="/opt/boost"               # Where Boost is installed
#BOOSTLIBS="/usr/lib"                # Use either BOOSTROOT or these two, if CMake
#BOOSTINCL="/usr/include"            #   is not able to find the Boost installation
BUILDMODE="Release"                  # Release or Debug mode
BUILDSTD="auto"                      # The C++ standard to use for building
BUILDTESTCODE="0"                    # Whether to build Geneva with test code
VERBOSEMAKEFILE="0"                  # Whether to emit compilationinformation
INSTALLDIR="/opt/geneva"             # Where the Geneva library shall go
```

You are now ready to compile the Geneva library.

Compiling the Geneva library

As a first step, you need to call make inside of the build-directory. A suitable Makefile was created when you called the build script in the last section.

Listing 10.6: Compiling the Geneva library

```
/home/developer/build> make -j 2
```

The -j 2 option tells make that it should simultaneously compile Geneva on two CPU cores, i.e. using two parallel "jobs". You can adjust this number to match the number of CPU cores in your system. But please note that the compiler requires additional memory for each additional compilation job you request. You should foresee at least 4 Gigabytes of RAM per job in the case of the GNU compiler, due to the deeply nested template constructs used in Boost. Your mileage may vary with different compilers and versions. If unsure, just call make without any arguments.

Depending on your system, the compilation can take a long time. On a reasonably well equipped system, though, it should not exceed 10-15 minutes[4] when using all cores. Note that, at the time of writing, the Geneva library collection comprised some 170000 lines of code.

[4]Core i5, 3GHz

If everything went okay, you should now have the Geneva libraries and a number of compiled examples. Try executing one of them (compare listing 10.7). You should then see an output similar to listing 10.8.

Listing 10.7: Executing a first example program

```
/home/developer/build> cd examples/geneva/01_GSimpleOptimizer
/home/developer/build/examples/geneva/01_GSimpleOptimizer> ./GSimpleOptimizer
```

Listing 10.8: A sample output for GSimpleOptimizer

```
Seeding has started
Starting an optimization run with algorithm "Evolutionary Algorithm"
0: 64.6073443050163
1: 25.9597623490252
2: 8.89715425355864
3: 1.45564799125829
4: 0.861887897798893
[...]
999: 7.37074272148514e-13
End of optimization reached in algorithm "Evolutionary Algorithm"
Done ...
```

Congratulations – you did it! You have just sought for the minimum of a paraboloid, using Geneva's multithreaded execution mode, and using an evolutionary strategy.

In case you wonder what configuration options are available for the program, try running it with the `-help` switch. Also have a look into the `config` sub-directory. You will find a number of configuration files in JSON format there.

NOTE: Geneva will refuse to run if the configuration files aren't found. If you do not have these files, they can be auto-generated for you. Simply make sure that an empty subdirectory `config` exists at the place where you call the Geneva-application. A set of configuration files with default values will then be created for you in this sub-directory.

Information on how to define your own optimization problem will be provided in chapter 11, and in far more detail in chapter 26.

Installing and using your new Geneva installation

Now that the library was compiled and tested to work correctly, what remains to be done is to let the build system deploy the library and associated header files to their final location, as well as updating your system environment to be able to use it.

One option for the first task it to run the following command:

Listing 10.9: Installing the compiled library

```
/home/developer/build> make install
```

This will copy the Geneva files to the location given by the value of the `INSTALLDIR` parameter in the `genevaConfig.gcfg` file. It is generally *not* a good idea to run the above command as *root*, as you might end up with all the files to deploy in an undesirable location in case of a configuration or build system error. A way around this is to create a target directory under the ownership of and with write-permission for the user that has also done the compilation, and to run the install command as that user.

However, if you want to make a system-wide installation of Geneva, and you cannot rely on the provided binary packages (see section 10.2), then at least on Linux another option is to create your own installation package first. This can be achieved with:

Listing 10.10: Creating an installation package

```
/home/developer/build> sudo make package
```

The resulting RPM or Debian package can then be installed in the usual way, deploying the Geneva library in the location `/opt/geneva`. E.g. under Ubuntu Linux, the following command would install the resulting Debian package:

Listing 10.11: Installing the installation package

```
/home/developer/build> dpkg -i geneva-opt-1.4.1-Linux.deb
```

Note that the resulting package will strongly depend on the Boost version being used. We thus generally only recommend this procedure, if you have used the Boost version already available on your system, which should have been installed through your normal package manager. Otherwise the Boost version the package expects might not match the version it was compiled with.

To be able to use your newly deployed copy of Geneva, you will need to to tell Linux where to find the new libraries. The easiest possibility is to add the Geneva `lib` folder to the `LD_LIBRARY_PATH` environment variable:

Listing 10.12: Setting the environment

```
/home/developer> export LD_LIBRARY_PATH=/opt/geneva/lib:${LD_LIBRARY_PATH}
```

This obviously assumes that Geneva was installed in `/opt/geneva/`.

To make this setting permanent you should add that statement to your shell's profile file, for instance `~/.bashrc` in the most common setups.

Another alternative is to set the dynamic loader search path system-wide in `/etc/ld.so.conf.d/`. Create a file `geneva.conf` there, with a single line in it: `/opt/geneva/lib`. Finally, run the command `ldconfig -a` to activate the new library path. Consult your system's documentation for more details.

NOTE that the procedure for making compiled libraries known to your system will differ on operating systems other than Linux.

Chapter 11.

Defining a first Optimization Problem

This chapter gives an overview of the steps you need to take in order to solve a first, custom optimization problem with the Geneva library.

Key points: (1) Geneva makes a hard distinction between optimization algorithms and problem definitions (2) The latter can be shared freely amongst available optimization algorithms (3) Some algorithms specifically act on a given parameter type and will leave the rest of a candidate solutions parameters untouched (4) Individuals (aka problem definitions) are classes that are derived from the `GParameterSet` class (5) A number of member functions needs be present in an individual (6) Of these, the `fitnessCalculation()`, `load_()` and `clone_()` functions as well as the `serialize()` function are the most important ones (7) The `Go2` class allows to freely add individuals, mix optimization algorithms to act on these individuals and perform optimization in various parallelization modes

No extensive coverage will be provided in this chapter on the implications of running Geneva in networked or multithreaded mode. Rather, we will concentrate on formulating the optimization problem itself. Going from here to parallel execution requires only minimal considerations on the side of the user and will be discussed in chapter 26.

Note that the example shown below – the search for the minimum of a parabola – is intentionally very simple, as the goal of this chapter is to give you a "high-level view" of the way Geneva works. Par II of this manual provides a far more in-depth explanation of the many options Geneva gives you.

This chapter assumes that you have at least a conceptual understanding of what the term "parametric optimization" entails. If you are unsure, we suggest that you first read chapter 2, as it tries to define the term "optimization" and what can – and cannot – be done with computer-based optimization algorithms.

We also assume that you have some background in the C++ programming language. As an example, you will need to understand terms like "class", "header", "template" or "derivation".

The complete example shown in this chapter is – including everything needed to build the example – available in the Geneva source tree in the sub-directory `examples/geneva/02_GParaboloid2D`. Once you have compiled Geneva, you should automatically also have access to the compiled example below the build directory.

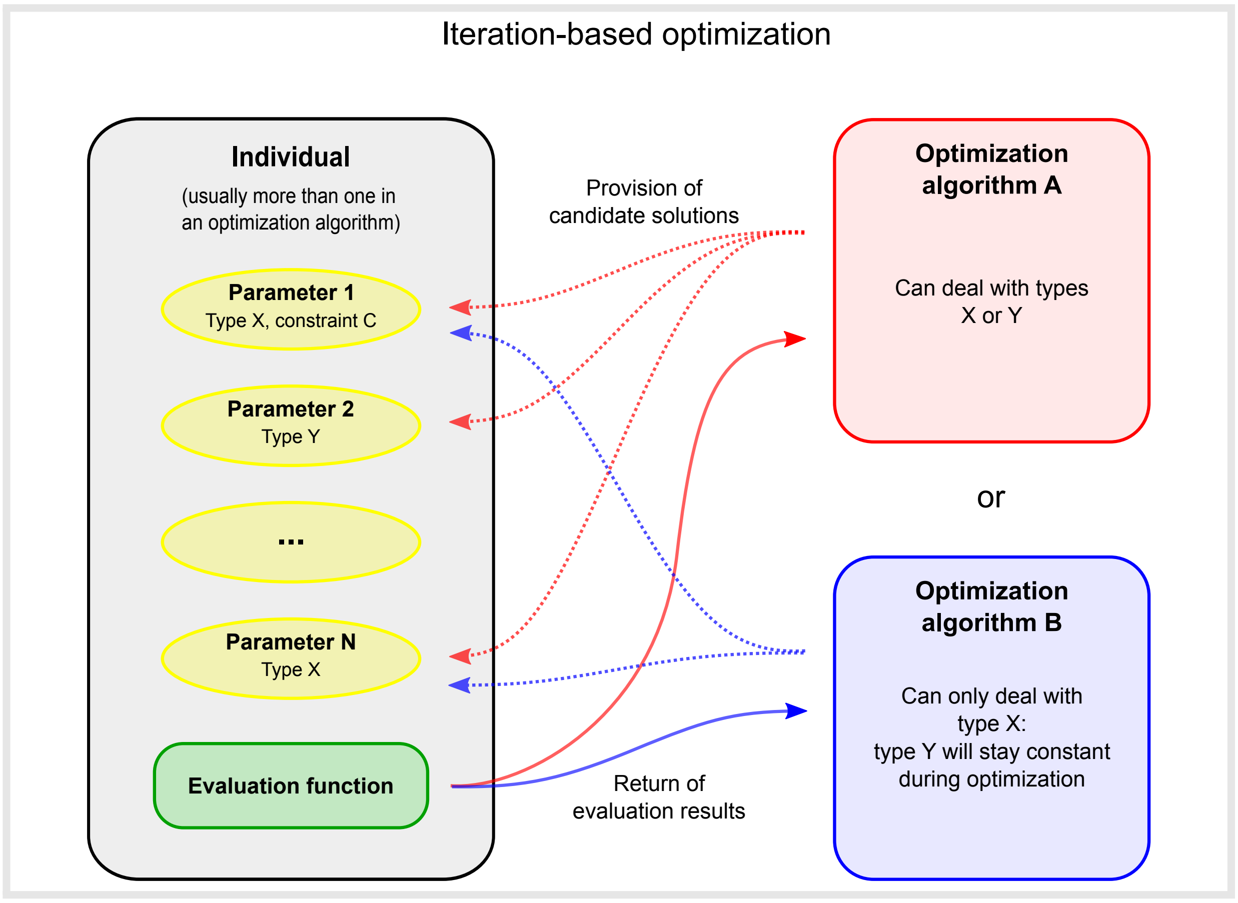

Iteration-based optimization
Individual
(usually more than one in an optimization algorithm)
Parameter 1
Type X, constraint C
Parameter 2
Type Y
...
Parameter N
Type X
Evaluation function
Provision of candidate solutions
Return of evaluation results
Optimization algorithm A
Can deal with types X or Y
or
Optimization algorithm B
Can only deal with type X:
type Y will stay constant during optimization

Figure 11.1.: *Geneva makes a hard distinction between the specification of optimization problems (left) and the optimization algorithms (right) trying to provide candidate solutions with increasingly better evaluation.*

11.1. Outline

Geneva distinguishes between optimization algorithms and the actual optimization problem. Conceptually, an optimization problem consists of a set of parameters (possibly equipped with constraints), and a definition of how to get from these to one or more numerical Evaluation Criteria.

All of this is expressed through C++ classes and objects, which in Geneva terminology are called *Individuals*. Individuals comprise the parameter definition as well as the evaluation function(s), which assign one or more evaluations to a parameter set. In Geneva, optimization problems can be described in terms of floating-point, integer or boolean parameters.

Individuals need to comply with the API and functionality Geneva expects, but are otherwise free-form. They are designed and programmed by the user and, by means of their adherence to the Geneva API, plug into the existing optimization framework provided by Geneva.

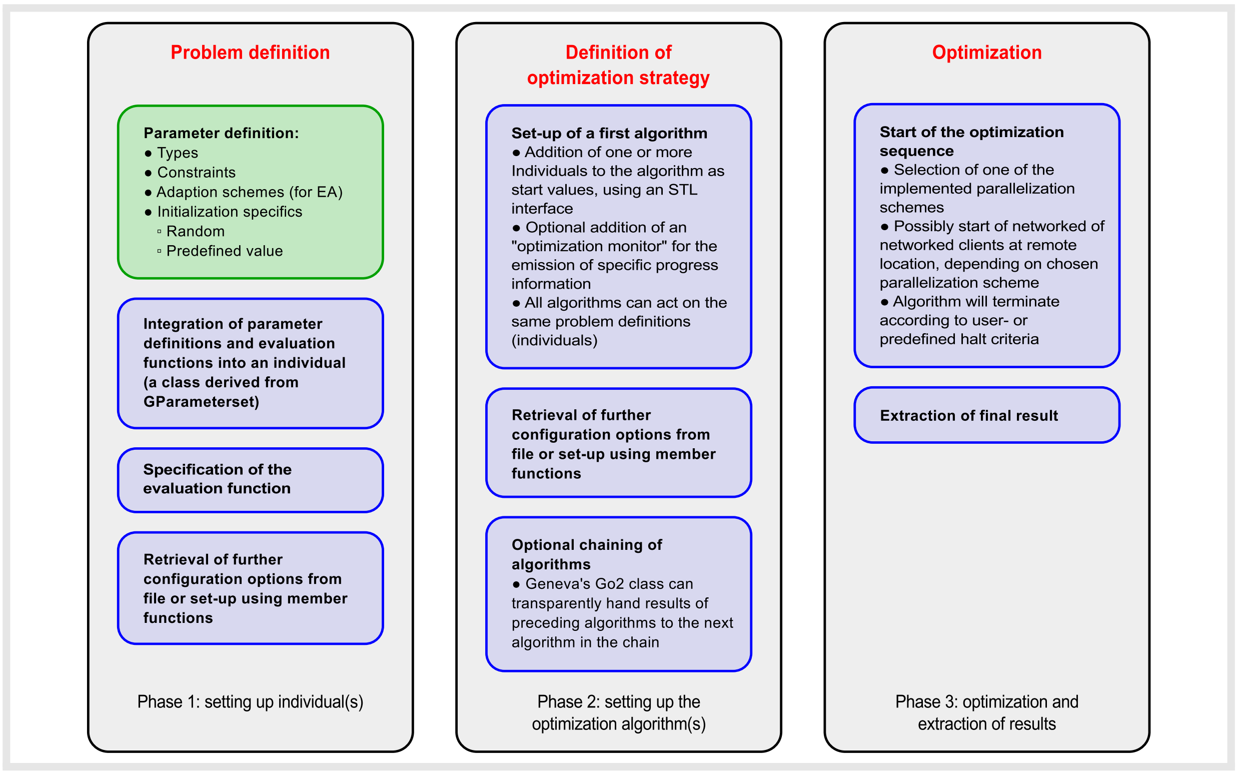

Problem definition
Parameter definition:
• Types
• Constraints
• Adaption schemes (for EA)
• Initialization specifics
▫ Random
▫ Predefined value
Integration of parameter definitions and evaluation functions into an individual (a class derived from GParameterset)
Specification of the evaluation function
Retrieval of further configuration options from file or set-up using member functions
Phase 1: setting up individual(s)
Definition of optimization strategy
Set-up of a first algorithm
• Addition of one or more Individuals to the algorithm as start values, using an STL interface
• Optional addition of an "optimization monitor" for the emission of specific progress information
• All algorithms can act on the same problem definitions (individuals)
Retrieval of further configuration options from file or set-up using member functions
Optional chaining of algorithms
• Geneva's Go2 class can transparently hand results of preceding algorithms to the next algorithm in the chain
Phase 2: setting up the optimization algorithm(s)
Optimization
Start of the optimization sequence
• Selection of one of the implemented parallelization schemes
• Possibly start of networked of networked clients at remote location, depending on chosen parallelization scheme
• Algorithm will terminate according to user- or predefined halt criteria
Extraction of final result
Phase 3: optimization and extraction of results

Figure 11.2.: *Parametric optimization with the Geneva library is done in three distinct phases.Phases of parametric optimization*

Specific "Individual-objects", i.e. Individuals equipped with a specific set of parameter values, are called *Candidate Solutions*.

It is the task of the algorithm side to equip candidate solutions with suitable parameter values in an iterative procedure, so that, over time, the evaluation of the candidate solutions improves.

Usually, optimization algorithms will require the evaluation of multiple parameter sets or candidate solutions in each iteration. In the easiest case with a single evaluation criterion, the parameter set with the best evaluation at the end of the optimization run represents the best (available[1]) solution to an optimization problem.

Individuals can be switched freely between the optimization algorithms implemented in Geneva, so that you can try out different algorithms with ease, or use the result provided by one algorithm as the starting point for another algorithm. Note, though, that not all algorithms will modify all implemented parameter types[2].

Figure 11.1 further illustrates this situation.

[1]See chapter 2 for a discussion of why we speak about the best *available* solution rather than *the best* or *ideal* solution.
[2]E.g., a gradient descent is not capable of modifying boolean values

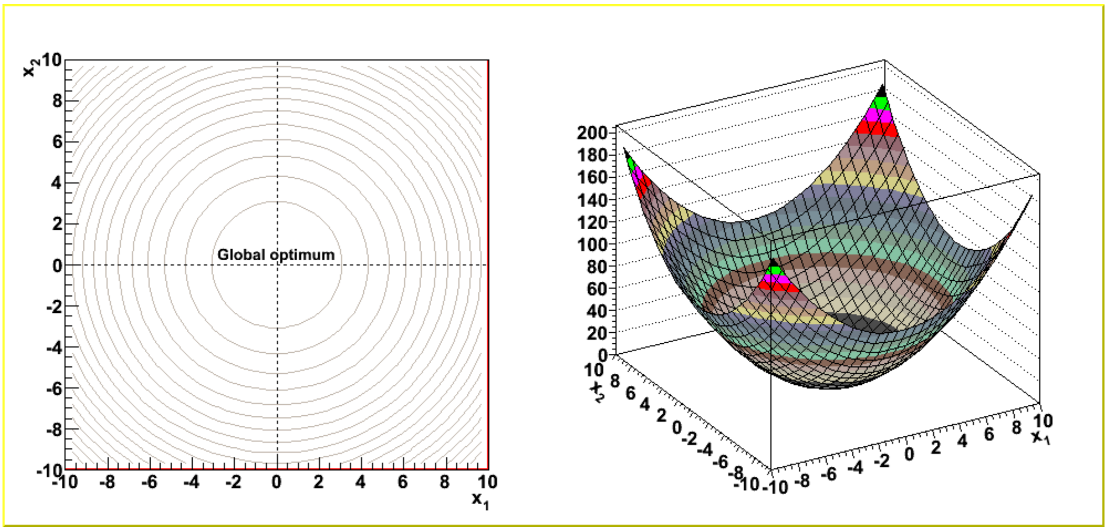


Figure 11.3.: *Two views of a two-dimensional paraboloid. Left: contour lines; right: three-dimensional view with function values.*

11.2. Defining a paraboloid

We will now express the complete optimization problem, including the definition of the individual, the `main()` function and a complete build environment.

We will start with the problem definition. In an n-dimensional paraboloid, the "quality" of the parameter set (n floating point numbers in our case) is defined by equation 11.1.

$$f(x_1,x_2,\ldots,x_n) = \sum_{i=1}^{n} x_i^2 = x_1^2+x_2^2+\ \ldots\ +x_n^2 \tag{11.1}$$

Figure 11.3 shows two views of a two-dimensional parabola[3] (as a contour- and a 3D-plot).

Ironically, a high quality in this case means a low function value – here we are searching for a set of parameters that **minimizes** the evaluation function. This is a quite common convention – one usually defines optimization problems so that a minimal value of the evaluation function is best.

As we will see, we have to perform two distinct steps to formulate this problem in the context of Geneva:

- We need to specify the parameters that describe the problem
- Geneva needs to be informed how to get from a given parameter set to an evaluation

All this is done by overloading the corresponding functions of a C++ class, including some "administrative" functions. One of the more important administrative duties is the ability to serialize the class, i.e. bring it into a form that can be transferred over a network. As we will see, though, this is simple,

[3]The plot was created with the ROOT analysis and visualization framework. See appendix C for a discussion

as Geneva supports you in this task.

11.3. Class Declaration

Listing 11.1 shows the smallest possible declaration of `GParaboloidIndividual2D`, the class which lets us search for the minimum of a two-dimensional parabola. It is derived from `GParameterSet`, which forms the base class of all individuals (i.e. problem definitions) in Geneva. Through `GParameterSet`, a comprehensive infrastructure becomes available to you, which lets you model your optimization problems in an intuitive way.

Listing 11.1: The declaration of the GParaboloidIndividual2D class

```
class GParaboloidIndividual2D :public GParameterSet
{
public:
  GParaboloidIndividual2D(); // default constructor
  GParaboloidIndividual2D(const GParaboloidIndividual2D&); // copy constructor
  virtual ~GParaboloidIndividual2D(); // destructor

protected:
  // Loads the data of another GParaboloidIndividual2D
  virtual void load_(const GObject*);
  // Creates a deep clone of this object
  virtual GObject* clone_() const;

  // Calculates the object's quality
  virtual double fitnessCalculation();

private:
  // Make the class accessible to Boost.Serialization
  friend class boost::serialization::access;

  // Triggers serialization of this class and its base classes.
  template<typename Archive>
  void serialize(Archive & ar, const unsigned int) {
    using boost::serialization::make_nvp;
    // Serialize the base class
    ar & BOOST_SERIALIZATION_BASE_OBJECT_NVP(GParameterSet);
    // Add other variables here like this:
    // ar & BOOST_SERIALIZATION_NVP(sampleVariable);
  }

  const double PAR_MIN_; // Lower boundary for parameters
  const double PAR_MAX_; // Upper boundary for parameters
};
```

You can find further, far more detailed information about individuals in chapter 15. The construction of an individual from parameter objects is illustrated in figure 15.1 on page 146 [4].

11.4. Member functions

In this section we will discuss `GParaboloidIndividual2D`'s member functions one by one.

The constructor

In Geneva, you will always need to provide a default constructor for your individuals[5].

Listing 11.2: The constructor of the GParaboloidIndividual2D class

```
GParaboloidIndividual2D::GParaboloidIndividual2D()
        : GParameterSet()
        , PAR_MIN_(-10.)
        , PAR_MAX_(10.)
{
        for(std::size_t npar=0; npar<2; npar++) {
                // GConstrainedDoubleObject is constrained to [PAR_MIN_:PAR_MAX_[
                boost::shared_ptr<GConstrainedDoubleObject>
                   gcdo_ptr(new GConstrainedDoubleObject(PAR_MIN_, PAR_MAX_));
                // Add the parameters to this individual
                this->push_back(gcdo_ptr);
        }
}
```

Apart from initializing the parent class and the local variables, we use the constructor to fill the individual with parameter objects in a loop. We have chosen a problem that demands floating point parameters[6], and we want them to be limited in their allowed value range.

A good choice to describe our parameters are `GConstrainedDoubleObject` objects. The object wraps a single variable, whose values are constrained to the `[PAR_MIN_:PAR_MAX_[` range. The actual value of the parameter is set to a random value in the allowed range automatically[7]. Another suitable parameter class would have been the `GConstrainedDoubleCollection`.

The `GConstrainedDoubleObject` is wrapped into a `boost::shared_ptr` smart pointer from the Boost library collection. Its main task is to delete the object pointed to when the last reference to it becomes invalid. **This way we do not have to care ourselves for the *when* and *if* of deleting dynamically allocated objects.** Its duties could thus be compared to what a garbage

[4] Note that you can also build "parameter trees", as Geneva comprises object collections that can themselves be stored in individuals and can hold objects of their own type.

[5] This requirement results from the Boost.Serialization library which, in order to be able to de-serialize an object, will first default-construct it, then fill its parameters with values. There is no requirement for a default-constructor to be public.

[6] As was said already, Geneva can also cope with integer and boolean values

[7] One can of course also specify the desired value, if needed

collector does in Java or C#. In most other respects a `boost::shared_ptr` behaves like a standard C/C++ pointer. `boost::shared_ptr` is discussed in more detail in appendix B.1.

Finally, we add the parameter object to the individual. Its interface is almost identical to a std::vector[8]. You can thus just use the usual `push_back()` function to add the smart pointer to the individual.

For simplicity reasons, we do this inside of the constructor. It would also have been possible to attach such objects from outside the individual. We recommend one of two options to attach parameters to inviduals:

- EITHER: Keep parameter definition and evaluation in one class and define parameters in the constructor or a member function of the individual.
- OR: Create a factory class that builds individuals of a given type. Geneva has a complete framework for this (see e.g. sections 33.4 and 26.2). This is the preferred way.

We can also set the value of the `GConstrainedDoubleObject` object with the `setValue()` member function (compare listing 11.3. Here, we choose a random value in the allowed value range. Chapter 31 discusses the topic of random number generation in the Geneva context in detail. **Please also note that, in all classes derived from `GParameterSet`, you have access to a random number generator.**

Listing 11.3: It is also possible to set the value if a GConstrainedDoubleObject manually

```
gcdo_ptr->setValue(gr.uniform_real<double>(PAR_MIN_, PAR_MAX_));
```

Naming schemes

We take a short look at the naming scheme of parameter objects:

- All Geneva classes start with an upper-case “G”
- Constrained parameter types clearly say so in the class name
- The target type is contained in the name
- As we are using a class to describe the parameter, we call it an “Object”. In comparison, there is also a `GDoubleCollection`, which is a `std::vector`-style collection of unconstrained double values. As these are not individually encapsulated by an object, but are instead contained in a collection, the class is called a `GDoubleCollection` (without the term `Object`). There are many other parameter classes, e.g. `GConstrainedInt32Object` (whose meaning should now be clear) or a `GDoubleObjectCollection`, which represents a collection of unconstrained double parameters, each of which is encapsulated in its own object and can thus be treated individually.

[8]Indeed, one of its parent classes wraps a std::vector<> .

Mixing parameter types

You can freely mix different parameter types in an individual. I.e., in the constructor you could also add a `GBooleanObject` or a `GConstrainedInt32Object`. We do not do this here, to keep the example simple. You could also have added collections of parameter objects (or collections of collections, if you really wanted to) to the individual, thus creating a sort of "tree" of parameters[9]. For the sake of simplicity, however, here we just add the parameters at the root level.

The copy constructor

Listing 11.4: The copy constructor of the GParaboloidIndividual2D class

```
GParaboloidIndividual2D::GParaboloidIndividual2D(const GParaboloidIndividual2D& cp)
        : GParameterSet(cp)
        , PAR_MIN_(-10.)
        , PAR_MAX_(10)
{ /* nothing */ }
```

As we only have constant local data, the only real task of `GParaboloidIndividual2D`'s copy constructor is to hand the object to be copied to the parent class. The actual copying of parameter objects is done for you by the `GParameterSet` class. For more complex use cases you will of course have to copy any necessary local data members of the individual as well, just like in any other copy constructor.

The `load_()` and `clone_()` functions

In an individual, Geneva can mix parameters of different types. This is possible, as the different parameter types share a common base class and API. The class at the root level is called `GObject`.

Consequently, all loading and cloning of Geneva objects needs to happen through pointers to this base class. For this purpose, Geneva mandates that every class that directly or indirectly derives from `GObject` implements a `load_()` function. Instantiable classes (those that are not purely virtual) in addition need to support the `clone_()` function.

User interaction with these functions then happens through two functions in `GObject`, called `load()` and `clone()` (note the missing underscores!) which are defined in `GObject`[10].

Both `load_()` and `clone_()` are declared `protected`, as the user doesn't need to call them directly. Derived classes need to be able to access particularly the `load_()` function, though.

The `clone_()` function is trivial, as it only returns a copy-constructed object. The implementation can be found in listing 11.5.

[9]This can be useful for formulating more complex problems. E.g., if you want to describe a feed-forward neural network, you might want to have individual collections of parameters for each layer

[10]The `clone()` function has additional functionality. For example, it can convert the result of the cloning procedure "on the fly" to a desired target type, using a template argument. Details can be found in part II.

Listing 11.5: The clone() function simply returns a copy-constructed GParaboloidIndividual2D object

```
GObject* GParaboloidIndividual2D::clone_() const {
        return new GParaboloidIndividual2D(*this);
}
```

The `load_()` function is slightly more complex, as it (usually[11]) needs to convert its argument to the correct type and also needs to take care of loading the parent class'es data. Listing 11.6 shows the details.

Listing 11.6: The load() function loads the classes own data and that of the parent class

```
void GParaboloidIndividual2D::load_(const GObject* cp)
{
  const GParaboloidIndividual2D *p_load
        = GObject::gobject_conversion<GParaboloidIndividual2D>(cp);

  // Load our parent's data
  GParameterSet::load_(cp);

  // No local data
  // sampleVariable = p_load->sampleVariable;
}
```

First, `cp` is converted to the target type. The function being used here is defined in `GObject`. Internally, checks are being made in DEBUG mode to ensure that one doesn't try to load an object's data into itself, or that the `GObject`-pointer is actually convertible to the target type. An exception will be thrown when this is not the case.

Strictly speaking, the conversion would not have been necessary here, as the next step – calling the parent class'es `load_()` function – would have been possible with the `GObject`-pointer alone. We have no local data, so we do not need to access any data members local to this class declaration in `cp`. But it is convenient to get the checks for self-assignment and conversion errors, so we do the conversion anyway. The commented out line at the end of the function shows how to load local data.

The actual fitness calculation

Listing 11.7 shows how to calculate the fitness from the parameters stored in the individual.

Listing 11.7: The fitness calculation

```
double GParaboloidIndividual2D::fitnessCalculation(){
  double result = 0.; // Will hold the result
  std::vector<double> parVec; // Will hold the parameters

  this->streamline(parVec); // Retrieve the parameters

  // Do the actual calculation
```

[11]Our case is so simple, though, that no local data needs to be loaded

```
	for(std::size_t i=0; i<parVec.size(); i++) {
		result += parVec[i]*parVec[i];
	}

	return result;
}
```

The first step is to extract the parameter values. Remember that parameters are usually encapsulated in objects, and that Geneva allows you to freely mix parameter objects holding different types, and to even arrange them in the form of a tree structure. Hence we need to make use of some tools to extract the parameter values. Three possibilities exist:

- The easiest possibility is shown in listing 11.7. The `streamline()` function extracts all parameter values of a chosen type (in this case `double`) in the order in which they were registered, and adds them to a std::vector. While `streamline()` actually is a function template, the desired type is derived from the type stored in the std::vector – you do not need to specify it separately (but can do so like this `streamline<double>()` if you want it for reasons of clarity).
- If you know the structure of the individual, you can access the parameter objects directly. Remember that the individual has a vector interface, so you can use the usual `at()` and `operator[]` functions). Keep in mind, though, that what you will get in this way will be base pointers to `GObject`, which need to be converted first.
- The `conversion_iterator` lets you iterate over all parameters of a given type on a given level of the tree. It will skip objects of different type, and it will return the objects readily converted to your desired target type. E.g., you could this way extract all GConstrainedDoubleObject objects from the `GParaboloidIndividual2D` object.

The last two options will be discussed in more detail in chapter 15.

As the next and final step we can use the extracted parameters to do the actual fitness or quality calculation – in our case a simple parabola.

There is one more thing to note: `fitnessCalculation()` is again labelled as `protected`. It is hence impossible for users to directly access this function. Instead, they will use the individual's `fitness()` function, as defined in a base class[12].

The reason for this setup is that we want to avoid to re-calculate the fitness whenever we want to know the quality of an individual. Re-calculation is after all *only* necessary when the parameters have changed. Thus `fitness()` will internally store the fitness for the current parameter set, once it has been calculated. As soon as the parameters change, `fitness()` will call `fitnessCalculation()` again. Otherwise it will return the cached value.

[12]The `GOptimizableEntity` class, discussed in more detail in section 12.1.4

Serialization

In order for an object to be transferred over a network connection (or for it to be stored on disk), it needs to be translated into a different format. This could e.g. be XML or the more efficient (but less portable in heterogeneous environments) binary format.

Thanks to the magic of the Boost.Serialization library (see appendix B.2), serializing an individual is mostly a matter of listing the data members to be serialized. This happens inside of the `serialize()` function, shown at the end of listing 11.1.

In our simple example without local data, all we need to do is to trigger serialization of the parent class. Chapter 26 will discuss a more complete example, involving local data.

In order to make the class known to the Boost.Serialization library, you will finally have to add a line each to the implementation (i.e. the `.cpp` file) of your individual (in this case `GParaboloidIndividual2D.cpp`) and to the declaration (i.e. the `.hpp` file):

Listing 11.8: The Boost.Serialization export statements

```
BOOST_CLASS_EXPORT_KEY(GParaboloidIndividual2D) // Goes into the header
BOOST_CLASS_EXPORT_IMPLEMENT(GParaboloidIndividual2D) // Goes into the .cpp file
```

The missing semicolon at the end of `BOOST_CLASS_EXPORT` statements is no mistake, as these are macros. We suggest that you have a look at the actual code of `GParaboidIndividual2D` in the Geneva distribution at this point.

11.5. The `main()` function

Now that we have the individual in place we need to take care of the main function (see listing 11.9).

Listing 11.9: The main() function

```
using namespace Gem::Geneva;
int main(int argc, char **argv) {
        Go2 go(argc, argv, "config/go2.json");

        //-------------------------------------------------------------------
        // Initialize a client, if requested
        if(go.clientMode()) return go.clientRun();

        //-------------------------------------------------------------------
        // Add individuals and algorithms and perform the actual optimization cycle

        // Make an individual known to the optimizer
        boost::shared_ptr<GParaboloidIndividual2D> p(new GParaboloidIndividual2D());
        go.push_back(p);

        // You could add an algorithm to the Go2 class here, which would always be
        // executed first. Not specifiying any algorithms results in the default
        // default algorithm, unless other algorithms specified on the command line.
19          // go & "ea";
20
21          // Perform the actual optimization
22          boost::shared_ptr<GParaboloidIndividual2D>
23                  bestIndividual_ptr = go.optimize<GParaboloidIndividual2D>();
24
25          // Do something with the best result
26  }
```

Geneva is a comprehensive toolkit, with many optimization algorithms and different execution modes. For the highest flexibility, you would need to access all of Geneva's classes directly.

In many cases, however, and particularly when trying to combine different optimization algorithms, the Go2 class provides a useful interface to the most important functionality. We will use it in this example.

In line 3 of listing 11.9, the `Go2` constructor is called with the command line parameters, as well as the name of its configuration file in JSON[13] format. The constructor will then parse any available command line parameters for applicable options and read in the configuration options from its configuration file.

Note that the application will terminate if the target directory (`config`, in this case) does not exist. If the directory exists, but the requested configuration file does not exist, it will create a configuration file with default values for you.

Next, in line 7, we check whether this application has been called in client mode, and if so, execute the client loop. Client mode can be triggered by passing the switches `−client −e 2` to the application on the command line. In networked mode, which is triggered through the option `−e 2`, Geneva applications may act as the server or one of usually many clients. If the option `−client` is not specified, it is assumed that the executable acts as the server. For completeness, please note that you would also have to specify a "Consumer" with a command line option like `−c tcpc`. This would indicate that we want to run Geneva in networked mode, through Boost.Asio. Call the executable with the option `−help` in order to see all available options.

Chapters 23 and 24 discuss the topic of parallelization and consumers in far more detail.

In line 14, an object of the `GParaboloidIndividual2D` class is created and added to the `go` object. We could have added more than one here – all individuals we add here will serve as distinct starting points for the optimization process. Note that we need to wrap the `GParaboloidIndividual2D` class in a `boost::shared_ptr<>` in order to add it to the `go` object. This convention helps Geneva to avoid memory leaks.

In Line 19, we could have added an evolutionary algorithm to the `go` object. Doing this, using the easy notation `go & "ea";` would force the `go` class to start every optimization run with an evolutionary algorithm.

However, the line is commented out in listing 11.9. As a consequence, unless the user specifies one or more optimization algorithms on the command line, the default algorithm will be used. This happens to be an evolutionary algorithm again. All configuration options of the evolutionary algorithm are read from a configuration file in this case, but could of course also be specified through member functions.

[13]JSON == **J**ava **S**cript **O**bject **N**otation

Note that we could have added other algorithms using the `&`-notation as well, and that we could also have added explicit objects instead of the mnemomic `"ea"`. As an example, we could have simply added a multi-threaded evolutionary algorithm object, followed by a networked swarm object. In this case, multi-threaded optimization with an evolutionary algorithm could have been followed by networked optimization with a swarm algorithm. The best solution found by preceding algorithms is then used as the starting point for the following optimization algorithm.

Finally, in line 22, we start the actual optimization cycle and extract the best individual for later inspection.

11.6. A note about performance

You might by now have gotten the impression that some of the techniques being used here could have a negative impact on Geneva's performance (in particular the extensive use of smart pointers and type conversions).

However, please keep in mind that the focus of the library is on problems with particularly complex and computationally expensive evaluation functions.

Thus a choice has been made in the design of the library to rate stability and consistency of the core library higher than cutting-edge efficiency. This decision does not affect the execution of evaluation functions for candidate solutions. As, in the chosen deployment scenario, the evaluation of candidate solutions will account for the most of the execution time, the effect of our decision on the performance of the optimization run should be minimal.

Through the chosen design, however, it is far less likely that the optimization crashes than if the core library would have been written solely with bare pointers instead of smart pointers and without any conversions.

11.7. What we didn't say ...

This chapter has covered the basics. More realistic examples will require further functionality, such as:

- Some optimization algorithms, like Evolutionary Strategies, associate "adaptors" with each parameter. In this chapter, however, we have just used the default settings, which will not be best for all problems.
- When you perform more complex optimizations, you will almost certainly want to read in external configuration information for your individual. This could be related to the settings of your adaptors, or it could be necessary to access an external data set, whose name needs to be provided to the individual. Geneva has an entire framework for parsing configuration files.
- Geneva allows you to let entire collections of candidate solutions compete against each other. This allows you to explore different areas of the parameter space in parallel.

These and many more topics will be discussed in chapter 26.

Chapter 12.

Class Hierarchies and Principles

At the time of writing, Geneva comprises three different class hierarchies, dealing with random number generation, communication and optimization[1]. The focus of this chapter will be on the optimization hierarchy, as the user will rarely have to deal with the other two hierarchies[2].

As a prelude to a description of common usage patterns, this chapter introduces the core class structure of the optimization library. The "outer" (user-visible) parts of the class tree will be discussed in the following chapters.

Key points: (1) Geneva comprises three class hierarchies, dealing with random number generation, communication and optimization (2) Additional utility classes are used throughout the hierarchy (3) The user will usually only have to deal with the optimization hierarchy (4) The optimization hierarchy is rooted in the `GObject` class, from which several branches emanate (5) The most important branches deal with optimisation algorithms and the definition / codification of the actual optimization problem (6) Other branches include adaptors (mainly used in Evolutionary Algorithms), monitor classes (used to monitor the progress of an optimization run), "trait" classes holding information specific to a given optimization algorithm (to be stored in the "individuals") and parameter definitions.

12.1. Core Optimization Classes

Figure 12.1 shows the innermost classes of the optimization-related class hierarchy. All classes in this hierarchy derive directly or indirectly from the `GObject` class.

All functionality that is directly related to optimization algorithms is available through the `libgeneva` library (named after the entire library collection) and associated header files[3].

[1]. . . plus a set of "common" utility classes
[2]. . . beyond a standard call from `main()`, which will usually stay the same
[3]Note that also a library of individuals for demonstration-, test- and profiling-purposes exists, called *geneva-individuals*

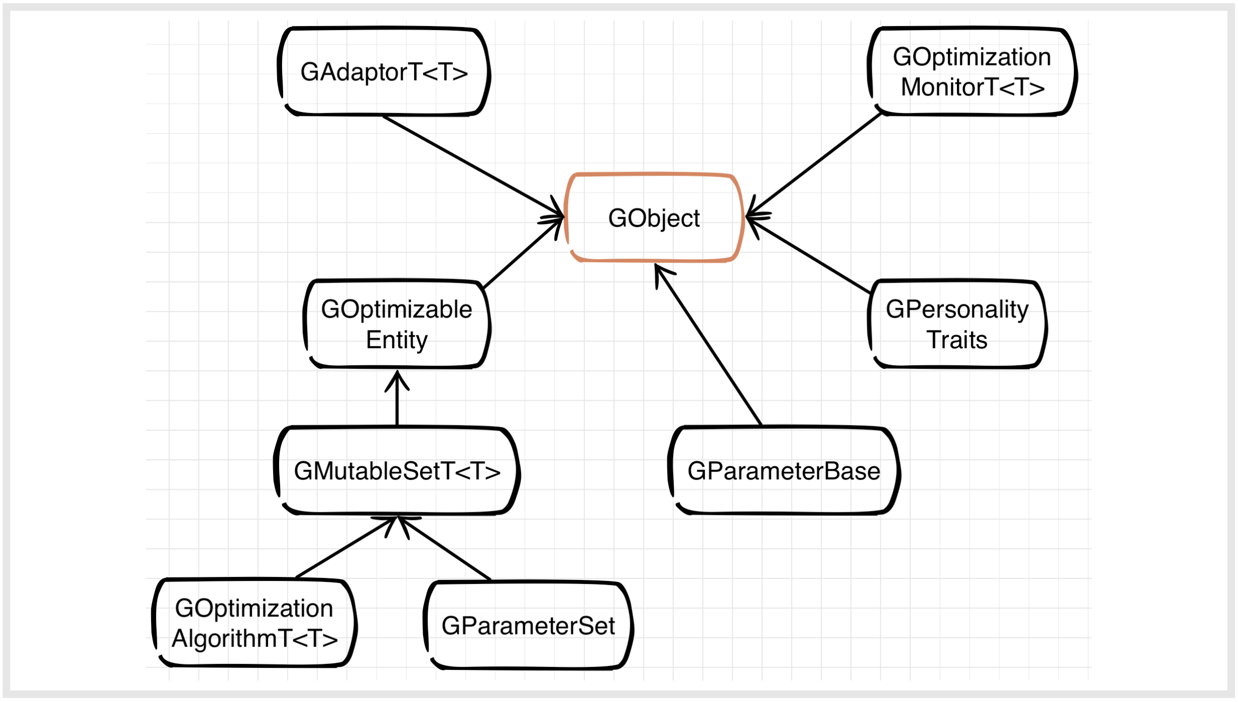


Figure 12.1.: *Geneva's optimization-related class hierarchy is rooted in the `GObject` class. The figure shows the innermost classes only.*

12.1.1. The GObject Class

`GObject` serves as the largest common denominator of all optimization related classes. The need for this setup arises from the fact that, from Geneva's perspective, in many cases the actual type of an object is not clear. As an important example, Geneva uses the STL vector interface in many places. Parameter sets, as discussed in section 12.1.5, allow to freely mix different parameter types. This gives a user a lot of freedom in describing his optimization problems. Loading and cloning of objects – which is a frequent operation in optimization algorithms – thus needs to happen through base pointers. `GObject` provides the necessary infrastructure to convert to the actual target type, and provides many convenience functions needed for (de-)serialization.

12.1.2. Parameter Definition with the `GParameterBase` class

All predefined parameter types of the Geneva library derive from the `GParameterBase` class. There are several important operations associated with this class-type[4]. In particular, `GParameterBase` objects can be:

[4]Note that in most cases, `GParameterBase` only defines the interface, and that the actual implementation is up to derived classes (i.e. the actual parameter types)

- ***adapted***. In the case of Evolutionary Algorithms and Simulated Annealing (compare chapters 4 and 5), a common motor for optimization is to associate *adaptors* with each parameter object. Not surprisingly, the `adapt()` call of `GParameterBase` triggers adaption.
- ***randomly initialized***. The `randomInit()` call triggers a type dependent initialization with a random value.

There are two kinds of `GParameterBase`-derivatives. Individual parameter objects encapsulate a single value. Collections encapsulate a set of parameter values of identical type. Note that there is also an object holding a collection of `GParameterBase` objects, so you can actually build “trees” of parameter objects. Further details are discussed in chapter 13.

12.1.3. Adaptors

Adaptors are associated with[5] `GParameterBase` objects and are used in the context of Evolutionary Algorithms and Simulated Annealing. Their most important operation is the `adapt(T& val)` call. `T` is a template parameter, i.e. a placeholder for a specific type. The function is implemented in the `GAdaptorT` class.

Note the trailing “`T`” in the class name. It indicates that this is a template class[6]. Derived classes specify both the type of parameter to be adapted, and the operations to be performed. Adaptions can be applied both to individual values or to collections of values of the same type.

`adapt(T& val)` encapsulates further functionality, such as the ability to define a likelihood for the adaption to be actually carried out.

Users will rarely need to define their own adaptors, as Geneva comes with a comprehensive collection.

12.1.4. Individuals

Individuals encapsulate the entity to be optimized by a given optimization algorithm. The two most important operations for individuals, implemented in the `GOptimizableEntity` class, are

- **`fitness()`** : There are actually two operations triggered by this call. Whenever the parameter values of an individual change, a dirty flag must be set. This happens automatically for all predefined parameter modifications in Geneva[7]. Thus, the function will first check a “dirty flag” in order to determine, whether re-calculation of an individual’s value is needed. Where this is not the case, it will return the last known fitness value. If the dirty flag is set, it will call a user-defined evaluation function. This rather complex setup happens, because the evaluation of a parameter set may be very costly – in extreme cases it might well take several hours. So we need to male sure that the actual calculation only happens when really needed.
- **`adapt()`** : This call will trigger adaption of all parameter objects, as discussed in sections 12.1.2 and 12.1.3.

[5]read: stored in a derivative of the GParameterBase class ...

[6]Chapter 29 discusses this and other coding conventions

[7]Note that, if a user chooses to manually modify the parameters, he must take care that re-evaluation is triggered

There are two main types of individuals in Geneva: Most importantly, the classes encapsulating the actual problem to be optimized (implemented through the `GParameterSet` class), plus the optimization algorithms themselves.

The fact that optimization algorithms implement the `GOptimizableEntity` interface means that it becomes possible to perform meta-optimization. This could simply involve the evolution of the configuration parameters of an algorithm, or could happen in the form of Multi-Populations, as discussed in section 4.5. Although this hasn't been tried yet with the Geneva library, it should even be possible to let different optimization algorithms compete against each other. Note that, within Geneva, only Evolutionary Algorithms and Simulated Annealing can be used as the host for this type of meta-optimization,

12.1.5. Parameter Sets

The `GParameterSet` class amalgamates the parameter definitions underlying a given optimization problem with the calculation of a parameter set's fitness. **It will likely be this class that users will be involved with mostly and that will consume most of their programming effort.**

A `GParameterSet` can be likened to a `std::vector<GParameterBase>`, i.e. a collection of `GParameterBase` objects, that can be modified with the usual STL algorithms. The class gains the ability to act like a `std::vector` by means of derivation from a template class, `GMutableSetT<T>`, whose template parameter is set to `GParameterBase`.

The user needs to define the fitness calculation in a derived class by overloading the `double fitnessCalculation()` function.

As an example, in chapter 11 we have derived `GParaboloidIndividual2D` from `GParameterSet`. In this new class, we have defined how a given parameter set can be translated into a fitness. In this case this happened by calculating the square of all floating point parameters and summing them up. We have also added parameter objects in the class's constructor.

Inside of `fitnessCalculation()` the reverse operation was performed and the individual's floating point parameters were extracted with the `streamline(std::vector<par_type>& parVec)` function. For a given parameter type, it fills `parVec` with all parameter values of this type stored in the object.

Note that, as a derivative of the `GOptimizableEntity` class, `GParameterSet`s can also trigger adaption of parameters (compare section 12.1.4).

Several of the following chapters will illustrate the usage of the `GParameterSet` class and its derivatives.

12.1.6. Personality Traits

Individuals are generally independent from the optimization algorithm used to modify them. However, optimization algorithms might need to associate further information with an individual. One example is the information about the best parameter set found so far in swarm algorithms. The easiest way is to store this information in the individual itself. The `GPersonalityTraits` class (and the derivatives available for each optimization algorithm) help to achieve this goal.

Note that the user will very rarely have to deal directly with this branch of the Geneva class tree, as it is mainly intended for internal use. When writing information providers[8], though, it may become necessary to access this object type directly. Chapter 25 and section 12.1.8 have further details on information providers.

12.1.7. Optimization Algorithms

The optimization algorithms implemented in the Geneva library (and likely the majority of all optimization algorithms in existence) share a common structure. After some sort of initialization, a main loop will execute a cycle logic, until a halt criterion is reached. After the execution of finalization code, the user can extract the best individual(s) found. Also, all optimization algorithms implemented in Geneva deal with a collection of individuals which, for want of a better word, will be called a *population* here.

One reason for choosing a population over individual candidate solutions is that the Geneva library is targeted at parallel and networked execution. As a consequence, when executed in a suitable environment, evaluation of multiple parameter sets in parallel does not come at additional cost (i.e., execution takes – roughly – equally long)[9].

All of this base functionality is implemented in the `GOptimizationAlgorithmT` class. The actual algorithms then need to overload some of this class'es functions, with `cycleLogic()` being the obvious, most important example. `GOptimizationAlgorithmT` also implementes several common halt criteria, such as a limit to the maximum number of iterations, the maximum number of iterations without improvement (a "stall counter") or the maximum amount of time allowed to elapse during an optimization. Listing 12.1 further illustrates this situation.

Listing 12.1: All optimization algorithms implemented in the Geneva library share a common work flow, which is implemented in the `GOptimizationAlgorithmT` class

```

do {
        cycleLogic();    // Perform the work specific to this algorithm
        informationRetrieval(); // Extraction of information about the optimization
        iteration++;     // Increment the iteration counter
}
while(!halt());          // Terminate optimization when a halt criterion triggers

finalize();      // Perform any necessary finalization work
returnBestIndividual(); // Return the best individual found
```

The algorithms differ slightly in what they may modify, though. Most optimization algorithms in Geneva derive directly or indirectly from `GOptimizationAlgorithmT<`**GParameterSet**`>`, while clas-

[8]Objects that emit information on the progress of the optimization – this is an advanced topic

[9]As an example, Simulated Annealing is usually implemented using a single "child", but has been amended in the Geneva library to act on an entire population of candidate solutions. See chapter 5 for further details.

ses meant to act on optimization algorithms rather than parameter sets (i.e. perform meta-optimization) will derive from `GOptimizationAlgorithmT`<**GOptimizableEntitity**>

All algorithms implemented in Geneva come in three types that derive from a common base class. In the case of Evolutionary Algorithms, `GSerialEA` allows serial execution only. It is derived from `GBaseEA` Then there is a class that performs optimization in parallel, using a configurable or auto-detected number of threads (e.g. `GMultiThreadedEA` and `GMultiThreadedSwarm`, `GMultiThreadedGD` for the other algorithms). Finally, there is a class which interacts with a broker infrastructure (`GBrokerEA` is one example). The broker infrastructure is discussed in detail in chapter 32 and also briefly in section 12.2. Its purpose is to communicate with different types of (likely networked) entities that perform the parallel tasks defined by an optimization algorithm.

Communication with a GPGPU through OpenCL has also been implemented through the broker (but has so far not been released as part of the main library – ask us for details, if you are interested).

As time permits, it is also intended to integrate an MPI[10]-comsumer into the broker infrastructure.

12.1.8. Optimization Monitors

During the course of the optimization cycle it is necessary to emit information. E.g., a user might want to terminate execution early, if either no sufficient progress can be observed or a sufficient quality has been achieved already.

The optimization algorithms implemented in Geneva have predefined, so called “optimization monitors”. At the very least, an optimization monitor should be able to emit the best fitness found in each iteration.

There is a very diverse set of information that can be extracted from the optimization classes, and this information differs from algorithm to algorithm. Hence Geneva needs to enable users to define their own information schemes.

For this purpose, `GOptimizationAlgorithmT` contains an embedded class which is, not surprisingly, called `GOptimizationMonitorT`. It has a predefined API, and its information-providers are called in each iteration. The actual implementations of a given algorithm derive their own classes from `GOptimizationMonitorT` and overload some of this functionality.

Users wishing to emit information in their own, custom format then just need to derive their own monitor from the existing hierarchy and supply it to the optimization algorithm. Note again that this class will be specific to this algorithm and will likely be unusable with the others.

Note that writing a good monitor is an advanced topic. **It is not necessary to provide your own optimization monitor for your first steps with Geneva.**

There is also a facility called Pluggable Optimization Monitor, which allows to quickly enable the extraction of some frequently used information. Geneva comes with a predefined set of information providers.

Further information on optimization monitors can be found in chapter 25.

[10]**M**essage **P**assing **I**nterface

12.2. Communication and Brokerage

Geneva is targeted at particularly complex optimization problems, with long-lasting evaluation functions. This makes it useful to parallelize Geneva mainly on the level of the evaluation of parameter sets[11]. If done right, then an individual's data will mainly consist of the parameter sets, plus some data needed for serialization purposes[12].

The optimization algorithms of the "broker type" (compare section 12.1.7) then hand a smart pointer[13] to the individuals to be evaluated to a broker. The broker accepts simultaneous connections from different providers of work items, through thread-safe queues. Work items are marked with an id so that the object they have originated from can be identified later. Items to be processed are then extracted from the queues in a round-robin fashion and handed to a "consumer" object for further processing.

Different consumer implementations exist. The most important implementation at the time of writing accepts connections from worker nodes, through network connections implemented with the Boost.Asio library. An implementation of a consumer using the Message Passing Interface would be relatively easy, using the abstractions provided by Geneva. Another consumer exists which simply hands the work item to processing threads on the same machine. It is mainly meant for stress-testing of the broker infrastructure.

Processed items are then handed back to the broker as soon as they come in. The broker makes the items available to the optimization algorithms, so they can be re-integrated into their data structures.

The communication- and brokerage-code is available through Geneva's *courtier* library and associated header files.

12.3. Random Number Creation

Many optimization algorithms require large quantities of random numbers to be readily available. However, in a potentially multi-threaded environment, creation of different random-number strains at the call site would require synchronisation of the seeding procedure, so different random number sequences are not (significantly) correlated. Also, with potentially hundreds of thousands of objects requiring random numbers (Geneva has been tested with up to 100000 parameter objects as part of an individual), the memory overhead of multiple generators would be immense.

And optimization algorithms work in cycles, with periods of high activity followed by short idle times. Performance would thus be hampered, if random numbers would be created only at the time they are needed.

Geneva thus centralizes the creation of random numbers inside of each executable, through a "random

[11]Evolutionary Algorithms also perform mutation in parallel in Geneva.

[12]On a side note this means that it is advisable to load external data needed for the fitness calculation either when the fitness function is called or, depending on the amount of data, once for the entire program duration. This can be done e.g. with a singleton. Geneva provides an infrastructure for this purpose.

[13]A `boost::shared_ptr<GParameterSet>`, really

number factory"[14].

Each consumer of random numbers has a random number proxy that has an array of "raw" random numbers (double numbers in the range $[0,1[$). Retrieving a new double random number thus simply requires incrementation of a counter. When the buffer has run empty, it is discarded and a new buffer is retrieved from the factory. The proxy has means to create other types of random numbers from this raw material, such as "integer" numbers or double random numbers with a gaussian distribution. Usage of the random number proxies is transparent to the consumer. To them the proxies look like ordinary random number generators.

The random number factory continuously fills buffers with "raw" random number packages from a number of threads. These threads will block when the buffers have reached their maximum capacity, and will then not consume CPU time. When random number packages are again extracted from the buffers by the random proxies, the threads resume operation, until the buffer is again full.

The random factory and the proxies are implemented in the *libhap* library inside of the Geneva library collection.

[14]In a networked environment there will be more than one Geneva executable running at the same time: one server and a number of workers. Each has its own random number factory.

Chapter 13.

Parameter Types

This chapter introduces the predefined parameter types of the Geneva library. These are the building blocks from which individuals can be assembled. As this discussion can be a bit boring, we suggest that you read section 13.1 now, and look at section 13.5, when you search for a suitable parameter object fitting your optimization problem.

Key points: (1) Parameter types can be categorized into single parameters and collections of parameters of the same type (2) Geneva supports boolean, integer and floating point parameters of different sizes (3) Geneva supports "meta collections", where `GParameterBase` objects are stored in a collection. This allows to create parameter hierarchies

13.1. Overview

Geneva's vanilla parameter types are built around three basic C++ types: `bool`, floating point and integer values. On the floating point side, Geneva's architecture generally allows `double` types. Until the necessity arises, Geneva currently does not support single precision `Ifloat` and `long double` paramters. One reason for this is that `long double` is not universally supported on all platforms (or, to be more exact, not all supported platforms support all required forms of `long double` math functions). `float` on the other hand seems to be too limited for most scientific and engineering use-cases, and most relevant hardware architectures support `double` parameters natively.

Note that GPGPU-hardware will currently show much better performance for `float` types[1]. However, Geneva will never run directly on the GPU (or when it does, it will be so advanced that `double` will be "en par" with `float`), but will rather "talk" to the GPU through a layer such as `OpenCL`. As double-values in the context of optimization will usually be used for their higher precision rather than their higher value range, their values can be easily transformed to `float` when needed.

[1]GPU-hardware with acceptable double precision performance is already available, but still quite expensive.

Integer values are expressed through Boost's `cstdint` framework in Geneva. In C++, the size of the standard integer types (e.g. `int` or `long`) is platform dependent. This makes it difficult to write platform-independent code. In order to solve this dilemma, Boost provides types like `boost::int32_t` (a 32 bit signed integer), `boost::int64_t` (64 bit signed integer) or `boost::uint64_t` (the unsigned version of `boost::int64_t`), which are then mapped to the native types of a given platform (or are left undefined, if they do not exist). Geneva currently supports `boost::int32_t` parameters only, but can be easily extended to cover other signed integer types as well, should the need arise. Unsigned integer types can be emulated by applying a constraint to the parameter type – see section 13.1.2.

13.1.1. Collections and Single Parameters

Parameter classes come in two flavours in Geneva: They can either contain a single value, or a collection of values. Collections can be treated almost the same as a `std::vector<par_type>` holding variables of the desired type.

Where adaptors need to be applied to a variable (like in the case of Evolutionary Strategies), collections apply the same adaptor to the entire collection[2]. This is significant, as adaptors may vary their internal configuration as part of the optimization procedure, and these modifications will then apply to the entire collection, instead of single parameters.

Parameter types holding only a single variable always have their own adaptor, which can individually and independently change its internal setup as necessary.

13.1.2. Constrained Types

It is quite often necessary to apply constraints to parameter types, so that they can only assume values in a given range. Geneva allows to constrain both parameter types based on floating point and integer variables. Note that an unsigned parameter type can be emulated by applying a lower constraint of 0.

The discussion of the `GConstrainedDoubleObject` class in section 13.5 gives an example of how constrained values are achieved in Geneva.

13.1.3. Naming Schemes

The naming of parameter types follows a predefined scheme:

- As is common in Geneva, all class names start with an uppercase *G*.
- If the parameter type is constrained, the next word of the class name is *Constrained*.
- Next comes the underlying base type, such as *Double* (uppercase) or *Int32* (referring to `boost::int32_t`).

[2]. . . with the exception of collections that carry parameter objects instead of individual "base" values. One example is the `GParameterObjectCollection`, discussed in section 13.5.

- If we are dealing with a single parameter value, the last word of the class name is `Object`.
- If we are dealing with a collection of values instead, the last word is `Collection`.
- If we are dealing with a collection of parameter objects, the last word is `ObjectCollection`.

Section 13.5 lists all parameter types that are currently implemented in Geneva. We recommend to read it when you are looking for a specific parameter type. Given the above naming scheme, there is probably no need to read it immediately.

Figure 15.1 on page 146 further illustrates how parameter objects can be integrated into an individual.

13.2. Value Access

Naturally, the most important user-visible operation of parameter types is the ability to access their values. The access is handled differently depending on whether the parameter type encapsulates a single variables (case "*A*") or variable collections (case "*B*").

13.2.1. Access to Individual Parameters

Retrieval of values in case *A* is possible through the `T GParameterT::value() const` function, where `T` is a template parameter. `GParameterT` is the base class of all individual parameter types. Reading access is also possible though the `operator()`. Note that it is not possible to directly modify the value through these functions.

However, the `void GParameterT::setValue(const T&)` function *does* allow modification of a parameter's value. `operator=(const T&)` provides a convenient alternative to `setValue()`.

Listing 13.1 illustrates the procedure on the example of the `GDoubleObject` parameter, which encapsulates a single, unconstrained `double` variable.

Listing 13.1: Access to the values of individual parameters

```
// [...]
GDoubleObject d;

d.setValue(1.234);
assert(d.value() == 1.234);
assert(d() == 1.234);

d = 2.468;
assert(d.value() == 2.468);
assert(d() == 2.468);

// [...]
```

13.2.2. Access to Values in Parameter-Collections

Throughout the Geneva library, collections of variables and objects use the `std::vector<>` interface. Still, two different situations need to be distinguished, as in Geneva parameter collections may either hold parameter objects or individual POD[3] values (such as `double`, `bool`, `...`).

Collections of POD Values

For all practical purposes, access to POD values in a Geneva collection is not different from access in an ordinary `std::vector<>`. Listing 13.2 illustrates this on the example of a `GDoubleCollection`, i.e. a collection of unconstrained `double` values.

Listing 13.2: Access to POD values in a collection

```
// [...]
GDoubleCollection dc;

dc.push_back(1.);
dc.push_back(2.);
dc.push_back(3.);

GDoubleCollection::iterator it;
for(it=dc.begin(); it!=dc.end(); ++it) {
  std::cout << *it << std::endl;
}

dc[0] = 4;
std::cout << dc.at(0) << std::endl;

dc.clear();
std::cout << dc.size() << std::endl;
// [...]
```

All properties of POD collections in Geneva are derived from the `GStdSimpleVectorInterface` class, which is itself a wrapper around a `std:vector<T>`[4].

Collections of Parameter Objects

Access to collections of parameter objects differs slightly from POD data, as the value being accessed is a smart pointer to a parameter object. One important consequence is that you can't just ask for the value at a given position. Rather, you need to go through the parameter object's interface functions.

[3]**P**lain **O**ld **D**ata

[4]With this design, we want to avoid direct derivation from `std:vector<T>`, as it does not provide a virtual destructor. Derivation from this class is thus commonly considered bad style and can even lead to errors (such as in cases, where the class hierarchy is accessed through a `vector` base class).

Listing 13.3 illustrates this on the example of a `GDoubleObjectCollection`, i.e. a collection of objects, each encapsulating individual `double` variables.

Listing 13.3: Sample access patterns to parameter objects in a collection

```

// [...]
GDoubleObjectCollection doc;

doc.push_back(boost::shared_ptr<GDoubleObject>(new GDoubleObject(1.)));
doc.push_back(boost::shared_ptr<GDoubleObject>(new GDoubleObject(2.)));
doc.push_back(boost::shared_ptr<GDoubleObject>(new GDoubleObject(3.)));

GDoubleObjectCollection::iterator it;
for(it=doc.begin(); it!=doc.end(); ++it) {
  // Note: (*it) returns a boost::shared_ptr<GDoubleObject>
  std::cout << (*it)->value() << std::endl;
}

// doc[0] and doc.at(0) return a boost::shared_ptr<GDoubleObject>. Hence we need to
// de-reference this pointer in order to modify or access the object's value.
*(doc[0]) = 4;
std::cout << (doc.at(0))->value() << std::endl;

// boost::shared_ptr<> will take care of the destruction of allocated objects
doc.clear();
std::cout << doc.size() << std::endl;
// [...]
```

A special purpose collection ist the `GParameterObjectCollection` class, which stores the parameter object base class `GParameterBase`. Hence, in order to access the data stored in the collection, you will first need to convert the `GParameterBase` object to the most-derived type. Geneva provides convenience functions for this purpose, particularly the member template `GParameterObjectCollection::at<parameter_type>(const std::size_t&)`, which returns the object at a given position and converts it to the desired target type `parameter_type` on the fly[5].

13.3. Access to Value- and Initialization-Boundaries

It is naturally possible to specify and retrieve boundaries for constrained parameter types. However, random initialization of unconstrained types also requires specification of boundaries, albeit only for the initialization process. Both boundary types also play a role for example in swarm algorithms, where particularly the initialization boundaries are interpreted as the “expected” value range. Some types – particularly boolean parameter types – do not support constraints.

Specification of boundaries usually happens at construction time, retrieval happens either through the

[5]Note that, for this to work, you obviously need to know the target type.

`getLowerBoundary()` / `getUpperBoundary()` functions in the case of constrained types or through `getLowerInitBoundary()` / `getUpperInitBoundary()` for unconstrained types. Section 13.5 has code examples for each parameter type.

13.4. De-activation of Parameters

labelsec:DeactivationOfParameters It is possible to de-active parameters using the call `GParameterBase::setAdaptionsInactive()`, and re-activated using the call `GParameterBase::setAdaptionsActive()`. The current status of a parameter may be obtained using the call `GParameterBase::adaptionsActive()`. All current optimization algorithms will then take care not to modify “inactive” parameters. The one exception are parameter scans, as it is assumed that, when a parameter is specified as a scan-target, its modification is indeed the desired outcome.

See section 15.2.1 for a discussion on how to selectively extract only active, inactive or all parameters.

13.5. Summary of Parameter Types

The following list of parameter types will provide examples for each type. The general access patterns have already been discussed in section 13.2, hence we will not comment the examples further here.

13.5.1. `GDoubleObject`

This parameter type holds a single, unconstrained double value. This might appear heavy-weight compared to collections of double values. However, particularly in Evolutionary Algorithms, there are many situations where it makes sense to apply separate mutations to each parameter. This might for example be the case if the quality surface has a complicated structure in one dimension, but is smooth in another. Listing 13.4 shows typical usage patterns for this parameter type.

Listing 13.4: Typical usage patterns of the GDoubleObject class

```
    //--------------------------------------------------------------
    // Construction
    GDoubleObject o1; // Default construction
    GDoubleObject o2(o1); // Copy construction
    GDoubleObject o3(2.); // Initialization by value
    GDoubleObject o4(0.,2.); // Random initialization in a given range
    // Construction and access frequently happens through smart pointers
    boost::shared_ptr<GDoubleObject> p(new GDoubleObject(0.,2.));

    //--------------------------------------------------------------
```

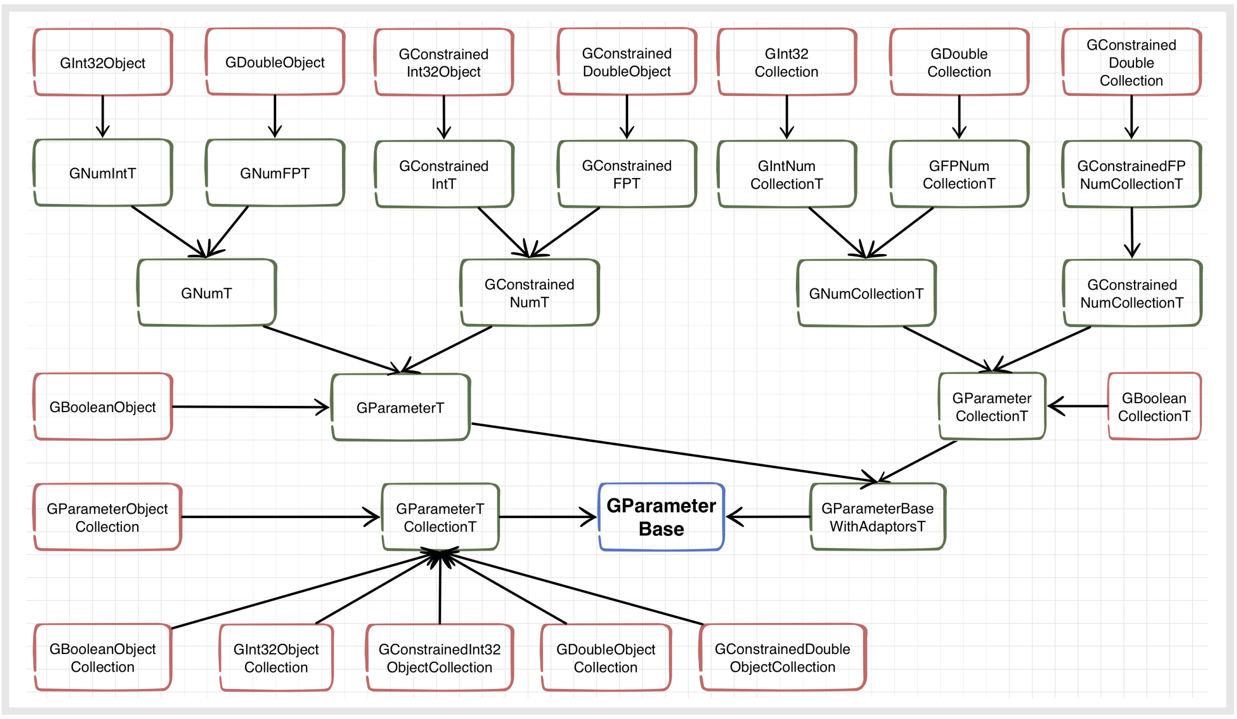


Figure 13.1.: *User-visible parameter-types (marked red) follow an intuitive naming scheme.*

```
    // Assignment, value setting and retrieval
    o1 = 1.; // Assigning and setting a value
    o2.setValue(2.);
    o4 = o1; // Assignment of another object
    std::cout << o4.value() << std::endl; // Value retrieval

    //———————————————————————————————————————————————
    // Boundaries
    // Retrieval of lower init boundary
    std::cout << o4.getLowerInitBoundary() << std::endl;
    // Retrieval of upper init boundary
    std::cout << o4.getUpperInitBoundary() << std::endl;

    //———————————————————————————————————————————————
    // Assignment of an adaptor (needed for Evolution Strategies)
    double sigma = 0.1; // "step width" of gauss mutation
    double sigmaSigma = 0.8; // adaption of sigma
    double minSigma = 0., maxSigma = 0.5; // allowed value range of sigma
    // 5% probability for the adaption of this object when adaptor is called
    double adProb = 0.05;
    boost::shared_ptr<GDoubleGaussAdaptor> gdga_ptr(
      new GDoubleGaussAdaptor(
        sigma, sigmaSigma, minSigma, maxSigma
        )
    );
```

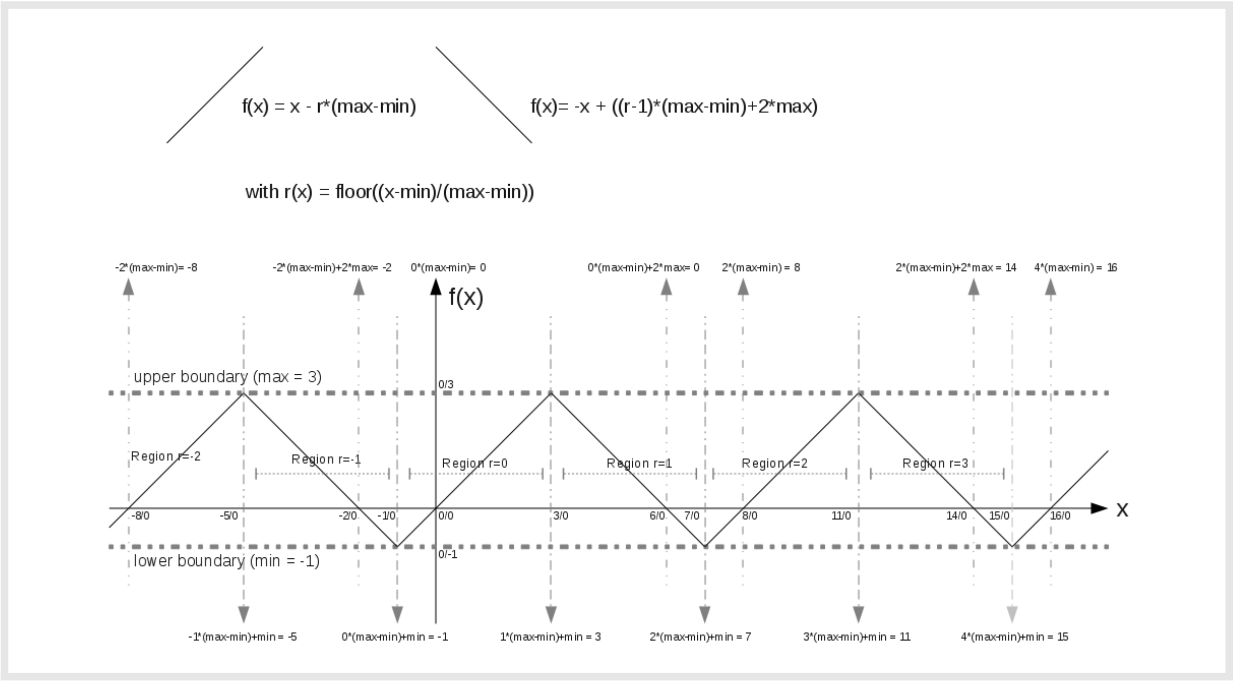


Figure 13.2.: *Constraints of floating point values are modelled as a mapping from an internal to a user-visible value. This allows to apply modifications of the core value to an unconstrained range, while presenting a constrained value to the user.*

```
        gdga_ptr->setAdaptionProbability(adProb);
        p->addAdaptor(gdga_ptr);
        //-----------------------------------------------------------
```

13.5.2. GConstrainedDoubleObject

This is a parameter type holding a single, *constrained* double value. Figure 13.2 shows the method that was used to constrain the value. Geneva uses a mapping from an internal to a user-visible value. The internal value is unconstrained, so that adaptors can be easily applied to the variable, without having to take into account its boundaries. The external value alternates between its two outer boundaries. Figure 13.3 shows the real mapping, as obtained from a `GConstrainedDoubleObject` by setting the internal value and extracting the external value.

One might suspect that applying constraints to an externally visible value while modifying an internal value using the gauss mutation (compare figure 4.2 and section 4.2.2) will severely distort the gaussian, which might degrade the performance particularly of Evolution Strategies. Figure 14.2 shows the internal values of a number of collections of gaussian distributed random numbers with different mean value, as well the externally visible values. It is evident that there is very little distortion close to the outer boundaries of the allowed value range.

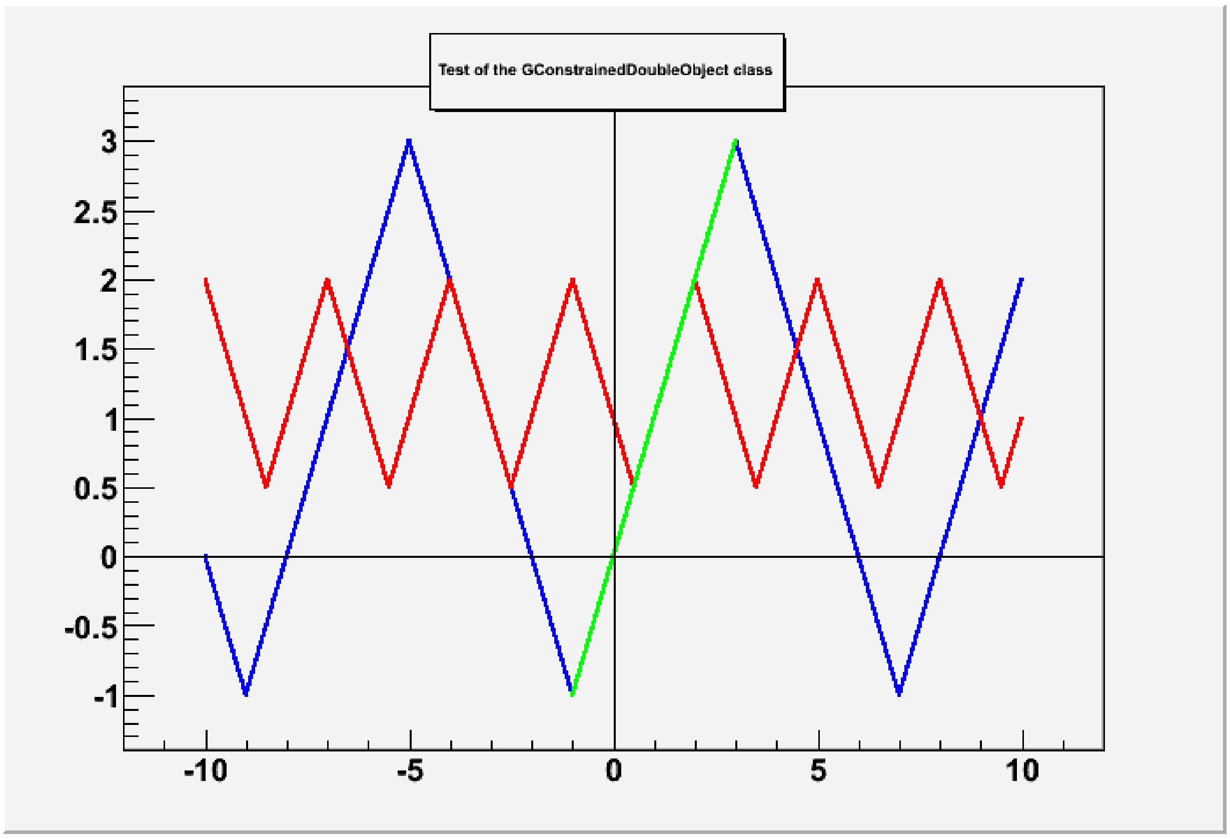


Figure 13.3.: *This picture shows the real mapping, as achieved for different constraints using the method illustrated in figure 13.2.*

Both boundaries of this object are inclusive, i.e., the `GConstrainedDoubleObject` may assume both boundary-values[6]. If you need an "open" upper boundary, i.e., the object may never assume the value of the upper boundary, you need to provide a suitable value. This is relatively easy using the Boost-function `boost::math::float_prior<fp_type>(x)`, where `fp_type` will usually represent a `double` parameter, and `x` is the upper boundary. In order to create a `GConstrainedDoubleObject` with a *closed* lower boundary 0, an *open* upper boundary 2 and a randomly initialized value, one would call the constructor `GConstrainedDoubleObject(0.,float_prior(2.))`[7].

Listing 13.5 shows typical usage patterns for this parameter type. Note that we can add the same adaptor as in listing 13.4. However, we do not show the construction of the adaptor here in order to keep the description short.

[6]Former versions of Geneva used to silently deduct the smallest representable double value from the upper boundary. However, this has led to confusion in a number of places and has made the code overly complex.

[7]C++ can deduct the template type from the function argument. We also assume that the namespace `boost::math` has been made known to the compiler using a `using namespace` command.

Listing 13.5: Typical usage patterns of the GConstrainedDoubleObject class

```
        //------------------------------------------------------------
        // Construction
        GConstrainedDoubleObject o1; // Default construction
        GConstrainedDoubleObject o2(o1); // Copy construction
        GConstrainedDoubleObject o3(2.); // Initialization by value
        GConstrainedDoubleObject o4(0.,2.); // Initialization of value boundaries
        // Initialization with value and value boundaries
        GConstrainedDoubleObject o5(1.,0.,2.);
        // Construction and access frequently happens through smart pointers
        // Note that both boundaries specified in the constructor are inclusive
        boost::shared_ptr<GConstrainedDoubleObject> p(
          new GConstrainedDoubleObject(0.,2.)
        );

        //------------------------------------------------------------
        // Assignment, value setting and retrieval
        o1 = 1.; // Assigning a value
        o2.setValue(1.5);
        o5 = o1; // Assignment of another object
        std::cout << o4.value() << " " << o5.value() << std::endl; // Value retrieval

        //------------------------------------------------------------
        // Boundaries
        // Retrieval of lower value boundary
        std::cout << o4.getLowerBoundary() << std::endl;
        // Retrieval of upper value boundary
        std::cout << o4.getUpperBoundary() << std::endl;

        //------------------------------------------------------------
        // Assignment of an adaptor
        // (setup of adaptor not shown here; identical to GDoubleObject)
        p->addAdaptor(gdga_ptr);
        //------------------------------------------------------------
```

13.5.3. GDoubleObjectCollection

This is a collection of `GDoubleObject` objects. Each of them may carry its own adaptor. Using this collection makes sense if you want to avoid repeated casts.

No adaptor can be assigned to the collection itself, rather you need to add adaptors to the `GDoubleObject` objects before adding them to the collection. The class has a `std::vector<boost::shared_ptr<GDoubleObject> >` interface, so access happens in the same way as is common for the Standard Template Library STL and throughout all of Geneva. Listing 13.6 shows typical usage patterns of this class.

Listing 13.6: Typical usage patterns of the GDoubleObjectCollection class

```
        //------------------------------------------------------------
        // Construction
        GDoubleObjectCollection c1; // Default constructor
        GDoubleObjectCollection c2(c1); // Copy construction
        // Copy construction inside of smart pointer
        // Note: Copy construction will create deep copies
        // of all objects stored in c1
        boost::shared_ptr<GDoubleObjectCollection> p_c3(
          new GDoubleObjectCollection(c1)
        );

        //--------------------------------------------------------------
        // Filling with objects
        for(std::size_t i=0; i<10; i++) {
                // Create a smart pointer wrapping a GDoubleObject
                boost::shared_ptr<GDoubleObject> p(new GDoubleObject());
                // Configure GDoubleObject as required. E.g., add adaptors
                // ...
                // Add to the collection
                c1.push_back(p);
        }

        // Note: No adaptor is added to the collection itself, only
        // to the objects contained in it.

        //--------------------------------------------------------------
        // Assignment through operator= . Note: This will create
        // deep copies of all objects stored in c1
        c2 = c1;
        *p_c3 = c1;
        //--------------------------------------------------------------
        // Access to parameter objects in the collection
        for(std::size_t i=0; i<10; i++) {
                std::cout << p_c3->at(i)->value() << std::endl;
                std::cout << c1[i]->value() << std::endl;
        }

        // Note: The iterator points to a smart pointer, so in order to
        // call a function on the parameter objects we first need to
        // dereference the iterator, then the smart pointer
        GDoubleObjectCollection::iterator it;
        for(it=c1.begin(); it!=c1.end(); ++it) {
                std::cout << (*it)->value() << std::endl;
        }
        //--------------------------------------------------------------
```

13.5.4. GConstrainedDoubleObjectCollection

This is a collection of `GConstrainedDoubleObject` objects. Each of them may carry its own adaptor. No adaptor can be assigned to the collection itself. The class has a `std::vector-`

`<boost::shared_ptr<GConstrainedDoubleObject> >` interface. Listing 13.7 shows typical usage patterns of this class.

Listing 13.7: Typical usage patterns of the GConstrainedDoubleObjectCollection class

```
//----------------------------------------------------------------------
// Construction
GConstrainedDoubleObjectCollection c1; // Default constructor
GConstrainedDoubleObjectCollection c2(c1); // Copy construction
// Copy construction inside of smart pointer. Note: Copy construction will
// create deep copies of all objects stored in c1
boost::shared_ptr<GConstrainedDoubleObjectCollection> p_c3(
  new GConstrainedDoubleObjectCollection(c1)
);

//----------------------------------------------------------------------
// Filling with objects
for(std::size_t i=0; i<10; i++) {
        // Create a smart pointer wrapping a GDoubleObject
        boost::shared_ptr<GDoubleObject> p(new GDoubleObject());
        // Configure GDoubleObject as required. E.g., add adaptors, ...
        // Add to the collection
        c1.push_back(p);
}

// Note: No adaptor is added to the collection itself, only
// to the objects contained in it.
//----------------------------------------------------------------------
// Assignment through operator= . Note: This will create
// deep copies of all objects stored in c1
c2 = c1;
*p_c3 = c1;
//----------------------------------------------------------------------
// Access to parameter objects in the collection
for(std::size_t i=0; i<10; i++) {
        std::cout << p_c3->at(i)->value() << std::endl;
        std::cout << c1[i]->value() << std::endl;
}

// Note: The iterator points to a smart pointer, so in order to
// call a function on the parameter objects we first need to
// dereference the iterator, then the smart pointer
GConstrainedDoubleObjectCollection::iterator it;
for(it=c1.begin(); it!=c1.end(); ++it) {
        std::cout << (*it)->value() << std::endl;
}
//----------------------------------------------------------------------
```

13.5.5. GDoubleCollection

This is a collection of unconstrained double values (i.e. C++ `double` types). A single adaptor is assigned to the collection and applied to all values contained in it. With few restrictions, the collection has a `std::vector<double>` interface. Listing 13.8 shows typical usage patterns.

Listing 13.8: Typical usage patterns of the GDoubleCollection class

```
        //---------------------------------------------------------------
        // Construction
        GDoubleCollection c1; // Default construction
        GDoubleCollection c2(c1); // Copy construction
        // Copy construction inside of smart pointer
        boost::shared_ptr<GDoubleCollection> p_c3(new GDoubleCollection(c1));
        // 100 double values, randomly initialized in the range [-3.,3[
        GDoubleCollection c4(100, -3., 3.);

        //---------------------------------------------------------------
        // Filling with objects
        for(double d=0.; d<100.; d+=1.) {
                c1.push_back(d);
        }
        //---------------------------------------------------------------
        // Adding an adaptor
        double sigma = 0.1; // "step width" of gauss mutation
        double sigmaSigma = 0.8; // adaption of sigma
        double minSigma = 0., maxSigma = 0.5; // allowed value range of sigma
        // 5% probability for the adaption of this object when adaptor is called
        double adProb = 0.05;
        boost::shared_ptr<GDoubleGaussAdaptor> gdga_ptr(
            new GDoubleGaussAdaptor(sigma, sigmaSigma, minSigma, maxSigma)
    );
        gdga_ptr->setAdaptionProbability(adProb);
        c1.addAdaptor(gdga_ptr);

        //---------------------------------------------------------------
        // Assignment through operator= . Note: This will create deep copies
        c2=c1;
        *p_c3 = c1;

        //---------------------------------------------------------------
        // Access to parameter objects in the collection
        for(std::size_t i=0; i<c1.size(); i++) {
                std::cout << c1[i] << std::endl;
                std::cout << c1.at(i) << std::endl;
        }
        GDoubleCollection::iterator it;
        for(it=c1.begin(); it!=c1.end(); ++it) {
                std::cout << *it << std::endl;
        }
        //---------------------------------------------------------------
```

13.5.6. GConstrainedDoubleCollection

This is a collection of constrained double values. All values have the same constraint, and a single adaptor is assigned to all of them. Construction is only possible with constraints[8]. Both boundaries supplioed to this class are closed, i.e., all variables stored in this class may assume the value of both boundaries. See section 13.5.2 (`GConstrainedDoubleObject`) for a discussion on how to achieve an open upper boundary, that will not be reached by the variable values.

Listing 13.9: Typical usage patterns of the GConstrainedDoubleCollection class

```
//———————————————————————————————————
// Construction
// Initialization with 100 variables and constraint [-10, 10[
GConstrainedDoubleCollection c1(100, -10, 200.);
GConstrainedDoubleCollection c2(c1); // Copy construction

// Note -- we do not currently fill in additional data items. This
// class is not yet at its final stage.

//———————————————————————————————————
// Adding an adaptor
double sigma = 0.1; // "step width" of gauss mutation
double sigmaSigma = 0.8; // adaption of sigma
double minSigma = 0., maxSigma = 0.5; // allowed value range of sigma
// 5% probability for the adaption of this object when adaptor is called
double adProb = 0.05;
boost::shared_ptr<GDoubleGaussAdaptor> gdga_ptr(
   new GDoubleGaussAdaptor(sigma, sigmaSigma, minSigma, maxSigma)
);
gdga_ptr->setAdaptionProbability(adProb);
c1.addAdaptor(gdga_ptr);

//———————————————————————————————————
// Assignment through operator= . Note: This will also create
// deep copies of the adaptor
c2 = c1;

//———————————————————————————————————
// Access to parameter objects in the collection
// Note: We currently recommend not to use the subscript and at()
// operators or iterators
for(std::size_t i=0; i<c1.size(); i++) {
        c1.setValue(i, double(i));
        std::cout << c1.value(i) << std::endl;
}
```

[8]This class is not at its final stage, as it still allows modifications of its underlying data set that are not intended this way. As an example, access to the individual parameter items should only happen through the `value()` and `setValue()` functions, as a transformation takes place from an internal value to an externally visible value. Access through the subscript- or (at())-operators wil give you the "raw" internal values only. Changes to the architecture of this class may occur in the future. We recommend to limit usage of this class to the patterns described in listing 13.9 below.

```
        //------------------------------------------------------------
```

13.5.7. GInt32Object

This parameter object holds a single integer value of type `boost::int32_t`. Listing 13.10 shows typical usage patterns of this class.

Listing 13.10: Typical usage patterns of the GInt32Object class

```
        //------------------------------------------------------------
        // Construction
        GInt32Object o1; // Default construction
        GInt32Object o2(o1); // Copy construction
        GInt32Object o3(2); // Initialization by value
        GInt32Object o4(0,2); // Random initialization in a given range
        // Construction and access frequently happens through smart pointers
        boost::shared_ptr<GInt32Object> p_o5(new GInt32Object(0,2));

        //------------------------------------------------------------
        // Assignment, value setting and retrieval
        o1 = 1; // Assigning and setting a value
        o2.setValue(2);
        o4 = o1; // Assignment of another object
        std::cout << o4.value() << std::endl; // Value retrieval

        //------------------------------------------------------------
        // Boundaries
        // Retrieval of lower init boundary
        std::cout << o4.getLowerInitBoundary() << std::endl;
        // Retrieval of upper init boundary
        std::cout << o4.getUpperInitBoundary() << std::endl;

        //------------------------------------------------------------
        // Assignment of an adaptor
        boost::shared_ptr<GInt32FlipAdaptor> ifa_ptr(new GInt32FlipAdaptor());
        ifa_ptr->setAdaptionProbability(0.05); // 5% probability
        p_o5->addAdaptor(ifa_ptr);
        //------------------------------------------------------------
```

13.5.8. GConstrainedInt32Object

A parameter object holding a single `boost::int32_t` value, which has been assigned a lower and upper boundary. Listing 13.11 shows typical usage patterns of this class.

Listing 13.11: Typical usage patterns of the GConstrainedInt32Object class

```
        //------------------------------------------------------------
        // Construction
        GConstrainedInt32Object o1; // Default construction
        GConstrainedInt32Object o2(o1); // Copy construction
        GConstrainedInt32Object o3(2); // Initialization by value
        // Initialization of allowed initialization range
        GConstrainedInt32Object o4(0,10);
        // Initialization with value and allowed initialization range
        GConstrainedInt32Object o5(1,0,10);
        // Construction and access frequently happens through smart pointers
        boost::shared_ptr<GConstrainedInt32Object>
                p_o6(new GConstrainedInt32Object(0,2));

        //-----------------------------------------------------------
        // Assignment, value setting and retrieval
        o1 = 1; // Assigning and setting a value
        o2.setValue(2);
        o4 = o1; // Assignment of another object
        std::cout << o4.value() << std::endl; // Value retrieval

        //-----------------------------------------------------------
        // Boundaries
        // Retrieval of lower init boundary
        std::cout << o4.getLowerBoundary() << std::endl;
        // Retrieval of upper init boundary
        std::cout << o4.getUpperBoundary() << std::endl;

        //-----------------------------------------------------------
        // Assignment of an adaptor
        boost::shared_ptr<GInt32FlipAdaptor> ifa_ptr(new GInt32FlipAdaptor());
        ifa_ptr->setAdaptionProbability(0.05); // 5% probability
        p_o6->addAdaptor(ifa_ptr);
        //-----------------------------------------------------------
```

13.5.9. GInt32ObjectCollection

A collection of `GInt32Object` objects, each with its own adaptor. No adaptor can be assigned to the collection itself. Listing 13.12 shows typical usage patterns of this class.

Listing 13.12: Typical usage patterns of the GInt32ObjectCollection class

```
        //-----------------------------------------------------------
        // Construction
        GInt32ObjectCollection c1; // Default constructor
        GInt32ObjectCollection c2(c1); // Copy construction
        // Copy construction inside of smart pointer
        boost::shared_ptr<GInt32ObjectCollection> p_c3(
          new GInt32ObjectCollection(c1)
        );
        // Note: Copy construction will create deep copies
        // of all objects stored in c1

        //----------------------------------------------------------------
        // Filling with objects
        for(std::size_t i=0; i<10; i++) {
                // Create a smart pointer wrapping a GInt32Object
                boost::shared_ptr<GInt32Object> p(new GInt32Object());
                // Configure GInt32Object as required. E.g., add adaptors
                // ...
                // Add to the collection
                c1.push_back(p);
        }

        // Note: No adaptor is added to the collection itself, only
        // to the objects contained in it.

        //----------------------------------------------------------------
        // Assignment through operator= . Note: This will create
        // deep copies of all objects stored in c1
        c2 = c1;
        *p_c3 = c1;
        //----------------------------------------------------------------
        // Access to parameter objects in the collection
        for(std::size_t i=0; i<10; i++) {
                std::cout << p_c3->at(i)->value() << std::endl;
                std::cout << c1[i]->value() << std::endl;
        }

        // Note: The iterator points to a smart pointer, so in order to
        // call a function on the parameter objects we first need to
        // dereference the iterator, then the smart pointer
        GInt32ObjectCollection::iterator it;
        for(it=c1.begin(); it!=c1.end(); ++it) {
                std::cout << (*it)->value() << std::endl;
        }
        //----------------------------------------------------------------
```

13.5.10. `GConstrainedInt32ObjectCollection`

A collection of `GConstrainedInt32Object` objects, each with its own adaptor. Listing 13.13 shows typical usage patterns of this class.

Listing 13.13: Typical usage patterns of the GConstrainedInt32ObjectCollection class

```
        //----------------------------------------------------------------
        // Construction
        GConstrainedInt32ObjectCollection c1; // Default constructor
        GConstrainedInt32ObjectCollection c2(c1); // Copy construction
        // Copy construction inside of smart pointer
        boost::shared_ptr<GConstrainedInt32ObjectCollection>
            p_c3(new GConstrainedInt32ObjectCollection(c1));
        // Note: Copy construction will create deep copies
        // of all objects stored in c1

        //————————————————————————————————————————
        // Filling with objects
        for(std::size_t i=0; i<10; i++) {
                // Create a smart pointer wrapping a GConstrainedInt32Object
                boost::shared_ptr<GConstrainedInt32Object>
                    p(new GConstrainedInt32Object());
                // Configure GConstrainedInt32Object as required. E.g., add adaptors
                // ...
                // Add to the collection
                c1.push_back(p);
        }

        // Note: No adaptor is added to the collection itself, only
        // to the objects contained in it.
        //————————————————————————————————————————
        // Assignment through operator= . Note: This will create
        // deep copies of all objects stored in c1
        c2 = c1;
        *p_c3 = c1;

        //————————————————————————————————————————
        // Access to parameter objects in the collection
        for(std::size_t i=0; i<10; i++) {
                std::cout << p_c3->at(i)->value() << std::endl;
                std::cout << c1[i]->value() << std::endl;
        }

        // Note: The iterator points to a smart pointer, so in order to
        // call a function on the parameter objects we first need to
        // dereference the iterator, then the smart pointer
        GConstrainedInt32ObjectCollection::iterator it;
        for(it=c1.begin(); it!=c1.end(); ++it) {
                std::cout << (*it)->value() << std::endl;
        }
        //————————————————————————————————————————
```

13.5.11. GInt32Collection

A collection of unconstrained `boost::int32_t` values (i.e. not objects), with a common adaptor, assigned to the entire collection. Listing 13.14 shows typical usage patterns of this class.

Listing 13.14: Typical usage patterns of the GInt32Collection class

```
        //————————————————————————————————————————
        // Construction
        GInt32Collection c1; // Default construction
        GInt32Collection c2(c1); // Copy construction
        // Copy construction inside of smart pointer
        boost::shared_ptr<GInt32Collection> p_c3(new GInt32Collection(c1));
        // 100 boost::int32_t values, with an initialization range of [-3,3]
        GInt32Collection c4(100, -3, 3);

        //------------------------------------------------------------
        // Filling with data
        for(boost::int32_t i=0.; i<100; i++) {
                c1.push_back(i);
        }

        //------------------------------------------------------------
        // Adding an adaptor
        boost::shared_ptr<GInt32FlipAdaptor> ifa_ptr(new GInt32FlipAdaptor());
        ifa_ptr->setAdaptionProbability(0.05); // 5% probability
        c1.addAdaptor(ifa_ptr);

        //------------------------------------------------------------
        // Assignment through operator= . Note: This will also create
        // deep copies of the adaptor
        c2=c1;
        *p_c3 = c1;

        //------------------------------------------------------------
        // Access to parameter objects in the collection
        for(std::size_t i=0; i<c1.size(); i++) {
                std::cout << c1[i] << std::endl;
                std::cout << c1.at(i) << std::endl;
        }
        GInt32Collection::iterator it;
        for(it=c1.begin(); it!=c1.end(); ++it) {
                std::cout << *it << std::endl;
        }
        //------------------------------------------------------------
```

13.5.12. GBooleanObject

A single boolean value, encapsulated in a parameter object so it can be assigned its own adaptor. Listing 13.15 shows typical usage patterns of this class.

Listing 13.15: Typical usage patterns of the GBooleanObject class

```
        //------------------------------------------------------------
        // Construction
        GBooleanObject o1; // Default construction
        GBooleanObject o2(o1); // Copy construction
        GBooleanObject o3(true); // Initialization by value
        // Construction and access frequently happens through smart pointers
        boost::shared_ptr<GBooleanObject> p(new GBooleanObject(true));

        //------------------------------------------------------------
        // Assignment, value setting and retrieval
        o1 = false; // Assigning and setting a value
        o2.setValue(false);
        o3 = o1; // Assignment of another object
        // Value retrieval and value emission
        std::cout << (o3.value()?true:false) << std::endl;

        //----------------------------------------------------------------
        // Assignment of an adaptor
        boost::shared_ptr<GBooleanAdaptor> bad_ptr(new GBooleanAdaptor());
        bad_ptr->setAdaptionProbability(0.05); // 5% adaption probability
        p->addAdaptor(bad_ptr);
        //----------------------------------------------------------------
```

13.5.13. GBooleanObjectCollection

A collection of `GBooleanObject` objects, each with its own adaptor. The collection itself cannot be assigned an adaptor. Listing 13.16 shows typical usage patterns for this class.

Listing 13.16: Typical usage patterns of the GBooleanObjectCollection class

```
        //----------------------------------------------------------------
        // Construction
        GBooleanObjectCollection c1; // Default constructor
        GBooleanObjectCollection c2(c1); // Copy construction
        boost::shared_ptr<GBooleanObjectCollection> p_c3(
                        new GBooleanObjectCollection(c1)
        ); // Copy construction inside of smart pointer
        // Note: Copy construction will create deep copies
        // of all objects stored in c1

        //----------------------------------------------------------------
        // Filling with objects
        for(std::size_t i=0; i<10; i++) {
                // Create a smart pointer wrapping a GBooleanObject
                boost::shared_ptr<GBooleanObject> p(new GBooleanObject());
                // Configure GBooleanObject as required. E.g., add adaptors
                // ...
                // Add to the collection
                c1.push_back(p);
        }

        // Note: No adaptor is added to the collection itself, only
        // to the objects contained in it.

        //----------------------------------------------------------------
        // Assignment through operator= . Note: This will create
        // deep copies of all objects stored in c1
        c2 = c1;
        *p_c3 = c1;
        //------------------------------------------------------------
        // Access to parameter objects in the collection
        for(std::size_t i=0; i<10; i++) {
                std::cout << p_c3->at(i)->value() << std::endl;
                std::cout << c1[i]->value() << std::endl;
        }

        // Note: The iterator points to a smart pointer, so in order to
        // call a function on the parameter objects we first need to
        // dereference the iterator, then the smart pointer
        GBooleanObjectCollection::iterator it;
        for(it=c1.begin(); it!=c1.end(); ++it) {
                std::cout << (*it)->value() << std::endl;
        }
        //------------------------------------------------------------
```

13.5.14. GBooleanCollection

A collection of `bool` values with a common adaptor, which is assigned to the collection object. Listing 13.17 shows typical usage patterns for this class.

Listing 13.17: Typical usage patterns of the GBooleanCollection class

```
        //------------------------------------------------------------
        // Construction
        GBooleanCollection c1; // Default construction
        GBooleanCollection c2(c1); // Copy construction
        GBooleanCollection c3(100); // Initialization with 100 random booleans
        // Initialization with 100 random booleans, of which 25% have a true value
        GBooleanCollection c4(100, 0.25);
        // Copy construction inside of smart pointer
        boost::shared_ptr<GBooleanCollection> p_c5(new GBooleanCollection(c1));

        //------------------------------------------------------------
        // Filling with data
        for(std::size_t i=0; i<100; i++) {
                c1.push_back(i%2==0?true:false);
        }

        //------------------------------------------------------------
        // Adding an adaptor
        boost::shared_ptr<GBooleanAdaptor> bad_ptr(new GBooleanAdaptor());
        bad_ptr->setAdaptionProbability(0.05); // 5% adaption probability
        p_c5->addAdaptor(bad_ptr);

        //------------------------------------------------------------
        // Assignment through operator= . Note: This will also create
        // deep copies of the adaptor
        c2=c1;
        *p_c5 = c1;

        //------------------------------------------------------------
        // Access to parameter objects in the collection
        for(std::size_t i=0; i<c1.size(); i++) {
                std::cout << (c1[i]?"true":"false") << std::endl;
                std::cout << (c1.at(i)?"true":"false") << std::endl;
        }
        GBooleanCollection::iterator it;
        for(it=c1.begin(); it!=c1.end(); ++it) {
                std::cout << (*it?"true":"false") << std::endl;
        }
        //------------------------------------------------------------
```

13.5.15. `GParameterObjectCollection`

Geneva has a special purpose collection capable of holding `GParameterBase` objects. This allows to build parameter hierarchies or "parameter trees". As a `GParameterObjectCollection` is itself derived from the `GParmeterBase` class, it can hold objects of its own type, or generally any other parameter type discussed in the current section 13.5. Listing 13.18 shows typical usage patterns of this class.

Listing 13.18: Typical usage patterns of the GParameterObjectCollection class

```
        //------------------------------------------------------------
        // Construction
        GParameterObjectCollection c1; // Default constructor
        GParameterObjectCollection c2(c1); // Copy construction
        boost::shared_ptr<GParameterObjectCollection> p_c3(
                        new GParameterObjectCollection(c1)
        ); // Copy construction inside of smart pointer
        // Note: Copy construction will create deep copies
        // of all objects stored in c1

        //------------------------------------------------------------
        // Filling with objects. Note that they may have
        // different types, but must all be derived from
        // GParameterBase

        // Create a smart pointer wrapping a GDoubleObject
        boost::shared_ptr<GDoubleObject> p_d(new GDoubleObject());
        // Configure GDoubleObject as required. E.g., add adaptors
        // ...
        // Add to the collection
        c1.push_back(p_d);

        // Create a smart pointer wrapping a GInt32Object
        boost::shared_ptr<GInt32Object> p_i(new GInt32Object());
        // Configure GInt32Object as required. E.g., add adaptors
        // ...
        // Add to the collection
        c1.push_back(p_i);

        // Create another GParameterObjectCollection object.
        // As it is derived from GParameterBase, we can store it
        // in GParameterObjectCollection objects and create
        // tree-like structures in this way
        boost::shared_ptr<GParameterObjectCollection> p_child(
                        new GParameterObjectCollection()
        );
        c1.push_back(p_child);

        // Note: No adaptor is added to the collection itself, only
        // to the objects contained in it (if they support this).

        //------------------------------------------------------------
        // Assignment through operator= . Note: This will create
        // deep copies of all objects stored in c1
        c2 = c1;
        *p_c3 = c1;

        //------------------------------------------------------------
        // Access to parameter objects in the collection

        // Direct conversion, if we know the target type
        boost::shared_ptr<GDoubleObject> p_d2 = c1.at<GDoubleObject>(0);

        // Conversion iterator -- will return all GDoubleObject items
        // stored on this level. Note that the conversion iterator will
        // *not* recurse into p_child .
        GParameterObjectCollection::conversion_iterator<GDoubleObject>
           it_conv(c1.end());

        for(it_conv=c1.begin(); it_conv!=c1.end(); ++it_conv) {
                boost::shared_ptr<GDoubleObject> p_conv = *it_conv;
                std::cout << p_conv->value() << std::endl;
        }
        //------------------------------------------------------------
```

Chapter 14.

Adaptors

Adaptors determine how parameter objects are *mutated* in the context of Evolutionary Algorithms and Geneva's Simulated Annealing implementation.

Key points: (1) Adaptors are selected by users and are stored in the parameter objects themselves. (2) The adaption process itself is handled by the Geneva library internally – users do not have to take care that their adaptors are applied, once they have been loaded into the parameter object. (3) Adaptors also do not influence the optimization process with other algorithms. (4) Some configuration parameters apply to all adaptors (5) The most used "general" parameter is the mutation probability. It determines the likelihood for an adaptor to be actually used (6) The mutation probability may itself be subject to mutation, as a high mutation probability usually works best far away from the global optimum, whereas close to the optimum a smaller mutation probability works best

This section gives an overview of the majority of adaptors implemented in Geneva so far, as well as general options applicape to all adaptors. Figure 14.1 shows the class hierarchy used for adaptors.

14.1. General adaptor options

Mutation Probability

With large amounts of free parameters in an optimization problem it can be useful to limit the number of changed parameters per mutation. The function `setAdaptionProbability()` allows to set the (possibly initial) probability. As the name suggests, the allowed value range of the adaption probability is $[0:1]$.

Variation of the Mutation Probability

In the beginning of the optimization process, far away from the global optimum, a high mutation probability (usually 1 is generally desired, whereas close to the optimum, lower mutation probabilities are usually more advantageous.

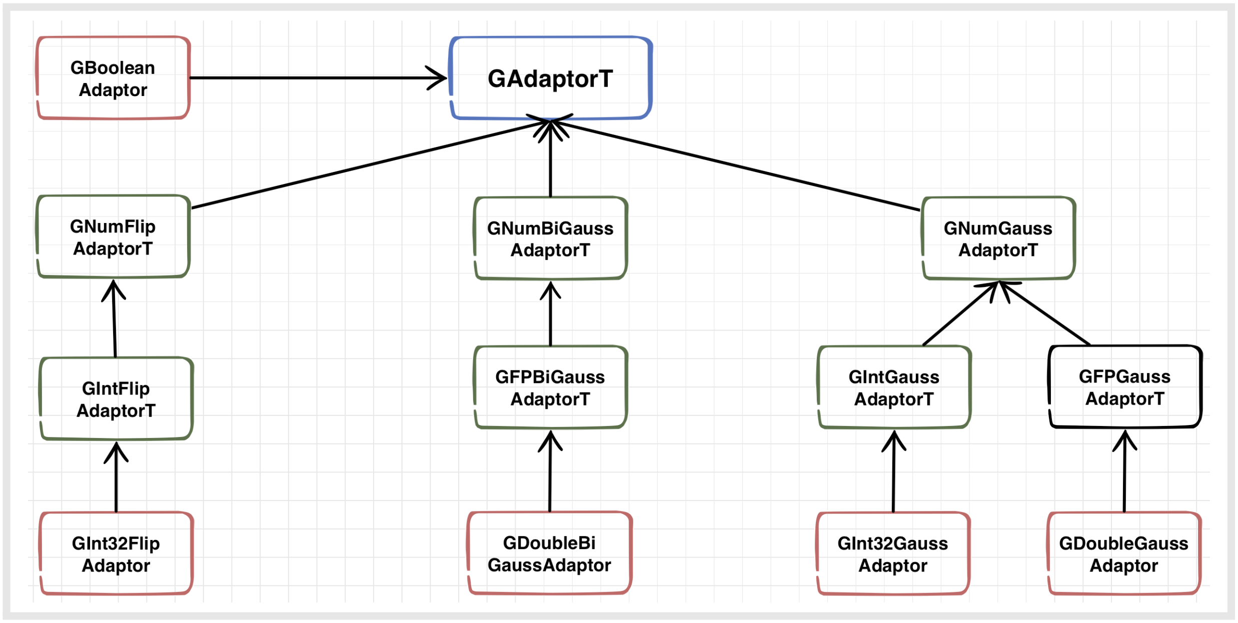


Figure 14.1.: *Adaptors derive from a common base class whose template parameter determines the types they can be applied to.*

For this reason, Geneva has the ability to adapt the mutation probability itself, as part of the optimization process. The speed of this adaption can be controlled with the `setAdaptAdProb(double)` function. The default value for this parameter is 0.1.

A somewhat tricky aspect of a variable mutation probability is the fact that for small numbers of parameters, a low mutation probability may lead to unchanged individuals. For this reason, Genevas individuals contain some logic that enforces at least one mutation per iteration and individual.

There is also a need to enforce lower and upper boundaries for the adaption probability. This may be done with the `setAdProbRange(double min, double max)` function.

Adaption Mode

Rarely used, but sometimes useful is the option to temporarily switch on or off an adaptor with the `setAdaptionMode()` function. When set to `ADAPTALWAYS`, every call to the adaptor will lead to an adaption. Not surprisingly, `ADAPTNEVER` temporarily switches off adaptions with this adaptor. `ADAPTWITHPROB` leads back to the default behaviour, where a parameter is only adapted with a given probability.

Adaption Threshold

Likewise, `setAdaptionThreshold(const boost::uint32_t&)` allows to limit execution of the adaptor to every nth call. I.e., when set to 1, every call to the adaptor will lead to an adaption (provided the adaption probability allows it). This is the recommended setting. When set to 2, only ever second call to the adaptor will lead to an adaption. **This option is rarely used, arguably redundant and might be removed from Geneva in the future.**

14.2. GDoubleGaussAdaptor

In the mutations implemented in the `GDoubleGaussAdaptor`, random numbers with a gaussian distribution are added to double parameters. Specific parameters of this adaptor type include the initial step width (the σ of the gaussian), as well as the minimum and maximum allowed values of σ.

σ may only assume values in the range $[0:1]$. It may thus be interpreted as a percentage of the allowed or expected value range of a floating point parameter. As an example, given an allowed value range of a parameter type of $[-10:10]$ and a σ of 0.1 (a typical value), the "step width" of gauss mutation would be $0.1*20=2$.

σ is itself subject to mutation. A configuration parameter which is currently called *sigmaSigma* influences the speed of adaption of σ. The adaption of σ as part of the optimization process allows the algorithm to adapt to varying geometries of the quality surface (e.g. "flat" and "mountaineous" regions).

"Gauss mutation" is described in detail in section 4.2.2. A gaussian distribution of random numbers, as might be used in this adaptor type, is shown in figure 4.2. Listing 14.1 shows typical usage patterns of this class.

Listing 14.1: Typical usage patterns of the GDoubleGaussAdaptor class

```
//-------------------------------------------------------------
// Construction
GDoubleGaussAdaptor a1; // Default construction
GDoubleGaussAdaptor a2(a1); // Copy construction

double adProb=0.05; // A 5% probability that adaption actually takes place
GDoubleGaussAdaptor a3(0.05); // Construction with adaption probability

double sigma=0.2, sigmaSigma=0.8, minSigma=0., maxSigma=2.;
//Construction with specific mutation parameters
GDoubleGaussAdaptor a4(sigma, sigmaSigma, minSigma, maxSigma);
//Construction with specific mutation parameters
GDoubleGaussAdaptor a5(sigma, sigmaSigma, minSigma, maxSigma, adProb);
// Construction inside of a smart pointer
boost::shared_ptr<GDoubleGaussAdaptor> p_a6(new GDoubleGaussAdaptor());
//-------------------------------------------------------------
// Assignment
a3 = a1;
*p_a6 = a1;
//-------------------------------------------------------------
// Setting and retrieval of specific configuration parameters
a1.setSigma(sigma);
double sigma2 = a1.getSigma();
```

```

        a1.setSigmaRange(minSigma, maxSigma);
        boost::tuple<double,double> t = a1.getSigmaRange();
        std::cout << t.get<0>() << " " << t.get<1>() << std::endl;

        a1.setSigmaAdaptionRate(sigmaSigma);
        double adaptionRate = a1.getSigmaAdaptionRate();

        a1.setAll(sigma, sigmaSigma, minSigma, maxSigma);
        //---------------------------------------------------------------
        // Parameters common to all adaptors
        a1.setAdaptionProbability(adProb);
        double adProb2 = a1.getAdaptionProbability();

        boost::uint32_t adaptionThreshold = 1;
        a1.setAdaptionThreshold(adaptionThreshold);
        adaptionThreshold = a1.getAdaptionThreshold();

        // Always adapt, irrespective of probability
        a1.setAdaptionMode(ADAPTALWAYS);
        // Adapt according to the adaption probability
        a2.setAdaptionMode(ADAPTWITHPROB);
        // Temporarily disable the adaptor
        a3.setAdaptionMode(ADAPTNEVER);
        boost::logic::tribool adaptionMode = a1.getAdaptionMode();

        //---------------------------------------------------------------
```

14.3. GDoubleBiGaussAdaptor

The `GDoubleBiGaussAdaptor` adaptor replaces the single Gauss of `GDoubleGauss-Adaptor` with two gaussians with distance "*delta*" from each other, centred around 0. The gaussians may optionally have different σ values. The σ values and the distance may be varied as part of the optimization process to adapt to varying geometries of the quality surface.

The rationale behind this adaptor is that we want to primarily search in regions that have not been explored yet, but which are still close to the best known solution(s). In comparison, a single gaussian, as implemented in the `GDoubleGaussAdaptor` favours the region closest to the best known solutions.

"Bi-Gauss mutation" is described in detail in section 4.2.2. A bi-gaussian distribution of random numbers, as might be used in this adaptor type, is shown in figure 4.5. The adaptor uses a function from the `GRandom` family of classes (compare section 31.2.1).

Listing 14.2: Typical usage patterns of the GDoubleBiGaussAdaptor class

```
        //---------------------------------------------------------------
        // Construction
        GDoubleBiGaussAdaptor a1; // Default construction
        GDoubleBiGaussAdaptor a2(a1); // Copy construction

        double adProb=0.05; // A 5% probability that adaption actually takes place
        GDoubleBiGaussAdaptor a3(0.05); // Construction with adaption probability

        // Construction inside of a smart pointer
        boost::shared_ptr<GDoubleBiGaussAdaptor> p_a4(new GDoubleBiGaussAdaptor());

        //----------------------------------------------------------------
        // Assignment
        a3 = a1;
        *p_a4 = a1;

        //----------------------------------------------------------------
        // Setting and retrieval of specific configuration parameters

        // sigma1 and sigma2 may differ
        a1.setUseSymmetricSigmas(false);
        bool useSymmetricSigmas = a1.getUseSymmetricSigmas();

        // Set/get sigma1 and sigma2
        a1.setSigma1(0.1);
        a1.setSigma2(0.2);
        double sigma1 = a1.getSigma1(), sigma2 = a1.getSigma2();

        // Set/get the allowed value range of sigma1 and sigma2
        a1.setSigma1Range(0.001,2.);
        a1.setSigma2Range(0.001,2.);
        boost::tuple<double,double> sigma1Range = a1.getSigma1Range();
        boost::tuple<double,double> sigma2Range = a1.getSigma2Range();

        // Set/get the adaption rate of sigma1 and sigma2
        a1.setSigma1AdaptionRate(0.8);
        a1.setSigma2AdaptionRate(0.8);
        double sigma1AdaptionRate = a1.getSigma1AdaptionRate();
        double sigma2AdaptionRate = a1.getSigma2AdaptionRate();

        // Set all sigma1 and sigma2 parameters at once. Note: We use
        // the lower/upper boundaries extracted above.
        a1.setAllSigma1(
                        sigma1
                        , sigma1AdaptionRate
                        , sigma1Range.get<0>()
                        , sigma1Range.get<1>()
        );
        a1.setAllSigma2(
                        sigma2
                        , sigma2AdaptionRate
                        , sigma2Range.get<0>()
                        , sigma2Range.get<1>()
        );

        // Set the initial distance between both peaks
        // and retieve the current value
        a1.setDelta(1.5);
        double delta = a1.getDelta();

        // Set/get the lower and upper boundaries of delta
        a1.setDeltaRange(0.,5.);
        boost::tuple<double,double> deltaRange = a1.getDeltaRange();

        // Set/get the adaption rate of delta
        a1.setDeltaAdaptionRate(0.8);
        double deltaAdaptionRate = a1.getDeltaAdaptionRate();

        // Set all delta parameters at once. Note: We use the
        // lower and upper boundaries that were extracted above
        a1.setAllDelta(
                delta
                , deltaAdaptionRate
                , deltaRange.get<0>()
                , deltaRange.get<1>()
        );

        //--------------------------------------------------------------
        // Parameters common to all adaptors
        a1.setAdaptionProbability(adProb);
        double adProb2 = a1.getAdaptionProbability();

        boost::uint32_t adaptionThreshold = 1;
        a1.setAdaptionThreshold(adaptionThreshold);
        adaptionThreshold = a1.getAdaptionThreshold();

        // Always adapt, irrespective of probability
        a1.setAdaptionMode(ADAPTALWAYS);
        // Adapt according to the adaption probability
        a2.setAdaptionMode(ADAPTWITHPROB);
        // Temporarily disable the adaptor
        a3.setAdaptionMode(ADAPTNEVER);
        boost::logic::tribool adaptionMode = a1.getAdaptionMode();

        //--------------------------------------------------------------
```

14.4. `GInt32GaussAdaptor`

The `GInt32GaussAdaptor` mimics the behaviour of the `GDoubleGaussAdaptor` for signed 32 bit integer types. Just like its role model, it adds gaussian distributed integer random numbers to the adaptor's argument. This means in particular that small changes are favored over large changes. However, the adaptor does need to take into account that at least a change of 1 is needed in order to have an effect. Listing 14.3 shows typical usage patterns.

Listing 14.3: Typical usage patterns of the GInt32GaussAdaptor class

```
//-------------------------------------------------------------
// Construction
GInt32GaussAdaptor a1; // Default construction
GInt32GaussAdaptor a2(a1); // Copy construction

// A 5% probability that adaption actually takes place
double adProb=0.05;
// Construction with adaption probability
GInt32GaussAdaptor a3(0.05);

//Construction with specific mutation parameters
double sigma=0.1, sigmaSigma=0.8, minSigma=0., maxSigma=20.;
GInt32GaussAdaptor a4(sigma, sigmaSigma, minSigma, maxSigma);
GInt32GaussAdaptor a5(sigma, sigmaSigma, minSigma, maxSigma, adProb);

// Construction inside of a smart pointer
boost::shared_ptr<GInt32GaussAdaptor> p_a6(new GInt32GaussAdaptor());

//-------------------------------------------------------------
// Assignment
a3 = a1;
*p_a6 = a1;

//-------------------------------------------------------------
// Setting and retrieval of specific configuration parameters
a1.setSigma(sigma);
double sigma2 = a1.getSigma();

a1.setSigmaRange(minSigma, maxSigma);
boost::tuple<double,double> t = a1.getSigmaRange();
std::cout << t.get<0>() << " " << t.get<1>() << std::endl;

a1.setSigmaAdaptionRate(sigmaSigma);
double adaptionRate = a1.getSigmaAdaptionRate();

a1.setAll(sigma, sigmaSigma, minSigma, maxSigma);

//-------------------------------------------------------------
// Parameters common to all adaptors
a1.setAdaptionProbability(adProb);
double adProb2 = a1.getAdaptionProbability();

boost::uint32_t adaptionThreshold = 1;
a1.setAdaptionThreshold(adaptionThreshold);
adaptionThreshold = a1.getAdaptionThreshold();

// Always adapt, irrespective of probability
a1.setAdaptionMode(ADAPTALWAYS);
```

```
            // Adapt according to the adaption probability
            a2.setAdaptionMode(ADAPTWITHPROB);
            // Temporarily disable the adaptor
            a3.setAdaptionMode(ADAPTNEVER);

            boost::logic::tribool adaptionMode = a1.getAdaptionMode();

            //-------------------------------------------------------------
```

14.5. GInt32FlipAdaptor

The `GInt32FlipAdaptor` adaptor flips an unsigned 32 bit integer up or down randomly. As this operation is very simple, there are no specific parameters – only the standard parameters available to all adaptors. Listing 14.4 shows typical usage patterns of this class.

Listing 14.4: Typical usage patterns of the GInt32FlipAdaptor class

```
        //-------------------------------------------------------------
        // Construction
        GInt32FlipAdaptor a1; // Default construction
        GInt32FlipAdaptor a2(a1); // Copy construction

        // A 5% probability that adaption actually takes place
        double adProb=0.05;
        // Construction with adaption probability
        GInt32FlipAdaptor a3(0.05);

        // Construction inside of a smart pointer
        boost::shared_ptr<GInt32FlipAdaptor> p_a4(new GInt32FlipAdaptor());

        //-------------------------------------------------------------
        // Assignment
        a3 = a1;
        *p_a4 = a1;

        //-------------------------------------------------------------
        // Parameters common to all adaptors
        a1.setAdaptionProbability(adProb);
        double adProb2 = a1.getAdaptionProbability();

        boost::uint32_t adaptionThreshold = 1;
        a1.setAdaptionThreshold(adaptionThreshold);
        adaptionThreshold = a1.getAdaptionThreshold();

        // Always adapt, irrespective of probability
        a1.setAdaptionMode(ADAPTALWAYS);
        // Adapt according to the adaption probability
        a2.setAdaptionMode(ADAPTWITHPROB);
        // Temporarily disable the adaptor
        a3.setAdaptionMode(ADAPTNEVER);

        boost::logic::tribool adaptionMode = a1.getAdaptionMode();

        //---------------------------------------------------------------
```

14.6. `GBooleanAdaptor`

The `GBooleanAdaptor` acts almost identical to the `GInt32FlipAdaptor`, but acts on boolean values. Thus, flipping is only possible into one direction. Listing 14.5 shows typical usage patterns of this class.

Listing 14.5: Typical usage patterns of the GBooleanAdaptor class

```
        //---------------------------------------------------------------
        // Construction
        GBooleanAdaptor a1; // Default construction
        GBooleanAdaptor a2(a1); // Copy construction

        // A 5% probability that adaption actually takes place
        double adProb=0.05;
        // Construction with adaption probability
        GBooleanAdaptor a3(0.05);

        // Construction inside of a smart pointer
        boost::shared_ptr<GBooleanAdaptor> p_a4(new GBooleanAdaptor());

        //---------------------------------------------------------------
        // Assignment
        a3 = a1;
        *p_a4 = a1;

        //---------------------------------------------------------------
        // Parameters common to all adaptors
        a1.setAdaptionProbability(adProb);
        double adProb2 = a1.getAdaptionProbability();

        boost::uint32_t adaptionThreshold = 1;
        a1.setAdaptionThreshold(adaptionThreshold);
        adaptionThreshold = a1.getAdaptionThreshold();

        // Always adapt, irrespective of probability
        a1.setAdaptionMode(ADAPTALWAYS);
        // Adapt according to the adaption probability
        a2.setAdaptionMode(ADAPTWITHPROB);
        // Temporarily disable the adaptor
        a3.setAdaptionMode(ADAPTNEVER);

        boost::logic::tribool adaptionMode = a1.getAdaptionMode();

        //———————————————————————————————————————————
```

14.7. Adaptors and Constrained Parameter Types

In the context of adaptors, we'd like to draw your attention to a specialty of the Geneva library. If you look again at figures 13.2 and 13.3, you will see that constraints are implemented by performing a mapping from an internal, unconstrained value to an externally visible, constrained value. With the exception of boolean values, adaptors always act on the internal value of constrained types. As a consequence, they do not need to take into account the constraints of the parameter type.

Figure 14.2 illustrates what happens to a gaussian close to the boundary of a constrained double type. In essence, "overlapping" regions of the gaussian are "reflected" to valid value ranges of the parameter range. Contrary to expectation, the gaussian does not appear to be strongly distorted by this process.

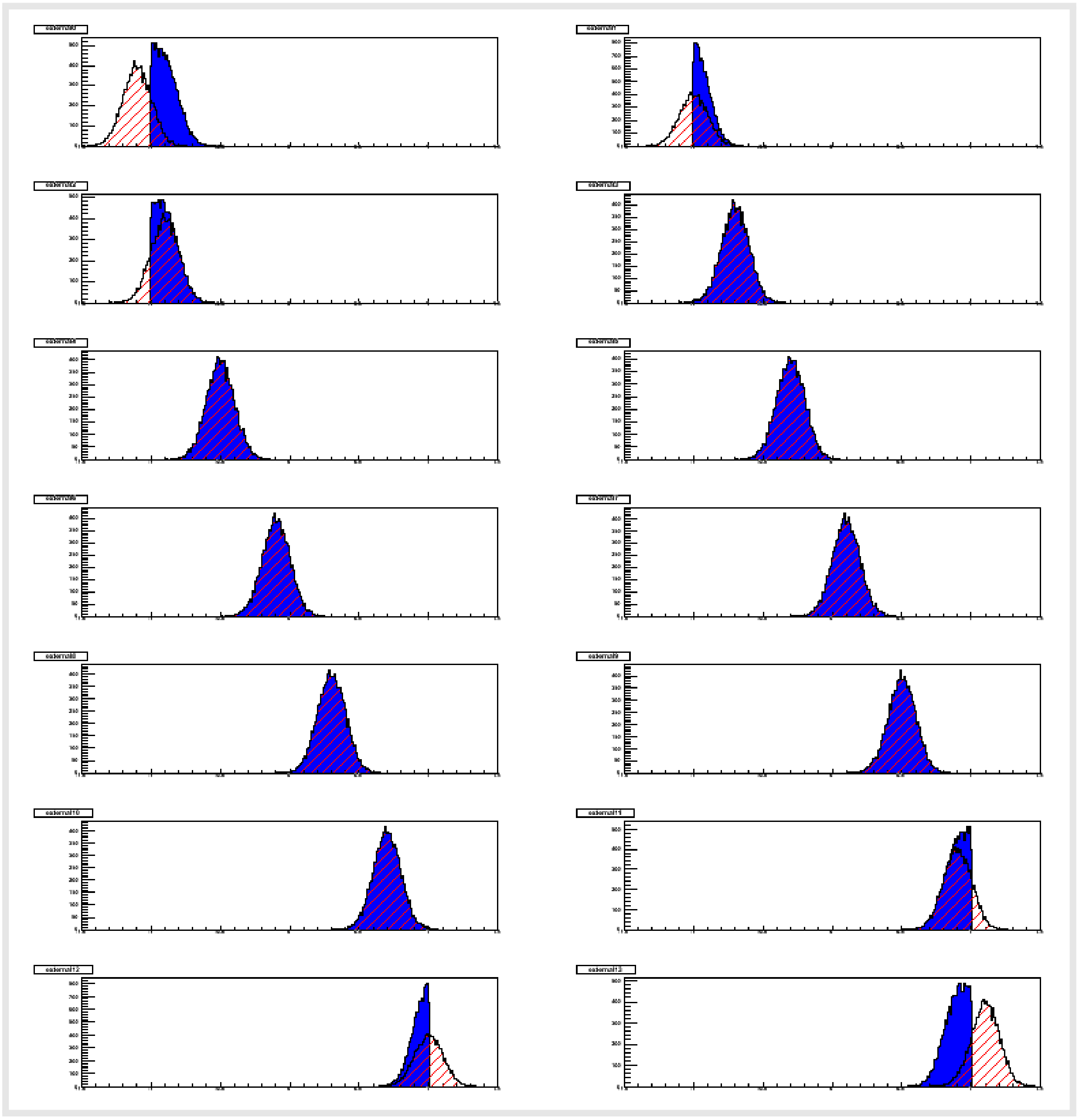


Figure 14.2.: *This figure shows the internal values of a number of collections of gaussian distributed random numbers with different mean value, as well the externally visible values. The blue curve shows the actual distribution, the curve with red stripes shows the "input" gaussian. It is evident that there is very little distortion close to the outer boundaries of the allowed value range.*

Chapter 15.

Individuals and Parameters

This chapter discusses the definition of an optimization problem, through the aggregation of parameter objects (possibly equipped with adaptors) and an evaluation function inside of "individuals"[1].

Key points: (1) In the simplest case, an optimization problem can be defined by a specification of the parameters to be varied, including constraints, and the evaluation function(s) needed to rate a given candidate solution (2) Individuals are always derived from the `GParameterSet` class. (3) Parameter objects can be added directly to a `GParameterSet`, the evaluation function is defined in the derived class provided by the user (4) Individuals can contain entire hierarchies of parameter objects (5) Access to parameter values involves the use of the `streamline()` template, the `conversion_iterator` or direct access of specific parameters (6) It is possible to selectively extract active, inactive or all parameters (7) Multiple evaluation criteria can be defined in the evaluation function (8) In the majority of cases, making Individuals fit for serialization does not require much more than the specification of the parameters to be serialized.

15.1. General Principles

A full definition of an optimization problem is possible by specifying the parameters, their constraints and the evaluation criteria used to assign one or more quality measures to a given parameter collection. Both the parameter definition and the evaluation function are highly problem-specific and need to be provided by the user. This task, however, is greatly facilitated by Geneva, which provides a full framework for this purpose.

[1]As discussed in chapter 12, optimization algorithms and parameter sets form two distinct entities. Optimization algorithms may act on `GParameterSet`-derivatives, which in turn comprise the entire definition of an optimization problem. Note that "parameter set" and `GParameterSet` are used interchangeably in this chapter. In the context of this document, parameter sets are also often called "individuals". Note, though, that a more exact definition of a "Geneva individual" would be "candidate solution, used as the input for Geneva's optimization algorithms". Geneva also allows a form of meta evolution, where optimization algorithms become subject to optimization themselves. So while each parameter set is an individual, not every individual is necessarily a parameter set. For the sake of simplicity, though, the terms "individual" and "parameter set" are used interchangeably in this chapter.

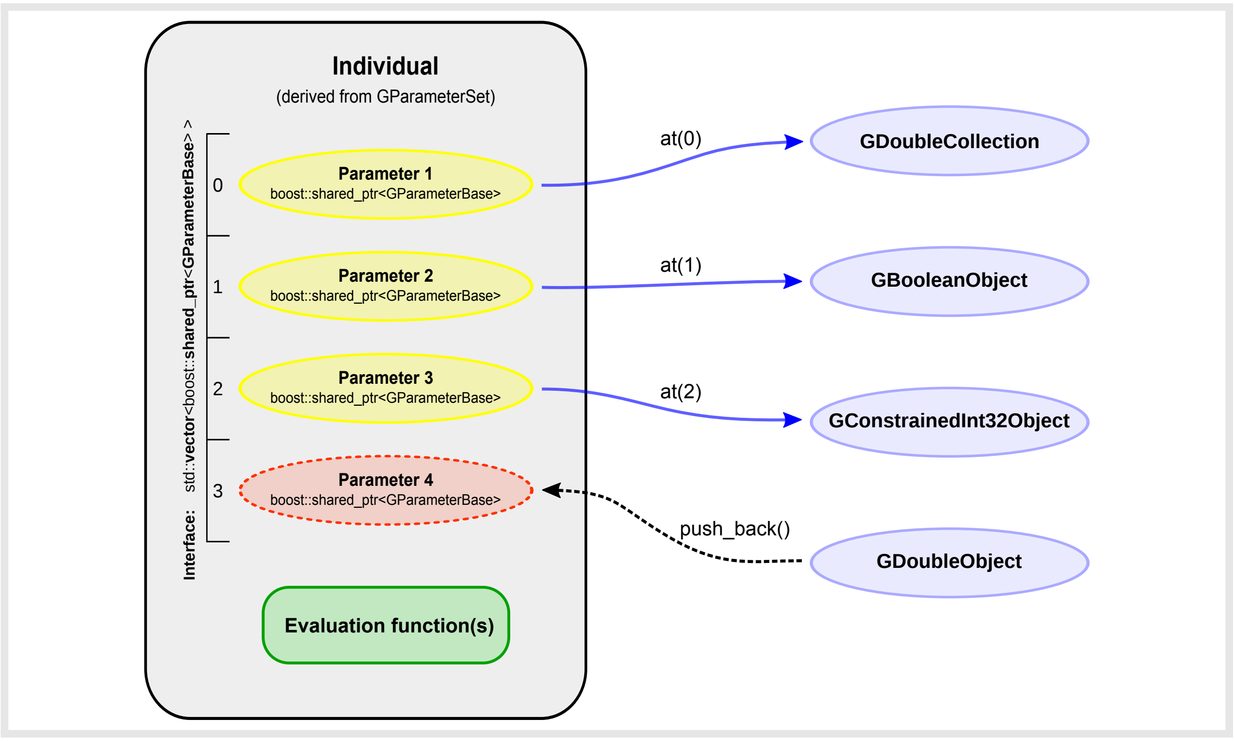

Individual
(derived from GParameterSet)
Interface: std::vector<boost::shared_ptr<GParameterBase> >
0
1
2
3
Parameter 1
boost::shared_ptr<GParameterBase>
Parameter 2
boost::shared_ptr<GParameterBase>
Parameter 3
boost::shared_ptr<GParameterBase>
Parameter 4
boost::shared_ptr<GParameterBase>
Evaluation function(s)
at(0)
at(1)
at(2)
push_back()
GDoubleCollection
GBooleanObject
GConstrainedInt32Object
GDoubleObject

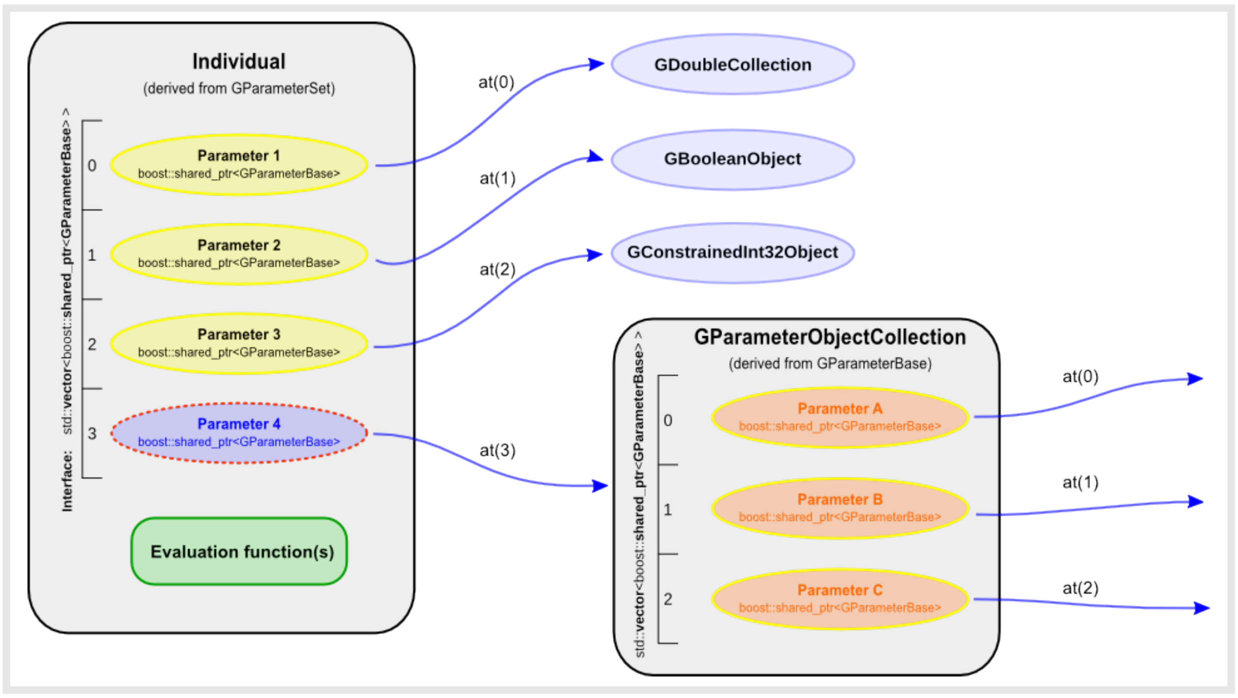
Figure 15.1.: *Geneva's individuals are derived from the GParameterSet class, which features a* `std::vector<>` *interface. The class stores smart pointers to* `GParameter-Base` *objects, so that it becomes possible to store different parameter types in the container.*

In programming terms, individuals are objects that are derived from `GParameterSet`. This class has a `std::vector<boost::shared_ptr<GParameterBase> >` interface. Or, in other words, it stores smart pointers to `GParameterBase` objects, using a `std::vector<>` interface. `GParameterBase` is the base class to a full set of parameter objects, ranging from single `bool` values to collections of constrained `double` values (compare figure 13.1 and the entire chapter 13). As `GParameterSet` only knows about base objects, it becomes possible to assemble an individual from many different parameter types.

To further illustrate this, let us look at an optimization problem that may best be described in terms of integral, boolean and floating point parameters. In our example, the integer variables may only assume a certain value range, while the floating point variables are unconstrained, but share common characteristics (in which case it may make sense to treat them as a collection).

This situation could be modelled by deriving a class from `GParameterSet`, and successively adding a `GDoubleCollection`, `GBooleanObject` and a `GConstrainedInt32-Object` to the object, using the usual `std::vector<>` function `push_back()`[2]. For good

[2]Note in this context that parameter types may carry further functionality, such as the ability to be *mutated* through an adaptor (compare section 12.1.3). If adaptors are required, it is advisable to add them to a parameter object before it



Individual
(derived from GParameterSet)
Interface: std::vector<boost::shared_ptr<GParameterBase> >
0
1
2
3
Parameter 1
boost::shared_ptr<GParameterBase>
Parameter 2
boost::shared_ptr<GParameterBase>
Parameter 3
boost::shared_ptr<GParameterBase>
Parameter 4
boost::shared_ptr<GParameterBase>
Evaluation function(s)
at(0)
at(1)
at(2)
at(3)
GDoubleCollection
GBooleanObject
GConstrainedInt32Object
GParameterObjectCollection
(derived from GParameterBase)
std::vector<boost::shared_ptr<GParameterBase> >
Parameter A
boost::shared_ptr<GParameterBase>
Parameter B
boost::shared_ptr<GParameterBase>
Parameter C
boost::shared_ptr<GParameterBase>
at(0)
at(1)
at(2)

Figure 15.2.: *Geneva's individuals can serve as the root of an entire hierarchy of parameters, using the GParameterObjectCollection class. It can be likened to an individual without attached evaluation function and also features a `std::vector<>` interface*

measure we also add a `GDoubleObject`, which might represent a single double value with a different role in the optimization problem than the doubles stored in the `GDoubleCollection`[3].

Figure 15.1 further illustrates the general architecture of an individual. Figure 15.2 shows another option. With the help of the `GParameterObjectCollection` – a collection of `GParameterBase`-derivatives and itself a derivative of `GParameterBase` – it becomes possible to define entire parameter hierarchies.

Chapters 11 and 26 illustrate the procedure of defining your own individual with concrete examples.

15.2. `fitnessCalculation()`: Evaluating Individuals

Optimization algorithms work by successively evaluating candidate solutions (individuals with different parameter values but identical structure) and deriving[4] new candidate solutions from older ones. An optimization problem can be uniquely defined by the type and constraints of the individuals' parameters and the evaluation function which assigns a value to a given parameter set.

is attached to an individual.

[3]Note again that `GParameterSet` expects these objects to be wrapped into a `boost::shared_ptr<>`.

[4]In this context, the term "deriving" does not refer to the C++ programming term "derivation", but is used in the sense of "using some of the older individuals' characteristics"

As a key duty in defining an optimization problem, users thus need to overload the `double GParameterSet::fitnessCalculation()` function[5]. Through this function, numeric quantities (often called quality or fitness) are assigned to the candidate solution. In the most common case, only a single evaluation criterion exists. However, Geneva also allows to define multiple evaluation criteria, with one master-criterion and multiple sub-criteria. This is needed for multi-criterion optimization (compare e.g. section 2.5).

Getting the evaluation function right can be an iterative procedure. Unless the optimization problem's structure is already clearly defined, the engineer's experience in modelling the problem will play a role. There will be hidden knowledge, maybe not even consciously known, unexpected effects and false conceptions. This implies that multiple optimization runs might be necessary before a suitable result is found. But once it is found, implicit knowledge might have become explicit and misconceptions might have been eradicated.

15.2.1. Accessing Parameters

Before it becomes possible to assign a fitness to a given parameter set, though, its values need to be extracted. Remember that Geneva's individuals store base pointers of parameter objects. Before accessing their values, they need to be converted to the target type. This adds some overhead, but also makes Geneva's individuals quite flexible, as different parameter types can be mixed in an intuitive way[6]. There are three ways of gaining access to the parameter values.

`streamline()`

The `streamline()` function has already been demonstrated in listing 11.7 on page 95.

Listing 15.1: The streamline function

```
double GParaboloidIndividual2D::fitnessCalculation(){
  // [...]
  std::vector<double> parVec; // Will hold the parameters
  this->streamline(parVec, ALLPARAMETERS); // Retrieve the parameters

  // Calculate a fitness based on the parVec vector
  // [...]

  return result;
}
```

`streamline()` is a member template of the `GParameterSet` class. It iterates over all `GParameterBase`-objects stored in it and extracts values of a given type. They are then attached to

[5]Note that it is not strictly necessary to always derive your own class from GParameterSet. Gemfony scientific has code that allows to call an external application for the evaluation step, with an easily customizable protocol for data exchange. Talk to us if you are interested in using this for your optimization problems.

[6]We'd also like to remind you that Geneva aims at optimization problems with particularly long-running evaluation functions, so that the overhead of the type conversion does not play a significant role.

a `std::vector<>` provided as argument. When `streamline()` encounters a `GParameterBase` object that represents a collection, it will recurse into it and extract all parameters of the desired type. The user then receives the parameter *values* in the same order in which the corresponding parameters were registered with the individual.

In listing 15.1, the type is determined automatically from the type stored in the `parVec` vector. Note that a more verbose way of specifying parameters of which type should be extracted would be `streamline<double>(parVec)`.

`streamline()` is the recommended way of accessing parameter values in individuals with different parameter types, when the parameter structure does not need to be preserved.

Note that there is a second version of `streamline`, which accepts a `std::map<std::string, std::vector<target_type> >`. The function returns the names of parameter objects together with their values. As parameter objects may carry multiple values (such as in the case of a `GDoubleCollection`), the value-type of the map needs to be a `std::vector<target_type>`.

As the last, optional parameter to `streamline` it is possible to specify, that only active (switch `ACTIVEONLY`), inactive (switch `INACTIVEONLY`) or all (switch `ALLPARAMETERS`) should be extracted.

conversion_iterator

`streamline()` will recurse into parameter collections (and go deeper, if the parameter collection itself contains parameter collections). In an individual with a hierarchical parameter structure, it is possible to iterate over all parameters of a given type on the same level of the tree with the `conversion_iterator` class template[7]. It will not only find parameters of a given type on this level, but also present them to you readily converted from the `GParameterBase` base class.

Listing 15.2 demonstrates the usage for an individual in which a number of `GConstrainedDoubleObject` objects have been stored. It calculates a parabola from their values.

Listing 15.2: The conversion_iterator class

```
double MyIndividual::fitnessCalculation(){
   double result = 0;

   // Note that we need to pass the end()-iterator to the
   // constructor of the conversion_iterator<> .
   MyIndividual::conversion_iterator<GConstrainedDoubleObject> it(this->end());
   for(it=this->begin(); it!=this->end(); ++it) {
      result += (*it)->value() * (*it)->value();
   }

   return result;
}
```

[7]The `conversion_iterator` is implemented as an embedded class inside of the `GStdPtrVectorInterface` – the class the provides the `std::vector<>` interface to many of Geneva's collection classes

Note that we could have done the same for a `GParameterObjectCollection` stored in the individual.

`conversion_iterator` does not currently have the ability to distinguish between active, inactive or all parameters, as the iterator is implemented in a container-base class `GStdPtrVectorInterface`, that does not have any knowledge about parameter objects. However, as the iterator gives you direct access to the parameter objects, checking for the "active"-state is easy.

Directly accessing the parameter objects

If we *do* know the parameter structure of the individual, then we can also directly access the parameter objects and convert them on the fly. As an example, if we *positively know* that a `GConstrainedDoubleObjectCollection` is stored in the first position of our individual, we can extract the collection as is shown in listing 15.3.

Listing 15.3: Directly accessing parmeter objects

```
double MyIndividual::fitnessCalculation(){
   double result = 0;

   // Extract the GConstrainedDoubleObjectCollection in position 0
   // Note: This call will throw if we accidently try to access an
   // object of another type
   boost::shared_ptr<GConstrainedDoubleObjectCollection> vC
                          = at<GConstrainedDoubleObjectCollection>(0);

   // We can now loop over the content of the collection just as if
   // it were an ordinary std::vector<GConstrainedDoubleObject> . E.g.:
   for(std::size_t i=0; i<vC->size(); i++) {
          result += vC->at(i)->value() * vC->at(i)->value();
   }

   // Note that we could instead also have used a
   // GConstrainedDoubleObjectCollection::const_iterator for the loop

   return result;
}
```

15.2.2. Defining Multiple Evaluation Criteria

So far we have assumed that, inside of the `fitnessCalculation()` function, only a single fitness is calculated. As discussed in section 2.5, though, it *can* be necessary to define multiple evaluation criteria.

A practical example might again be given by the simulation of a car engine, which returns multiple criteria, such as the fuel consumption, horse power and the amount of pollutants produced by the motor. All of them may be important.

On the other hand, Geneva comprises many optimization algorithms, and not all of them can deal with multiple evaluation criteria. Geneva handles this by identifying a master-evaluation criterion, represented by the value returned by the `fitnessCalculation()` function. This fitness will be used for the assessment of individuals by algorithms that cannot handle more than one evaluation criterion (a Gradient Descent would be an obvious example).

In addition to the master criterion, it is also possible to specify secondary results inside of `fitnessCalculation()`. This is demonstrated in the `04_GMultiCriterionParabola` example in the Geneva distribution. Listing 15.4 shows the evaluation function of this example.

Listing 15.4: Main results and secondary results

```
double GMultiCriterionParabolaIndividual::fitnessCalculation(){
	double main_result = 0.; // Will hold the main result
	std::vector<double> parVec; // Will hold the individual parameters

	this->streamline(parVec); // Retrieve the parameters

	// Do the actual calculations. Note that the first calculation
	// counts as the main result and that we can register other,
	// secondary evaluation criteria.
	main_result = GSQUARED(parVec[0] - minima_[0]);
	for(std::size_t i=0; i<parVec.size(); i++) {
		registerSecondaryResult(GSQUARED(parVec[i] - minima_[i]));
	}

	return main_result;
}
```

In this example, we have multiple, one-dimensional parabolas, each with its own optimum[8]. An algorithm capable of dealing with multiple evaluation criteria then needs to find parameter values that simultaneously minimize all parabolas.

As a word of precaution, we suggest not to chain algorithms capable of dealing with multiple criteria with those that cannot. If you do have to mix them, run the "single-criterion" algorithm first.

15.3. Serialization

Geneva's individuals need to be serializable, so they can be easily sent over a network for remote-evaluation[9]. Geneva's serialization framework is based on the *Boost.Serialization* library[61]. Detailed explanations of this library can be found at the Boost website (`http://www.boost.org`). A good introduction to the serialization library is also available online through a portal[69].

Note, that, in the vast majority of cases, you do *not* need to understand the details of Boost.-Serialization, though.

[8]`GSQUARED` is a macro that calculates the square of its argument.
[9]On a side note, this feature is also used for check-pointing

Geneva has all necessary code for the serialization of its optimization-related objects already built in, so that you can concentrate on the description of those variables of your individuals that are required for the evaluation of your objects. Listing 15.5 gives an example of the code that is required in your individuals. It is taken from the `GFunctionIndividual` class that can be found in the Geneva distribution below the path `$GENEVAHOME/include/geneva-individuals`

Listing 15.5: Serialization of user-defined individuals usually only requires the specification of those variables that need to be serialized

```
class GFunctionIndividual : public GParameterSet
{
private:
  friend class boost::serialization::access;

  template<class Archive>
  void serialize(Archive & ar, const unsigned int v) {
          ar & BOOST_SERIALIZATION_BASE_OBJECT_NVP(GParameterSet)
             & BOOST_SERIALIZATION_NVP(demoFunction_);
  }

  [...]
};
```

The code in listing 15.5 holds the required specifications both for the serialization *and* the de-serialization of `GFunctionIndividual` objects.

First, (de-)serialization of the parent class `GParameterSet` is triggered, then the serialization of the one local variable (denoting the type of evaluation function used in this individual) is specified. Note that the `serialize()` template does *not* need to be public, but can be private. However, the friend declaration in listing 15.5 is required, if the `serialize()` function is declared private.

In the serialization step, *Boost.Serialization* then writes all required (and specified) data to a stream, either in plain text, XML or binary format[10]. During de-serialization, *Boost.Serialization* then creates a default-constructed object of your individual[11]. *Boost.Serialization* then loads all data specified in the `serialize()` function back into the corresponding variables, so that, for the purpose of evaluation, the de-serialized object is identical to the original object (as long as you have specified all necessary data, obviously).

Boost.Serialization also needs to be made aware of your individual. In order to achieve this, in the case of `GFunctionIndividual` just add the macro `BOOST_CLASS_EXPORT_KEY(class name)` to the header file of your individual, and the macro `BOOST_CLASS_EXPORT_IMPLEMENT(class name)`. So, in the case of `GFunctionIndividual`, the required statements would be `BOOST_CLASS_EXPORT_KEY(Gem::Geneva::GFunctionIndividual)` and `BOOST_CLASS_EXPORT_IMPLEMENT(Gem::Geneva::GFunctionIndividual).`

[10]. . . usually the most effective, but not recommended for cross-platform use
[11]i.e. there needs to be a default constructor in your individual. Note, though, that it may be private.

Beyond this, usually no other user-action is required. For some advanced cases (e.g. different code for serialization and de-serialization) we suggest to have a look at the Boost.Serialization documentation.

15.3.1. Dealing with Large Data-Sets

Complex evaluation functions will often require large data sets as input. It is not useful to serialize these data sets and ship them over a network, if they are static (i.e. not specific to each new or modified individual). Instead, they should be loaded just once remotely.

Geneva has a framework for this. The details of this operation are described in section 23.3.1, together with a discussion of networked execution.

As an example when training feed-forward neural networks with Geneva (compare section 9.3), the training data doesn't change, only the network's weights do. Hence only the weights need to be transferred to the remote site.

15.4. Further Interface Functions

Apart from the `fitnessCalculation()` function, a number of other functions need to be defined for each individual. Most were already discussed in the introductory example in chapter 11. In order to make this chapter self-contained, though, we again list the requirements below.

The default constructor

Each individual needs to have a default constructor[12] Note that it does not have to be public, nor does it have to do anything useful. However, Geneva uses the `Boost.Serialization` library, which requires serializable objects to have a default constructor.

The copy constructor

There needs to be a public copy constructor[13]. It is particularly required for the `clone_()` function (see below).

The destructor

It is advisable to define a *virtual* destructor for your individuals.

[12]I.e. a constructor that does not take any arguments,

[13]A copy constructor copies the state of another object of the same type

`operator=()`

It is recommended (but not required) to have a public `operator=()`. Note that it can be easily implemented through the function `load()` which is available for every individual. `operator=()` should return a constant reference to your individual.

`load_()` and `clone_()`

Every Geneva individual needs to have protected `load_()` and `clone_()` functions. Their general duties should be self-explanatory. Their implementation is straight forward. Please note that particularly loading also needs to take care of the parent class by calling the `GParameterSet::load()` function. Cloning is trivial and is achieved by just copy-constructing an individual. See chapter 11 for examples.

15.5. Personalities

Individuals are independent from optimization algorithms. However, optimization algorithms need to be able to associate data with individuals. This is done with the help of the `GPersonalityTraits` class and its derivatives (such as the `GEAPersonalityTraits` class in the case of Evolutionary Algorithms). Pointers to these classes can be stored in an individual. Users will not usually need to interact with the personality trait classes. The one exception occurs when users wish to write their own information providers, i.e. classes that emit information on the progress of the optimization in regular intervals. The details are discussed in chapter 25.

Here we just want to say that it is possible to access the personality trait objects through the `GOptimizableEntity::getPersonalityTraits<personality_type>()` member template.

Chapter 16. Advanced Constraint Handling

This chapter discusses generalized constraint handling beyond individual parameters and the facilities provided by Geneva to deal with such constraints. It also discusses a facility to flag invalid solutions after the evaluation function has run.

Key points: (1) Constraints are not limited to single parameters (2) A parameter's constraints can depend on the current value of another variable (3) With many such "inter-parameter constraints", the *valid* parameter space can be very small compared to the parameter space described by individual parameter boundaries alone (4) Optimization algorithms need to avoid spending too much time in "invalid" areas of the parameter space, but might not be able to visit invalid areas altogether (5) Optimization algorithms (and/or individuals) must make sure not to call evaluation functions with invalid parameter sets, as these might return invalid results or could even crash the entire program

Chapter 13 has already discussed various constrained value types, such as the `GConstrained-DoubleObject` class. Such constrained types can be used whenever it is known that a given variable will never exceed given boundaries. For example, the velocity of a car will never be lower than 0 and, for all practical purposes, will never reach 400 kilometers per hour. So the variable might be sufficiently described using a `GConstrainedDoubleObject` object with boundaries `[0,400[`.

However, there may be situations, where variable constraints depend on the current value of other variables. E.g. – in order to stay with the "car" example – the *allowed* speed of a car will certainly depend on its current geographic location, as different speed restrictions apply inside and outside of city boundaries, on motorways, etc. . So an optimization of the traffic flow (such as the minimization of the average time needed from A to B) would have to take into account dependencies between variable constraints and values.

16.1. Visualization

The effects of such dependencies can be graphically visualized only for simple cases, as any kind of dependency between any number of variables (and their constraints) may be possible. So a generalized description will be difficult. In order to better illustrate the problem, though, we will consider a

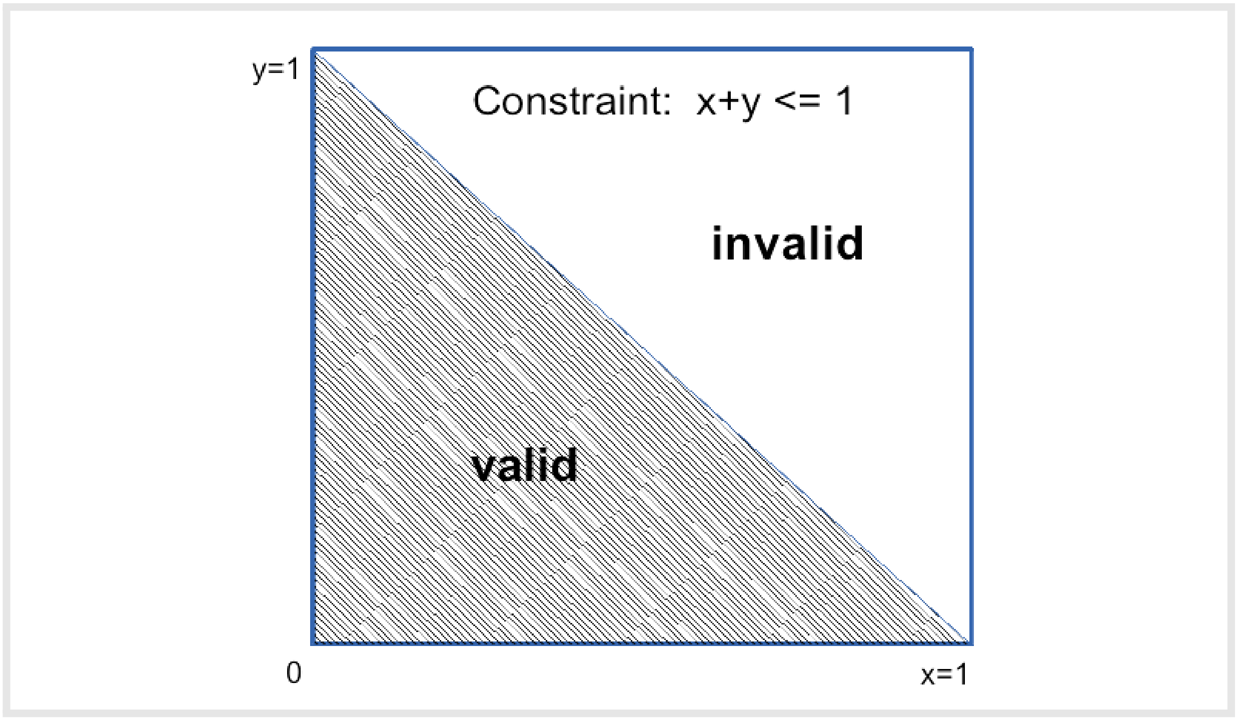


Figure 16.1.: *A constraint x+y <= 1 renders part of the parameter space invalid*

situation, where two variables `x` and `y` may only assume values in the range `[0,1]`, and the sum of both variables is also constrained by 1 (compare equation 16.1).

$$x+y<=1 \tag{16.1}$$

Thus only variable combinations, whose sum is smaller or equal 1 are considered valid. Figure 16.1 shows valid (grey) and invalid (white) areas resulting from this setup[1].

16.2. Problem Definition

In other words, half of the parameter space violates the constraint. The situation becomes even more severe with more than 2 variables. E.g., with three variables, with a "sum-constraint" equivalent to equation 16.1 only 25% of the parameter space is valid, with 4 variables 12.5% of the parameter space, and so on. Note that the evaluation function of parameter sets may (or may not) be defined in the whole parameter space, but valid results of the optimization problem may certainly only be found in those areas that do not violate any "inter-parameter constraint".

[1]Another, more complex example is presented by the Ramachandran plot. It plots backbone dihedral angles against amino acid residues of proteins. Only part of the parameter space leads to solutions which can actually be found in nature, which is an important fact to know when performing protein folding.

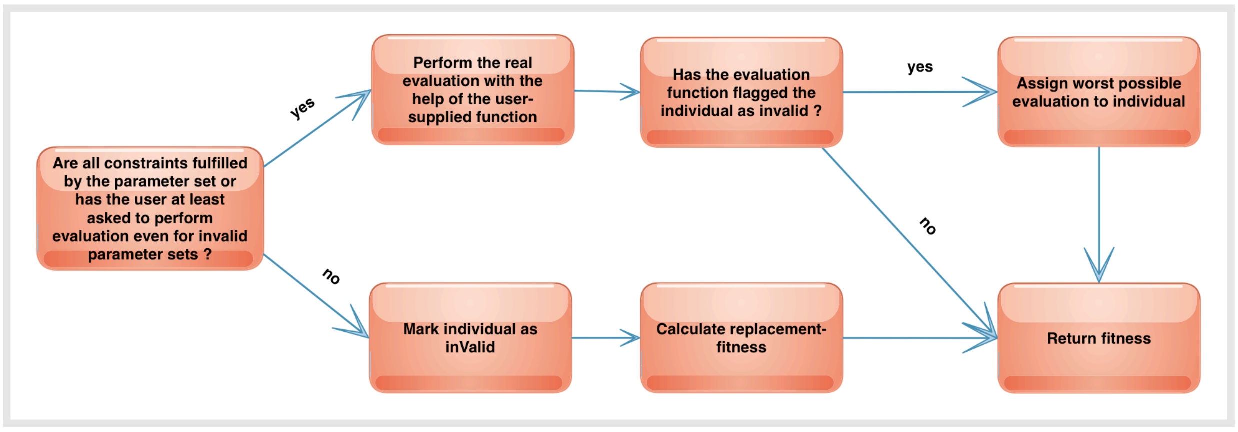

Are all constraints fulfilled by the parameter set or has the user at least asked to perform evaluation even for invalid parameter sets ?
yes
no
Perform the real evaluation with the help of the user-supplied function
Has the evaluation function flagged the individual as invalid ?
yes
no
Assign worst possible evaluation to individual
Mark individual as inValid
Calculate replacement-fitness
Return fitness

Figure 16.2.: *Evaluation workflow in the presence of potentially invalid solutions*

So the optimization algorithm needs to avoid “visiting” invalid areas of the parameter space, because

- the evaluation function might not be defined in this area and return erroneous results or might even crash
- the best (achievable) result of the optimization problem will not be found there. Spending too much time there will make the entire optimization process less effective

On the other hand, as no “standard description” of invalid areas can be achieved easily, invalid areas of the parameter space must be dealt with as part of the optimization procedure. This becomes even more important as it will be difficult for optimization algorithms to just “jump” from one valid area to another, if these are seperated by invalid solutions. Some algoritms, such as “Evolutionary strategies” could cope with invalid areas, if these had a consistently worse evaluation than valid areas (but no constant evaluation).

Hence an “ideal” solution to this dilemma would have to fulfill the following conditions:

1. It must be possible to identify invalid solutions even with multiple “inter-parameter constraints”
2. It would be advantageous if a “level of invalidity” could be defined for invalid solutions[2]
3. Calling the evaluation function in invalid areas of the parameter space must be avoided, as it might return invalid results or crash the program. A replacement value should be provided.
4. Distinction between valid and invalid areas should be done on the level of individuals, not optimization algorithms.
5. There should be no need for optimization algorithms to distinguish between valid and invalid areas of the parameter space
6. Spending too much time in invalid areas should be avoided
7. An evaluation must be assigned to invalid areas of the parameter space that is consistently worse than any possible evaluation of valid areas
8. The evaluation of invalid areas should improve when nearing valid areas

[2]. . . which may be difficult for the Ramachandran plot

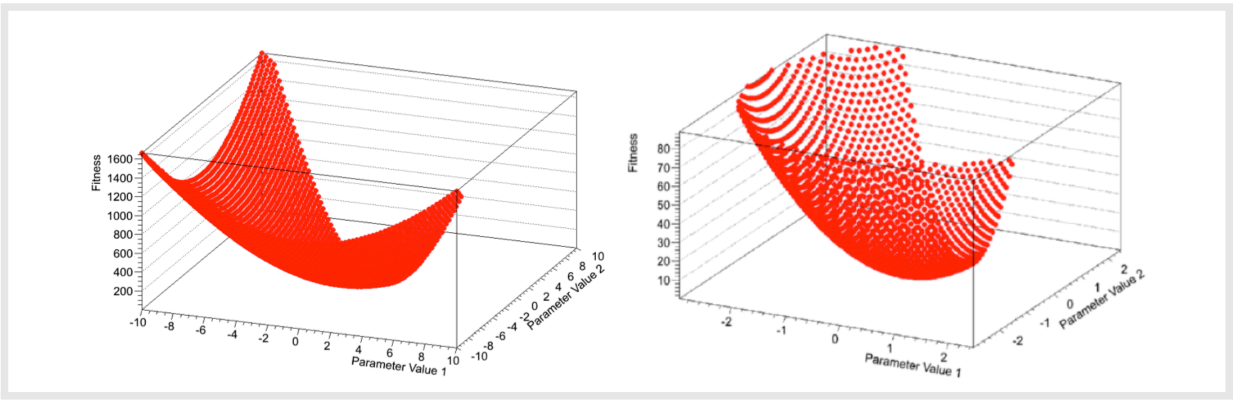


Figure 16.3.: *Valid solutions of a parabola with a "sum-constraint" (left) and an additional "sphere" constraint (right). The solutions were determined with Genevas parameter scan*

16.3. Identifying invalid candidate solutions with Geneva

Geneva contains a class infrastructure allowing to specify whether a constraint is fulfilled by the parameters identifying an individual, or to what extent a constraint has been violated. The idea is to associate a "validity level" with candidate solutions.

Note that validity checks may be performed prior to the evaluation of a condidate solution, so custom checkes may not depend on an up-to-date fitness. Indeed calling the fitness function on an un-evaluated individual may throw.

The validity level is calculated from the parameter values of a candidate solution and does not require the evaluation function to be called. Validity levels > 1 indicate that a violation has occurred. The difference between the actual return value and 1 is meant as an indication, how severe the violation is and is the most important ingredient to satisfy condition 2 in section 16.2. Values < 0 are either considered to be an error or valid, depending on a setting of the constraint objects (`GPreEvaluationValidityCheckT<ind_type>::setAllowNegative()`). If such values are considered to be invalid, a "replacement validity level" > 1 is calculated for negative validity levels. Values in the range $[0, 1]$ always indicate that the constraint wasn't violated[3].

Multiple constraints may be aggregated, and a joint "validityLevel" can be calculated according to user-defined policies. The most likely choice seems to be the product of the validity levels of all "invalid" constraints or 0, if no constraint was violated (policy "`Gem::Geneva::MULTIPLYINVALID`"). Another policy (`Gem::Geneva::ADDINVALID`) simply adds the validity levels of invalid candidate solutions.

In order to define a constraint, a user needs to derive a class from `Gem::Geneva::GParameterSetConstraint`. Listing 16.1 shows a strongly simplified header of a constraint check, whether the sum of a number of variables exceeds a given threshold.

[3]Usually, a value of 0 is returned for all valid solutions.

Listing 16.1: Simplified overload of the GParameterSetConstraint class defining a "sum" constraint

```
class GDoubleSumConstraint : public GParameterSetConstraint {
public:
   GDoubleSumConstraint();
   GDoubleSumConstraint(const double& C);
   GDoubleSumConstraint(const GDoubleSumConstraint&);

   virtual ~GDoubleSumConstraint();

protected:
   virtual double check_(const GParameterSet *) const;

   virtual void load_(const GObject*);
   virtual GObject* clone_() const;

private:
   double C_; // The constant that should not be exceeded by the sum of parameters
};
```

Note that the class derives indirectly from `GObject` and thus needs to implement all functions expected by that class, in particular the `load_()` and `clone_()` functions. Not shown in listing 16.1 is the serialization code, which should be added, but is trivial in this case.

For further details and the complete code we suggest to also have a look at the `GFunctionIndividual.hpp/.cpp` files, which have various examples for constraint objects.

The most important component of the constraint check ist the `double check_(const GParameterSet *) const` function. It gets “public” access to a GParameterSet object and returns a `double` value indicating whether (and to what extent) the constraint is violated or not.

Listing 16.2: The definition of the actual check for constraint violation

```
double GDoubleSumConstraint::check_ (
   const GParameterSet *p
) const {
   std::vector<double> parVec;
   p->streamline(parVec);

   double sum = 0.;
   std::vector<double>::iterator it;
   for(it=parVec.begin(); it!=parVec.end(); ++it) {
      sum += *it;
   }

   if(sum < C_) {
      return 0.;
   } else {
      return sum/C_;
   }
}
```

Listing 16.2 shows the definition of the constraint check. The “amount of violation” is simply calculated by normalizing the sum with the constraint value `C_`.

It is also possible to aggregate several constraints. Example `14_GDependentConstraints` shows how this is done and how the “constraint collection” can be added to an individual. Listing 16.3 shows an excerpt from this example.

Listing 16.3: Constraints can be aggregated

```cpp
using namespace boost;

boost::shared_ptr<GFunctionIndividual> p = gfi_ptr->get<GFunctionIndividual>();

// Create the constraint objects
shared_ptr<GDoubleSumConstraint>
   doublesum_constraint_ptr(new GDoubleSumConstraint(1.));
shared_ptr<GSphereConstraint>
   sphere_constraint_ptr(new GSphereConstraint(3.));
shared_ptr<GParameterSetFormulaConstraint>
   formula_constraint(new GParameterSetFormulaConstraint(
      "fabs(sin({{x}})/min({{y}}, 0.000001))"
      )
   ); // sin(x) < y

// Create a check combiner and add the constraint objects to it
shared_ptr<GCheckCombinerT<GOptimizableEntity> >
   combiner_ptr(new GCheckCombinerT<GOptimizableEntity>());
combiner_ptr->setCombinerPolicy(Gem::Geneva::MULTIPLYINVALID);

combiner_ptr->addCheck(doublesum_constraint_ptr);
combiner_ptr->addCheck(sphere_constraint_ptr);
combiner_ptr->addCheck(formula_constraint);

// Register the combiner with the individual (note: we could also have registered
// one of the "single" constraints here (see below for commented-out examples)
p->registerConstraint(combiner_ptr);

// p->registerConstraint(doublesum_constraint_ptr);
// p->registerConstraint(sphere_constraint_ptr);
// p->registerConstraint(formula_constraint);
```

As a first step, an individual is obtained from a factory class. Then various constraints are instantiated (one of the ist the `GDoubleSumConstraint` discussed above).

In the next step, a “check combiner” object is created, and the constraint objects are registered with it. As we now have multiple return values of constraint objects, we need to set a policy for the aggregation of return values. The most useful option seems to be to muliply all invalid values (policy `Gem::Geneva::MULTIPLYINVALID`) or return 0, if no constraint was violated.

Finally, the `GCheckCombinerT<>` object is registered with the individual. Only a single constraint object may be registered with an individual, which is why we use the “check combiner”. Alternatively, each of the constraints could have been exclusively registered with the GFunctionIndividual object.

One possibly interesting, predefined constraint object used in the example is called `GParameterSetFormulaConstraint`. It takes a textual formula (here `fabs(sin(x)/max(y, 0.00001))`) and evaluates whether the individual violates this formula. This can be used to dynamically add checks on the command line or through a configuration file.

The formula accepts parameter names – these must be identical to the names assigned to parameter objects using the `GParameterBase::setParameterName()` function (which is optional – but it is an error when a variable name used in the formula isn't found in the parameter object). For ease of parsing, variable names must be enclosed in double curly braces, such as **{{x}}** and **{{y}}**. Apart from that, the formula syntax is mostly identical to the C/C++ notation, including most predefined functions.

The underlying parser is described in section 33.10. Further information can also be found in the `GFormulaParser` test in the Geneva distribution.

Figure 16.3 shows the result of the aggregation of a "sum constraint" and a "sphere constraint" (only solutions inside of a sphere with a given radius are considered to be valid). The underlying evaluation function is a simple parabola. The plots were created using Genevas parameter scan implementation (compare chapter 22).

Note that Geneva caches the "invalidity" values, as their calculation may be computationally expensive. Re-calculation thus only happens for individuals whose dirty flag is set, or if the evaluation id[4] does not match the one stored for the individual.

16.4. Transparent solution handling

From the point of view of the optimization algorithm it would be ideal if it could treat all candidate solutions alike, independent of whether they are valid or invalid. This is equivalent with the request for a common quality surfance both for valid and invalid solutins.

However, valid solutions must have a better evaluation than invalid solutions, and invalid parameter sets "close" to valid areas of the parameter space should have a better evaluation than those far away (Requirement 7 in section 16.2), so that the optimization process will move to valid regions of the parameter space as quickly as possible.

With the validity checks described in section 16.3 all this becomes possible, as long as constraints fulfill the conventions described there[5].

Four strategies have been implemented in Geneva so far:

- The simplest (and aguably least satisfying) solution assigns a constant value to invalid parameter sets. The value chosen is the worst possible double value (i.e. `MAX_DOUBLE` in the case of minimization, `MIN_DOUBLE` otherwise). This solution is not very satisfying, as it presents

[4]A unique id assigned to a given evaluation

[5]i.e. a validity level in the range $[0,1]$ indicates a valid parameter set, values > 1 indicate invalidity.

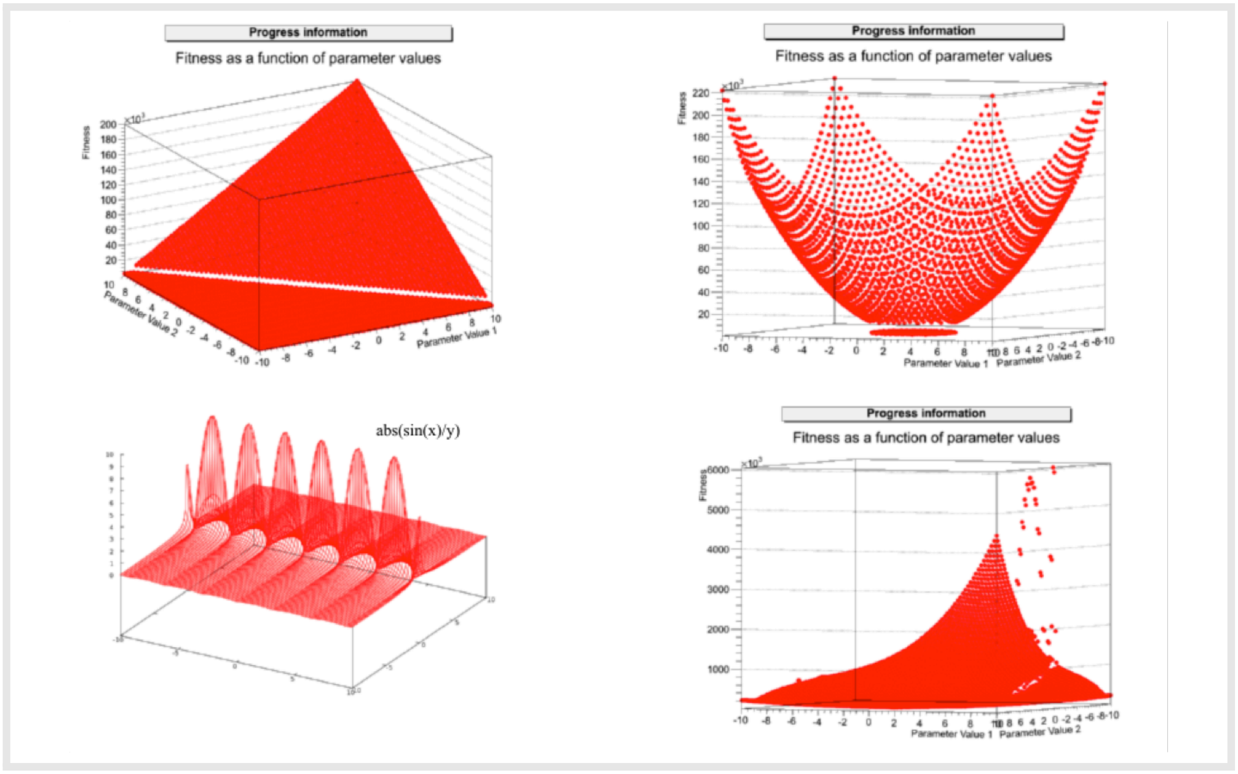


Figure 16.4.: *Quality surface including invalid in the case of the USESIGMOID policy*

the optimization algorithm with a flat quality surface in the invalid areas of the parameter space. Hence there is no indication for the algorithm where it should search for better solutions [Policy: “USEWORSTCASEFORINVALUD”].

- Geneva knows the worst known *valid* solution of each iteration. Evaluations of invalid regions are then calculated as a multiple of its fitness. The multiplier is calculated from the validiy level(s) of the constraint check(s). This way it is ensured that invalid solutions are always worse than valid solutions [Policy “USEWORSTKNOWNVALIDFORINVALID”]. The disadvantage of this solution is that the same area of the parameter space may receive different evaluations, depending on what the worst known valid solutions is at the moment.
- Geneva can apply a sigmoid function (with configurable parameters) to valid solutions. By default, the function converges to 10000. Invalid solutions receive a multiple of this threshold as a rating, calculated again from the level of invalidity. This method is mathematically more satisfying, but has the disadvantage that the evaluations stored in the `GParameterSet` objects are no longer the same as the “real” evaluation. Geneva thus makes these ratings accessible as well [Policy “USESIGMOID”].
- Finally, it is possible to tell Geneva to just call the user-defined evaluation function anyway, regardless of whether the individual is valid or not. It is then up to the user to find ways of dealing with parameter sets that are tagged as “invalid”.

	valid primary	valid secondary	invalid primary	invalid secondary
USESIMPLE EVALUATION	Normal evaluation result	Normal evaluation result	Result dependent on evaluation function	Result dependent on evaluation function
USEWORSTCASE FORINVALID	Normal evaluation result	Normal evaluation result	Worst possible evaluation, e.g. MAX_DOUBLE for minimization	Worst possible evaluation, e.g. MAX_DOUBLE for minimization
USEWORST KNOWN VALIDFOR INVALID	Normal evaluation result	Normal evaluation result	Worst known valid *primary* fitness of the current iteration, multiplied by level of invalidity	Worst known valid *secondary* fitness of the current iteration, multiplied by level of invalidity
USESIGMOID	Primary evaluation result transformed by sigmoid function	Secondary evaluation result transformed by sigmoid function	Upper or lower boundary of sigmoid, multiplied by level of invalidity	Upper or lower boundary of sigmoid, multiplied by level of invalidity
Marked as invalid by the user during the evaluation	none	none	Worst possible evaluation, e.g. MAX_DOUBLE for minimization	Worst possible evaluation, e.g. MAX_DOUBLE for minimization

Table 16.1.: Possible evaluation modes and values assigned to parameter sets

16.5. Constrained optimization with the USESIGMOID policy

The `USESIGMOID` policy seems to present the most satisfying solution of th options presented in section 16.4. Contrary to the `USEWORSTKNOWNVALIDFORINVALID` policy, the evaluation of invalid solutions does not depend on the progress of the optimization so far (i.e. the worst valid solution found up to that point). Invalid solutons are rated consistently worse than valid solutions, and the mapping for valid and invalid solutions is far easier to implement for the `USESIGMOID` policy than for `USEWORSTKNOWNVALIDFORINVALID`. The invalidity can be applied to a constant value (the boundary of the sigmoid function). Thus, for most cases, we recommend this policy. Figure 16.5 demonstrates the effect the sigmoid function has on a parabola (plotted in blue).

Note that a potential disadvantage of `USESIGMOID` is a loss of accuracy for parametersets, whose evaluations are close to the boundaries of the sigmoid. However, where this becomes apparent, one may raise the boundaries of the sigmoid, which are available as configuration parameters of the

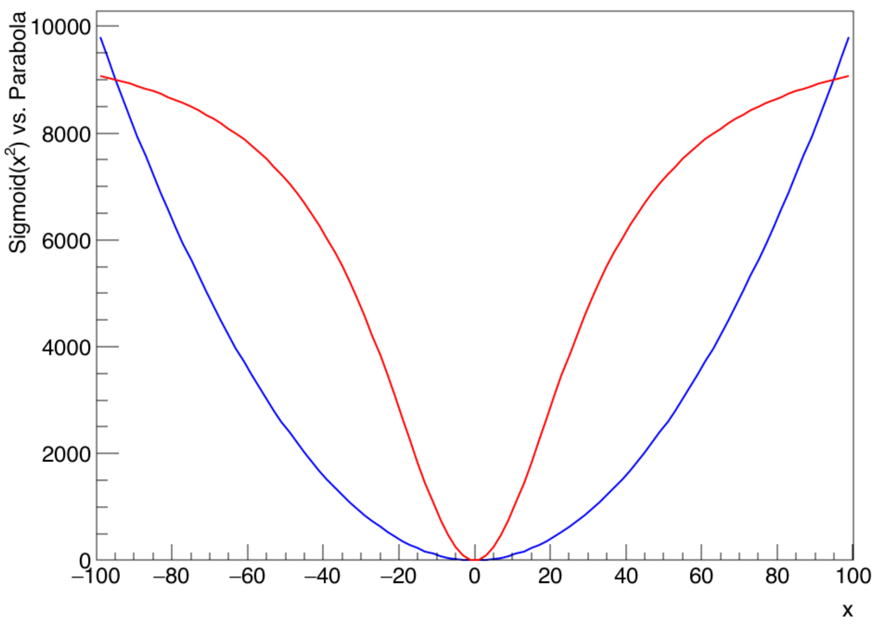


Figure 16.5.: *The effect that the application of a sigmoid function has on a parabola (plotted in blue)*

algorithm. Another possible disadvantage is the fact that the “true” evaluation is transformed by a function (the sigmoid). This may be confusing in cases where the absolute evaluation of parameter sets is important. However, please note that Geneva internally stores the “true” evaluation and makes it available to the user (see section 16.7 for further information).

Figure 16.4 shows the assembled quality surfaces of a parabola for the constraints discussed in section 16.3. The plot on the upper left-hand side shows the quality surface for the “sum-constraint” with both valid and invalid solutions. The valid solution (of a parabola) appear as a flat surface at the bottom, as the invalid solutions assume consistently (far) higher values. Please note the gap between valid and invalid solutions, resulting from the difference between the worst known valid solution and the upper boundary of the parabola.

The plot on the upper right-hand side of figure 16.4 shows the “sphere” constraint for the 2-dimensional parabola. Again the gap between valid and invalid solutions is visible.

The lower left-hand side illustrates a constraint, where only those solutions are considered to be valid that comply with $\left|sin(x)/y\right| < 1$. It is plotted with Gnuplot instead of the usual parameter scan in order to make the resulting structure easier to recognize.

The plot on the lower right-hand side finally shows the entire quality surface for the aggregation of all three constraints. It is immediately visible that the optimization algorithm will attempt to follow the “invalidity slope” towards valid solutions.

16.6. Other ways of identifying invalid solutions

It may happen that it is not known until after the evaluation function has run that a given parameter set is invalid. One example for such a situation could be a simulation that is used for the evaluation of a parameter set. It might then not be known until after the simulation has run that the parameter set does *not* represent a useful solution. Users may in this situation mark a parameter set as invalid through a call to the `GOptimizableEntity::userMarkAsInvalid()` function. Note that this call must be made from within the `fitnessCalculation()` function.

When an invalid solution of this type is found, Geneva will simply assign the worst possible evaluation to the corresponding individual, so that it is sorted out in the next iteration. Users may alternatively assign a "valid" evaluation to the individual, but should make sure that its value is worse than any valid solution.

Figure 16.2 shows the overall workflow used to determine sensible return values in the presence of potentially invalid solutions. Table 16.1 gives on overview of the consequences of each of the available ways of dealing with invalid solutions.

16.7. Accessing "true" and "transformed" fitness values

Users may be interested in both the true evaluation (provided there is one . . .) and the transformed evaluation. For this purpose, individuals in Geneva provide several functions.

The standard function to retrieve access to the "true" (i.e. untransformed) fitness is called `double fitness()`, or more generally `double fitness(const std::size_t&)`, if one needs to get access to both primary and secondary fitness values.

Note that the term "fitness" does not make sense for an individual, whose parameters have been changed without re-evaluation. Hence, by default, the above fitness functions will throw, when they are called for an individual, whose "dirty flag" is set.

If you want to explicitly allow re-evaluation, you may pass an additional boolean parameter `ALLOWREEVALUATION` to the `fitness(const std::size_t&)` function.

Access to the transformed fitness is given through the `double transformedFitness() const` and `double transformedFitness(const std::size_t&) const` functions. These functions assume that the individual has already been evaluated and the "dirty flag" isn't set. They will throw, if this is not the case.

Chapter 17.

Common Traits of Optimization Algorithms

This chapter describes the common features and characteristics of optimization algorithms and associated collections of individuals, as implemented in the Geneva library collection. The description particularly pertains to the `GOptimizationAlgorithmT` class, which forms the basis of all algorithms implemented in Geneva.

Key points: (1) All of Geneva's optimization algorithms derive from a common base class (2) This base class, `GOptimizationAlgorithmT`, implements the actions common to all algorithms, such as the main loop, information emission, or halt criteria (3) Each algorithm must implement a number of purely virtual functions of the base class, of which `double cycleLogic()` is arguably the most important (4) Information is emitted in regular intervals (5) Custom halt criteria, such as the maximum allowed number of iterations or the maximum amount of time for the optimization run, are implemented in the base class and apply to all algorithms (6) Custom halt criteria may be added (7) Each algorithm cares for its own parallelization, but must implement a serial and multi-threaded mode, as well as a variant that communicates through a broker (8) Addition of individuals to optimization algorithms mostly happens through a `std::vector<>` interface (9) As an experimental feature, `GOptimizationAlgorithmT` implements a check-pointing infrastructure.

We will start with a description of the class layout chosen for optimization algorithms and then discuss population types, as well as how candidate solutions may be added to algorithms.

Note that many of the configuration options described below can also be passed to the algorithm through a configuration file instead of direct calls to member functions. Further details on this topic are provided in the chapters describing each algorithm.

17.1. Class Layout

All of Geneva's optimization algorithms inherit from a common base class called `GOptimizationAlgorithmT`. The `T` at the end of the name indicates that this is a template class. In the algorithms implemented so far, the template argument is either `GParameterSet` or `GOptimizableEntity`[1]. `GParameterSet` is used for algorithms that act directly on problem

[1] Note that `GParameterSet` derives indirectly from `GOptimizableEntity`.

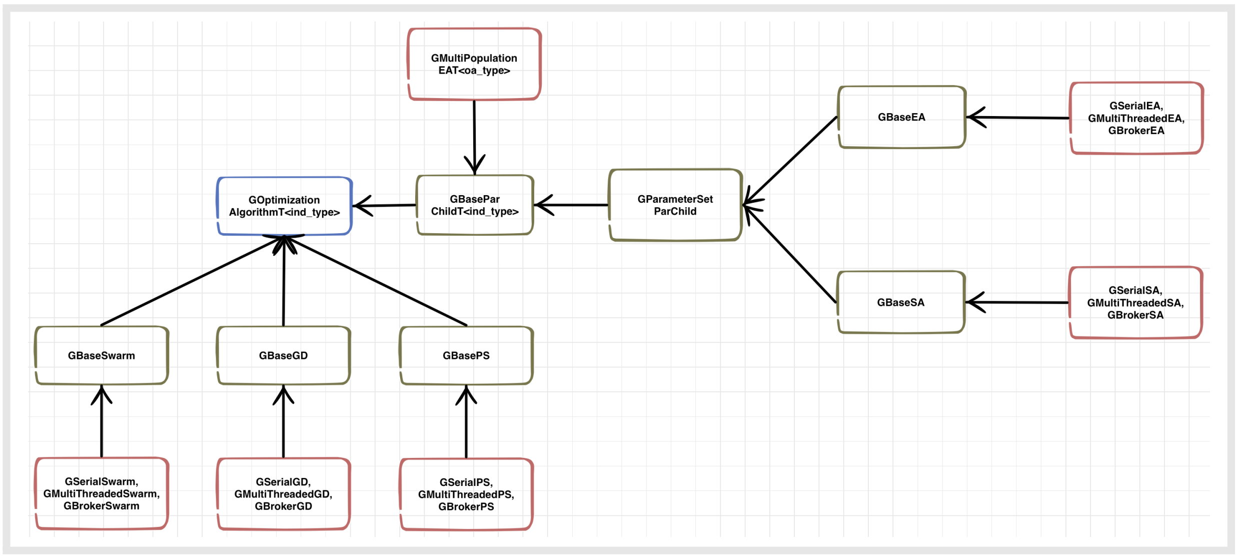

GMultiPopulation EAT<oa_type>
GOptimization AlgorithmT<ind_type>
GBasePar ChildT<ind_type>
GParameterSet ParChild
GBaseEA
GSerialEA, GMultiThreadedEA, GBrokerEA
GBaseSA
GSerialSA, GMultiThreadedSA, GBrokerSA
GBaseSwarm
GBaseGD
GBasePS
GSerialSwarm, GMultiThreadedSwarm, GBrokerSwarm
GSerialGD, GMultiThreadedGD, GBrokerGD
GSerialPS, GMultiThreadedPS, GBrokerPS

Figure 17.1.: *Optimization algorithms share common features, such as the definition of halt criteria, the main optimization cycle or the basic population structure. They can be implemented in a common base class, called* `GOptimizationAlgorithmT` *in Geneva.*

descriptions (aka candidate solutions). `GOptimizableEntity` is used for algorithms that also allow meta optimization (which is currently only true for Evolutionary Algorithms).

Base classes for each algorithm implement the features that cannot easily be parallelized. Each algorithm then implements at least a serial variant (used mainly for debugging) a multi-threaded variant used for multi-core machines large enough for the optimization problem at hand, and a variant that communicates with a broker, which can itself deal with different forms of parallelization. The broker's most prominent duty is to communicate with networked clients, through a dedicated “consumer”. See chapter 32 covering the courtier library for further information.

Note that it would have been possible to implement all parallelization modes in the common base class. However, this would have enforced strong restrictions on what can be parallelized in Geneva[2]. The current design allows each algorithm a finer control over what it wishes to parallelize and how, but implies some overhead in implementing new algorithms. All algorithms may act on the same problem descriptions. Figure 17.1 shows the inheritance tree for the optimization algorithms implemented in Geneva at the time of writing.

Note that, at the time of writing, optimization algorithms comprise less than 15% of the entire code base of Geneva. This means that it should be possible to add new algorithms with relative ease, as not much code is involved.

[2]To be more exact, it would have restricted parallelization solely to the concurrent evaluation of candidate solutions.

17.2. The Optimization Loop

Optimization algorithms generally act in cycles, basing the actions of the current iteration on the preceding iterations' results. It thus becomes possible to implement the "event loop" in an abstract base class. The actual optimization algorithms then only need to implement those parts of the business logic that are particular to their way of working. Listing 17.1 presents a rough sketch of this main loop. The actual implementation is slightly more complex.

Note that, as a user, you will practically never have to deal directly with the main optimization loop. A full description is provided here as we believe it may be important for users to understand some of the inner workings of the Geneva library collection.

Listing 17.1: All algorithms implemented in the Geneva library share a common main loop.

```
// [...]
// Output any initial information for the user
if(reportIteration) doInfo(INFOINIT);

// Initialize the optimization run
init();

do {
   // The actual business logic
   bestCurrentFitness = cycleLogic();

   // We want to provide feedback to the user in regular intervals.
   if(reportIteration && (iteration%reportIteration==0)) doInfo(INFOPROCESSING);

   // update the iteration counter
   iteration++;
} while(!halt());

// Clean up
finalize();

// Finalize the info output
if(reportIteration) doInfo(INFOEND);

// [...]
```

We will go through the listing from top to bottom.

doInfo()

As one of the first actions of a new optimization run, the `doInfo()` function is executed. It may be called in three modes – `INFOINIT`, `INFOPROCESSING` and `INFOEND` – which should be self-explanatory. The function is virtual and may be overloaded by derived classes. By default, though, it will try to execute a function in what is called an *optimization monitor*. Such monitors may be stored in the algorithm by the user and allow fine-grained control over what information is emitted. Chapter 25 discusses the topic of optimization monitors in detail.

Geneva provides default optimization monitors, which will print the best evaluation of the current iteration, and will in addition output a script in ROOT-format (compare appendix C) that allows to visualize the progress of the optimization run after it has terminated. The script will by default be called `result.C`.

The user may set the frequency of information emission using the `setReportIteration()` function. When `reportIteration` is set to 0, no emission of information is taking place. When set to 1, information will be emitted in every iteration; when set to 2, information will be emitted in every second iteration, and so on.

`init()` and `finalize()`

Optimization algorithms get the opportunity to execute custom code right before and after the main loop, so they can set up their internal data structures appropriately. This can be done by overloading the corresponding `init()` and `finalize()` functions.

The `do{ /* code */ } while(!halt())` loop

The most important call in the main loop happens right at the beginning. `cycleLogic()` is a function that must be overloaded by derived optimization algorithms. It performs all actions particular to a given algorithm and emits the best evaluation available when the function terminates. In the next step, information is emitted, if required. Finally, the iteration counter is incremented. The loop terminates when the `halt()` function returns true.

17.3. Geneva's Halt Criteria

An important part of the main optimization loop in listing 17.1 is the call to the `halt()` function. It wraps several halt criteria that may be combined or activated separately. It also wraps a custom halt criterion that may be overloaded in derived classes. Setting halt criteria is practically the only situation where users take direct influence on the main loop. The following halt criteria are implemented at the time of writing:

- The user may set the maximum number of iterations allowed for an algorithm using the `setMaxIteration()` function. This is the most common halt criterion.
- When no further progress is observed in an optimization algorithm, a "stall counter" is increased. The user may specify that the optimization run should be terminated when that counter has reached a certain value. This is possible using `setMaxStallIteration()`.
- It may be known in advance that a certain quality of a candidate solution is sufficient. So the user may choose to set a quality threshold, beyond which the optimization run should be

terminated. This is possible using the `setQualityThreshold()` function.

- Finally, it may be known in advance that only a certain amount of time may have passed before the optimization run must be terminated. So it is possible to set the maximum allowed time with the `setMaxTime()` function. Note that this function expects an argument with the type `boost::posix_time::time_duration`[27].

17.4. The Population Interface

It is generally assumed in the Geneva library that optimization algorithms act on collections (called *populations*) of candidate solutions (called *individuals* in this chapter). Geneva parallelizes mainly on the level of the parallel and distributed evaluation of candidate solutions. Hence, where needed, the library extends existing algorithms such that they act on populations rather than single candidate solutions. This allows to perform several evaluations in parallel.

17.4.1. Adding Candidate Solutions

In order to ease the addition of candidate solutions to optimization algorithms, `GOptimization-AlgorithmT<T>` exhibits an `STL` interface to the user. Starting points in the parameter space are thus simply added to algorithms using the usual `push_back()` function. Note that all interaction happens through smart pointers. Listing 17.2 shows a simple example of adding an individual to an evolutionary algorithm.

Listing 17.2: Individuals can be added to algorithms by means of the push_back() function

```
// [...]
boost::shared_ptr<GFunctionIndividual> ind(new GFunctionIndividual());
boost::shared_ptr<GSerialEA> ea(new GSerialEA());
ea.push_back(ind);
// [...]
```

As discussed before, `GOptimizationAlgorithmT<T>` is designed to accept either `boost-::shared_ptr<GParameterSet>` or `boost::shared_ptr<GOptimizableEn-tity>` objects as the template parameter `T`, depending on the chosen optimization algorithm.

This ensures that Evolutionary Algorithms may accept all optimization algorithms implemented in the Geneva library as “candidate solution”, this way implementing a form of *meta-optimization*. Likewise, all of Geneva’s algorithms accept `boost::shared_ptr<GParameterSet>` objects, so they can act on traditional individuals, implemented as collections of parameter objects.

17.4.2. Meaning of different positions in the vector

An algorithm may assign a particular meaning to a given position in a population. E.g., Geneva’s swarm algorithms segment the population into neighborhoods which are identified by their position in the vector. And the parents of evolutionary algorithms are located at the front of the collection.

17.5. Checkpointing

As an **experimental feature**, the Geneva library allows to perform checkpointing. The state of an optimization algorithm may be saved in regular intervals, or whenever a better candidate solution was found than was known so far.

The interval in which checkpointing should take place can be set using the `setCheckpointInterval()` function. `setCheckpointBaseName()` accepts the name of a directory and the (base-)name of a file. Jointly they determine when and where checkpoint files are stored. The current iteration and the quality of the best individual found are added to the base name in order to distinguish checkpoint files. `setCheckpointSerializationMode()` allows to specify whether checkpoints are stored in human-readable XML format or in the more compact binary format. Finally, `loadCheckpoint()` allows to load existing checkpoint files, e.g. in order to continue an optimization run.

The handling is identical for all implemented algorithms. **Use with care!**

Chapter 18.

Evolutionary Algorithms with Geneva

This chapter describes specific features relevant to Geneva's implementation of Evolutionary Algorithms. Geneva started as a pure implementation of this algorithm type, so this code is the longest established in the library, and arguably also the most advanced.

Key points: (1) Many core features of Geneva's Evolutionary Algorithms ("EAs") are implemented in the base classes `GOptimizationAlgorithmT` and `GBaseParChildT` (2) EAs can thus be treated like a `std::vector` of (smart pointers to) candidate solutions (3) EAs also share the same halt criteria implemented in `GOptimizationAlgorithmT` (4) Construction can happen either through the factory class `GEvolutionaryAlgorithmFactory`, the various constructors, or through the `Go2` class (compare chapter 24) (5) The factory class will read all settings from a configuration file, so that no manual configuration is necessary anymore (6) Apart from a (currently) fixed amount of parents and children, Geneva also allows dynamic growth of a population (7) Note that the number of parents may in the future become variable in the context of multi-criterion optimization (8) User-visible configuration options particularly relate to the duplication of parent individuals at the beginning of a new iteration and the selection of new parents from the best results of the current iteration (9) Mutation happens at the level of individuals and is only triggered by the EA-implementation. I.e., the chosen EA strategy does not have to have knowledge about details of the mutation strategy. (10) Geneva's implementation of Evolutionary Algorithms can be neither characterized as Evolution Strategy nor as Genetic Algorithm, as different parameter types may be mixed freely in the individuals

18.1. Looking Back at the Theory

Chapter 4 contains a full description of the theory behind Evolutionary Algorithms, as implemented in the Geneva library. However, in order to make this chapter more self-contained, we want to shortly review what was said there.

Evolutionary algorithms work through a sequence of duplication/recombination of parents, mutation of the ensuing children and selection of the best children (and sometimes parents) to form a new set of parents. Listing 18.1 shows again the basic work flow of both Evolution Strategies and Genetic Algorithms. It is evident that it shares many features with the main loop implemented in the `GOptimizationAlgorithmT` class (compare listing 17.1).

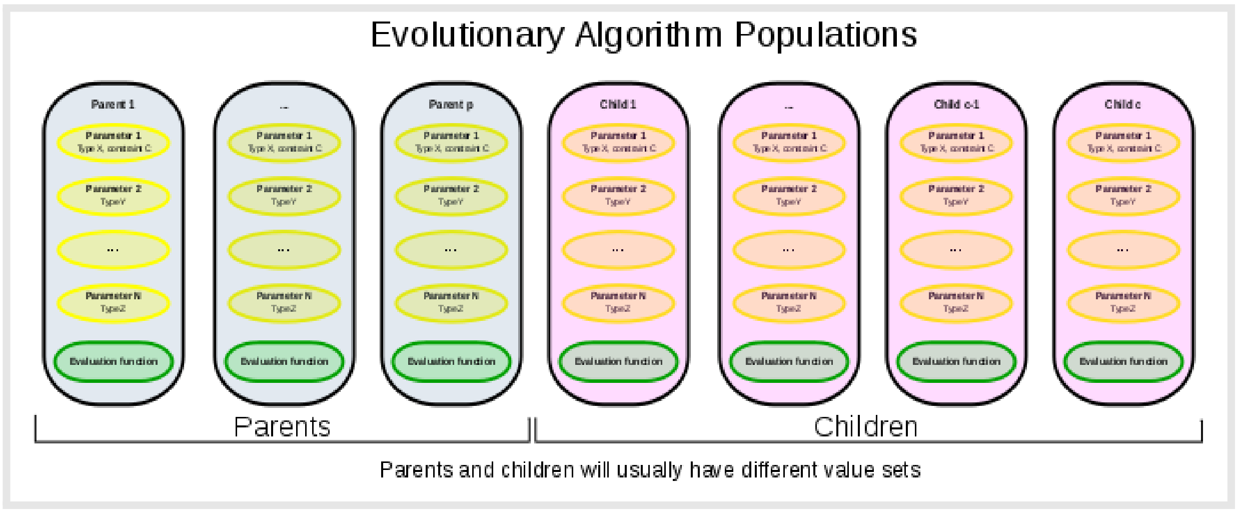

Evolutionary Algorithm Populations
Parent 1
...
Parent p
Child 1
...
Child c-1
Child c
Parameter 1 Type X, constraint C
Parameter 2 Type Y
...
Parameter N Type Z
Evaluation function
Parents
Children
Parents and children will usually have different value sets

Figure 18.1.: *Evolutionary Algorithm populations consist of $p >= 1$ parents and $c >= p$ children*

Listing 18.1: Evolutionary Algorithms can be described in a few lines of code

```
// [...]
do {
        recombine();     // create copies of parents or recombine their features
        mutate();        // modify individual parameters
        select();        // evaluate candidate solutions and select the best

        generation++;    // Increment the generation counter
} while(!halt());        // Terminate optimization when a halt criterion triggers
// [...]
```

Indeed, Geneva's implementation of Evolutionary Algorithms does not use a separate main loop. As one consequence, the same halt criteria implemented in `GOptimizationAlgorithmT` are also available for Geneva's Evolutionary Algorithms. Likewise, adding initial parents to the EA population can be done using a `std::vector<>` interface, namely the `push_back()` function.

18.2. Construction of Evolutionary Algorithm Objects

Direct construction of Geneva's Evolutionary Algorithm implementation can generally be done in one of two ways, described below. **A more indirect, but even easier possibility is also available through the Go2 class, which will be described in chapter 24.**

18.2.1. Using a Factory Class

A very convenient way is the construction through a factory class, called `GEvolutionaryAlgorithmFactory`. Listing 18.2 shows an example.

Listing 18.2: Evolutionary Algorithms can be easily created through a factory class

```cpp
#include "geneva/GEvolutionaryAlgorithmFactory.hpp"

int main(int argc, char** argv) {
  // [...]
  GEvolutionaryAlgorithmFactory
      f("./config/GEvolutionaryAlgorithm.json", PARMODE_SERIAL);

  // The factory will emit a "smart base pointer" to Geneva's EA implementation
  // The actual object is pre-configured with the desired parallelization mode
  // and the configuration options contained in the JSON file
  boost::shared_ptr<GBaseEA> ea_ptr = f();

  // You can make manual configuration changes, if desired. Here we request
  // a population size of 100, with 1 parent individual.
  ea_ptr->setDefaultPopulationSize(100,1);
  // [...]

  // Add individuals
  // [...]
}
```

The factory will emit a smart pointer `boost::shared_ptr<GBaseEA>` to an object with the desired parallelization mode (compare also chapter 23 and the class tree in figure 17.1), labelled as either `PARMODE_SERIAL`, `PARMODE_MULTITHREADED` or `PARMODE_BROKERAGE`.

The object will be pre-configured using the settings in a configuration file, which is specified as argument to the factory constructor. Note that, if the configuration file is not found, the factory will try to create a copy with default values below "`./config`". It will throw an exception if the target path isn't found. The configuration file uses the easily readable JSON format for its settings. Listing 18.3 shows an excerpt from the actual configuration file.

Listing 18.3: Excerpt from a configuration file in JSON format used for Evolutionary Algorithms

```
{
    // [...]
    "maxIteration":
    {
        "comment": "The maximum allowed number of iterations",
        "comment": "[GOptimizationAlgorithmT<ind_type>]",
        "default": "1000",
        "value": "1000"
    },
    "maxStallIteration":
    {
        "comment": "The maximum allowed number of iterations without",
        "comment": "improvement0 means: no constraint.",
        "comment": "[GOptimizationAlgorithmT<ind_type>]",
        "default": "0",
        "value": "0"
    },
    "reportIteration":
    {
        "comment": "The number of iterations after which a report should be issued",
        "comment": "[GOptimizationAlgorithmT<ind_type>]",
        "default": "1",
        "value": "1"
    },
    // [...]
}
```

Only changes to entries labelled `value` will have an effect. We suggest that you also read the following sections to understand the options available through the configuration file. Each option corresponds to a member function, through which the settings can also be made manually.

18.2.2. Through Constructors

Another obvious way is the usage of the default constructor. In this case you will directly construct one of the EA objects, which are called `GSerialEA` (serial execution), `GMultiThreadedEA` (for multi-threaded execution) and `GBrokerEA` (for communication through Geneva's broker).

Construction always happens through the default constructor. Once the object has been created, you will have to modify the default settings using the API available for the object. Many options and their meanings are described below.

18.3. Specifying the Amount of Parents and Children

The most prominent duty in setting up an evolutionary algorithm is to let it know about the number of parents and children, and, in effect, the total size of the population. Geneva's evolutionary algorithms *do* provide default values for these numbers. However, it is advisable to adapt the population size and number of parents to the problem at hand.

This is done with the help of the `setDefaultPopulationSize(std::size_t pop-Size, std::size_t nParents)` function. It accepts the size of the entire population as its first argument, and needs information on the desired number of parents (as a fraction of the entire population size).

The function can be called at any time before the start of the optimization loop. More commonly, the required information will be read from a configuration file (in which case no manual setup of the population size is required).

An obvious condition that must be met before the main optimization loop starts is that at least one individual has been registered with the evolutionary algorithm. Registered individuals will be interpreted as parent individuals (up to the number of parents specified by the user).

Listing 18.4 shows how to specify the population size.

Listing 18.4: Setting the population size is an important step

```
// [...]
std::size_t popSize = 100, nParents = 1;

GSerialEA ea;
ea.setDefaultPopulationSize(popSize, nParents);
// further population setup

// Adding an individual
boost::shared_ptr<GFunctionIndividual> ind(new GFunctionIndividual());
// further setup of the individual
ea.push_back(ind);

// Start the optimization run
ea.optimize()
// [...]
```

18.3.1. Population Size and Dynamic Population Growth

It is impossible to make general recommendations for the amount of parents and children in a population, as constraints also include the amount of available computing resources and the allotted time for an optimization run. For real-life optimizations these may even be the all-dominant criteria.

Larger populations will yield a better coverage of the parameter space and are thus more likely to succeed. On the other hand, where the population size is larger than the amount of compute units available, the execution time might last longer than the allotted time frame. Hence one important recommendation would be to choose a population size suitable for the available hardware.

So, if there are n compute nodes available for the optimization, a population with n children (resulting in n evaluations per iteration[1]) might be suitable. Population sizes of less then 10 children might be too small. So where not enough compute units are available, one might want to choose a multiple of the number of units for the number of children.

On small systems, it can be useful to restrict the population size to small amounts of children in the beginning and to increase the size gradually with the progress of the optimization run. Experience shows that the progress of the optimization run is large in the beginning and smaller populations are needed. However, as the algorithm is getting closer to the global optimum (or gets stuck in a local optimum), larger population sizes are needed.

Geneva's Evolutionary Algorithm allow to do this with the `setPopulationGrowth(std::size_t growthRate, std::size_t maxPopulationSize)` function, which accepts the growth rate per iteration as the first parameter, and the maximum allowed size of the population as the second. One would then set the initial population size to a comparatively small value (e.g. 10) in the beginning, using the `setDefaultPopulationSize()` function, and allow for a steady growth up to an upper limit.

[1] ... past the first iteration ...

Note that this feature does not help when there is a sufficient number of compute units available. One should also keep quantization effects in mind, as described in section 8.5.5. However, when there is a very large population (say: 400 children) and a small number of compute units (e.g. a quad-core Linux box), this feature can significantly reduce the amount of compute time needed to achieve satisfactory results.

18.4. Duplication Schemes

Looking back at listing 17.1, `GOptimizationAlgorithmT::cycleLogic()` (plus some other functions of `GOptimizationAlgorithmT`) is overloaded in the EA implementation. `cycleLogic()` then comprises the duplication/recombination, mutation and selection steps of listing 18.1. We will concentrate on these three steps in this section, starting with the duplication of parent individuals.

The duplication of parent individuals follows a possible recombination of two or more parents. It is an alternative method for the creation of new children. In the duplication step, children are created as identical copies of their parents, with the view of later mutation (described below in section 18.5). The important decision taken in the selection step is thus, *which* parent is duplicated[2].

VALUEDUPLICATIONSCHEME

In this duplication scheme, parents with a higher quality have a higher likelihood to be duplicated than those with a lower quality. The degradation of the selection likelihood happens linearly.

Table 18.1 illustrates the effects of the `VALUEDUPLICATIONSCHEME` scheme. The underlying example is the search for the minimum of a simple, 1000-dimensional parabola, performed in a population with 5 parents and 95 children over 2000 generations in a pure evolutionary strategy.

Internally, an array of 5 values between 0 and 1 is calculated, of which each indicates the likelihood for selection as a new parent. Parents with higher fitness get a higher likelihood for selection. In order to achieve this, for each duplication step a random number between 0 and 1 is calculated. Starting with the parent with the highest fitness, it is then compared in sequence to the thresholds assigned to each parent. When the random number is found to be lower than the threshold of a given parent, this parent is selected and copied into a new child.

RANDOMDUPLICATIONSCHEME

In the `RANDOMDUPLICATIONSCHEME` scheme, children are selected randomly from the available parents. Thus, parents with a high quality will not be favoured over parents with a lower quality.

[2]. . . which of course only plays a role if the user has asked for more than one parent in a population.

Parent position	0	1	2	3	4	
Assigned threshold	<= 0.34	<= 0.57	<= 0.75	<= 0.89	<= 1	
Likelihood for selection	0.34	0.23	0.17	0.14	0.11	
Actual selection numbers	65744	43508	32713	26238	21797	19000 (total)
% of total selections	0.35	0.23	0.17	0.14	0.11	

Table 18.1.: Effects of the VALUEDUPLICATIONSCHEME duplication scheme

DEFAULTDUPLICATIONSCHEME

This is the default (thus recommended) duplication scheme for Geneva's evolutionary algorithms that you get when you do not specify a scheme manually. It will point to one of the other schemes shown in this section. At the time of writing, `RANDOMDUPLICATIONSCHEME` is used as the default.

18.5. Mutation

The mutation step is governed mostly by the settings of adaptors and parameter objects, as described in chapters 14 and 13. Geneva's EA implementation simply triggers mutation of each parameter object and leaves the details to each object. This way, optimization with Evolutionary Algorithms stays generic and can be easily extended to other areas, such as meta-evolution.

There are consequently no user-accessible configuration options of the mutation step. **Users should rather take care to configure adaptors and parameter objects as desired.**

18.6. Evaluation and Selection

The recombination and mutation steps result in new candidate solutions. Their quality needs to be evaluated, before parents for the next iteration can be selected. Particularly the selection step involves a number of configuration options that may be set by the user. We will list the selection schemes in the following. Selection schemes may be chosen by the user with the `setSortingScheme()` function[3].

MUPLUSNU_SINGLEEVAL

As shown again in figure 18.1, populations in Evolutionary Algorithms consist of parent- and child-individuals. `MUPLUSNU_SINGLEEVAL` describes a selection scheme, where new parents are selected from both the preceding iteration's parents and children, according to their fitness. This involves sorting the entire population, so the best individuals are in the front of the population.

[3]The function name reflects the inner workings of the selection step, which is implemented as a sorting procedure.

As a consequence, the quality of the population will never decrease. On the other hand, experience shows that, with this selection scheme, the optimization gets more easily stuck in a local optimum and stagnates. This is also demonstrated on a concrete example in figure 9.3 on page 68.

Select this scheme with a call to `ea.setSortingScheme(MUPLUSNU_SINGLEEVAL)`, where `ea` is an object of Geneva's Evolutionary Algorithms implementation.

Note that `MUPLUSNU_SINGLEEVAL` only takes into account a single evaluation criterion. If you want to perform multi-criteria optimization in the context of evolutionary algorithms, use the `MUPLUSNU_PARETO` selection scheme instead.

MUCOMMANU_SINGLEEVAL

In the `MUCOMMANU_SINGLEEVAL` scheme, new parents are selected from the last iteration's children only. The quality of the population may thus generally decrease from iteration to iteration, and could in theory even diverge. However, there is always a selection pressure towards better solutions.

Experience shows that, overall, the optimization will make better progress than in the case of `MUPLUSNU_SINGLEEVAL`, so **we generally recommend this selection scheme for single evaluation criteria.** See again figure 9.3 for a concrete comparison between `MUCOMMANU_SINGLEEVAL` and `MUPLUSNU_SINGLEEVAL`.

Select this scheme with a call to `ea.setSortingScheme(MUCOMMANU_SINGLEEVAL)`, where `ea` is an object of Geneva's Evolutionary Algorithms implementation.

Use `MUCOMMANU_PARETO` instead of this selection scheme if you have more than one evaluation criterion.

MUNU1PRETAIN_SINGLEEVAL

This selection scheme is a compromise between `MUPLUSNU_SINGLEEVAL` and `MUCOMMANU_SINGLEEVAL`, applicable to populations with more than one parent individual and a single evaluation criterion for each individual.

Here, the best parent is always retained, unless a better child is found. Other parents of lower quality are replaced by the best children. So the population's quality will never decrease, but the optimization procedure will nevertheless cover a wider range than in the case of `MUPLUSNU_SINGLEEVAL` alone.

Select this scheme with a call to `ea.setSortingScheme(MUNU1PRETAIN_SINGLEEVAL)`, where `ea` is an object of Geneva's Evolutionary Algorithms implementation.

Note that we didn't benchmar this procedure against `MUCOMMANU_SINGLEEVAL` yet, so that we do recommend the latter.

MUPLUSNU_PARETO

Section 2.5 has introduced multi-criterion optimization in general and pareto optimization in particular. If you have more than one evaluation criterion, `MUPLUSNU_PARETO` lets you switch on pareto optimization, using the $(\mu+\nu)$ selection scheme.

Select this scheme with a call to `ea.setSortingScheme(MUPLUSNU_PARETO)`, where `ea` is an object of Geneva's Evolutionary Algorithms implementation.

Note that, at the time of writing, this is still considered to be an experimental feature.

MUCOMMANU_PARETO

The `MUCOMMANU_PARETO` is the equivalent of `MUCOMMANU_SINGLEEVAL` for multiple evaluation criteria.

Select this scheme with a call to `ea.setSortingScheme(MUCOMMANU_PARETO)`, where `ea` is an object of Geneva's Evolutionary Algorithms implementation.

Note that, at the time of writing, this is still considered to be an experimental feature.

18.7. Mixing Parameter Types

The most prominent representatives of Evolutionary Algorithms are Evolution Strategies and Genetic Algorithms, dealing with either floating point variables or boolean collections. Geneva's initial focus was on Evolution Strategies, and this is arguably still the area of the library that is the most advanced.

Geneva's individuals, however, allow to freely mix different parameter types, namely floating point, boolean and integer values, in order to describe a given optimization problem. As it is each parameter object that, through its adaptor, triggers mutation, Geneva's implementation of Evolutionary Algorithms can no longer be characterized as either "pure" Evolution Strategy or "pure" Genetic Algorithm. This allows more freedom to express a given optimization problem.

Chapter 19.

Simulated Annealing with Geneva

Key points: (1) Many core features of Geneva's Simulated Annealing implementation ("SAs") are implemented in the base class `GOptimizationAlgorithmT` and `GBaseParChildT` (2) SAs can thus be treated like a `std::vector` of (smart pointers to) candidate solutions (3) SAs also share the same halt criteria implemented in `GOptimizationAlgorithmT` (4) Construction can happen either through the factory class `GSimulatedAnnealingFactory`, the default constructor, or through the `Go2` class (compare chapter 24) (5) The factory class will read all settings from a configuration file, so that no manual configuration is necessary anymore (6) In Geneva, Simulated Annealing is closely related to Evolutionary Algorithms (7) Particularities of Simulated Annealimng are implemented as a special selection scheme in the context of Evolutionary Algorithms. For this reason, both algorithms share a common base class (`GBaseParChildT<T>`). (8) Geneva's SA implementation differs from the usual setup in that it maintains parents and children, i.e., it may follow multiple optimization paths. (9) Geneva's SA implementation may thus perform optimization in parallel, whereas the "standard" SA algorithm always deals with a single candidate solution. (10) The "start temperature" and the degradation strength form two particularly important settings of Simulated Annealing in Geneva

Traditionally, Evolutionary Algorithms and Simulated Annealing followed different strategies. However, as explained in chapter 5, Geneva implements Simulated Annealing on the same foundation as Evolutionary Algorithms. Both share a common base class: `GBaseParChildT<T>`.

In contrast to traditional SA-implementations, Geneva uses more than one candidate solution, though. Simulated Annealing is then implemented as a special selection scheme. This setup allows us to use Geneva's infrastructure for parallelization, whereas traditionally only one candidate solution would heve been created at a time.

Thanks to the common base class, many parameters of the Evoltionary Algorithm classes are also available in Simulated Annealing. In particular, setting of population sizes remains an important duty.

There are two additional parameters that play an important role for Simulated Annealing. Their background is described in chapter 5, so we will just list the configuration options that may be changed by the user.

`ea.setT0(double)` allows to set the start temperature – in a sense it it is a measure for the entropy in the molten metal (compare chapter 5 if you need more information). Likewise, `ea.setTDegradationStrength(double)` allows to specify, how quickly the "molten metal" cools down.

Duplication / recombination work identical to Geneva's Evolutionary Algorithms, as this is implemented in one of the base classes. Likewise, Simulated Annealing offers the same parallelization modes as all other optimization algorithms (serial, multithreaded or throuh the broker). Also, the same adaption/-mutation operators may be used as for evolutionary algorithms.

Again, Geneva's Simulated Annealing implementation may be instantiated through the `Go2`-class (mnemonic "sa"), by direct calls to the constructors or with the help of the factory class `GSimulatedAnnealingFactory`

Chapter 20.

Particle Swarm Optimization with Geneva

This chapter describes Geneva's implementation of the Particle Swarm Optimization algorithm. This is the first algorithm that was added, after Geneva's Evolutionary Algorithm implementation was completed. At the time, it has triggered many architectural changes in the Geneva library collection, so that it has now become far easier to add new algorithms.

Key points: (1) Swarm algorithms work through repetitive position updates of a population's individuals (2) Individuals are drawn to different extents towards a personal best solution, the best solution known so far in their neighborhood and possibly also towards a globally best solution (3) Swarm algorithms, as implemented in Geneva, act on floating point values only. Other parameter types of an individual will remain constant. (4) Construction may either happen through a factory class or (with more work) through the default constructor (5) In comparison to Evolutionary Algorithms, fewer configuration parameters need to be set

20.1. Looking Back at the Theory

Chapter 6 contains a description of the theory behind swarm algorithms (or "**P**article **S**warm **O**ptimization", or **PSO** for short), as implemented in the Geneva library. The algorithm described in listing 6.1 is fully implemented. We will aim to add the (far easier) algorithm shown in listing 6.2 in the near future, as it can be implemented with the infrastructure that is already available.

In a nutshell, in the variant shown in listing 6.1, individuals are drawn to differing extents in each iteration towards the personally best solution, a neighborhood-best and a globally best solution. The movement of each individual also contains a component taking into account the best solutions of past iterations. The "best" solutions are updated in each iteration. Figure 20.1 illustrates this situation.

Just like in the case of Evolutionary Algorithms, Geneva's PSO implementation uses the main loop implemented in the `GOptimizationAlgorithmT` class. Hence particularly the halt criteria and main loop, as discussed in sections 17.2 and 17.3 can be reused. This reduces the amount of code to be implemented for PSOs in Geneva.

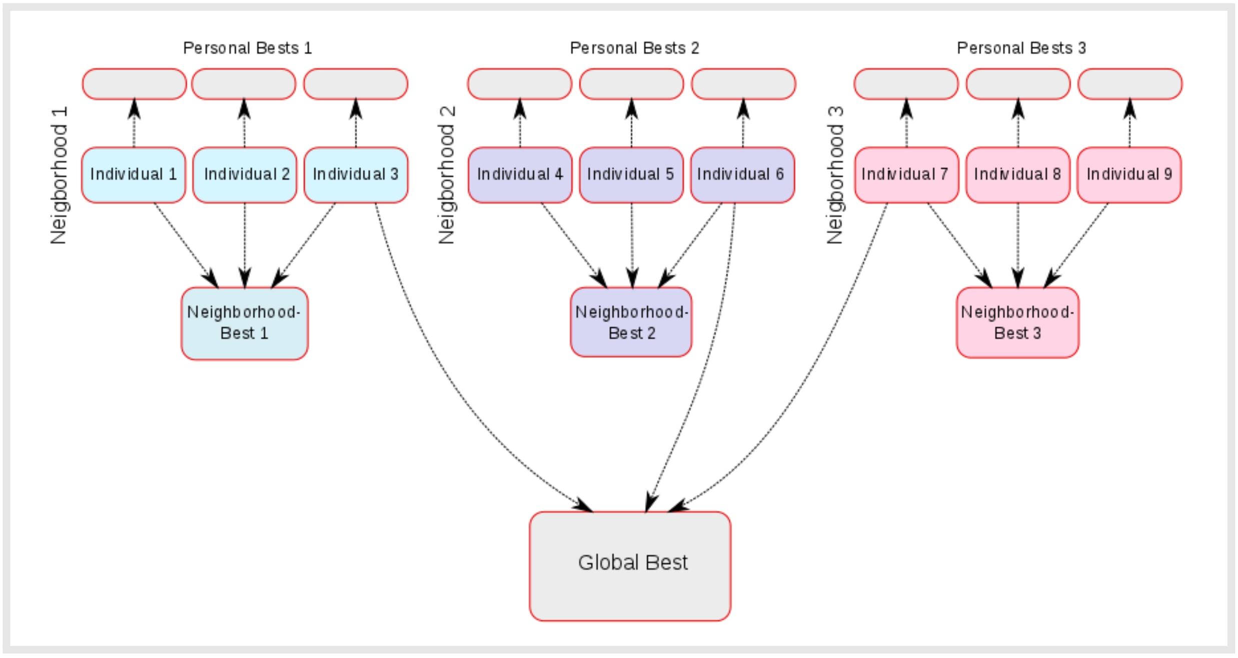


Figure 20.1.: *A swarm population is segmented into neighborhoods. In each iteration, individuals are drawn to a different extent towards their respective personal best-, neighborhood best and globally best solution known so far.*

Like everywhere in Geneva, individuals can be added to the algorithm by means of the `push_back()` function. An important difference to Evolutionary Algorithms is that Geneva's swarm algorithms only act on floating point values.

20.2. Construction of PSO Objects

Construction of Geneva's PSO objects can generally be done in one of two ways, which are described below. **A more indirect, but arguably even easier possiblity is also available through the Go2 class, which will be described in chapter 24.**

20.2.1. Using a Factory Class

Geneva comprises a factory class for its swarm algorithms, called `GSwarmAlgorithmFactory`. Its construction follows the same rules as already discussed for the `GEvolutionaryAlgorithmFactory` class. Listing 20.1 shows an example.

The factory class in listing 20.1 reads all data from a configuration file in JSON format (compare listing 18.3 for an example). This does not prevent you from making manual changes to the swarm algorithm object, though.

Listing 20.1: Swarm algorithms can be easily created through a factory class

```
#include "geneva/GSwarmAlgorithmFactory.hpp"

int main(int argc, char** argv) {
  // [...]
  GSwarmAlgorithmFactory
      f("./config/GSwarmAlgorithm.json", PARMODE_SERIAL);

  // The factory will emit a "smart base pointer" to Geneva's swarm implementation
  // The actual object is pre-configured with the desired parallelization mode
  // and the configuration options contained in the JSON file
  boost::shared_ptr<GBaseSwarm> swarm_ptr = f();

  // You can make manual configuration changes, if desired
  swarm_ptr->setSwarmSizes(3,20);
  // [...]

  // Add individuals
  // [...]
}
```

20.2.2. Through Constructors

Construction is of course also possible through the constructor. In this case you need to directly create the objects for the chosen parallelization mode, called `GSerialSwarm`, `GMultiThreadedSwarm` and `GBrokerSwarm`.

Construction always happens through the default constructor. Once the object has been created, you may modify the default settings using the API provided. A number of important settings is described below. It is recommended that you read it even if you choose the factory method. The information will give you a better understanding of the configuration options available in the configuration file.

20.3. Neighborhood-Sizes and Numbers of Neighborhoods

Apart from choosing good start values, arguably the most important duty in setting up a Geneva swarm algorithm is the specification of the number of neighborhoods, as well as the number of individuals in them. The total size of the population can then be calculated from these two numbers. Note that all neighborhoods are required to have the same size[1]. The population size may be set using the `setSwarmSizes(std::size_t, std::size_t)` function. It accepts the number of neighborhoods as its first argument, and the number of members in each neighborhood as the second argument. Listing 20.2 shows how to specify the population size.

[1]While the size of each neighborhood may change in each iteration, e.g. due to missing responses in case of networked execution, the neighborhoods have the ability to repair themselves. This way each iteration may start with a defined number of candidate solutions in each neighborhood.

Listing 20.2: Setting the number of neighborhoods and members in them is an important step

```
// [...]
std::size_t nNeighborHoods = 5, nNeighborhoodMembers = 20;

GSerialSwarm swarm;
swarm.setSwarmSizes(nNeighborHoods, nNeighborhoodMembers);
// further population setup

// Adding an individual
boost::shared_ptr<GFunctionIndividual> ind(new GFunctionIndividual());
// further setup of the individual
swarm.push_back(ind);

// Start the optimization run
swarm.optimize()
// [...]
```

20.4. Setting Progress Factors

Listing 6.1 contains a number of progress factors, labelled `w`, `c0`, `c1` and `c2`. They determine the amount by which individuals are drawn towards their personal best, the corresponding neighborhood's best and the globally best solution respectively (factors `c0-c2`). `w` is a multiplicative factor for the last iteration's "velocity" (called `Delta` in listing 6.1). In a way, it can be compared with the strength of an individual's recollection of past "good" regions.

In the Geneva API, these factors can be set with the functions `setCPersonal(double)`, `setCNeighborhood(double)` and `setCGlobal(double)`. The constant `w` can be set with the function `setCVelocity(double)`. Usual values for `cPersonal` and `cNeighborhood` are 2. As all individuals are also drawn towards a global best, the constant `w` (or `cGlobal`) also represents the magnitude of correlation between different neighborhoods. Small values are recommended.

20.5. Constraints for Position Updates

If `Delta` in listing 6.1 gets too large, there is a danger that the swarm algorithm diverges. Hence the size of the velocity may be constrained to a percentage of the allowed value range of the floating point parameters constituting each individual[2].

The function `setVelocityRangePercentage(double percentage)` allows to set this constraint. The default value of `percentage` is 0.15. This means that the velocity update of each floating-point parameter may not exceed 15% of its allowed value range.

[2]For unconstrained values, the initialization range is used instead of the allowed value range of a parameter.

Note that the algorithm takes care that all parameters are scaled down by the same factor (if at all necessary), so that the velocity vector does not change direction.

20.6. The Update Rule

In Listing 6.1, a joint set of random numbers is calculated for all parameters of the individual. This is called the “linear update rule” (expressed through the constant `SWARM_UPDATERULE_LINEAR` in Geneva).

Another possibility is to calculate a new random number for each parameter and each constant. This mode is called `SWARM_UPDATERULE_CLASSIC` in Geneva.

Users can choose between both modes with the function `setUpdateRule(ur)`.

Chapter 21.

Gradient Methods with Geneva

This chapter describes gradient methods, as implemented in the Geneva library. In a nutshell, gradient methods aim to utilize the shape of the quality surface, e.g. by seeking the path of steepest descent. This implies a number of advantages and disadvantages. On the one hand, gradient methods will find efficiently the next optimum. On the other hand, this algorithm type will usually not be able to leave that optimum again to proceed towards the global optimum. Gradient methods can only be applied to floating point parameters.

Key points: (1) Geneva implements a steepest descent algorithm (2) It works by calculating the difference quotient, which requires small variations of each floating point variable (3) As a consequence, for each iteration at least $n+1$ evaluations are required, where n is the number of floating point parameters (4) Geneva allows several concurrent starting points are allowed (5) Just three parameters describe the steepest descent: The number of concurrent starting points, the size of the finite step into the direction of each parameter and the size of the step in the approximate direction of steepest descent

21.1. Geneva's Steepest Descent Implementation

The steepest descent algorithm, as implemented in Geneva, works by calculating the difference quotient for all floating point variables. In other words, it makes a small step into the direction of each floating point parameter and calculates the quality at the new position. As a result, $n+1$ calculations are needed per iteration, where n is the number of floating point parameters[1].

The algorithm then calculates the approximate direction of steepest descent from these values and makes a step into this direction.

Geneva's steepest descent implementation also allows to perform several optimizations in parallel. The rationale behind this is that, given enough computing resources, several optimizations can be performed in parallel. And due to the tendency of gradient methods to get stuck in local optima, it can become important to start the optimization from a number of different starting points in parallel.

[1] This implies that for very large numbers of parameters, other algorithms are more efficient than gradient methods.

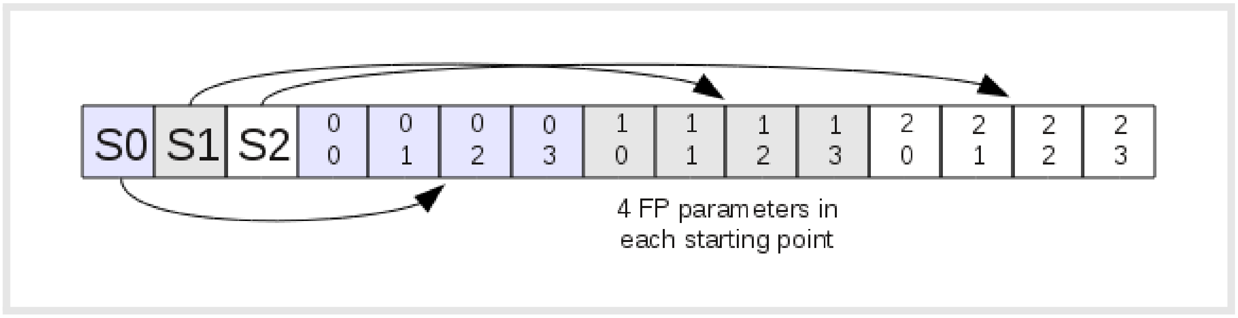


Figure 21.1.: *In a gradient-descent population, the individuals representing the current position in each iteration are followed by collections of individuals representing each of the floating point parameters.*

This feature results in a special layout of the gradient descent "population". Note that it is not up to the user to define the size of the population with gradient descents. Instead, the exact size is determined by the number of floating point parameters in an individual and by the number of concurrent starting points.

21.1.1. Population Layout and Addressing of Individuals

In Geneva's gradient descent population, "child individuals"[2] can be addressed in the following way:

$$child(i,j) = N_S + i * N_{FP} + j \tag{21.1}$$

where N_S represents the number of simultaneous gradient descents performed by the algorithm, N_{FP} is the number of floating point parameters in the individual, $i = [0,\ldots,N_S[$ and $j = [0,\ldots,N_{FP}[$ are indices used to address starting points and floating point parameters.

21.2. Construction of Gradient Descent Objects

Construction of Geneva's gradient method objects can generally be done in one of two ways, which are described below. **A more indirect, but arguably even easier possibility is also available through the Go2 class, which will be described in chapter 24.**

21.2.1. Using a Factory Class

Geneva comprises a factory class for its gradient descent algorithms, called `GGradientDes-centFactory`. Its construction follows the same rules as already discussed for the `GEvolu-`

[2]Individuals representing the step into the direction of each floating point parameter.

tionaryAlgorithmFactory class. Listing 21.1 shows an example.

Listing 21.1: Gradient Descent algorithms can be easily created through a factory class

```
#include "geneva/GGradientDescentFactory.hpp"

int main(int argc, char** argv) {
  // [...]
  GGradientDescentFactory
      f("./config/GGradientDescentAlgorithm.json", PARMODE_SERIAL);

  // The factory will emit a "smart base pointer" to Geneva's GD implementation.
  // The actual object is pre-configured with the desired parallelization mode
  // and the configuration options contained in the JSON file
  boost::shared_ptr<GBaseGD> gd_ptr = f();

  // You can make manual configuration changes, if desired
  gd_ptr->setNStartingPoints(1);
  // [...]

  // Add individuals
  // [...]
}
```

The factory class in listing 21.1 reads all data from a configuration file in JSON format (compare listing 18.3 for an example of JSON[3]). This does not prevent you from making manual changes to the swarm algorithm object, though.

21.2.2. Through Constructors

Construction is of course also possible through the constructors. In this case you need to directly create the objects for the chosen parallelization mode, called GSerialGD, GMultiThreadedGD and GBrokerGD.

Construction may happen through the default constructor. Once the object has been created, you may modify the default settings using the API provided. The most important settings are described below. It is recommended that you read it even if you choose the factory method. The information will give you a better understanding of the configuration options available in the configuration file.

Construction may also happen through a constructor that accepts various configuration parameters, whose meaning will become clearer below. Use GSerialGD(nStartingPoints, finiteStep, stepSize) constructor for this purpose (or GMultiThreadedGD, GBrokerGD).

[3]Note that the JSON code in listing 18.3 was generated for Evolutionary Algorithms, not Geneva's Gradient Descents

21.3. Important Configuration Options

There are just three important configuration options for Geneva's steepest descent implementation: The number of simultaneous starting points (set with the `setNStartingPoints(std::si-ze_t np)` function), the step size (set with the `setStepSize(float sz)`), and the size of the finite step into each parameter's direction (set with `setFiniteStep(float fs)`). The step size represents the length of the step made into the direction of steepest descent. All three options may also be passed directly to the constructor discussed in section 21.2.2.

Chapter 22.

Parameter Scans with Geneva

This chapter describes Geneva's implementation of parameter scans.

Key points: (1) Geneva implements both random parameter scans and scans on a grid (2) The algorithm should only be applied to small numbers of parameters, with a limited number of steps in each direction (3) The parameter scan uses the same infrastructure as all other algorithms (4) Parallelization is thus seemless (5) Parameter scans may be used to find suitable start valus for another optimization algorithm (6) They may also be used to test how sensitive a result is to the variation of a given parameter (7) Scans may be specified through member functions or through an easy text syntax, e.g. on the command line.

This algorithm is a relatively new addition to Geneva. The need for its implementation arose from a situation, where users wanted to perform manual scans of parameter ranges in one or two dimensions. Given Geneva's ability to "chain" algorithms, parameter scans may however also help to find a good starting point for other optimization algorithms. Another use-case is a check for the "sensitivity" of an optimization result when a chosen parameter is varied. Users must be aware, however, that with many parameters and many steps in each direction the time needed for the computation may quickly become too high (compare section 2.3.1 for further information).

For this reason, Geneva comprises the ability to perform both random parameter scans or scans on a grid. Scans may comprise all "native" parameter types of Geneva, such as integer, boolean and floating point numbers. Parameters can be chosen by name or through their position in an individual, and scan-ranges may be specified.

22.1. Construction of Parameter Scan Objects

Construction of Geneva's parameter scan objects can generally be done in one of two ways, which are described below. **A more indirect, but arguably even easier possiblity is also available through the Go2 class, which will be described in chapter 24.** `Go2` also makes available convenient command line options for the specification of parameter scans.

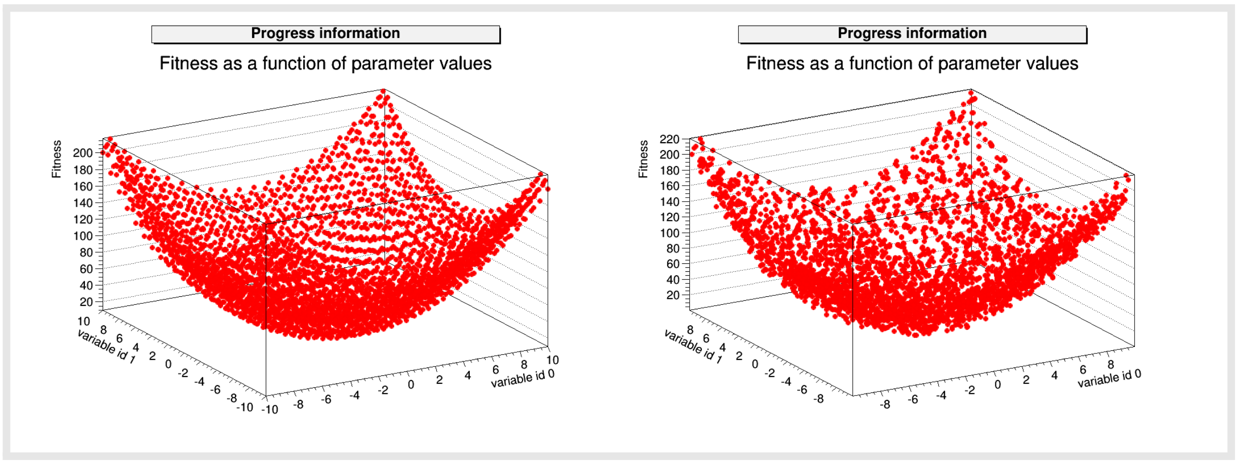


Figure 22.1.: *Parameter scans of the Rastrigin function (compare appendix A.5), on a regular 60x60 grid (left) and with random scan points (right)*

22.1.1. Construction through a Factory Class

Geneva comprises a factory class for its parameter scan algorithms, called `GParameterScanFactory`. Its construction follows the same rules as already discussed for the `GEvolutionaryAlgorithmFactory` class. Listing 22.1 shows an example.

Listing 22.1: Parameter scans can be easily created through a factory class

```
#include "geneva/GParameterScanFactory.hpp"

int main(int argc, char** argv) {
  // [...]
  GParameterScanFactory
      f("./config/ParameterScan.json", PARMODE_SERIAL);

  // The factory will emit a "smart base pointer" to Geneva's parameter scan
  // implementation. The actual object is pre-configured with the desired
  // parallelization mode and the configuration options contained in the JSON
  // file
  boost::shared_ptr<GBasePS> ps_ptr = f();

  // You can make manual configuration changes, if desired
  // Here we ask for the first and second double parameter to be
  // scanned in the range [-10:10] in 100 steps each, resulting in
  // 10000 evaluations of the objective function
  ps_ptr->setParameterSpecs("d(0, -10., 10., 100), d(1, -10., 10., 100)");
  // [...]

  // Add individuals
  // [...]
}
```

The factory class in listing 22.1 reads all data from a configuration file in JSON format (compare listing 18.3 for an example, albeit taken from Evolutionary Algorithms). This does not prevent you from setting configuration options manually, though.

22.1.2. Direct Construction through Constructors

Construction is of course also possible through the constructor. In this case you need to directly create the objects for the chosen parallelization mode, called `GSerialPS`, `GMultiThreadedPS` and `GBrokerPS`. Construction always happens through the default constructor.

Once the object has been created, you may modify the default settings using the API provided. A number of important settings is described below. It is recommended that you read it even if you choose the factory method. The information will give you a better understanding of the configuration options available in the configuration file, and you may always reconfigure selected options after the object has been created using the options in the configuration file.

22.2. Random scan versus scan on a grid

Parameter scans can be performed in two modes: Either a regular grid is defined, with a predefined number of steps in each direction, or the total number of evaluations may be specified, with a random selection of scan points throughout the parameter space.

Figure 22.1 illustrates this possibility on the example of the Rastrigin function. A scan on a 60x60 grid is shown on the left side of the plot, the right side shows a scan with random scan-points.

The option `GBasePS::setScanRandomly(bool)` allows to specify, whether the specified parameters should be scanned randomly (`true`) or on a grid (`false`).

The number of scan points for random scans is determined through the population size and the maximum number of iterations. For scans on a grid the number of scan points is determined through a "step" parameter of the scan-specification (22.3).

22.3. Specifying which parameters to scan

Specification of the parameters to be scanned happens through a string. Listing 22.1 already shows one example: `d(0, -10., 10., 100), d(1, -10., 10., 100)` means: "*scan the first and second double parameter that was registered with the individual in the range* $[-10:10]$ *in* 100 *steps each*".

Note that "100 *steps each*" really means that for two variables $100*100 = 10000$ evaluations are performed, or with just three variables 1 million evaluations. This happens, as every value of parameter 0 needs to be combined with every value of parameter 1. You can leave out the number of steps for random scans. Leaving it in the parameter specification for random scans will have no effect.

Instead of `d` for `double`, one may also use `i` for `boost::int32_t` and `b` for `bool`. Boundaries and the number of steps are specified just like in the case of `double` parameters, with the obvious exception of boolean parameters. For these, you may indeed give boundaries and steps, but they will have no effect. Leaving out the number of steps will lead to the usage of a default value for all parameter types.

A specifyer `s(10000)` means: scan the entire parameter space (covering all registered variables of an individual) with up to 10000 scan points. Combining this specification with additional parameter specifications will lead to an error.

There may be cases where it is not clear what the order of parameters is. However, Geneva allows to assign names to parameter objects. Its parameter scan uses this option to allow the following syntax for variables: `d(myVariableName, -10., 10., 100)` and likewise for the other parameter types. I.e., we can refer to the variables by name instead of the order of registration with the individual. Referring to a variable by name that wasn't registered with the individual will lead to an error.

Geneva also comprises container types, with the `GConstrainedDoubleCollection` just being one example. For these types, the following syntax is possible: `d(myContainerName[3], -10., 10., 100)`. This would scan the fourth entry (counting starts at 0, as is common in C/C++) of `myContainerName` and scan it in 100 steps from -10 to 10.

Chapter 23.

Parallelization Modes

Geneva's optimization algorithms can be executed in three different parallelization modes. This chapter discusses the available options. Each mode is represented by a different class (compare figure 17.1). This design was chosen over a centralized implementation (e.g. in the `GOptimizationAlgorithmT` class) in order to allow for additional parallelization to be implemented for specific algorithms. Chapter 24 shows how to access the different parallelization modes more easily.

Key points: (1) Optimization algorithms are available in a serial, multi-threaded and brokered mode. (2) The serial mode is mostly used for debugging purposes, but might also be useful if only a single license for an external solver is available (3) In the case of networked execution, it is not necessary to ship all constant data together with individuals. (4) Direct access to the three different execution modes of the available optimization algorithms can be tedious, but gives very direct control over the inner workings of your optimization program. (5) Chapter 24 describes an easier option for choosing between different algorithms and execution modes.

There are three execution modes `EXECMODE_SERIAL`, `EXECMODE_MULTITHREADED` and `EXECMODE_BROKERAGE` in Geneva, defined in the enum `execMode`. Using these constants might be useful when reading the parallelization mode from a configuration file or the command line.

23.1. Serial Execution

This is the simplest execution mode. It is primarily meant for debugging, if you encounter problems while integrating individuals with Geneva. Multithreading is reduced to the creation of random numbers in this mode (and you can effectively switch off even this, if needed). Thus any race conditions still encountered in serial execution mode are likely caused by your implementation rather than Geneva. On the other hand, if the problems do go away when switching to serial mode, they might indeed be related to Geneva. **Make sure to file a bug report in this case (compare section 30.1.1)**.

In order to gain direct access to serial execution, each optimization implements classes like `GSerialEA`, `GSerialSwarm` or `GSerialGD`. Other algorithms name the serial optimizer accordingly. There are no specific configuration options (other than the general ones inherited from `GBaseXX` and its parent classes, of course) that apply to the `GSerialXX` family of classes.

23.2. Multithreaded Execution

This execution mode uses the Boost.Thread library to execute different parts of optimization algorithms in parallel on the same system (without networking). As was discussed in chapter 8, the most important task to be parallelized in optimization algorithms is the evaluation of candidate solutions. This is the case for all implemented algorithms. Other parts of the algorithm might also be executed in parallel, depending on the chosen algorithm. Evolutionary Algorithms, as one example, might also perform the mutation step in parallel, as it is limited to each individual. Recombination, on the other hand, is likely not worth parallelizing, due to the task switching overhead[1].

In order to gain direct access to multithreaded execution, use one of the `GMultiThreadedXX` class (such as `GMultiThreadedEA`, `GMultiThreadedSwarm` and `GMultiThreadedGD`).

23.2.1. Configuation Parameters

Multithreaded execution adds one additional steering parameter over the options inherited from the `GBaseXX` family of classes: In each `GMultiThreadedXX` class, the number of threads for parallel execution of the optimization algorithm can be set with the `setNThreads(boost::uint16_t)` function, where `boost::uint16_t` is a 16 bit unsigned integer. Note that it is usually not necessary to set the amount of threads by hand, as Geneva tries to determine the number of execution units (e.g. physical cores, when hyper-threading is disabled) in your machine[2]. This is usually a good value for the number of threads as well. Where Geneva fails to determine that number, however, the number of threads is set to the default value of 2. When this happens, you might want to set the number of threads by hand, using this function.

Specifying 0 as the parameter of `setNThreads(boost::uint16_t)` will switch on automatic detection of the number of compute units. Any other value (>0, of course, as the parameter is unsigned) will switch automatic detection off.

23.3. Brokered Execution

Brokered execution uses the services of the `GBrokerConnector2T<>` class described in section 32.5 to submit individuals to the broker. Depending on the consumer which has been plugged into the broker, execution may either happen locally or at a remote site (such as a different node of a cluster or a worker node in a compute cloud). Both multi-threaded and serial execution are available as consumers locally – the latter mostly for debugging purposes[3].

[1]Note that, for the sake of simplicity and like all other algorithms, EA currently only parallelizes on the level of the evaluation. This has allowed to simplify the class structure.

[2]Note that it cannot distinguish between physical and virtual cores – a system with hyperthreading enabled will appear to Geneva as having more than the physical number of cores

[3]Another usage scenario might be a situation where you only have a single license for an external solver

As the "brokered" classes (such as `GBrokerEA`, `GBrokerSwarm` and `GBrokerGD`) derive from the `GBrokerConnector2T<>` class, the options added over what has already been implemented in the `GBaseXX` familiy of classes is limited the options of `GBrokerConnector2T<>`. This class is described in detail in section 32.5.

Of particular importance is the `setSubmissionReturnMode()` function. Depending in the chosen policy, the object will wait indefinitely for items of the current submission to return, or will timeout and optionally resubmit unprocessed items. Usually, this setting is specific to the chosen optimization algorithm[4], so that users do not need to care for this setting.

23.3.1. Loading Static Data at a Remote Site

The broker was initially designed only to deal with remote execution. In this usage scenario, individuals are serialized and shipped to a remote site. Serialization is computationally quite expensive, and the transfer particularly over a wide area connection costs time. Hence one will try to minimize the amount of data that needs to be serialized.

Think of an individual that represents a particular state of a feed-forward neural network. The individual will hold data for the weights between its nodes (compare section 9.3 and figure 9.5), which certainly needs to be shipped to the remote site for the evaluation, as they are different for each individual. However, the evaluation will also crucially depend on the training data. As this data is the same for each individual, the question must be asked how shipping it to the remote site can be avoided.

The templated network clients implemented in the Geneva's Courtier library therefore require that work item implement the `loadConstantData()` function. It accepts an object of the same type and loads – pretty much in the style of a copy constructor – any data specified by the user. As this is highly application-specific, users wishing to use this feature need to overload the (emtpy) `loadConstantData()` function. Once done, they can register a "constant" work item with the class handling client interactions. They also need to make sure that constant data in their individuals isn't serialized. This simply means omitting constant data in the `serialize()` function.

23.4. Direct Instantiation of Algorithms

We have now seen that different parallelization modes are implemented in different classes. This has the consequence that switching between different modes (for example with a command line parameter) can be somewhat tedious. The situation becomes worse when we also want to switch between different optimization algorithms. We will explain in chapter 24, how this can be avoided.

On the other hand, direct interaction with the available optimization algorithms will provide you with a very direct control of the actions of your optimization program. Hence we want ot explain in this section what needs to be done to achieve this. The code is based on the `GDirectEA` example in the Geneva distribution.

[4]e.g., gradient methods require a full return of work items

Listing 23.1 shows how to directly instantiate the different parallelization modes of Geneva's Evolutionary Algorithms. The variables in the example use tell-tale names, so we do not explain them further.

Listing 23.1: Direct instantiation of different parallelization modes of Geneva's EA Implementation

```
/**************************************************************************/
// If this is a client in networked mode, we can just start the listener and
// return when it has finished
if(EXECMODE_BROKERAGE==parallelizationMode && !serverMode) {
  boost::shared_ptr<GAsioTCPClientT<GParameterSet> >
     p(new GAsioTCPClientT<GParameterSet>(ip, boost::lexical_cast<std::string>(port)));

  p->setMaxStalls(maxStalls); // 0 equals an infinite number of stalled retrievals
  p->setMaxConnectionAttempts(maxConnectionAttempts);

  // Start the actual processing loop
  p->run();

  return 0;
}

/**************************************************************************/
// We can now start creating populations. We refer to them through the base class

// This smart pointer will hold the different population types
boost::shared_ptr<GBaseEA> pop_ptr;

// Create the actual populations
switch (parallelizationMode) {
//-------------------------------------------------------------------------
case EXECMODE_SERIAL: // Serial execution
{
        // Create an empty population
        pop_ptr = boost::shared_ptr<GSerialEA>(new GSerialEA());
}
break;

//-------------------------------------------------------------------------
case EXECMODE_MULTITHREADED: // Multi-threaded execution
{
        // Create the multi-threaded population
        boost::shared_ptr<GMultiThreadedEA> popPar_ptr(new GMultiThreadedEA());

        // Population-specific settings
        popPar_ptr->setNThreads(nEvaluationThreads);

        // Assignment to the base pointer
        pop_ptr = popPar_ptr;
}
break;

//------------------------------------------------------------------------------
case EXECMODE_BROKERAGE: // Execution likely with networked consumer
{
        // Create a network consumer and enrol it with the broker
        boost::shared_ptr<GAsioTCPConsumerT<GParameterSet> >
          gatc(new GAsioTCPConsumerT<GParameterSet>(port, 0, serMode));
        GBROKER(Gem::Geneva::GParameterSet)->enrol(gatc);

        if(addLocalConsumer) { // This is mainly for testing and benchmarking
                boost::shared_ptr<GBoostThreadConsumerT<GParameterSet> >
                  gbtc(new GBoostThreadConsumerT<GParameterSet>());
                gbtc->setNThreadsPerWorker(nEvaluationThreads);
                GBROKER(Gem::Geneva::GParameterSet)->enrol(gbtc);
        }

        // Create the actual broker population
        boost::shared_ptr<GBrokerEA> popBroker_ptr(new GBrokerEA());

        // Assignment to the base pointer
        pop_ptr = popBroker_ptr;
}
break;

//------------------------------------------------------------------------------
default:
{
   glogger
   << "In main(): Received invalid parallelization mode "
   <<  parallelizationMode << std::endl
   << GEXCEPTION;
}
break;
}

/**************************************************************************/
// Create a factory for GFunctionIndividual objects and perform
// any necessary initial work.
GFunctionIndividualFactory gfi("./config/GFunctionIndividual.json");

// Create the first set of parent individuals with random initialization
std::vector<boost::shared_ptr<GFunctionIndividual> > parentIndividuals;
for(std::size_t p = 0 ; p<nParents; p++) {
   parentIndividuals.push_back(gfi.get<GFunctionIndividual>());
}

/**************************************************************************/
// Create an instance of our optimization monitor

boost::shared_ptr<progressMonitor> // demo function is only known to individual
   pm_ptr(new progressMonitor(parentIndividuals[0]->getDemoFunction()));

```

```

pm_ptr->setProgressDims(xDim, yDim);
pm_ptr->setFollowProgress(followProgress); // Shall we take snapshots ?
pm_ptr->setXExtremes(gfi.getMinVar(), gfi.getMaxVar());
pm_ptr->setYExtremes(gfi.getMinVar(), gfi.getMaxVar());

/****************************************************************************/
// Now we have suitable populations and can fill them with data

// Add individuals to the population. Many Geneva classes, such as
// the optimization classes, feature an interface very similar to std::vector.
for(std::size_t p = 0 ; p<parentIndividuals.size(); p++) {
  pop_ptr->push_back(parentIndividuals[p]);
}

// Specify some general population settings
pop_ptr->setPopulationSizes(populationSize,nParents);
pop_ptr->setMaxIteration(maxIterations);
pop_ptr->setMaxTime(boost::posix_time::minutes(maxMinutes));
pop_ptr->setReportIteration(reportIteration);
pop_ptr->setRecombinationMethod(rScheme);
pop_ptr->setSortingScheme(smode);
pop_ptr->registerOptimizationMonitor(pm_ptr);

// Perform the actual optimization
pop_ptr->optimize();

/****************************************************************************/
// Do something with the best individual found
boost::shared_ptr<GFunctionIndividual> p =
   pop_ptr->getBestIndividual<GFunctionIndividual>();

// Here you can do something with the best individual ("p") found.
// We simply print its content here, by means of an operator<< implemented
// in the GFunctionIndividual code.
std::cout
<< "Best result found:" << std::endl
<< p << std::endl;

/****************************************************************************/
// Terminate
return(0);
```

The code has been taken from the `main()` function.

Chapter 24.

Unified Access to Optimization Algorithms

We have seen in section 23.4 that the direct instantiation of optimization algorithms can be somewhat tedious, as different parallelization modes have been implemented as seperate classes[1]. The situation gets worse if a user wants to be able to switch between different algorithms, e.g. using a command line argument or a setting in a configuration file.

This chapter introduces the `Go2` class, which not only facilitates access to the implemented algorithms in all of their parallelization modes, but in addition also allows to "chain" algorithms, making the best result of one algorithm the input of another algorithm. `Go2` also handles command line arguments, which it collects from the available consumers and optimization algorithms. All of this makes the handling of Geneva's algorithms far easier in every-day usage scenarios.

Key points: (1) The `Go2` class provides users with easy access to the implemented algorithms with just a few lines of code. (2) Individuals can be added to `Go2` with the `push_back()` function or through the provision of a factory class (3) More than one optimization algorithm can be added to `Go2` with the `&` sign (4) Algorithms can either be added through (smart-pointers to) optimization algorithm objects or through placeholders (5) Where placeholders are used, the parallelization mode can be set dynamically on the command line.

In a way, this chapter is the culmination point of the entire manual, as it is the `Go2` class that provides users with the easiest access to Geneva's optimization capabilities and even adds further functionality, such as the chaining of algorithms. Still, the description will be relatively short, as `Go2` does most work automatically for you behind the scenes.

24.1. The `main()` function

With `Go2`, the `main()` function can be reduced to just a few lines of code. Listing 24.1 shows a complete example. It should be very visible that it significantly reduces the code overhead compared to listing 23.1[2].

[1] ...with the rationale that a "one size fits all" approach to parallelization for all available optimization algorithms would lead to the neglection of many opportunities.

[2] which doesn't even show the entire `main()` function ...

Listing 24.1: A typical `main()` function, as implemented with the `Go2` class

```
#include "geneva/Go2.hpp"
#include "geneva-individuals/GFunctionIndividual.hpp"

using namespace Gem::Geneva;

int main(int argc, char **argv) {
   Go2 go(argc, argv, "./config/Go2.json");

   //---------------------------------------------------------------------------
   // Client mode
   if(go.clientMode()) {
               return go.clientRun();
   } // Execution will end here in client mode

   //---------------------------------------------------------------------------
   // As we are dealing with a server, register a signal handler that allows us
   // to interrupt execution "on the run"
   signal(SIGHUP, GObject::sigHupHandler);

   //---------------------------------------------------------------------------
   // Create a factory for GFunctionIndividual objects and perform initial work.
   boost::shared_ptr<GFunctionIndividualFactory>
         gfi_ptr(new GFunctionIndividualFactory("./config/GFunctionIndividual.json"));

   // Add a content creator so Go2 can generate its own individuals, if necessary
   go.registerContentCreator(gfi_ptr);

   // Add a default optimization algorithm to the Go2 object. This is optional.
   // Indeed "ea" is the default setting anyway. However, if you do not like it, you
   // can register another default algorithm here, which will then be used, unless
   // you specify other algorithms on the command line. You can also add a smart
   // pointer to an optimization algorithm here instead of its mnemonic.
   go.registerDefaultAlgorithm("ea");

   // Perform the actual optimization
   boost::shared_ptr<GFunctionIndividual> p
     = go.optimize<GFunctionIndividual>();

   // Here you can do something with the best individual ("p") found.
   std::cout << "Best result found:" << std::endl << p << std::endl;
}
```

24.1.1. Go2 Instantiation and Client Mode

We will now go through this example in detail. In the first line, the `Go2 go` object is instantiated. It will read some configuration options from file and makes sure to parse the command line for further options. Command line options are not only taken from `Go2` itself, but are assembled from registered optimization algorithms and consumers. Their addition happens behind the scenes, without the need for user-interaction.

Note that, if `./config/Go2.json` does not exist, it will be automatically created for you, provided that a directory `./config/` exists. `Go2` will not create the directory for you, as this would imply the danger of cluttering your file system with unwanted directories (imagine a very long path). An existing configuration file with the same name will be kept intact.

We then ask whether the program has been called in client- or server-mode[3]. In client-mode, the program will try to contact the server behind the scenes. No user-interaction is needed for this. Users will need to provide the IP and port of the server on the command-line, though. A typical call to a *networked client* with the `main()` function from listing 24.1 is shown in listing 24.4.

Listing 24.2: Executing a networked client with the `main()` function of listing 24.1

```
/home/developer> ./myNetworkedClient --client --ip=192.168.0.1 --port=12345
```

This assumes that the server can be reached via the ip $192.168.0.1$ with port 12345. The `-client` switches on the client mode. The program can thus easily be submitted to worker nodes in a batch submission system, and can also be run over wide-area networks. The only condition is that the server can be reached from the client, when it initiates a connection. Geneva works in pull mode.

A networked client will return from main, once `go.clientRun()` returns.

24.1.2. Installing a signal handler

It is now possible to interrupt an optimization run by sending it a `SIGHUP` signal on Linux (and most Unix-systems) or a `CTRL_BREAK_EVENT` on Microsoft Windows. A signal handler for both events can be installed transparently by adding the following line to the start of your `main()` function:

Listing 24.3: Registering a signal handler to cathc hang-up signals

```
signal(G_SIGHUP, GObject::sigHupHandler);
```

Here, `G_SIGHUP` is a define for `SIGHUP` on Linux and for `CTRL_BREAK_EVENT` on Microsoft Windows. On a Linux-system, it is then possible to interrupt execution using the command

Listing 24.4: Terminating a Geneva run, when a signal handler was installed

```
/home/developer> killall -HUP myNetworkedClient
```

Note that the signal is evaluated only once per evaluation. Currently running evaluations of a given iteration will not be interrupted. However, `Go2` will make an attempt to recover all data up to the current iteration, when the `HUP`-signal was received. See `01_GSimpleOptimizer` for an example on how to register the signal handler.

[3]Client-mode may e.g. refer to a networked client, using `Boost.Asio` for the transfer of data.

24.1.3. Content Creators and Factories

We have discussed factory classes in section 33.4. We now use the factory made available with the `GFunctionIndividual` individual, which comes together with the Geneva distribution. The factory again reads its configuration options from a configuration file.

`Go2` accepts so called “content creators”, which are just factories for individuals. Once registered, `Go2` takes care itself of adding the required number of individuals to its algorithms.

As we can create any desired number of `GFunctionIndividual` objects just through a call to `gfi()` (where `gfi` is the name of the `GFunctionIndividualFactory` object) and `Go2` uses a `std::vector<>` interface, we could also have added the individuals to the `Go2` object with calls to `go.push_back(gfi())` manually.

24.1.4. Creating and adding an Optimization Algorithm Object

We are now ready to throw a first optimization algorithm into the cauldron. The following options are available:

- We could do nothing. `Go2` uses Evolutionary Algorithms as default algorithm, when no other algorithms were specified. Note that we may change the default algorithm with a call to the `Go2::registerDefaultAlgorithm()` function, which accepts a mnemonic for the chosen default algorithm (e.g. “ea” for Evolutionary Algorithms or “swarm” for particle swarm optimization). This is what we have done in listing 24.1.
- We could have added an algorithm like this: `go & "swarm"`. This would add a swarm algorithm instead of an evolutionary algorithm. Note that, instead of a mnemonic, we may also add smart pointers to optimization algorithms to `Go2` using the “&” operator. This allows us a stronger control over the algorithms that were added to `Go2`.
- We may add additional algorithms on the command line, again using the mnemonics. See the `-help` switch of `Go2`. Note that algorithms added in `main()` take precedence over those added on the command line.

Just like in the case of the `Go2` class itself, configuration files will be created for you automatically in the specified path, if they do not exist yet for a given optimization algorithm. Default values will then be used.

24.1.5. Peforming the actual Optimization

We can now start the actual optimization run, with a call to `go.optimize<GFunctionIndividual>()`. It will return a smart pointer to the best `GFunctionIndividual` object found. Note that Geneva internally uses the `GObject` base class of `GFunctionIndividual` for all of its actions, which is however automatically converted to the target class for you.

Once we have received the final result, we can further process it or output results, depending on what the user wishes to do in this part of the program.

24.1.6. Go2's Help Mode

`Go2` offers you some built-in help for its command-line options. Simply call the program with `--help` on the command line.

24.2. Adding further Algorithms

We have so far only used a single optimization algorithm. You can do much more with, Go2, though. As one example, the class allows you to "chain" different optimization algorithms. The best result of one algorithm then becomes the input of the next. Listing 24.5 shows an example.

Listing 24.5: Optimization algorithms can be chained using the & sign.

```
// Create an evolutionary algorithm in brokerage mode
GEvolutionaryAlgorithmFactory
    ea("./config/GEvolutionaryAlgorithm.json", EXECMODE_BROKERAGE);

// Create a gradient descent in multi-threaded mode
GGradientDescentFactory
    gd("./config/GGradientDescentFactory.json", EXECMODE_MULTITHREADED);

// Chain the two algorithms
go & ea() & gd();

// Perform the actual optimization
boost::shared_ptr<GFunctionIndividual> p = go.optimize<GFunctionIndividual>();
```

As we can see, it is possible to *chain* algorithms with the help of the `&` operator. Algorithms do not even have to share the same parallelization mode.

Note that `ea()` and `gd()` emit `boost::shared_ptr<>` smart pointers to algorithms. You could thus also have captured the output of `ea()` or `gd()` and perform some further, manual configuration work with them. The smart pointers can then again be added to `go` with the `&` operator.

But it can get even simpler. If you do not want to do any manual configuration of the optimization algorithms and rather modify the options in the configuration files, you can replace the explicit instantiation of algorithms with a placeholder. If we add in a swarm algorithm just for kicks (which might not make sense), then listing 24.5 becomes

Listing 24.6: Listing 24.5 can be further simplified when using place holders

```
// Chain the algorithms
go & "ea" & "swarm" & "gd";
// Perform the actual optimization
boost::shared_ptr<GFunctionIndividual> p = go.optimize<GFunctionIndividual>();
```

This has the big advantage that you can choose the parallelization mode dynamically on the command line, using the `-p` switch (compare the `--help` switch for all available options). Note that there is also a slight catch in that you cannot currently assign different parallelization modes to different algorithms from the command line.

You can also choose to mix both methods by creating and configuring some algorithms manually, as shown in listing 24.5 and adding others with the placeholders shown in listing 24.6. A parallelization mode specified on the command line will then only apply to those algorithms that were added with place holders.

Chapter 25.

Optimization Monitors

The previous chapters have concentrated on means to optimize a given problem. Over the course of an optimization run, Geneva emits information, such as the best result of a given iteration. Geneva also emits a file in ROOT format which can be used to visualize the progress of the optimization run (compare figure 9.3 and appendix C).

There are times, however, where this information isn't enough, and you want to get a deeper insight into what happens inside of the algorithm during the optimization, or emit additional intermediate results. For this purpose, Geneva implements the `GOptimizationMonitorT` class, as well as derivatives for each implemented optimization algorithm. They can be registered with the algorithm and will henceforth emit the desired information.

This has been used for example to write out the best picture during each iteration in the Mona Lisa example of section 9.1.1 (compare also figures 9.1 and 9.2). This chapter describes how to create a custom optimization monitor.

Key points: (1) A `doInfo()` function is called by `GOptimizationAlgorithmT<>` in different modes before and after the optimization cycle, and for every iteration (2) `doInfo()` uses the services of the `GOptimizationMonitorT<>` class and its derivatives (3) Custom optimization monitors may be loaded into the optimization algorithm. (4) They have access to all public information of the current algorithm (5) Default optimization monitors exist for all "stock" optimization algorithms implemented in Geneva. (6) "Pluggable optimization monitors" allow easy access to individual-specific information (7) A number of predefined pluggable optimization monitors exist

25.1. Internal Architecture

We want to look back at listing 17.1, which has been reproduced in this chapter in listing 25.1. Three phases can be distinguished: Initialization, processing and finalization. Correspondingly, the base class of all optimization algorithms, `GOptimizationAlgorithmT<>`, contains a member function `doInfo()`, which can be called in the three modes `INFOINIT`, `INFOEND` and `INFOPROCESSING`. The first two modes are only called once, `INFOPROCESSING` is called in every iteration. `doInfo()` is responsible for emitting (possibly user-defined) information on the progress of the optimization run.

Listing 25.1: `doInfo()` is called before and after the main loop, as well as for every iteration.

```

// [...]
// Output any initial information for the user
if(reportIteration) doInfo(INFOINIT);

// Initialize the optimization run
init();

do {
   // The actual business logic
   bestCurrentFitness = cycleLogic();

   // We want to provide feedback to the user in regular intervals.
   if(reportIteration && (iteration%reportIteration==0)) doInfo(INFOPROCESSING);

   // update the iteration counter
   iteration++;
} while(!halt());

// Clean up
finalize();

// Finalize the info output
if(reportIteration) doInfo(INFOEND);

// [...]
```

Geneva has been built as modular as possible, so many parts of the library can be adapted to the users' needs. Consequently, `doInfo()` does not emit any information itself[1], but uses the help of the `GOptimizationMonitorT<>` class template.

Optimization algorithms can store `boost::shared_ptr<>` smart pointers, holding `GOptimizationMonitorT<>` objects or their derivatives. `doInfo(mode)` will call `GOptimizationMonitorT<>::informationFunction(...)`, which receives the desired information mode (`INFOINIT`, `INFOEND` and `INFOPROCESSING`) and a constant `this` pointer of the current optimization algorithm as argument. Note that `GOptimizationMonitorT<>` is implemented as a member class of `GOptimizationAlgorithmT<>`, so that the `this` pointer will appear as a `GOptimizationAlgorithmT<> *`, i.e. a pointer to the base class of the corresponding optimization algorithm.

Internally, `informationFunction()` calls three different functions, depending on the chosen information mode: `firstInformation()` is called for the `INFOINIT` mode, `lastInformation()` for the `INFOEND` mode, and `cycleInformation()` is called in all or some

[1]. . . although it has been declared `virtual`, so you *could* overload it in derived classes.

iterations of the optimization process. How often the function is called is user-defined – listing 25.1 shows the details.

All three functions are virtual, so that users may derive custom optimization monitors from `GOptimizationMonitorT<>` and load them into their algorithm. Their overloaded information functions can reveal every detail of the current optimization algorithm. Note that they will have to perform a cast internally to the target algorithm, as the information functions receive a base pointer to `GOptimizationAlgorithmT<>` only.

`GOptimizationAlgorithmT<>` implements a suitable `GOptimizationMonitorT<>` class, which is loaded by default, so in general no user interaction is required. However, it only outputs the best result of the current iteration.

25.2. Specifics for the Algorithms

In order to facilitate access to the optimization monitor, each of the "stock" optimization algorithms of the Geneva library collection implements its own version of the optimization monitor, as a derivative of `GOptimizationMonitorT<>`. As an example, `GBaseEA` as the base class of all Evolutionary Algorithms, implements the `GEAOptimizationMonitor` class. The three information functions described in section 25.1 are overloaded. They cast the `GOptimizationAlgorithmT<> *` base pointers to `GBaseEA *`, so the monitor can get easy access to the internals of the algorithm. As `GEAOptimizationMonitor` is implemented as an embedded class of `GBaseEA`, which is loaded by the object itself, fast static casts can be used.

If you intend to write your own optimization monitor, you may have to derive your own class from `GOptimizationMonitorT<>` and overload the three functions `firstInformation()`, `lastInformation()` and `cycleInformation()`.

A complete usage example, including the development of a custom optimization monitor, will be shown in chapter 26.

25.3. Pluggable Optimization Monitors

Geneva comprises a second mechanism for monitoring, which is designed to observe individuals more than entire algorithms. These so called "pluggable optimization monitors" carry less overhead, and there is a number of general purpose monitors already predefined for common tasks.

- `GCollectiveMonitorT<>` allows to aggregate different monitors
- `GProgressPlotterT<>` monitor a given set of variables inside of all or of the best individuals of a population, creating a graphical output using ROOT. It supports floating point types only. double and float values may not be mixed.
- `GAllSolutionFileLoggerT<>` allows to log all candidate solutions found to a file. **Note that the file may become very large!** Results are output in the following format: `param1`

param2 ... param_m eval1 eval2 ... eval_n. By default, no commas are printed between values.

- GNAdpationsLoggerT<> allows to log the number of adaptions made inside of adaptors to a file. This is mostly needed for debugging and profiling purposes. The number of adaptions made is a good measure for the current adaption probability.
- GAdaptorPropertyLoggerT<> allows to log chosen properties of adaptors. Such properties are limited to numeric entities, that may be converted to double. This monitor thus allows direct measurement of the adaption probability.

If you intend to write your own pluggable monitor, we recommend to have a look at these predefined classes (see file GPluggableOptimizationMonitorsT.hpp). Further monitors will be added over time.

Listing 25.2 shows how to register two monitors with go, with the help of the collective monitor.

Listing 25.2: Registering a pluggable optimization monitor

```
boost::shared_ptr<GCollectiveMonitorT<GParameterSet> >
   collectiveMonitor_ptr(new GCollectiveMonitorT<GParameterSet>());

boost::shared_ptr<GAllSolutionFileLoggerT<GParameterSet> >
  allsolutionLogger_ptr(new GAllSolutionFileLoggerT<GParameterSet>(logAll))
allsolutionLogger_ptr->setUseTrueFitness(false);
allsolutionLogger_ptr->setShowValidity(true);

boost::shared_ptr<GAdaptorPropertyLoggerT<GParameterSet, double> >
  sigmaLogger_ptr(
     new GAdaptorPropertyLoggerT<GParameterSet, double>(
        logSigma,
        "GDoubleGaussAdaptor",
        "sigma"
     )
  );
sigmaLogger_ptr->setMonitorBestOnly(false);

collectiveMonitor_ptr->registerPluggableOM(allsolutionLogger_ptr);
collectiveMonitor_ptr->registerPluggableOM(sigmaLogger_ptr);

if(collectiveMonitor_ptr->hasOptimizationMonitors()) {
   go.registerPluggableOM(
         boost::bind(
               &GCollectiveMonitorT<GParameterSet>::informationFunction
               , collectiveMonitor_ptr
               , _1
               , _2
         )
   );
}
```

The code is taken from example 13_GPluggableOptimizationMonitors.

Chapter 26.

A More Complex Example

The purpose of this chapter is to demonstrate many of the past chapters' techniques on a practical example. It builds on the introduction of chapter 11.

There will be an easy start, as we will first refine the `GParaboloid2D` example, so that it can handle an arbitrary amount of dimensions, and an additional target function alongside the parabola. We will then develop a factory class for the associated `GFMinIndividual`, in order to read configuration options from file. In the next step, we will set up a new, `Go2`-based `main()` function and present a custom optimization monitor, which produces a graphical view of the optimization progress.

Over the course of this chapter, a complete environment is created that can well serve as a starting point for your own optimization problems[1].

The entire example, called `GFunctionMinimizer`, can be found in the Geneva distribution, in the directory `examples/geneva/07_GFunctionMinimizer`.

Key points: (1) `GFMinIndividual` further develops the idea of an individual whose target function can be switched. It is mainly used for profiling and efficiency tests. (2) Individuals can add their own configuration options to a `GParserBuilder` object, which will then automatically create and parse configuration files. (3) This facilitates the creation of factories that set up individuals according to variables specified in the configuration files (4) Optimization Monitors can be added "on demand" to optimization algorithms (5) They may depend on the internals of a given algorithm. Creating a "one size fits all" monitor is thus difficult (6) Setting up the `main()` function is facilitated by the `Go2` class

26.1. Setting Up the Individual

In this section, we want to extend listing 11.1, so that a new individual `GFMinIndividual` becomes available. It can handle an arbitrary number of dimensions, as well as more then one target function. We will also discuss how the configuration information can be serialized. If you have skipped

[1] Another option would be the example `examples/10_GStarter`, which is more refined than example 7 and can in addition be compiled independent from the main Geneva source tree.

chapter 11, we recommend that you read it first, as it contains a number of explanations which will not be replicated in this chapter. Listing 26.1 shows that, compared to listing 11.1, the declaration of `GFMinIndividual` hasn't changed much.

Listing 26.1: The declaration of the GFMinIndividual class

```
namespace Gem {
namespace Geneva {

class GFMinIndividual : public GParameterSet {
public:
        GFMinIndividual();
        GFMinIndividual(const GFMinIndividual&);
        virtual ~GFMinIndividual();

        const GFMinIndividual& operator=(const GFMinIndividual&);

        void setTargetFunction(targetFunction);
        targetFunction getTargetFunction() const;

        virtual void
           addConfigurationOptions(Gem::Common::GParserBuilder&, const bool&);

        double getAverageSigma() const;

protected:
        virtual void load_(const GObject*);
        virtual GObject* clone_() const;

        virtual double fitnessCalculation();

private:
        friend class boost::serialization::access;
        template<class Archive>
        void serialize(Archive & ar, const unsigned int) {
                ar & BOOST_SERIALIZATION_BASE_OBJECT_NVP(GParameterSet)
                   & BOOST_SERIALIZATION_NVP(targetFunction_);
        }

        double parabola(const std::vector<double>& parVec) const;
        double noisyParabola(const std::vector<double>& parVec) const;

        targetFunction targetFunction_;
};

} /* namespace Geneva */
} /* namespace Gem */

BOOST_CLASS_EXPORT_KEY(Gem::Geneva::GFMinIndividual)
// Remember that the .cpp file needs to hold a
// corresponding BOOST_CLASS_EXPORT_IMPLEMENT() statement!
```

There are still default- and a copy constructors as well as a destructor. An additional variable has been added, allowing to switch between two functions `parabola()` and `noisyParabola()` (compare sections A.1 and A.2 in the appendix for an illustration of these functions). The implementation of both functions in private helper functions is trivial. They are called from within `fitnessCalculation()`.

The `targetFunction_` variable can be set and retrieved using two "getter" and "setter" functions. `targetFunction_` itself needs to be initialized by the constructor.

There are also `load_()` and `clone_` functions. `load_()` takes care of loading the parent class'es data (by calling its `load_()` function), and also copies `targetFunction_`. `clone_` uses the copy constructor to create an exact copy of this object.

`targetFunction_` has also been added to the `serialize()` function, as we need to give remote entities information on the target function to call inside of `fitnessCalculation()`.

If you have read chapter 11, then there cannot have been many surprises for you up till now, and we do not need to further refine the structure discussed up till here. The only notable additions over listing 11.1 are the `addConfigurationOptions()` and the `getAverageSigma()` function.

`getAverageSigma()` will be used in section 26.3 to implement a custom optimization monitor[2].

The reasoning behind `addConfigurationOptions()` is described in section 33.3. Our goal is to facilitate the creation of a factory class for `GFMinIndividual`, which reads all its data from a configuration file and configures the `GFMinIndividual` objet as required.

All of Geneva's core optimization classes are all equipped with this function. Its purpose is to add local configuration variables to a configuration file and retrieve parsed values for these variables. Listing 26.2 shows the function's implementation.

Listing 26.2: The implementation of the GFMinIndividual::addConfigurationOptions() function

```
namespace Gem {
namespace Geneva {

void GFMinIndividual::addConfigurationOptions (
        Gem::Common::GParserBuilder& gpb
        , const bool& showOrigin
) {
        std::string comment;

        // Call our parent class'es function
        GParameterSet::addConfigurationOptions(gpb, showOrigin);

        // Add local data
        comment = ""; // Reset the comment string
        comment += "Specifies which target function should be used:;";
        comment += "0: Parabola;";
        comment += "1: Noisy Parabola;";

        if(showOrigin) comment += "[GFMinIndividual]";
```

[2]This happens for illustration reasons, as the pluggable optimization monitors already contain a facility to extract sigma

```
        gpb.registerFileParameter<targetFunction>(
                "targetFunction" // The name of the variable
                , GO_DEF_TARGETFUNCTION // The default value
                , boost::bind(
                        &GFMinIndividual::setTargetFunction
                        , this
                        , _1
                  )
                , Gem::Common::VAR_IS_ESSENTIAL
                , comment
        );
}

} /* namespace Geneva */
} /* namespace Gem */
```

Essentially, the local `targetFunction_` option is added to a `GParserBuilder` object, which has been passed to the function as a parameter. We do not pass a reference to the variable, but rather ask `registerFileParameter<targetFunction>()` to call a call-back function with the parsed parameter.

`addConfigurationOptions()` also passes the object to the `GParserBuilder` parent class, which will recursively add its own options and those of its parent classes. This way any change to a class's architecture is immediately reflected in the configuration files.

If you have watched carefully, you will have noticed that listing 26.1 shows no sign of the dimension of the target function, and also the lower and upper boundaries of the variables have vanished from the declaration.

In order to understand this, please think back at the architecture of individuals. They are essentially `STL`-containers, to which parameter objects can be added. Compare figure 15.2 for the details. In chapter 11 we had asked the constructor to equip parameter objects with adaptors and add them to the class.

With the current example, however, it is more flexible to let an external entity configure the parameter objects and add them to the `GFMinIndividual` object. This external entity – a factory class – will be described in the next section.

26.2. Creating a Factory

We have already learned about the `GFactoryT<>` class template in section 33.4. We will now use its services to create a factory for `GFMinIndividual` that can read its options from a configuration file. Listing 26.3 shows the class declaration of the factory.

Listing 26.3: A factory for `GFMinIndividual` objects

```
namespace Gem {
namespace Geneva {

```

```
class GFMinIndividualFactory
        : public Gem::Common::GFactoryT<GFMinIndividual>
{
public:
        GFMinIndividualFactory(const std::string&);
        virtual ~GFMinIndividualFactory();

protected:
        virtual boost::shared_ptr<GFMinIndividual> getObject_(
           Gem::Common::GParserBuilder&
           , const std::size_t&
        );
        virtual void describeLocalOptions_(Gem::Common::GParserBuilder&);
        virtual void postProcess_(boost::shared_ptr<GFMinIndividual>&);

private:
        GFMinIndividualFactory();

        double adProb_;
        double sigma_;
        double sigmaSigma_;
        double minSigma_;
        double maxSigma_;
        std::size_t parDim_;
        double minVar_;
        double maxVar_;
};

} /* namespace Geneva */
} /* namespace Gem */
```

We see two public constructors. The "standard" constructor accepts the name of a configuration file as argument and initializes local data. The destructor does essentially nothing. A default constructor is intentionally private and undefined, so this factory can *only* be instantiated with the name of a configuration file.

The only important functions are `getObject_()`, `describeLocalOptions_()` and `postProcess_()`. It is here that `GFMinIndividual` objects are created and configured. The functions have been overloaded from the parent class `GFactoryT<Gem::Common::GFMinIndividual>`. We will discuss them one by one.

Listing 26.4: The `GFMinIndividualFactory::getObject_()` function

```
boost::shared_ptr<GFMinIndividual> GFMinIndividualFactory::getObject_(
        Gem::Common::GParserBuilder& gpb
        , const std::size_t& id
) {
        // Will hold the result
        boost::shared_ptr<GFMinIndividual> target(new GFMinIndividual());

        // Make the object's local configuration options known
        target->addConfigurationOptions(gpb, true);

        return target;
}
```

Listing 26.4 shows the implementation of the `getObject_()` function. It first creates a default-constructed `GFMinIndividual` object inside of a `boost::shared_ptr<>` smart pointer. It then gives it an opportunity to register any local configuration options (compare listing 26.2) before returning the smart pointer to the caller.

Listing 26.5: The `describeLocalOptions_()` function

```
void GFMinIndividualFactory::describeLocalOptions_(Gem::Common::GParserBuilder& gpb) {
        using namespace Gem::Courtier;

        // Describe our own options
        std::string comment;

        comment = "";
        comment += "The probability for random adaptions of values;";
        gpb.registerFileParameter<double>(
                "adProb"
                , adProb_
                , GFI_DEF_ADPROB
                , Gem::Common::VAR_IS_ESSENTIAL
                , comment
        );

        comment = "";
        comment += "The sigma for gauss-adaption in ES;";
        gpb.registerFileParameter<double>(
                "sigma"
                , sigma_
                , GFI_DEF_SIGMA
                , Gem::Common::VAR_IS_ESSENTIAL
                , comment
        );

        // Registration of further configuration options has been erased

        // Allow our parent class to describe its options
        Gem::Common::GFactoryT<GFMinIndividual>::describeLocalOptions_(gpb);
}
```

Listing 26.5 shows part of the implementation of the `GFMinIndividualFactory::describeLocalOptions_()` function. Its purpose is to register local options of the factory with the `GParserBuilder` object. Not all registered options are shown for space reasons. Compare listing 26.3 for all options that need to be added. Typically these are variables that are needed to configure the target objet to be created by the factory. They will be read from (and written to, if needed) a configuration file, once they have been registered.

As soon as these options have become available (i.e. have been read from file), the factory needs to post-process the `GFMinIndividual` object. This happens in the `postProcess_()` function and is described in listing 26.6.

Listing 26.6: The `GFMinIndividualFactory::postProcess_()` function

```
void GFMinIndividualFactory::postProcess_(boost::shared_ptr<GFMinIndividual>& p) {
  // Set up a collection with parDim_ values
  boost::shared_ptr<GConstrainedDoubleCollection>
         gcdc_ptr(new GConstrainedDoubleCollection(parDim_, minVar_, maxVar_));

  // Randomly initialize
  gcdc_ptr->randomInit();

  boost::shared_ptr<GDoubleGaussAdaptor>
        gdga_ptr(new GDoubleGaussAdaptor(sigma_, sigmaSigma_, minSigma_, maxSigma_));
  gdga_ptr->setAdaptionProbability(adProb_);
  gcdc_ptr->addAdaptor(gdga_ptr);

  // Make the parameter collection known to this individual
  p->push_back(gcdc_ptr);
}
```

The function receives the smart pointer that was created in the `getObject_()` function in listing 26.4. It then creates a parameter object and equips it with an adaptor. Once done, the parameter object is added to the `GFMinIndividual` object.

At this point the creation of the individual is done and it can be returned to the user. The user will typically receive it from the factory by means of an `operator()`, i.e. he can treat the factory like a normal function. See the description in section 26.4 for an example on how to use the factory. Note that the user of the factory will not get to see the three functions implemented above, as they are protected. They are for internal use by the factory only, but need of course to be implemented by the author of the factory.

26.3. Adding a Custom Optimization Monitor

By default, Geneva's optimization monitors will only produce information about the progress of the optimization, namely the fitness of the best individual(s). Sometimes this is not enough and you might want to extract further information about the progress of the optimization run. This may be for debugging purposes, or you might want to output additional data in every iteration.

Note that the creation of an optimization monitor is entirely optional, but can help with some advanced scenarios.

As information will generally be extracted from the optimization algorithm, the creation of monitors cannot be seen independently of the optimization strategy. In this section, we want to lead you through the creation of an optimization monitor for an Evolutionary Algorithm ("EA"). As was discussed in

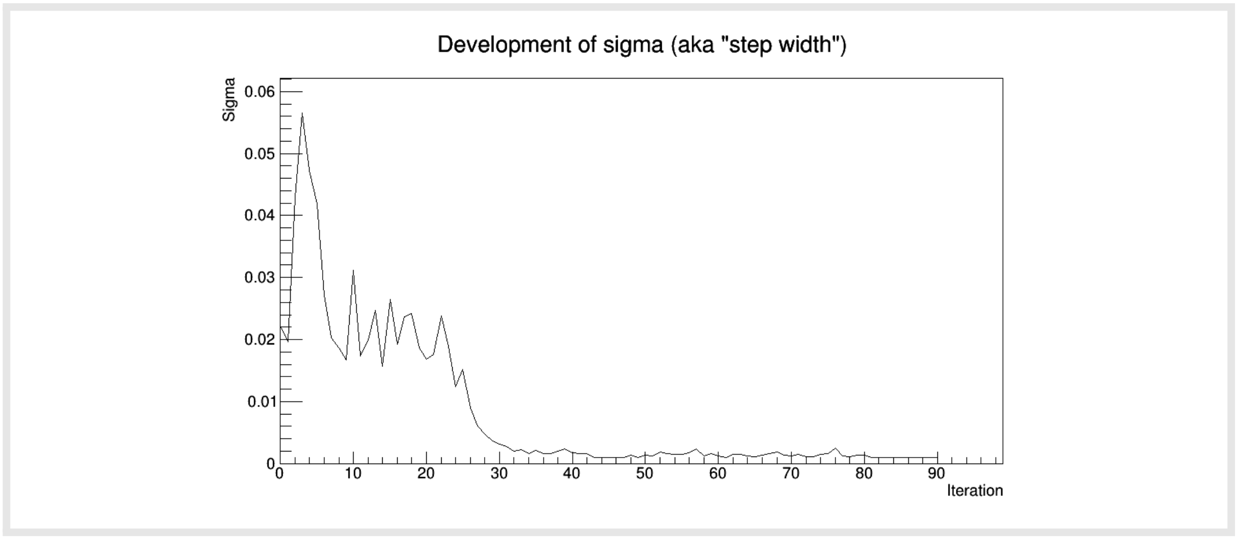


Figure 26.1.: *Geneva's default optimization monitor would only write out the progress of the optimization run in ROOT format. The figure shows the progress for a standard 2D parabola, as output by the optimization monitor discussed in this chapter*

chapters 4 and 18, mutations are carried out by the adaptors assigned to the parameter objects stored in individuals. The most common mutation of Evolutionary Strategies uses gaussian-distributed random numbers, which are added on a feature vector of double values. A gaussian has a σ value, denoting the "width" of the curve (compare figure 4.2). In this section, we will now try to extract the current σ value of the adaptor assigned to the `GConstrainedDoubleCollection` of the `GFMinIndividual`. Once the optimization run is over, we want to create a plot of the σ values of each iterations' best individual.

The example is based on the discussion of chapter 25. Note that our implementation applies to Evolutionary Algorithms only, so we derive from `GEAOptimizationMonitor`. We also need to overload the `firstInformation()`, `cycleInformation` and `lastInformation()` function. The declaration of `GSigmaMonitor` is shown in listing 26.7.

Listing 26.7: The declaration of GSigmaMonitor

```

class GSigmaMonitor
    :public GBaseEA::GEAOptimizationMonitor
{
public:
    GSigmaMonitor(const std::string fileName);
    GSigmaMonitor(const GSigmaMonitor& cp);
    virtual ~GSigmaMonitor();

protected:
    virtual void firstInformation(GOptimizationAlgorithmT<GParameterSet> * const goa);
    virtual void cycleInformation(GOptimizationAlgorithmT<GParameterSet> * const goa);
    virtual void lastInformation(GOptimizationAlgorithmT<GParameterSet> * const goa);
```

```

  virtual void load_(const GObject* cp);
  virtual GObject* clone_() const;

private:
  GSigmaMonitor();

  std::string fileName_; ///< The name of the output file

  // Ease recording of essential information
  Gem::Common::GPlotDesigner gpd_;
  boost::shared_ptr<Gem::Common::GGraph2D> progressPlotter_;
  boost::shared_ptr<Gem::Common::GGraph2D> sigmaPlotter_;
};
```

The standard constructor accepts the name of a file to which the result should be written and initializes local variables. The copy constructor and destructor should be self-explanatory. `load_()` and `clone_()` follow the same rules as for any other optimization-related Geneva class and will not be further explained here. The default constructor has been disabled by labelling it private and keeping it undefined.

Recorded data is stored directly in plotters associated with Geneva's `GPlotDesigner` class (`progressPlotter_` and `progressPlotter_`). Both will later be added to a `GPlot-Designer` object (`gpd_`), which takes care of the output of results. See chapter 33.9 for further information on the plot designer.

The only functions that need further explanations are thus `firstInformation()`, `cycle-Information()` and `lastInformation()`. We will look at them one by one.

Listing 26.8: The implementation of `firstInformation()`

```
void GSigmaMonitor::firstInformation(GOptimizationAlgorithmT<GParameterSet> * const goa) {
  // Initialize the plots we want to record
  progressPlotter_->setPlotMode(Gem::Common::CURVE);
  progressPlotter_->setPlotLabel("Fitness as a function of the iteration");
  progressPlotter_->setXAxisLabel("Iteration");
  progressPlotter_->setYAxisLabel("Best Result (lower is better)");

  sigmaPlotter_->setPlotMode(Gem::Common::CURVE);
  sigmaPlotter_->setPlotLabel("Development of sigma (aka \\\"step width\\\")");
  sigmaPlotter_->setXAxisLabel("Iteration");
  sigmaPlotter_->setYAxisLabel("Sigma");

  gpd_.setCanvasDimensions(P_XDIM, P_YDIM);
  gpd_.registerPlotter(progressPlotter_);
  gpd_.registerPlotter(sigmaPlotter_);

  // We call the parent classes firstInformation function,
  // as we do not want to change its actions
  GBaseEA::GEAOptimizationMonitor::firstInformation(goa);
}
```

Listing 26.8 shows the `firstInformation()` class. It sets up the progress plotters and calls the corresponding parent class'es function. `firstInformation()` is called before the actual optimization starts.

Listing 26.9 is where the σ of the best individual of each iteration is extracted.

Listing 26.9: The implementation of `cycleInformation()`

```cpp
void GSigmaMonitor::cycleInformation(GOptimizationAlgorithmT<GParameterSet> * const goa) {
  // Convert the base pointer to the target type
  GBaseEA * const ea = static_cast<GBaseEA * const>(goa);

  // Extract the requested data. First retrieve the best individual.
  // It can always be found in the first position with evolutionary algorithms
  boost::shared_ptr<GFMinIndividual> p = ea->clone_at<GFMinIndividual>(0);

  // Retrieve the best "raw" fitness and average sigma value
  progressPlotter_->add(
    boost::tuple<double,double>((double)ea->getIteration(), p->fitness())
  );
  sigmaPlotter_->add(
    boost::tuple<double,double>((double)ea->getIteration(),p->getAverageSigma())
  );

  //----------------------------------------------------------------
  // Call our parent class'es function
  GBaseEA::GEAOptimizationMonitor::cycleInformation(goa);
}
```

We know that in evolutionary algorithms the best individual is located in the first position of the population. Hence we can extract the individual and ask it for its average sigma value. This is done by means of the `GFMinIndividual::getAverageSigma()` function, which we had introduced in listing 26.1. So essentially, all necessary work is done by extracting the best individual and asking it for its average σ value. `cycleInformation()` is called in regular intervals during the optimization, according to the user settings[3].

In contrast, `lastInformation()` is called by `GOptimizationMonitorT<>`'s main loop after the optimization has ended. It is here that we want to write out the results. Listing 26.10 shows the very simple details – all work is done by the `GPlotDesigner` object.

Listing 26.10: The implementation of `eaLastInformation()`

```cpp
void GSigmaMonitor::lastInformation(GOptimizationAlgorithmT<GParameterSet> * const goa) {
  // Write out the result
  gpd_.writeToFile(fileName_);

  // We just call the parent classes eaLastInformation function,
  // as we do not want to change its actions
  GBaseEA::GEAOptimizationMonitor::lastInformation(goa);
}
```

[3]By default it will be called in every iteration.

`GPlotDesigner` dynamically creates a pre- and postamble, and injects the dynamic σ measurements in-between. The resulting ROOT script is shown in listing 26.11.

Listing 26.11: The result file created with `GPlotDesigner`

```
{
 gROOT->Reset();
 gStyle->SetCanvasColor(0);
 gStyle->SetStatBorderSize(1);
 gStyle->SetOptStat(0);

 TCanvas *cc = new TCanvas("cc", "cc",0,0,1200,1400);

 TPaveLabel* canvasTitle = new TPaveLabel(0.2,0.95,0.8,0.99, "Progress information");
 canvasTitle->Draw();

 TPad* graphPad = new TPad("Graphs", "Graphs", 0.01, 0.01, 0.99, 0.94);
 graphPad->Draw();
 graphPad->Divide(1,2);

 //=================== Header Section ====================
 double x_array__0[91];
 double y_array__0[91];

 double x_array__1[91];
 double y_array__1[91];

 //=================== Data Section =====================

 x_array__0[0] = 0;    y_array__0[0] = 6.76806;
 x_array__0[1] = 1;    y_array__0[1] = 2.06513;
 x_array__0[2] = 2;    y_array__0[2] = 0.0374993;
 // Entries removed for ease of readability
 x_array__0[88] = 88;  y_array__0[88] = 3.79166e-06;
 x_array__0[89] = 89;  y_array__0[89] = 1.67342e-06;
 x_array__0[90] = 90;  y_array__0[90] = 3.53337e-08;

 x_array__1[0] = 0;    y_array__1[0] = 0.0223039;
 x_array__1[1] = 1;    y_array__1[1] = 0.0195696;
 x_array__1[2] = 2;    y_array__1[2] = 0.0429653;
 // Entries removed for ease of readability
 x_array__1[88] = 88;  y_array__1[88] = 0.001;
 x_array__1[89] = 89;  y_array__1[89] = 0.001;
 x_array__1[90] = 90;  y_array__1[90] = 0.001;

 //=================== Plot Section =====================

 graphPad->cd(1);
 TGraph *graph_0 = new TGraph(91, x_array__0, y_array__0);
 graph_0->GetXaxis()->SetTitle("Iteration");
 graph_0->GetYaxis()->SetTitle("Best Result (lower is better)");
 graph_0->SetTitle("Fitness as a function of the iteration");
```

```
  graph_0->Draw("APL");

  graphPad->cd(2);
  TGraph *graph_1 = new TGraph(91, x_array__1, y_array__1);
  graph_1->GetXaxis()->SetTitle("Iteration");
  graph_1->GetYaxis()->SetTitle("Sigma");
  graph_1->SetTitle("Development of sigma (aka \"step width\")");
  graph_1->Draw("APL");

  graphPad->cd();
  cc->cd();
}
```

The sigma-part of the resulting plot can be seen in figure 26.1. The above code also records the fitness alongside the sigma value.

Assessment

We have now seen that writing a custom optimization monitor is rather easy. The `firstInformation()`, `cycleInformation()`, `lastInformation()` family of functions gets access to the optimization algorithm and can ask it for any information that can be publicly accessed. A difficulty arises through the fact that information is specific to the given optimization algorithm, making it difficult to write a “one size fits all” monitor.

In our example, we have done more than “just” overloading the `ea` family of functions. We have in addition called the parent class’s functions. As a consequence, we have received two output files, one showing the progress of the optimization run, the other the development of the best σ as a function of the iteration.

The next section (26.4) will show you how to set up `main()` and also how to register the optimization monitor with your optimization algorithm.

26.4. Setting up `main()`

With the help of `Go2`, the `main()` function becomes rather simple. It is shown in listing 26.12. We have omitted the `#include` section.

Listing 26.12: The `main()` function, created with the `Go2` class

```
using namespace Gem::Geneva;
namespace po = boost::program_options;

int main(int argc, char **argv) {
   bool printBest = false;
   std::vector<boost::shared_ptr<po::option_description> > od;
   boost::shared_ptr<po::option_description> print_option(
      new po::option_description(
          "print"
          // This allows you say both --print and --print=true
          , po::value<bool>(&printBest)->implicit_value(true)->default_value(false)
          , "Switches on printing of the best result"
       )
    );
    od.push_back(print_option);

    Go2 go(argc, argv, "./config/Go2.json", od);

    //---------------------------------------------------------------------
    // Client mode
    if(go.clientMode()) {
          return go.clientRun();
    }

    //---------------------------------------------------------------------
    // Server mode, serial or multi-threaded execution

    // Create a factory for GFMinIndividual objects and perform
    // any necessary initial work.
    GFMinIndividualFactory gfi("./config/GFMinIndividual.json");
    // Retrieve an individual from the factory and make it known to the optimizer
    go.push_back(gfi());

    // Register an optimization monitor for evolutionary algorithms. This allows
    // the GEvolutionaryAlgorithmFactory to find suitable monitors in the store.
    GOAMonitorStore->setOnce("ea", boost::shared_ptr<GSigmaMonitor> (
      new GSigmaMonitor("./sigmaProgress.C"))
    );

    // Create an evolutionary algorithm in multi-threaded mode
    GEvolutionaryAlgorithmFactory ea(
      "./config/GEvolutionaryAlgorithm.json"
      , EXECMODE_MULTITHREADED
    );
    boost::shared_ptr<GBaseEA> ea_ptr = ea.get<GBaseEA>();

    // Add the algorithm to the Go2 object
    go & ea_ptr;

    // Perform the actual optimization
    boost::shared_ptr<GFMinIndividual> bestIndividual_ptr
      = go.optimize<GFMinIndividual>();

    // Do something with the best result. Here: Simply print it, if requested
    if(printBest) {
       std::cout << "Best individual found has values" << std::endl
       << bestIndividual_ptr << std::endl;
    }
}
```

Most of the details of listing 26.12 have already been explained in chapter 24 (also compare listing 24.1). However, take note of how we have added a custom command line option (`-print`), which will be handled automatically be `Go2`.

Beyond this, we will only describe the additions we have made for the optimization monitor. It can be added by registering it with a global monitor store (function `setOnce()`, with key “ea”). Whenever a new evolutionary algorithm is created through the factory, it will now search in the global monitor store for a suitable optimization monitor and load it into its data structures.

Note that we could also have added the optimization monitor directly to our EA object, using `ea_ptr->registerOptimizationMonitor(mon_ptr)`, where `mon_ptr` would be a `boost::shared_ptr` to our optimization monitor. However, the monitor would then only be available to this particular EA object.

Chapter 27.

Caveats and Restrictions

We have now described the Geneva library on well over 250 pages. We do believe that it is quite comprehensive and offers ample opportunity for further development. With its over 130000 lines of code, though, there are some areas where design compromises had to be made. This chapter thus wants to collect restrictions, caveats and plain oddities that you might encounter up over time. It can also be understood as a breeding ground for future improvements, as we will aim to amend the library where needed. As this is a "living" document that will be changed frequently, this chapter will not be preceded by the "*key points*", as is common in other chapters.

27.1. Floating Point Accuracy

Geneva does not enforce restrictions regarding the allowed value range of floating point variables. Users *must* however be aware that optimization techniques can return senseless results where very large floating point values are being used. This is due to the way **IEEE 754** floating point numbers work. Obviously, as a `double` variable is represented by a fixed number of bits, it cannot assume every possible value in its value range. Instead, the distance between two allowed values will grow with growing absolute size of the value. Close to 0, a `double` precision value can carry 16 digits – this is where the precision is highest. On the other hand, the distance between two allowed values close to the upper or lower limit can be as high as 10^{290} ...

It is thus recommended that problem definitions are written in such a way that the parameters to be modified act in the range $[0,1]$.

This is a general issue which is independent of the Geneva library collection.

27.2. Gradient Descent and Varying Parameter Value Ranges

A Gradient Descent, as implemented in the Geneva library, calculates the direction of steepest descent and makes a finite step in that direction. This procedure can be problematic if some parameters have a constrained value range, possibly even of different size. It is thus recommended to formulate an optimization problem in such a way that all parameters have the same value range. It is always possible to scale the value range inside of an evaluation function.

27.3. "Silent changes" to parameter values

Geneva was designed for optimization problems with particularly long-running evaluation functions. As the evaluation will remain constant as long as the parameters don't change, Geneva caches the fitness values. Re-evaluation only happens when the parameters have changed. While automatic detection of parameter-changes would technically be possible, it would be computationally quite expensive. Hence Geneva relies on a "dirty flag", which must be set whenever changes to parameter values are made. As long as these changes are made by Geneva, this happens automatically. Users should however avoid to make parameter changes without setting the dirty flag, otherwise Geneva will not function properly. The one exception are individuals that are newly created – their "dirty flag" will be set by default.

27.4. Individuals with a Variable Architecture

There are valid optimization use cases, where the number of parameters changes in the course of the optimization. E.g., if we look back to the the Mona Lisa example (see section 9.1.1), one might want to start with a single triangle and then add further triangles, when the optimization stalls. 10 new parameters will be added for each new triangle.

Similarly, the training of a feed-forward neural network (see section 9.3) might involve a simultaneous modification of the network's architecture (see e.g. [8, pp. 521–526]). Adding or removing nodes to or from the network obviously changes the number of parameters subject to optimization.

Geneva tries to provide you with a set of optimization algorithms, all of which are based on the same data structures for the specification of candidate solutions. Unfortunately, some algorithms, such as gradient descents, cannot cope with varying dimensions of the parameter space. Hence the above approach might not work for your setup.

Note, though, that at least in theory, Evolutionary Strategies should give you a means to nevertheless optimize problems with varying dimension. Problems might however arise even there when the `streamline()` function is used outside of the fitness calculation scope (which should not change the architecture). Reloading a vector of fixed size into an individual with a changed architecture will result in an exception or the proverbial "undefined behaviour".

Note that you can simulate variable-sized individuals by allocating a larger number of parameters, and disallowing the modification or ignoring the values of some of them.

27.5. The Effect of the Mutation Probability

In Evolutionary Algorithms, one needs to be careful when setting the initial mutation probability of parameters (or their adaptors) to a low value using the `setAdaptionProbability()` call. Generally, this setting is meant to lead to a more directed search process.

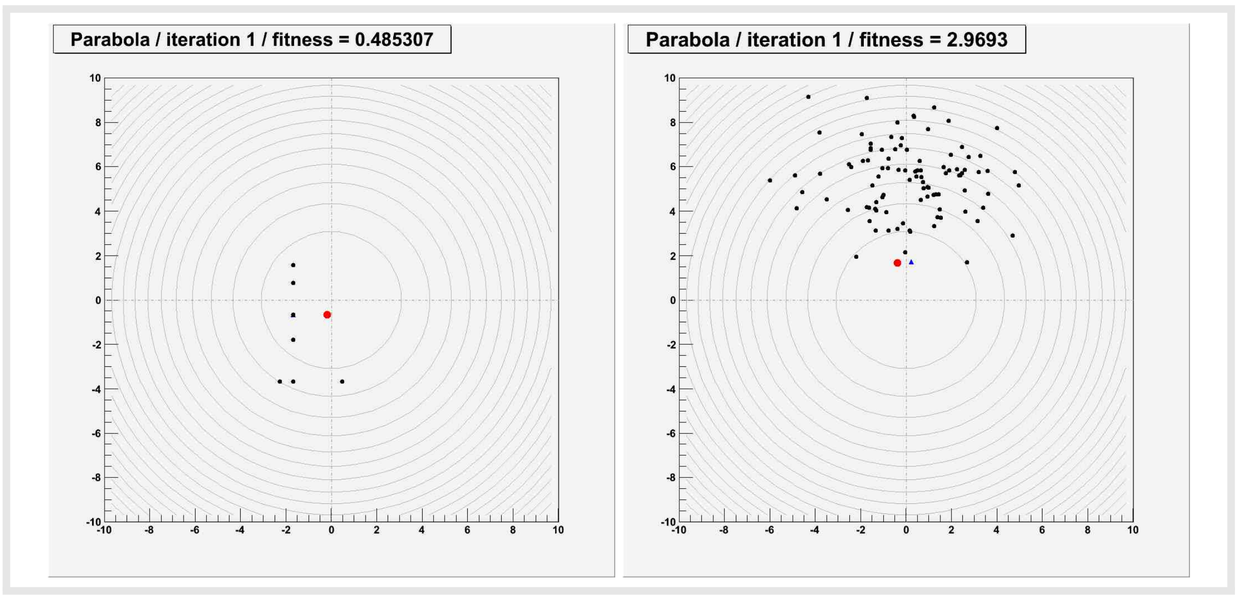


Figure 27.1.: *Choosing a good mutation probability can be important for the success of an evolutionary algorithm. In the left side of the picture, a 5% mutation probability was chosen. Several individuals didn't get updated at all, for some others only some parameters were updated, resulting in a sort of line search. On the right side, a 100% mutation probability was chosen. Except for the mutation probability, all configuration options of the optimization were identical.*

It is generally recommended to use higher initial mutation probabilities for lower numbers of parameters.

Even thow the adaption probalility may itself be subject to mutation, if this advice is not followed, you can get a situation where some individuals do not get changed at all, or where the evolutionary algorithm degrades to a sort of line search. Figure 27.1 illustrates this on an example with just two parameters.

A mutation probability of 5% means in this case, that the majority of individuals remains unchanged. Hence, with two parameters, a mutation probability of at lest 50% should be used.

27.6. Value Range of Constrained Paramters

Parameter classes like `GConstrainedDoubleObject` or `GConstrainedInt32Object` will not make the full value range of the underlying basic type available to the user. The reason for this lies in the mapping that is performed from the internal to the external representation of a parameter. There are various occasions where the difference between upper and lower boundary is calculated. As a statement like `2*std::numeric_limits<double>::max()` will leave the allowed value range for a `double`, calculating `(max-min)` would not be possible if `max` and `min` have been assigned the largest possible values. Hence the allowed value range is reduced by a factor of 10 in Geneva. More complicated calculations would allow to use the entire value range. However, as the above restriction is meaningless for most practical purposes (compare section 27.1), the more efficient solution has been chosen.

27.7. Broker-flooding

Imagine you are dealing with a large population of 1000 individuals, and that your optimization runs in networked mode. Now let us assume that we have chosen a very small upper limit to the `waitFactor` of 1, and there is just one workernode active.

In the example of an Evolutionary Algorithm, a `waitFactor` of 1 means that the population will only be allowed to spend the amount of time in each iteration that is needed for the first item to return.

So in each iteration, the population will submit 1000 individuals to the broker, but will only be able to get back a single item. This will lead to a flooding of the broker with individuals and ultimately to items being discarded by the broker.

Thus, clearly, the `waitFactor` needs to take into account the number of worker nodes and the amount of items being submitted to them in each iteration.

27.8. Assigning the worst possible evaluation

Users should always make sure to mark an individual as invalid using the protected `GOptimizableEntity::markAsInvalid()` function, if they wish to assign the worst known possible evaluation (such as `MAX_DOUBLE`) to an individual. The reason for this is that the worst known *valid* solution plays a special role in Geneva, if parameter constraints exist. This tagging must happen from inside the evaluation function.

27.9. Secure communication

Early versions of Geneva featured optional encrypted communication. In the meantime, however, we have removed this feature. This has happened for the following reasons:

- We expect the most likely use-case for Geneva's network modes to be local clusters rather than WANs (i.e. Grids and Clouds). As cluster nodes will usually have a private IP, encypted communication does not seem to be very useful but will rather lead to bad performance.
- Where users want to use WANs, they may create static VPNs between client nodes and the server. These can be configured in a far more versatile way than the static security setup which would be required for Geneva.
- Geneva's network client will open a connection to the server whenever they ask for work ("pull-modus"). Likewise they will open (and subsequently close) a connection when they are finished

with a calculation. For short evaluation times this will lead to very poor performance in the presence of encryption, at least when a hand-shake is required

- While we do have in-house experience in security of the Linux Operating System, the same does not apply for application design. And as the OpenSSL desaster has shown, even people with years of experience under their belt may make programming mistakes that lead to security-nightmares. Thus we follow the “KISS”[1] principle in Geneva, particularly as there are more versatile, secure and efficient possibilities available (such as the usage of Client-side VPNs).

[1]Keep it simple and stupid

Part III.

Details and Advanced Topics

Chapter 28.

Performing Meta-Optimization with Geneva

This chapter discusses the topic of meta-optimization. In the context of this chapter, this term comprises various techniques. Geneva has the ability to treat optimization algorithms as individuals. In the most simple case, a given type of algorithm is loaded into an Evolutionary Algorithm. The parameter space can then be explored from different starting points. Secondly, it can make sense to let an optimization algorithm optimize the configuration parameters of another algorithm, so that it solves a given problem more quickly. More of a research topic is a situation where different types of optimization algorithms compete against each other.

Key points: (1) Many different forms of meta optimization exist (2) Geneva allows to treat optimization algorithms as individuals, hence allowing to let identical oder different algorithms compete against each other (3) Algorithms can be themselves embedded in individuals, allowing the direct optimization of their configuration parameters (4) A useful target function is the number of solver calls (i.e. the number of calls to the optimization criterion)

28.1. Multi-Populations

Looking back at figure 12.1 we can see that the `GOptimizationAlgorithmT` class is indirectly derived from the `GOptimizableEntity` class. Evolutionary Algorithms are capable of directly hosting `GOptimizableEntity` objects as the subject of optimization[1]. Through this "master algorithms" cycle of duplication, mutation and selection it then becomes possible for algorithms to compete against each other, implementing a simple form of meta-optimization. Note that, for optimization algorithms, the step of mutation and selection is one, and is represented by a full optimization cycle. The *quality* or *fitness* of an algorithm at the end of the cycle is then represented by the fitness of the individual with the best quality. Even meta-meta-optimization (or any other depth of hierarchy) could be done in this way, although it is doubtful whether this makes sense.

[1]Note that, at the time of writing, the other implemented algorithms are not suitable for this purpose. There is no way a gradient method or a swarm could modify entire algorithms. Evolutionary algorithms, on the other hand, mainly deal with mutations, which can be directly mapped to the initialization and optimization of a population.

The Geneva software distribution comes complete with an example illustrating multi-populations (see `08_MultiPopulation`).

28.2. Optimizing Configuration Parameters

Most optimization algorithms depend on a large number of configuration options; and the success or failure of an optimization run will often crucially depend on the right choice of parameters.

Whether an optimization run can be considered "good" or "bad" can be rated by many criteria. For problems with very complex (hence computationally intensive) solvers, however, the most common choice will be represented by the amount of time needed to reach a satisfactory result. This figure of merit in turn directly depends on the total number of solver calls needed to reach a given optimum.

Where good results for a given optimization problem are already known, one can thus treat the optimization run itself as the solver function, with the figure of merit being the number of calls to the solver needed to reach the known result. Optimizing an optimization algorithm for speed thus involves varying the configuration paramters in such a way that the number of solver calls is minimized. Sometimes it is then possible to utilize the same set of parameters for other problems of the same nature.

As many optimization problems involve the use of random numbers and will thus follow a non-deterministic path through the parameter space, one needs to take care to run an algorithm multiple times and calculate averages and standard deviations, as one might otherwise treat an "outlier" incorreclty as a result, which is commonly achieved by an algorithm with this set of parameters. Hence this form of meta-optimization requires some patience. Nevertheless it may pay off well, as the mutual dependencies of different configuration parameters is not easily visible.

Using this procedure for a class of optimization problems can give indications for choices such as:

- Suitable population sizes, numbers of parents and children in evolutionary algorithms?
- How large should a swarm be and how should it be partitioned ?
- Which step widths should be chosen ?
- With what likelihood should parameters indeed be mutated ?
- Which cooling schedule is suitable for simulated annealing ?

Implementing this kind of procedure can be a bit tricky, but will generally involve the creation of a dedicated individual, whose parameters reflect the configuration options of a given algorithm. The `fitnessCalculation()` function will then instantiate an algorithm repeatedly with these settings (and a given optimization problem) and determine the number of calls to the solver. The result of the fitness function is then the average number of calls, which needs to be minimized.

This kind of problem will generally involve different variable types, most notably integral, boolean and floating point parameters. In the context of Geneva, Evolutionary Algorithms are the most suitable type to perform this kind of optimization.

It must be stressed, that the success of this procedure crucially depends on the validity of the generalization from a given optimization problem with known outcome to another problem, which is allegedly of the same type (and thus can be treated with similar configuration parameters).

28.2.1. A concrete example

Figure 28.1 shows the procedure on a concrete example. An evolutionary algorithm was asked to optimize the configuration parameters (most notably the number of parents and children, step widths and variation ranges as well as adaption probabilities for parameters) in such a way that the average number of solver calls was minimized.

The procedure uses Geneva's `GFunctionIndividual`, which implements different benchmark functions, among them the so called "noisy parabola" (a.k.a. "Berlich function", compare figure A.2). In figure 28.1, the algorithm was applied to the two-dimensional form of this function, which has a minimum at $(0,0)$ and has a very high number of local optima.

The plot in the upper left corner of figure 28.1 shows the average number of solver calls, as the optimization progressed. It is visible the algorithm converged quicky and was able to decrease the number of solver calls by more than a factor of two. Looking at the parameters being optimized, it is obvious that the reduction of the number of children played an important role in the procedure. It is also visible that the internal adaption of the step width (i.e. the width of the gaussian in Gauss mutations) was allowed to cover a very wide range by the procedure.

Figure 28.2 shows the results for the 8-dimensional version of the noisy parabola.

28.3. Letting different Algorithms Compete

Finally, with Geneva, it is generally feasible (albeit so far untested) to let different optimization algorithms compete against each other. The easiest way would be an evolutionary algorithm, whose individuals are indeed represented by optimization algorithms. Note, though, that there are some problems with this approach.

As just one example, one has to take care that algorithms with faster convergence do not totally dominate the optimization process. E.g., an algorithm might initially converge quickly and essentially throw out other algorithms, and might later get stuck in a local optimum. Some of the slower-moving algorithms might have better success here, but are no longer part of the meta-optimization. Hence care has to be taken in the design of the objective function.

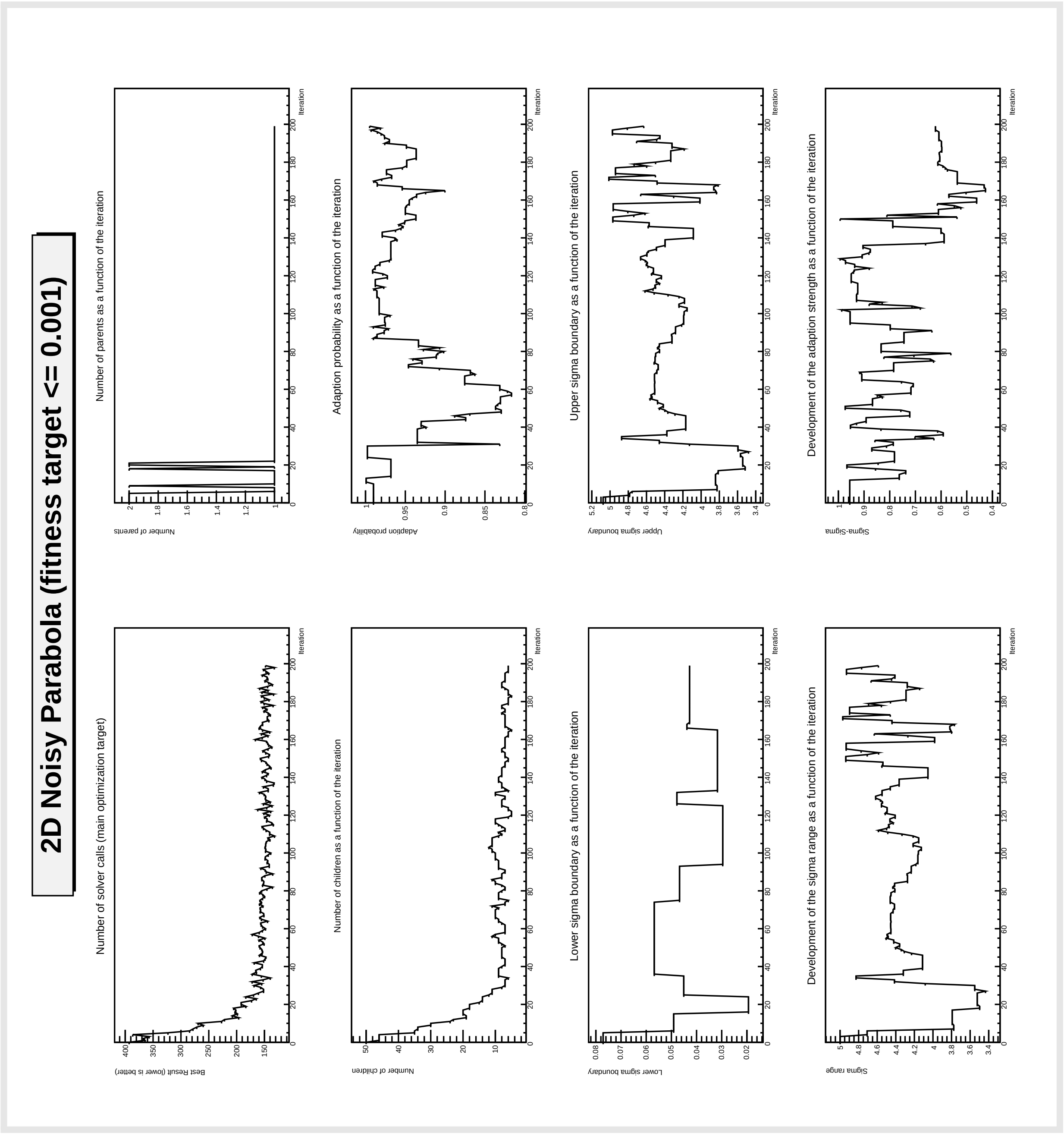


Figure 28.1.: *Meta optimization, shown here on the example of a two-dimensional "Noisy Parabola", allows to optimize the parameters of optimization algorithms. The figure in the upper left corner represents the best average number of solver calls in each iteration. The other plots show the development of some of the variables being optimized.*

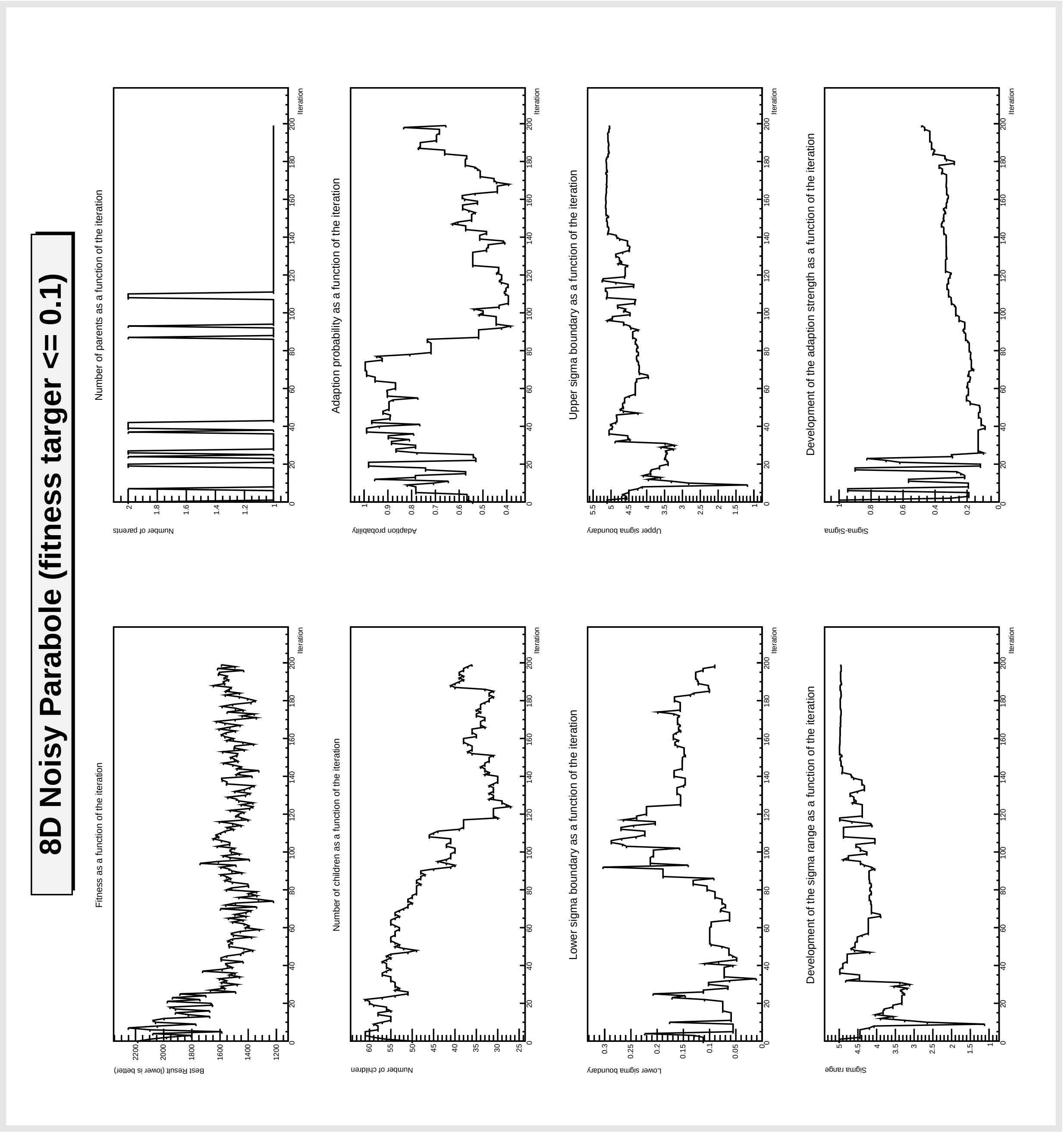


Figure 28.2.: *Same procedure as in figure 28.1, albeit for the 8-dimensional noisy parabola*

Chapter 29.

Coding Conventions

This chapter outlines some guidelines on how to write Geneva code. Note that we consider these conventions to be more than recommendations. In order to help users to understand the Geneva code, it appears to be of the utmost importance to adopt a consistent coding style.

By the same token, the Geneva code has evolved over time, and parts of the library might not follow with our own guidelines. Thus, if you encounter code that does not comply with our rules, you are encouraged to contact the authors via `contact@gemfony.eu`.

29.1. Code Documentation

Geneva aims to be as easy to understand as possible. Extensive documentation is therefore of high importance, and contributors to the code are encourage to document their code thoroughly.

29.1.1. Generating reference documentation, License information

Every file, function and class is prepended by a comment in Doxygen format. The intention is to generate reference documentation directly from the source code. Doxygen allows to create reference manuals in different formats, including LaTeX and HTML, provided the library authors have complied with Doxygen's conventions.

Reference documentation for Doxygen is available from the Doxygen web page[76]. Comments should be made for all entities that comprise a C++ program, including files, (member-)functions, classes and variables.

Files

Each file, including code files (both headers and implementation) and scripts, should be prepended with information regarding the name of the file, the author and copyright, as well as the license under which the file is offered to users. This may look like this:

Listing 29.1: The file preamble

```
/**
 * @file GParameterBase.hpp
 */

/* Copyright (C) Dr. Ruediger Berlich and Karlsruhe Institute of Technology
 * (University of the State of Baden-Wuerttemberg and National Laboratory
 * of the Helmholtz Association)
 *
 * Contact: info [at] gemfony (dot) com
 *
 * This file is part of the Geneva library, Gemfony scientific's optimization
 * library.
 *
 * Geneva is free software: you can redistribute it and/or modify
 * it under the terms of version 3 of the GNU Affero General Public License
 * as published by the Free Software Foundation.
 *
 * [Some further information]
 */
```

Classes, member functions and data members

All classes should be prepended by a general description of their role, like this:

Listing 29.2: The class preamble

```
namespace Gem {
namespace GenEvA {

/**
 * The purpose of this class is to provide a common base for all parameter
 * classes, so that a GParameterSet can be built from different parameter
 * types. The class also defines the interface that needs to be implemented
 * by parameter classes.
 *
 * Note: It is required that derived classes make sure that a useful
 * operator=() is available!
 */
class GParameterBase
    : public GMutableI
    , public GObject
{
  // [...]
```

Note the `/**`, which indicates to doxygen that this is a comment that should be included in the reference documentation.

The declaration of member functions should be prepended by a Doxygen-style `brief` comment:

Listing 29.3: The brief description before member function declarations

```
/** @brief The copy constructor */
GParameterBase(const GParameterBase&);
```

Definitions of member functions should be prepended by a more thorough description, including a full explanation of function arguments:

Listing 29.4: The full description before member function definitions

```
/******************************************************************************/
/**
 * Checks for equality with another GParameterBase object
 *
 * @param  cp A constant reference to another GParameterBase object
 * @return A boolean indicating whether both objects are equal
 */
bool GParameterBase::operator==(const GParameterBase& cp) const {
  // comparison code
}
```

Definitions of member functions should be separated from each other by a line, as shown in the above listing.

Stand-alone functions

Stand-alone functions follow the same conventions as member functions.

29.1.2. In-Code Comments

In-code comments are mostly free-style. The coding rules below give a few examples as to how code should be commented.

29.2. Coding Rules

29.2.1. Function argument lists

In Geneva, argument lists follow certain conventions. First of all, if there are too many arguments, each argument should be listed on a separate line, like so:

Listing 29.5: Long argument lists should be wrapped

```
void myComplicatedFunction(
       const double& argument1
     , const double& argument2
     , const double& argument3
     , double& writeToMe
) {
  /* some implementation */
}
```

Argument names in header files

Function arguments in header files should be listed without argument name. This facilitates subsequent changes of the argument names in the implementation file.

Listing 29.6: Argument names in headers should be omitted

```
class myClass {
public:
   myClass(); /// The default constructor
   myClass( const myConstructor& ); ///< The copy constructor

   const myClass& operator=(const myClass&); ///< Assignment operator

   // [...]
};
```

Argument names in the actual implementation

Argument names in the actual implementation should as always be descriptive. In the case of setters for private variables, argument names should reflect the names of the private variable. Likewise the function name should reflect the name of the variable being set.

Listing 29.7: Arguments to setters of private variables should mimic their name

```
void setMyPrivateVar(const boost::uint32_t& myPrivateVar) {
  myPrivateVar_ = myPrivateVar;
}
```

Empty argument lists

C++ allows to specify (member-)functions without arguments in two ways:

Listing 29.8: Functions with empty argument lists

```
void myFunctionOne(void) { /* ... */ }
void myFunctionTwo() { /* ... */ } // This option should be used
```

Geneva has adopted the second possibility.

29.2.2. Parentheses and Initialization of Variables

For Functions . . .

Curly brackets should be used in the following way for functions:

Listing 29.9: Parentheses in functions

```
boost::int32_t myFunction() {
   // Some code here
   return 1;
}
```

It is often necessary to initialize local variables of a class in constructors. If so, then each initialization should be on its own line, as should be the opening bracket of the function definition. Furthermore, commas and the initial colon should be aligned. E.g.:

Listing 29.10: Parentheses in constructors, when initializing member variables

```
myClass::myClass()
   : localVariableOne(1)
   , localVariableTwo(2)
{
   // Some code
}
```

For Classes and Structs . . .

The conventions for classes and structs are similar to those used for functions. When dealing with a base class, then the opening bracket should follow the class name:

Listing 29.11: Parentheses in declarations of base classes

```
class myBaseClass {
public:
   myBaseClass();
};
```

The opening bracket of the class declaration should be on its own line, if the class is derived from one or more parents. In this case, each parent class should be listed on a new line. The initial colon and following commas should be aligned.

Listing 29.12: Parantheses in declarations of derived classes

```
class myDerivedClass
  : public myClass
  , private boost::noncopyable
{
public:
  myDerivedClass();
};
```

try/catch blocks

When testing for exceptions, code should be formatted in the following style:

Listing 29.13: Parentheses in try/catch blocks

```
try {
  // some code that should be tested for exceptions
}
catch (...) {
  // Code to be executed when an exception is caught
}
```

29.2.3. Descriptive naming schemes

The names of classes, member functions, stand-alone functions and variables should be descriptive, amalgamating words in the following way:

Listing 29.14: The naming of variables, classes and functions should be descriptive

```
class GParameterBase {
public:
  void setAdaptionsActive();
private:
  bool adaptionsActive_;
};
```

Note that classes of the core Geneva framework usually start with an upper case `G`. Another, albeit rarely used, convention is that the names of interface classes end with uppercase `I`.

Template classes and structs should end with an upper-case `T` (as used e.g. in `GAdaptorT`).

29.2.4. Empty Functions

Empty (member-)functions should be clearly marked as such:

Listing 29.15: Marking empty functions

```
virtual void ~myClassWithEmptyDestructor()
{ /* nothing */ }
```

This often plays a role when a destructor is specified, but is empty (which may be useful if it is declared virtual).

29.2.5. Templates

Templates should generally use the `typename` placeholder instead of `class`, as it states the intent much more clearly. Some more formatting conventions apply:

Template functions

The `template` statement in template functions should be put on a separate line like so:

Listing 29.16: Formatting template functions

```cpp
template <typename T>
bool myTemplateFunction(T var) {
  // some code
  return true;
}
```

Class templates

Like in the case of template functions, the `template` statement of class templates should be put on a separate line. All other conventions regarding the formatting of classes remain valid.

Listing 29.17: Formatting class templates

```cpp
template <typename T>
class someClassTemplate {
public:
   void setMyVar(const T& myVar);

private:
   T myVar_;
};
```

Member templates

Member templates follow the same conventions as standard function templates:

Listing 29.18: Formatting member templates

```cpp
class myClass {
public:
   myClass();
   virtual ~myClass();

   template <typename T>
   void doSomeFancyStuff(const T& myVar) {
     // Some inline code
   }
};
```

29.2.6. Namespaces

The closing of namespaces should be clearly marked, like so:

Listing 29.19: The closing brackets of namespaces should be clearly marked

```
namespace Gem {
namespace GenEvA {

// some code inside of namespace scope

} /* namespace GenEvA */
} /* namespace Gem */
```

29.2.7. Include guards

Header files should contain include guards in the following style, in order to prevent multiple inclusion of the same header and resulting compilation errors.

Listing 29.20: Naming of include guards follows the naming scheme of header files

```
#ifndef MYCLASSHEADER_HPP_
#define MYCLASSHEADER_HPP_

class myClass {
  // some code
};
#endif /* MYCLASSHEADER_HPP_ */
```

29.2.8. `const`-correctnes

Quite a comprehensive discussion of the topic of const-correctnes can be found at `http://www.parashift.com/c++-faq-lite/const-correctness.html`. The Geneva team regards this topic as particularly import. All recommendations presented at the above URL should be followed.

29.2.9. Control Flow

Control statements should be formatted in the following way:

if/else if/else

Listing 29.21: Formatting `if` statements

```
/*
 * Some general statements. Mandatory for complex cases.
 */
if(condition) { // Optional comment
  // some code
}
else if(otherCondition) { // Optional comment
  // some other code
}
else { // Optional comment
  // code for all remaining cases
}
```

switch

Listing 29.22: Formatting `switch` statements

```
/*
 * Some general statements. Mandatory for complex cases.
 */
switch(value) {
// some optional comment
case one:
  // some code
  break;

// some other optional comment
case two:
  {
    boost::int32_t i; // optional comment
    // some code which requires local variables
  }
  break;

default:
  // code to be executed if no other case mathches
  break;
}
```

for

Listing 29.23: Formatting `for` statements

```
/*
 * Some general statements. Mandatory for complex cases.
 */
std::vector<double>::iterator it;
for(it=myVec.begin(); it!=myVec.end(); ++it) { // optional local comment
  // some code
}
```

Note the general recommendation to prepend the ++. This becomes particularly important when dealing with iterators (and makes no difference for integer counters).

while

Listing 29.24: Formatting `while` statements

```
/*
 * Some general statements. Mandatory for complex cases.
 */
while(condition) { // optional local comment
   // some code
}
```

do/while

Listing 29.25: Formatting `do/while` statements

```
/*
 * Some general statements. Mandatory for complex cases.
 */
do { // optional local comment
   // some code
}
while(condition);
```

29.2.10. Private variables and get/set-functions

Private variables in classes (including STL containers, local objects, etc.) should be marked with a trailing underscore like so:

Listing 29.26: Formatting `do/while` statements

```
class myClass {
public:
  myClass();

  void setMyClassVariable(const boost::int32_t& myClassVariable) {
    if(myClassVariable%2 == 0)
      myClassVariable_ = myClassVariable;
    else
      throw();
  }

  boost::int32_t getMyClassVariable() const {
    return myClassVariable_;
  }

private:
  boost::int32_t myClassVariable_;
};
```

Setter and getter functions should use the same name as the private variable, but without the underscore. Also, even if only trivial getter and setter functions are needed (i.e. no checks are performed by the setter function), class variables should nevertheless be declared private and accompanied by appropriate *getters* and *setters*. Checks in the setter might become necessary at a later time, and using functions for the access to private variables (instead of declaring them public) makes the code far more modular and maintainable.

29.2.11. Omitting throw() in member function declarations

In Geneva, member functions are not marked with the exceptions they may throw. An in-depth rationale is available online in an article by Herb Sutter[67].

29.2.12. Avoiding global variables

Non-const global variables (with the possible exception of Mutex-variables) are not allowed, as Geneva is a multi-threaded library.

29.2.13. Portable use of integer variables

The Geneva library makes extensive use of the Boost library collection. One reason for this is portability. Unfortunately C++ does not guaranty a specific size for integer variables. For this reason Boost has introduced the *Standard Integer Types* library[5]. Instead of using e.g. an `int` type, users can now specify the desired size and signedness of the integer variable, e.g. boost::uint32_t for a 32 bit unsigned integer. These types are then mapped on matching, platform-dependent local types.

Users of the Geneva library should follow this convention wherever possible. Exceptions to this rule may occur where external libraries are called, which require e.g. a “short” to be passed as argument. In this case it is recommended to resort to the `boost::numeric_cast<>()` function for additional protection against overflows. It will then throw in DEBUG mode, so you know exactly where the problem lies.

29.2.14. Defines, macros and const variables

Function macros should be avoided at all costs, as templates usually offer a far better solution than macros.

Constructs like

Listing 29.27: Defines may not be used for specific values

```
#define SOMENUMBER 3
```

which are often used to give a number a tell-tale name, may not be used in Geneva. Instead, use the following:

Listing 29.28: Constant global variables should be used instead of defines

```
const unsigned boost::uint32_t SOMENUMBER = 3;
```

This is far more specific and allows the compiler to warn on type mismatches. Indeed, where const values need to be supplied as parameters, such global definitions allow different classes to share the same naming conventions. Values then need to be changed in only one place.

29.2.15. `#ifdef`

`#ifdef`'s can be used but should also be avoided in favour of general configuration options. When using this option, follow the following structure:

Listing 29.29: Begin and end of an `#ifdef` statement should be clearly marked

```
#ifdef DEBUG
//some debugging code
#endif /* DEBUG */
```

29.2.16. Namespaces and Explicit Scope

The Geneva library makes extensive use of the Boost set of libraries, as well as the C++ standard library. All `using` statements in the library have been eliminated. Instead, calls to Boost functions or std functions should be prepended with the scope, such as in the following example:

Listing 29.30: Calls to external library components should be prepended by the explicit scope

```
// [...]
std::vector<boost::shared_ptr<SomeGenevaClass> > dataVector;
// [...]
```

So far, for readability reasons, we do not enforce the scope for objects from the Gem::GenEvA namespace. Other Gem namespaces (e.g. Gem::Util namespace) must be explicitly mentioned.

29.3. File naming schemes

29.3.1. Descriptive names

The naming of the files that constitute the Geneva library should reflect the main classes contained in them. Where appropriate, only one class should be stored in one file. An exception to this rule are little helper classes and structs that are closely related to the main class of the file. Please also note again the convention that template classes end with an upper-case `T` (see subsection 29.2.3).

29.3.2. File extensions

The Geneva library follows the convention that header files use the file extension `.hpp`, while the actual implementation is contained in files with the extension `.cpp`.

Chapter 30.

Helping Each other

The Geneva library is an Open Source project, targetted at users from industry and science alike. And parametric optimization is so generic a topic that users of the Geneva library will come from a wide area of technical disciplines. It is thus one of the most prominent goals of Geneva to help and unite different fields of knowledge. This chapter wants to give you pointers to where help can be found and also wants to explain, where *you* can help, should you have ideas for the future development of Geneva.

30.1. Finding Help

If you read this chapter, then chances are that you have tried out Geneva and have stumbled upon a usage-related question, a topic that needs improvement or that is just plain wrong. We have tried to document the Geneva library thoroughly, and we hope that this manual is a good starting point for using Geneva. We do not maintain, however, that it is easy to understand under all circumstances. After all, parametric optimization is a complex topic.

Industrial code is estimated to contain 4–5 errors per 1000 lines of code after delivery. According to this estimate, Geneva would contain about 500–600 errors of all levels of severity. We do not (and indeed cannot) promise that Geneva is free of bugs. However, when we are made aware of a problem, we will aim to solve it whenever possible.

As a user of the Open Source edition, the most direct route to get help is through the **Usage** mailing list. Got to our web page `http://www.gemfony.eu`, then click on *Forum*. You will see three forums – *Announcements*, *Usage* and *Development*. We suggest to post your usage-related questions to the *Usage* forum. We will monitor the list and try to answer in a timely manner. And chances are that other users will have stumbled across the same problem before, so they might also be willing to help.

In order to post in the forum, you will need a *Nabble* account. Links to the registration are available on the page listed above.

30.1.1. Bug Reports

Reports about possible bugs can be submitted in one of three ways:

- You may post them on the *Usage* list
- You can also post them through a form on the Gemfony web page `http://www.gemfony.eu` (click on the *Bug Reports* link)
- You can post them through the Launchpad portal (`http://www.launchpad.net/geneva` – see the "How to report a bug" link)

No matter which way of submitting a bug you chose, we would like to ask you to make available to us the following information in order to resolve the problem:

- A description of the way in which you use the Geneva library (type of application, parallelization mode, . . .)
- An exact description of the system(s) the problem happens on
- A description of the circumstances (what happens, which error messages appear)
- **Ideally some code that replicates the problem**

Note that, if we cannot reproduce the problem, then chances are that we cannot solve it for you.

30.1.2. As a Commercial User

If you use the Geneva library in a commercial setting and are looking for help, we suggest that you consider using one of our support and consulting offers. Please contact us for further details via `contact@gemfony.eu`. Please also note that we can offer you other licensing options than the Affero GPL v3.

30.2. Suggesting Improvements

Geneva covers a wide range of possible application scenarios. Given the generic nature of parametric optimization, however, it is unlikely that we have even covered a small portion of what users want to do with Geneva. It is thus likely that you will come across a missing feature, or that a given feature is more complicated than it needs to be. We would thus like to encourage you to make us aware of possible improvements. The easiest way is to do so on the *Usage* list accessible through our web page. Alternatively, please feel free to contact us through `contact@gemfony.eu`.

30.2.1. Donating Code and Fixing Bugs

You may also donate code and bug fixes to Geneva. Please do contact us with your suggestions via `contact@gemfony.eu`. Among the things we are looking for are:

- Implementations of new optimization algorithms (even if they are in pseudo code). We'd appreciate if you accompany your code with a publication of the corresponding algorithm (papers, discussions of the topic), so we get a better understanding of the scope of the suggestion
- "Standard" individuals for new deployment scenarios
- Cool demos
- Code that improves efficiency
- Integration with new means of parallelization (i.e. Consumers)
- Pinpointing bugs

Note that, depending on the type and amount of code we might have to ask you to sign a transfer agreement, so that we may include your donation into our code. We have made our own code available as Open Source, but, as a commercial entity, do rely on the fact that we may also use Geneva commercially. This requires that we have full control over the code and all rights to use it as is needed.

30.3. Monetary donations

Monetary donations are of course welcome. For example, if you want a given feature to appear in the Open Source edition within short time, you could decide to sponsor its development. Please contact us via `contact@gemfony.eu`.

30.4. Licensing

You have in front of you a copy of a high quality software that has been designed according to high scientific standards. The typical cost of similar products lies in the range of possibly thousands of Euros per copy. Yet, you have the opportunity to use this code free of charge, subject to the terms abd conditions of the GNU Affero General Public License ("AGPL").

Please note that this *does* imply responsibilities on your side. Geneva is neither freeware nor can its code be used without restrictions in other projects. We suggest that you read the license in its entirety to understand your rights and obligations. It is quoted verbatim in appendix D.1. Copies are also available at `http://www.gnu.org/licenses/`, and the text is shipped with the Open Source version of the Geneva library collection.

30.4.1. The GNU Affero General Public License

Note that the meaning of "derived work" in the Affero GPL is not always clear. **In our interpretation of this term, we follow a statement made by the Free Software Foundation**. Note that, under most circumstances, this means that you, as a user, also accept responsibilities. Please *do* make sure to carefully read the AGPL and to comply with its terms and conditions.

Part IV.

Independent Geneva Libraries

Chapter 31.

Creating Random Numbers with Hap

This chapter discusses the creation of random numbers with Geneva's *libhap* library. It comprises two main parts – a random number factory and proxy objects, giving interested parties (individuals, parameter objects, optimization algorithms) transparent access to a variety of different random number types and distributions. Production of random numbers is thus centralized, while consumption happens decentrally. **Hap is implemented independently of the optimization use case and can be used in other projects (subject to the conditions of the library's license).**

Key points: (1) Geneva's hap library distinguishes between a random number factory and a random proxy (2) The factory produces random numbers in multiple threads, taking care of the synchronisation of the seeds of all its random number generators (3) The use of the factory results in better utilization of idle CPU cycles. (4) In the use case "parametric optimization", there are idle times in regular intervals (5) The random number proxy presents a transparent interface to objects in need for random numbers, that allows concurrent access to a pool of random numbers (6) To users, the proxy looks like a local, independent random number generator (7) Many random number distributions are modelled in the proxy objects (8) They use internally "raw" random numbers (evenly distributed random double numbers in the range $[0,1[$).

31.1. The Random Number Factory

Optimization happens in cycles, with periods of strong activities (e.g. evaluation of individuals) followed by relatively calm times (sorting, book-keeping, ...). Random numbers are used in many optimization algorithms – **high-quality numbers of different characteristics are needed in large quantities.** Many objects may be interested in random numbers, possibly requiring parallel access. But instantiating individual random number generators in potentially hundreds of thousands of objects is inefficient and requires synchronization of seeds, so the results of different individuals are not correlated or, in the worst case, identical.

Geneva has thus centralized the production of random numbers in a factory class, which is accessed through a singleton. It produces evenly distributed double random numbers in the range $[0,1[$ concurrently in a small[1] number of threads. Arrays of random numbers are added to thread-safe buffers, until

[1] The exact number is usually modelled after the number of processing units in the system, but can also be set by hand.

these reach their maximum capacity[2].

The synchronization of the seeds of the random number generators in the producer threads is handled internally in the factory. In particular, it has been taken care that the various random number generators are not seeded with correlated numbers. Instead, seeds are taken from a random number generator itself[3].

Once the buffers are full, the producer threads block (in particular this means that they do not consume any computing power when blocked) and only continue to add random number arrays to the buffer once there is free space again. Consumers of random numbers can concurrently retrieve the random number buffers from the thread-safe queue when needed.

Given a sufficient number of producer threads, random numbers will thus be predominently produced in periods of low activity of the optimization algorithm, while a sufficient amount of random numbers is available when needed by the optimization algorithms.

The random number factory will be instantiated without any direct interaction by the user. All that needs to be done is to include the header file `GRandomT.hpp` in your code.

31.2. The Random Number Proxy `GRandom`

In order to shield users of random numbers from having to deal with the random number factory directly, Geneva comprises a proxy class `GRandom`, which handles the interaction with the factory behind the scenes. Users can ask for a double random number at any time. When a local random number buffer is available and not exhausted, the random numbers are taken from there. Once it is empty, a new buffer is obtained from the factory and stored locally. To the user, the proxy thus looks exactly like a local random number generator.

`GRandom` models various random number distributions and random number types. These are produced from the “raw” double values obtained from the factory.

Access to the random number proxy is possible by instantiating the `GRandom` class. See the examples in section 31.2.1 for further information.

31.2.1. Random Number Types and Distributions

This section gives an overview of the available random number types and distributions in `GRandom`. All code samples are also available in the Geneva distribution, in the example `05_GHapUsagePatterns`. Figures 31.1, 31.2 and 31.3 illustrate some of the distributions.

[2]Production of random numbers could even happen in a different location, such as on a GPGPU. Note, though, that this is not currently implemented in Geneva.

[3]We have observed in our tests that a particular generator type produced highly correlated random number sequences when adjacent seeds were used.

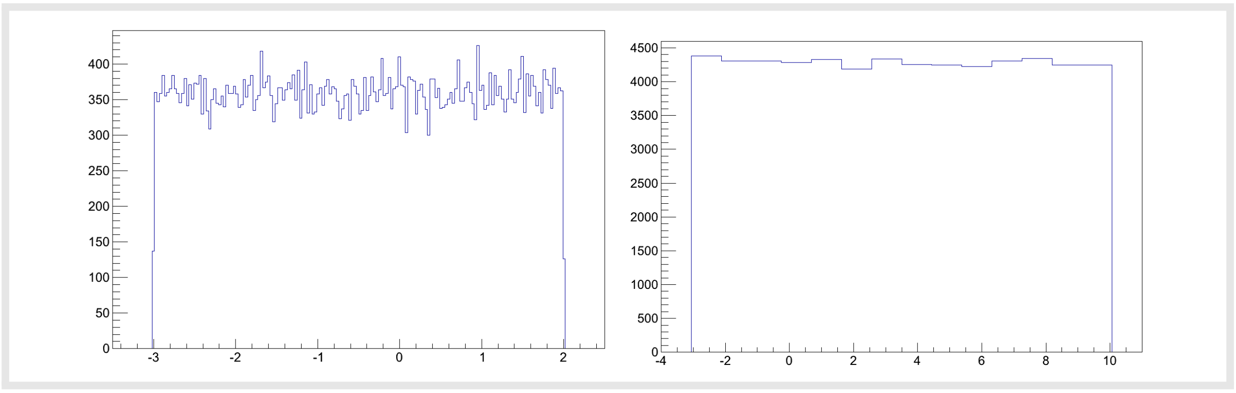


Figure 31.1.: *Floating-point and integer random numbers can be constrained in their value ranges.*

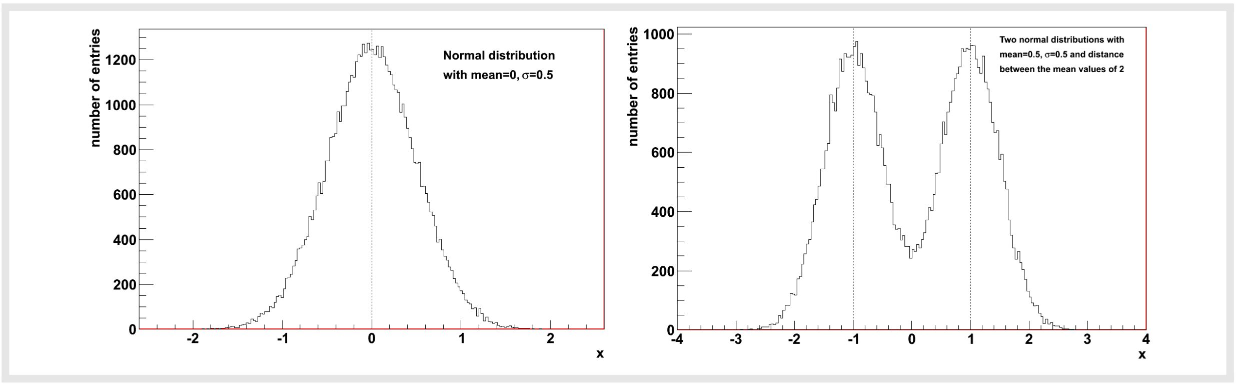


Figure 31.2.: *Two types of gaussian distributions are availble – a standard gaussian with a user-defined mean and σ, and two gaussians of equal σ superimposed. Both are used as part of mutation operators in Evolutionary Algorithms (compare chapter 4.2.2).*

Uniform `double` random numbers in the range $[0,1[$

The `GRandom::uniform_01<double>()` function is just a wrapper around the function used to retrieve random number packages from the factory. This is the native random number type of Geneva and has the least overhead. Listing 31.1 shows how to use this function.

Listing 31.1: Uniformly distributed double random numbers in the range $[0,1[$

```
#include "hap/GRandomT.hpp"

const int NPROD = 1000;

// All GRandom-related code is in the namespace Gem::Hap
using namespace Gem::Hap;

int main(int argc, char** argv) {
  // Instantiate a random number generator
  GRandom gr;
```

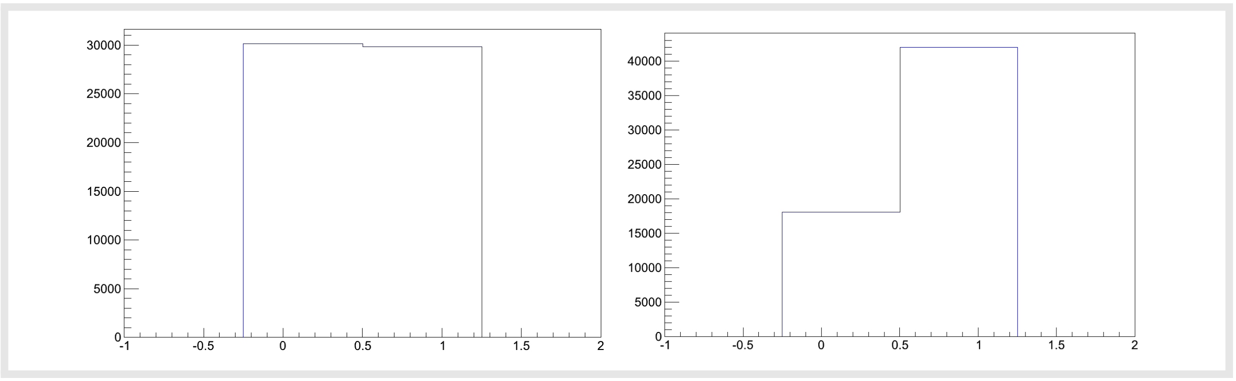


Figure 31.3.: *Boolean random numbers can either be produced with a pre-defined probability distribution for* `true` *and* `false` *or an even likelihood for both.*

```
    double d_even_01 = 0.;

    for(int i=0; i<NPROD; i++) {
      // Random numbers with an even distribution of
      // double values in the range [0,1[
      d_even_01 = gr.uniform_01<double>();

      // Note: GRandomBase defines an operator(), hence
      // you could also use gr() to obtain a random number
      // of this type.
    }

    return 0;
}
```

Uniform `double` values in the range $[0, max[$

The `GRandom::uniform_real(const double&)` function creates uniformly distributed double random numbers in the range $[0, max[$. Internally, `GRandom::uniform_real(const double&)` wraps the `GRandom::uniform_01<double>()` function.

The usage example in listing 31.2 concentrates on the calling conventions and omits the header file and `main()` function. See listing 31.1 for a complete example.

Listing 31.2: Uniformly distributed double random numbers in the range $[0, max[$

```
    // Instantiate a random number generator
    GRandom gr;

    double max=10.;
    double d_even_0_max = 0.;

```

```
  for(int i=0; i<NPROD; i++) {
    // Random numbers with an even distribution of
    // double values in the range [0.,max[
    d_even_0_max = gr.uniform_real<double>(max);

    // The following form is also possible, as the
    // template type can be determined from the type of max
    d_even_0_max = gr.uniform_real(max);
  }
```

Uniform `double` values in the range $[min, max[$

The `GRandom::uniform_real(const double&, const double&)` function creates uniformly distributed double random numbers in the range $[min, max[$. Internally, the function wraps the `GRandom::uniform_01<double>()` function. Note that `max` may also be negative.

Listing 31.3: Uniformly distributed double random numbers in the range $[min, max[$

```
  // Instantiate a random number generator
  GRandom gr;

  double min=0., max=10.;
  double d_even_min_max = 0.;

  for(int i=0; i<NPROD; i++) {
    // Random numbers with an even distribution of
    // double values in the range [0.,max[
    d_even_min_max = gr.uniform_real<double>(min, max);

    // The following form is also possible, as the
    // template type can be determined from the type of min/max
    d_even_min_max = gr.uniform_real(min, max);
  }
```

Normal Distribution with mean$==0$ and $\sigma==1$

Particularly evolutionary strategies need random numbers with a gaussian distribution. This is the most simple form of this random number type, as provided by the GRandom class. Internally, two double random numbers in the range $[0,1[$ are used to create two random number with a gaussian distribution.

Listing 31.4: Random numbers with a normal distribution (mean==0, $\sigma==1$

```
  // Instantiate a random number generator
  GRandom gr;

```

```
double d_std_gauss = 0.;

for(int i=0; i<NPROD; i++) {
  // A normal ("gaussian") distribution of random numbers
  // with mean 0 and sigma 1
  d_std_gauss = gr.normal_distribution<double>();
}
```

Normal Distribution with mean$==0$ and configurable σ

This function creates random numbers with a gaussian distribution, a mean value of 0 and a configurable σ value.

Listing 31.5: Random numbers with a normal distribution, mean 0 and configurable σ

```
// Instantiate a random number generator
GRandom gr;

double d_gauss_sigma = 0.;
double sigma = 2.;

for(int i=0; i<NPROD; i++) {
  // A normal ("gaussian") distribution of random numbers
  // with mean 0 and sigma "sigma"
  d_gauss_sigma = gr.normal_distribution<double>(sigma);

  // Note: Thanks to the "double" argument you could leave out
  // the <double> here
}
```

Normal Distribution with configurable mean and σ

This function creates random numbers with a gaussian distribution, with configurable mean and σ values.

Listing 31.6: Random numbers with a normal distribution (configurable mean and σ)

```
// Instantiate a random number generator
GRandom gr;

double d_gauss_mean_sigma = 0.;
double mean = 1.;
double sigma = 2.;

for(int i=0; i<NPROD; i++) {
  // A normal ("gaussian") distribution of random numbers
  // with configurable mean and sigma
  d_gauss_mean_sigma = gr.normal_distribution<double>(mean, sigma);
```

```

    // Note: Thanks to the "double" argument you could leave out
    // the <double> here
  }
```

Bi-Normal Distribution with Equal σ

This function adds two gaussians, centered around mean “mean”, with a configurable (but equal) σ, a distance "distance" from each other. The idea is to use this function in conjunction with evolutionary strategies, so we avoid searching with the highest likelihood at a location where we already know a good value exists. Rather we want to shift the highest likelihood for probes a bit further away from the candidate solution.

Listing 31.7: Random numbers with a bi-normal distribution (configurable mean and configurable, but identical σ)

```
  // Instantiate a random number generator
  GRandom gr;

  double d_bi_gauss;
  double mean = 1.;
  double sigma = 2.;
  double distance = 3.;

  for(int i=0; i<NPROD; i++) {
    d_bi_gauss
      = gr.bi_normal_distribution<double>(mean, sigma, distance);
  }
```

Bi-Normal Distribution with different σ values

This function adds two gaussians with sigmas "sigma1", "sigma2" and a distance of "distance" from each other, centered around mean. The idea is to use this function in conjunction with evolutionary strategies, so we avoid searching with the highest likelihood at a location where we already know a good value exists. Rather we want to shift the highest likelihood for probes a bit further away from the candidate solution.

Listing 31.8: Random numbers with a bi-normal distribution (configurable mean and configurable σ values)

```
  // Instantiate a random number generator
  GRandom gr;

  double d_bi_gauss_difsigma;
  double mean = 1.;
  double sigma1 = 2.;
  double sigma2 = 1.;
```

```
double distance = 3.;

for(int i=0; i<NPROD; i++) {
  d_bi_gauss_difsigma
    = gr.bi_normal_distribution<double>(mean, sigma1, sigma2, distance);
}
```

Boolean Random Numbers

This function produces boolean random numbers with an equal likelihood for `true` and `false`.

Listing 31.9: Boolean Random numbers with equal probability for true and false

```
// Instantiate a random number generator
GRandom gr;

bool bool_rnd = true;

for(int i=0; i<NPROD; i++) {
  bool_rnd = gr.uniform_bool();
}
```

Boolean Random Numbers with Configurable Probability

This function produces boolean random numbers with a configurable probability for `true` (and `true`).

Listing 31.10: Boolean Random numbers with individual probability for true and false

```
// Instantiate a random number generator
GRandom gr;

bool bool_rnd_weight = true;
double prob = 0.25; // 25% probability for "true"

for(int i=0; i<NPROD; i++) {
  bool_rnd_weight = gr.weighted_bool(prob);
}
```

Integer Random Numbers with Configurable min/max

This function produces integer random numbers inside (and including) a range `[min,max]`.

Listing 31.11:]Integer Random numbers in range [min, max]

```
// Instantiate a random number generator
GRandom gr;
```

```

boost::int32_t int_rand_min_max = 0.;
boost::int32_t min = -10, max = 10;

for(int i=0; i<NPROD; i++) {
  int_rand_min_max = gr.uniform_int(min, max);

  // Note: There is another function optimized for small integers:
  int_rand_min_max = gr.uniform_smallint(min, max);
}
```

Integer Random Numbers with Configurable max

This function produces integer random numbers inside (and including) a range `[0,max]`.

Listing 31.12:]Integer Random numbers in range [0, max]

```
// Instantiate a random number generator
GRandom gr;

boost::int32_t int_rand_max = 0.;
boost::int32_t max = 10;

for(int i=0; i<NPROD; i++) {
  int_rand_max = gr.uniform_int(max);

  // Note: There is another function optimized for small integers:
  int_rand_min_max = gr.uniform_smallint(min, max);
}
```

Chapter 32.

Brokering with the Courtier Library

This chapter discusses Geneva's *Courtier* library. As its core component, it includes a broker, which is complemented with a number of special purpose consumers. Jointly they form a generic system for the submission of work items to compute units, whose nature is determined by the consumer being used. Producers can submit work items to the broker through a generic interface. The `GBroker-Connector2T` class helps producers to submit and retrieve work items to/from the broker, and to handle missing responses.

Note that this chapter describes techniques that you will not usually have to deal with directly, if you are a user of a Geneva-based optimization application. However, the library might come in handy when designing a custom work submission system.

Key points: (1) Four main components make up Geneva's broker architecture: The broker itself, consumers, buffer ports and the `GBrokerConnector2T` class (2) Consumers process work items either locally or submit them to remote sites (3) Buffer ports contain two thread-safe queues, one for raw and one for processed items. (4) Multiple consumer and buffer port objects can be plugged into the broker. (5) Producers submit work items to the buffer ports' raw queues (6) Upon request from the consumer, the broker queries the buffer ports raw queues and hands the work items to the consumer (7) The consumer takes care that the work items get worked on (8) Processed work items are handed back to the "processed" queue of the buffer port (9) Work items particularly need to implement the `bool process()` function and need to be able store and retrieve an id associated with them (10) It is not guaranteed that work items passing through a consumer actually return (e.g. in the case of networked execution) (11) Producers thus need to be aware that not all submitted items might return (12) Fault tolerance is achieved through the `GBrokerConnector2T` class, which can take care of all communication with the broker for the producers, and handle missing responses.

The following section discusses the broker, including its architecture and the available configuration options. Note that the entire system is template-based. So except for very few interface functions, *Courtier* does not make any assumptions regarding the work items being processed by the system.

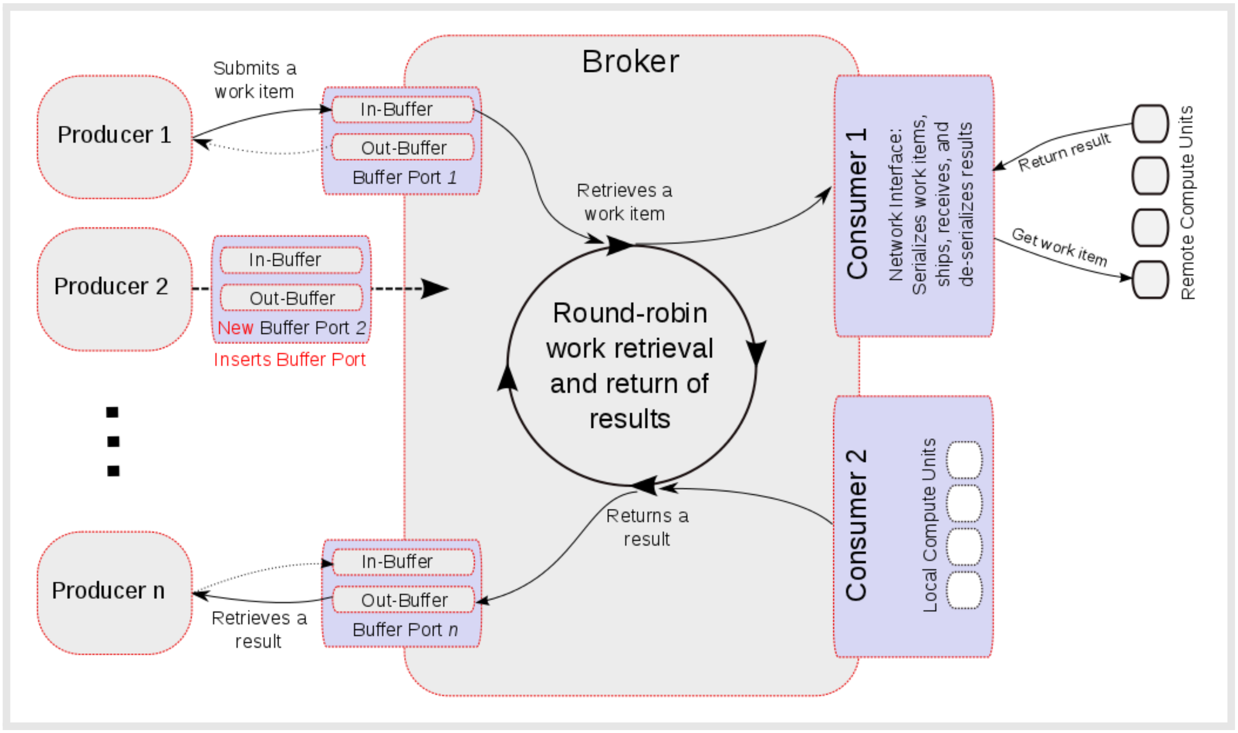


Figure 32.1.: *Several entities may submit items simultaneously through "buffer ports", which are plugged into the broker. The broker will deliver items in a round-robin fashion to one or more consumers that ask for work items. Consumers might then deliver the work items to remote sites, or process them locally (depending on the consumer type). Results are shipped back to the original buffer port, from where they can be retrieved by the producer.*

32.1. Architecture

The broker sits at the heart at the entire task submission system. Its architecture can be easily understood when looking at figure 32.1.

32.1.1. Buffer Ports

Entities (called "**E**" in this section) wishing to submit work items for further processing may register "buffer ports" with the broker. A buffer port contains two thread-safe queues, to which items may be concurrently written to and read from[1]. The first queue is used for the submission of "raw" work items, and processed items can be retrieved by **E** from the second queue. More than one buffer port may be plugged into the broker simultaneously.

[1]See section 33.1 on page 281 for a detailed description of the thread-safe queue.

32.1.2. Consumers

A consumer is a class that initiates the processing of work items. In the easiest case[2] it will just spawn a number of threads for the local processing of work items on the available CPU cores.

Consumers may also wait for connections of networked clients. They will then need to care for the serialization of work items (which is built in already into Geneva's optimization-related classes), ship the data to the remote entity and give back processed items to the broker.

Geneva's *Courtier* library currently implements the following consumers:

- `GBoostThreadConsumerT` processes work items in parallel in a number of threads. By default, it will try to start as many threads as computing cores are available on the system.
- `GAsioTCPConsumerT` uses the `Boost.ASIO` library to ship serialized work items to networked clients. The `GAsioTCPClientT` class can be used to implement networked clients with ease.
- `GSerialConsumerT` is meant for debugging only. It processes one work item after the other locally.

We plan to add a consumer for the **M**essage **P**assing **I**nterface (MPI) soon. Given the separation between consumer implementation and job submission shown in figure 32.1, the task is expected to be simple from a technical perspective.

Communication with a GPGPU through OpenCL has been implemented, but has not been made available publicly.

32.1.3. Fault Tolerance

Note that it is *not* guaranteed that all work items that pass through a consumer actually do return. This is particularly true when dealing with networked execution. Hence producers who submit work items to the broker need to be able to deal with missing responses. See section 32.5 for one way of achieving fault tolerance.

32.1.4. Work Flow

The broker's work flow is driven by the consumers. When a consumer requests a work item, the broker will query the available buffer ports in succession and hand available work items to the consumer. The consumer will then take care to process the item locally or ship it to remote work units.

Processed work items are handed back to the broker, which checks the id of the originating buffer port and adds the work item to it. The producer **E** may then retrieve the final result from the queue.

[2]... intended for production use

32.2. Requirements for Work Items

The broker is implemented as a template library. Hence it assumes a specific interface available for work items. If you are interested in the details, have a look at the `GSubmissionContainerT` class in the Geneva distribution. It may (but does not need to) serve as the base class of work items submitted to the broker. Work items must implement the following functionality (or acquire it by deriving from `GSubmissionContainerT`):

- `bool process()` triggers the processing of a work item. All processing logic needs to be implemented by the work item. The function returns a boolean which should indicate whether processing has lead to a useful result. This may be used by remote entities to decide whether processed items should be shipped back to the broker. E.g. in optimization algorithms, if the processing of an individual didn't lead to an improved fitness, the remote client may decide to discard the work item completely, based on the result of the `process()` call. **Implementation of this function is required even when deriving from `GSubmissionContainerT`, as it is purely virtual there.**
- A work item **must** be able to store what is called a "courtier id". It has the type `boost::tuple<Gem::Courtier::ID_TYPE_1, Gem::Courtier::ID_TYPE_2>`. The id is used to identify the buffer port from which the work item has originated. Implementation of this function may be omitted when deriving from `GSubmissionContainerT`.
- A work item **must** implement the `setCourtierId()` and `getCourtierId()` functions, so the broker can set and get the id. Implementation of this function may be omitted when deriving from `GSubmissionContainerT`.

`void loadConstantData(boost::shared_ptr<T>)` is an **optional** function. When implemented by derived classes, work items holding large amounts of constant data may be deposited at a remote site as part of the client code. The work item then needs to implement code to load the constant data from the other entity (which has the same type as the work item). This can help reduce the amount of data to be serialized and shipped to remote locations.

When a networked consumer is used, work items also **need to be serializable** using the conventions of the `Boost.Serialization` library. See the discussion of `loadConstantData()` above for how to avoid serializing large amounts of constant data[3].

32.3. Accessing the Broker

The broker is implemented as a global singleton. User-code should `#include "courtier/GBrokerT.hpp"`. The broker can then be accesses through a call to `GBROKER(T)`, where `T` represents the type of the work item. As an example, the broker used for shipping the Geneva optimization system's individuals to consumers can be accessed through `GBROKER(Gem::Geneva::GParmameterSet)`.

[3]Example: Training data of a neural network

32.4. Configuration Options of the Broker

There aren't many configuration options for the broker, as most of the real action happens on the level of the consumers. Unless you intend to write your own consumer, the only options you need to know are both called `enrol()`. One accepts buffer ports wrapped in a `boost::shared_ptr`, the other accepts a consumer which is also wrapped into a `boost::shared_ptr`.

The function `void enrol(boost::shared_ptr<GBufferPortT<boost::shared_ptr<work_item> > >)` allows you to register a buffer port object with the broker. The buffer port accepts a work item, which is again wrapped into a smart pointer. More than one buffer port may be enrolled with the broker.

`void enrol(boost::shared_ptr<GConsumer>)` expects a `boost::shared_ptr<>` to a consumer. The broker only talks to the consumer base class and does not know what a particular consumer does. More than one consumer may be enrolled with the broker.

We strongly recommend to `typedef` the arguments of `enrol()` – we only show them here in full beauty so you understand what the function expects.

32.5. Submission of Work Items

It is usually not necessary for producers of work items to talk directly to the broker. Instead, it is recommended to use the `GBrokerConnector2T` class. It will instantiate a buffer port object for you and establish the connection to the broker.

Three different working modes exist, which are specified as a constructor argument (type `Gem::Courtier::submissionReturnMode`):

- **`INCOMPLETERETURN`** means that it is expected that some work items of the current submission cycle might not return. The `GBrokerConnector2T` object will then calculate a suitable timeout and return to the caller, even if not all work items have returned. If all work items have returned to the caller before the timeout, the `GBrokerConnector2T` object will return to the caller immediately. In the context of parametric optimization, this mode is usefull in all cases where an optimization algorithm can cope with missing returns and repair itself. This is e.g. the case with Evolutionary- and Swarm Algorithms.
- **`RESUBMISSIONAFTERTIMEOUT`** means that work items that have not returned after a timeout will be resubmitted to the broker. There is a configurable maximum number of resubmissions. It may happen that the same work item returns more than once (e.g. a work item from the first submission and one from a re-submission). Redundant work items are discarded and the `GBrokerConnector2T` keeps track of which work items have returned. In the context of parametric optimization this mode is usefull for gradient descents, as these cannot continue without all work items having returned.
- **`EXPECTFULLRETURN`** means that `GBrokerConnector2T` will wait indefinitely for returns. This mode of operation is useful in situations where one is sufficiently sure that all work items will return, e.g. in the case of a multi-threaded consumer, or possibly also in a cluster

environment with an MPI-Consumer[4]. Choosing this mode avoids the overhead associated with the other two modes.

It then provides you with an interface to submit work items to the broker using different variants of the `workOn` member function:

Listing 32.1: Work submission with exact control over which work items should be submitted, which items did not return, plus retrieval of work items from previous job submissions

```
bool workOn(
   std::vector<boost::shared_ptr<processable_type> >& workItems
   , std::vector<bool>& workItemPos
   , std::vector<boost::shared_ptr<processable_type> >& oldWorkItems
   , const std::string& originator = std::string()
);
```

The variant shown in listing 32.1 accepts a std::vector of work items, a std::vector of boolean values indicating which work items should be submitted[5], and a std::vector for storing work items from previous job submissions. A boolean return value indicates, whether all submitted work items have returned.

When the function has returned, the `workItems` vector might hold a mixture of successfully processed and "raw" work items that could not be processed. The `workItemPos` vector will hold the value `Gem::Courtier::GBC_PROCESSED` at the position of each work item that has been processed successfully, and `Gem::Courtier::GBC_UNPROCESSED` for every unprocessed work item. Thus, if all work items were processed, all entries of `workItemPos` are set to `GBC_PROCESSED`.

As the function may return after a timeout, before all work items of the current submission have returned, we also need a storage location for older items. They are put into the `oldWorkItems` vector and are always processed.

As a final, optional argument it is possible to give the function information on the caller. This is helpfull for debugging messages and strictly optional.

Listing 32.2: Work submission in a range, plus retrieval of work items from previous job submissions

```
bool workOn(
   std::vector<boost::shared_ptr<processable_type> >& workItems
   , const boost::tuple<std::size_t, std::size_t>& range
   , std::vector<boost::shared_ptr<processable_type> >& oldWorkItems
   , const bool& removeUnprocessed = true
   , const std::string& originator = std::string()
);
```

The `workOn`-variant in listing 32.2 accepts an index range instead of an array of specific positions as input, in addition to the workItems vector. It does not, however, return the positions of workItems that did not return from processing. Instead, by default, such work items will be erased from the

[4]. . . which has so far not been implemented

[5]A value of `Gem::Courtier::GBC_UNPROCESSED` indicates that this is a work item that is still unprocessed and needs processing. A value of `Gem::Courtier::GBC_PROCESSED` indicates that this item is already processed and does not need to be submitted

`workItems` vector, unless the caller specifically instructs the function not to do this. As in the previous version, completed work items from previous submission cycles will be stored in the `oldWorkItems` vector.

Listing 32.3: Work submission in a range, plus retrieval of work items from previous job submissions

```
bool workOn(
   std::vector<boost::shared_ptr<processable_type> >& workItems
   , std::vector<boost::shared_ptr<processable_type> >& oldWorkItems
   , const bool& removeUnprocessed = true
   , const std::string& originator = std::string()
)
```

The final `workOn`-variant shown in listing 32.3 works similar to 32.2, but submits all items in the `workItems` vector.

Chapter 33.

Common Functionality and Classes

This chapter describes miscellaneous classes and utility functions that have been implemented for the Geneva library. They are available through the *common* library. Note that, over time, some of these classes might become independent libraries.

Key points: (1) Geneva's thread-safe queue allows concurrent threads to read and write simultaneously. It takes into account allowed maximum sizes of the queue. Producers and consumers will inform each other once an empty queue has been filled again or space has again become available if the queue was full. Blocked threads will thus not consume compute time. (2) The `raiseException` macro emits `gemfony_error_condition` objects, which are in addition equipped with diagnostic information about the location where the error has occurred. (3) The `GParserBuilder` class implements a complete framework for creating and parsing configuration files in JSON format (4) The `GFactoryT<>` class template allows to implement factoy classes that produce custom objects and read options from a configuration file. (5) Geneva's Singleton implementation emits `boost::shared_ptr<>` smart pointers in order to reduce the dependencies between different singletons using each others services (6) The `GGlobalOptionsT<>` class template can be used to store arbitrary data at a global scope. (7) The `GThreadGroup` class template implements a simple thread group. It is based on the corresponding Boost class. (8) The `GThreadPool` class template implements a simple thread pool (9) The `GPlotDesigner` lets you create ROOT scripts for your data with relative ease

33.1. A Thread-Safe Queue

The Geneva optimization library is heavily multithreaded, from the implementation of a global random number factory to the concurrent evaluation of individuals. Thus, thread-safe queues as a means of concurrently submitting and retrieving data and work items from multiple threads were needed. They have been implemented in the `GBoundedBufferT` template[1]. Geneva's implementation is based on the `bounded_buffer_comparison.cpp` example by Jeff Garland, which is shipped together with the Boost library collection.

[1]Note that also a `GBoundedBufferWithIdT` class exists which adds a unique id to `GBoundedBufferT`, in order to make it recognizable by Geneva's broker.

Note that the actual implementation in the Geneva library is more elaborate, offering for example *pushs* and *pops* with a timeout. Listing 33.1 illustrates the general principles[2].

Listing 33.1: General principles of a thread-safe queue

```cpp
template <class T>
class TSQueue {
public:
  explicit TSQueue(std::size_t capacity) : capacity_(capacity) {}

  void push_front(T item) {
    boost::unique_lock<boost::mutex> lock(mutex_);
    not_full_.wait(lock, boost::bind(&TSQueue<T>::is_not_full, this));
    container_.push_front(item);
    lock.unlock();
    not_empty_.notify_one();
  }

  void pop_back(T* pItem) {
    boost::unique_lock<boost::mutex> lock(mutex_);
    not_empty_.wait(lock, boost::bind(&TSQueue<T>::is_not_empty, this));
    *pItem = container_.back();
    container_.pop_back();
    lock.unlock();
    not_full_.notify_one();
  }

private:
  TSQueue(const TSQueue&);                // Intentionally private and undefined
  TSQueue& operator=(const TSQueue&);    // Intentionally private and undefined

  bool is_not_empty() const { return container_.size() > 0; }
  bool is_not_full() const { return container_.size() < capacity_; }

  const std::size_t capacity_;

  std::deque<T> container_;

  boost::mutex mutex_;
  boost::condition_variable not_empty_;
  boost::condition_variable not_full_;
};
```

It would be comparatively easy to create a thread-safe queue, simply by protecting each access to the `std::deque<>` with a Mutex. However, such an implementation does not take into account that the queue may be empty or that its size might have reached a user-defined upper limit. Producers and consumers would thus have to implement a “busy wait” until space becomes available again in the queue or items have been added.

[2]. . . but does not represent the actual implementation, which is too long to be shown here. See also the excellent book “Concurrency in Action” by Anthony Williams [144] for an in-depth introduction into the topic.

It would be much nicer if producers could inform consumers (and vice versa), once this has happened. In Boost (and the new C++11 thread facilities, which are modelled after Boost.Thread), this can be done with the help of condition variables. This is realized in listing 33.1 as well as in the more complex Geneva implementation.

33.2. Raising Exceptions and Logging

Geneva implements its own exception class, meant to facilitate raising exceptions. Its main addition over standard exception classes is that it is possible to store a text in it, by passing a string to the constructor. It is also possible to submit the class – called `gemfony_error_condition` – to a stream, thus facilitating the output of stored data. Two possibilities exist for raising exceptions.

33.2.1. raiseException

A **macro** with the name of `raiseException()` facilitates raising these exceptions. It also adds information about the location of the "throw" to the output. Note that the output is specific to the Geneva library, as it also asks the user to submit bug reports, should an error occur. Usage of `raiseException()` is shown in listing 33.2.

Listing 33.2: Raising exceptions with a Geneva macro

```
[...]
int someInt = 0;

try
{
  if(someInt = 1) { // This will always trigger
    raiseException(
      "Got someInt == " << someInt << std::endl
      << "You might want to write \"if(1 == someInt)\" next time" << std::endl
      << "This way the compiler will catch a missing \"=\"" << std::endl
    );
  } else {
    std::cout << someInt << std::endl;
  }
} catch(Gem::Common::gemfony_error_condition e) {
        std::cout << "The end is near: " << std::endl
                  << e << std::endl;
        std::terminate();
}

[...]
```

Listing 33.3 shows an example of what an exception might look like. Note that we have submitted the exception class directly to to the stream.

Listing 33.3: Raising exceptions with a Geneva macro

```
ERROR in file /home/developer/Geneva/src/geneva/GMultiThreadedEA.cpp
near line 212 with description:

In GMultiThreadedEA::finalize():
Invalid number of serverMode flags: 1000/100

If you suspect that this error is due to Geneva,
then please consider filing a bug via
http://www.gemfony.eu (link "Bug Reports") or
through http://www.launchpad.net/geneva

We appreciate your help!
The Geneva team
```

On a side note, this means that the algorithm had expected a given number of flags (preventing re-evaluation), but has found a population size exceeding the number of flags. The (now corrected) error happened as a consequence of the ability of Geneva's evolutionary algorithm populations to grow.

33.2.2. `glogger` for exception handling

The `raiseException` macro is rather light-weight. A more heavy-weight (but far more versatile) possibility for raising exceptions in Geneva is the GLogger class, and the supplied global singleton `glogger`. It is available as soon as you have included the `GLogger.hpp` header and linked with the `Common` library. Once this is done, code like it is shown in listing 33.4 becomes possible.

Listing 33.4: Raising exceptions with glogger

```
[...]
int someInt = 0;
try {
  if(someInt = 1) { // This will always trigger
    glogger
    << "Got someInt == " << someInt << std::endl
    << "You might want to write \"if(1 == someInt)\" next time" << std::endl
    << "This way the compiler will catch a missing \"=\"" << std::endl
    << GEXCEPTION;
  } else {
    std::cout << someInt << std::endl;
  }
} catch(Gem::Common::gemfony_error_condition e) {
        std::cout
        << "The end is near: " << std::endl
        << e << std::endl;
        std::terminate();
}
[...]
```

So it becomes possible to actually stream any kind of information to glogger that could otherwise be streamed to a normal `std::cout`. The “manipulator” `GEXCEPTION` makes sure an exception is raised, complete with diagnostic information about the location of the error (file name and approximate line of code inside of the file)[3].

The output will be very similar to the one shown for the `raiseException` macro, but will in addition be complemented with a time stamp and logged to a file named `GENEVA-EXCEPTION.log`.

Note that the exception is indeed raised through a global singleton, so that no information should be lost, even when the exception is raised from within a thread without a `try/catch` block.

A modifier `GTERMINATE` instead of `GEXCEPTION` will internally call `std::terminate`, which (arguably) might be more suitable for usage inside of a constructor. The output will appear on the console as well as in a file named `GENEVA-TERMINATION.log`.

33.2.3. glogger for logging

The `glogger` class has further abilities, such as logging and emitting warning messages. The first step for accessing this functionality is to register “log targets” with the `GLogger` class.

Logging is done through each of the channels identified through these objects. Geneva comes equipped with two log targets by default, the `GConsoleLogger` class, which will output information to the console (through std::cout), and the `GFileLogger` class, whose output ends up in a file[4].

Such logging targets can be registered through code such as the one shown in listing 33.5.

Listing 33.5: Adding log targets to the logger

```
[...]
boost::shared_ptr<GBaseLogTarget> gcl_ptr(new GConsoleLogger());
boost::shared_ptr<GBaseLogTarget> gfl_ptr(new GFileLogger("./somePathToLogFile.txt"));

glogger.addLogTarget(gcl_ptr);
glogger.addLogTarget(gfl_ptr);
[...]
```

Once log targets have been registered, it is possible to perform logging in a very similar way to the procedure shown in section 33.2.2 exception handling. However, instead of `GEXCEPTION`, one should use the `GLOGGING` manipulator. Output will be written to the channels that have been registered with the `GLogger` class (through the `glogger` singleton), as shown in listing 33.5.

Another manipulator, called `GWARNING`, will emit warnings instead of simple log messages. In contrast to normal logging messages, they are equipped with time stamps and the location from where the warning was triggered.

[3]Note that `GEXCEPTION` is indeed a macro which wraps the `GManipulator` class.

[4]On a side note, it should be possible to design further targets without too many problems, such as a network logging target.

33.2.4. Instant logging to files, stdout and stderr

Unfortunately, `std::cout` is not thread-safe, so that output written to this stream from different threads might be garbled. `glogger` can be used to output information to `stdout` and `stderr`, using the two manipulators `GSTDOUT` and `GSTDERR`. Any log targets registered with the `GLogger` class will be ignored in this case.

Likewise, it is possible to output data to specific files without registration of a corresponding log target. This can be useful for debugging purposes, if the output of some part of a function should end up in a file.

Listing 33.6 shows examples for all three cases. Note that the name of the file is given as an argument to glogger here.

Listing 33.6: Adding log targets to the logger

```
// [...]
// Output to a specific file
glogger("file.txt")
<< "Some information " << 3 << " " << 4 << std::endl << GFILE;

// Output to stdout
glogger << "std::out-information" << std::endl << GSTDOUT;

// Output to stderr
glogger << "std::err information" << std::endl << GSTDERR;
// [...]
```

33.3. Parsing Configuration Files

Geneva implements the `GParserBuilder` framework for parsing configuration files in JSON[5] format. The idea behind choosing this format is to facilitate the creation of web interfaces for the editing and creation of configuration files. The techniques have been used extensively throughout the optimization framework. Every object adds its options to a `GParserBuilder` object, so that changes to the class can be directly mapped to the configuration process. `GParserBuilder` can both create and read configuration files. Its use is not limited to the optimization framework, as it has no dependencies on it.

The Geneva distribution has a complete example (`common/GConfigFileCreation`) on how to create and read configuration files. In this section we only want to present the principles.

Listing 33.7: Creating and parsing configuration files with `GParserBuilder`

```
//[...]
const std::string fileName = "./config/configFile.json";

// Create the parser builder
```

[5]Java Script Object Notation

```
Gem::Common::GParserBuilder gpb;

//------------------------------------------------------------------
// We can directly set a variable by providing a reference to it.
int i = 0; const int IDEFAULT = 0;

gpb.registerFileParameter<int>(
        "iOption"
        , i
        , IDEFAULT
        , Gem::Common::VAR_IS_ESSENTIAL // Could also be VAR_IS_SECONDARY
        , "This is a comment; This is the second line of the comment"
);

//------------------------------------------------------------------
// Registering a call-back function (which in this
// case sets a globally defined integer variable
gpb.registerFileParameter<int>(
        "iOption2"
        , SOMEGLOBALINTDEFAULT
        , setGlobalInt
        , Gem::Common::VAR_IS_SECONDARY // Could also be VAR_IS_ESSENTIAL
        , "This is a comment for call-back option"
);

//------------------------------------------------------------------
// Adding a reference to a vector of configurable type to the collection.

std::vector<double> targetDoubleVector; // Will hold the read values
std::vector<double> defaultDoubleVec5; // The default values
defaultDoubleVec5.push_back(0.);
defaultDoubleVec5.push_back(1.);

gpb.registerFileParameter<double>(
        "vectorOptionsReference"
        , targetDoubleVector
        , defaultDoubleVec5
        , Gem::Common::VAR_IS_ESSENTIAL // Could also be VAR_IS_SECONDARY
        , "And yet another comment"
);

//------------------------------------------------------------------
// [...]

// Check the number of registered options
std::cout << "Got " << gpb.numberOfOptions() << " options." << std::endl;

if(createConfigFile) {
        std::string header = "This is a not so complicated header;with a second line";
        bool writeAll = true; // If set to false, only essential are written out
        gpb.writeConfigFile(fileName, header, writeAll);
} else if(readConfigFile){
       gpb.parseConfigFile(fileName);
}

// [...]
```

The entire process is relatively simple. After default-construction of a `GParserBuilder` object you can add different kinds of configuration options using the `registerFileParameter()` command. Different overloads exist. At the time of writing, `GParserBuilder` implements the following possibilities:

- Registering a reference to a single configuration parameter
- Registering a call-back function which will be called with a single, parsed value
- Registering a call-back function which will be called with two related parameters (example: lower and upper boundary of a random number generator)
- Registering a reference to a `std::vector<>`. This can be used if it is not yet clear how many configuration options there will be
- Registering a call-back function that will be called with a `std::vector` of values.
- Registering a reference to a `boost::array` of fixed size. This can be used to simultaneously store and read multiple variables when there is a strict requirements for a fixed number of values
- Registering a call-back function which will be called with a `boost::array` of fixed size, when the configuration file has been parsed.

Once the options have been registered with their default values (or at least one default value in the case of the `std::vector<>`), the call `writeConfigFile(...)` will create a configuration file. The function accepts three parameters: The name and path of the configuration file, a `std::string` holding a description for the file header, and a specification whether only "essential" variables should be written to file. If so, variables which are tagged `VAR_IS_SECONDARY`, will not be written out.

As of version 1.4.1 of Geneva it is also possible to stream comments to the `GParserBuilder::registerFileParameter()` call. Examples are again shown in `common/GConfigFileCreation`. Listing 33.8 gives an example.

Listing 33.8: Comments may also be streamed to the `registerFileParameter()` call

```
gpb.registerFileParameter<int>(
    "iOption"
        , i
        , IDEFAULT
)
<< "This is a comment" << std::endl
<< This is the second line of the comment";
```

Reading data from the configuration file is achieved by means of the `readConfigFile()` function, which only receives the file name as argument.

As a small caveat, if you register references to variables, you need to make sure that they are valid for the entire lifetime of the `GParserBuilder` object. Thus it is recommended to use `GParserBuilder` only inside of a single function and apply it to local variables, or to apply it to member variables of a struct or class, of which the `GParserBuilder` object is a part.

Listing 33.9 shows an example of a configuration file that was created with this method. Only part of the file is shown.

Listing 33.9: An example for a configuration file that was created with `GParserBuilder`

```
//------------------------------------------------------------------------
// This is a not so complicated header
// with a second line
// File creation date: 2011-Oct-02 17:48:45
//------------------------------------------------------------------------

{
    "iOption":
    {
        "comment": "This is a comment",
        "comment": " This is the second line of the comment",
        "default": "0",
        "value": "0"
    },
    "iOption2":
    {
        "comment": "This is a comment for call-back option",
        "default": "1",
        "value": "1"
    },
    "combinedLabel":
    {
        "iOption3":
        {
            "comment": "A comment concerning the first option",
            "default": "3",
            "value": "3"
        },
        "dOption1":
        {
            "comment": "A comment concerning the second option",
            "comment": "with a second line",
            "default": "3",
            "value": "3"
        }
    },

    // [...]
}
```

Note that, if you have specified a relative path for your configuration file, it is possible to specifiy an offset to your configuration file by setting the environment variable **`GENEVA_CONFIG_BASENAME`**.

Parsing, particularly of Geneva configuration files, can in this way become independent of the place of execution of an executable wishing to parse a given file.

33.4. Creating Factories

Geneva uses factories in many areas. They are particularly useful for the creation of individuals, but are likewise also being used for the creation of optimization algorithms. The *Common* library tries to facilitate the creation of factory classes through the `GFactoryT<>` class template. **It has been designed particularly with the `GParserBuilder` class in mind.**

Examples for the use of the `GFactoryT<>` class can be found throughout the Geneva distribution. It is suggested to have a look at some of the sample individuals. We will give a short illustration of the factory which was implemented for the `GFunctionIndividual` example. Listing 33.10 shows the declaration of the `GFunctionIndividualFactory` class[6].

Listing 33.10: Simplified declaration of the GFunctionIndividualFactory

```
class GFunctionIndividualFactory
        : public Gem::Common::GFactoryT<GParameterSet>
{
public:
        GFunctionIndividualFactory(const std::string& configFile);
        virtual ~GFunctionIndividualFactory();

protected:
        virtual boost::shared_ptr<GParameterSet>
              getObject_(Gem::Common::GParserBuilder& gpb, const std::size_t& id);
        virtual void describeLocalOptions_(Gem::Common::GParserBuilder& gpb);
        virtual void postProcess_(boost::shared_ptr<GParameterSet>& p);

private:
        // The default constructor. Intentionally private and undefined
        GFunctionIndividualFactory();

        Gem::Common::GOneTimeRefParameterT<double> adProb_;
        Gem::Common::GOneTimeRefParameterT<boost::uint32_t> adaptionThreshold_;
        // Further variables are not shown
};
```

The factory is derived from the `Gem::Common::GFactoryT<T>` class, whose template parameter is set to `GParameterSet` – items produced by the factory will have this type. The default constructor is disabled. The only available constructor accepts the name of the configuration file.

It will typically initialize a set of local variables to default values. These variables really represent configuration options of the objects this factory is supposed to create. The constructor will of course also store the name of the configuration file locally.

[6]Note that we have left out quite a few non-essential functions for reasons of readability.

The factory now needs to overload three functions of the base class:

- `getObject_(...)` is given a reference to a `GParserBuilder` object (compare section 33.3) and an id. The id is incremented by the base class upon each call to `getObject_()`. It allows the factory function to perform special actions for specific ids. For example, the first call might want to create different objects than all consecutive calls. `getObject_(...)` will typically ony default-construct the desired object and return it inside of a smart pointer (`boost::shared_pointer<>`). However, the object to be constructed may choose to add own configuration options to the parser builder. This is used extensively in Geneva's optimization framework.
- `describeLocalOptions_(Gem::Common::GParserBuilder&)` is the location where configuration options local to the factory class will be registered with the `GParserBuilder` object. Typically these are options that will be used later to configure the object to be created by the factory. The parsed values will be stored locally in the factory class.
- `postProcess_(boost::shared_ptr<GFunctionIndividual>&)` is where the stored options from the last step are used to further configure the target object.

The user can now instantiate the factory class. As it implements an `operator()`, creating new objects is very much like a function call, albeit much more versatile.

Note, though, that the items returned have the type `boost::shared_ptr<GParameterSet>`, i.e. they point to the base class of the actual object produced. You can extract the target type itself using the `get<GFunctionIndividual>()` function of `GFactoryT<>`.

33.5. Singletons

Singletons are global objects that live outside of the `main()` function. They can be used to gain access to functionality relevant to all entities of a program. Only a single object is ever produced, so that all users of this object share the same data. Geneva uses Singletons in many places, of which the broker architecture (compare chapter 32) and the random number factory (see chapter 31) are important examples. Geneva's singleton implementation is special in that it emits `boost::shared_ptr<>` smart pointers. The reason behind this is that in C++03, the order of destruction of singletons is not defined. Thus, if one singleton uses services of another, the program might stall. If it can however store a reference-counted smart pointer such as `boost::shared_ptr<>`, the object will only go out of scope when the last smart pointer is erased. Geneva's factory `GSingletonT` is implemented as a template, so that it can host practically any type of object. If this object is not meant to be default-constructed, you may overload the `TFactory_GSingletonT<>` function template for your use case. `GSingletonT` will call this function for the first creation of the object.

33.6. Global Options

The `GGlobalOptionsT<>` template is an experimental feature meant to facilitate access to global options. It is based on the Singleton described in section 33.5. As one example, the Mona Lisa example described in section 9.1 uses this class to store a picture of the Mona Lisa, so it doesn't need to get loaded from disk upon every iteration of the optimization process.

33.7. A Thread Group

The `GThreadGroup` class implements a simple group of `boost::thread` objects that can be simultaneously created and stopped. It is based on the `Boost.ThreadGroup` class, but has been augmented with additional features.

33.8. A Thread Pool

The Geneva library collection contains a simple thread pool implementation (called `GThread-Pool`), based on Boost.ASIO. A thread pool receives work items from a pool. A user-defined number of threads process these work items. When a thread finishes with one work item, it obtains the next item from the queue or waits idle, until the queue is filled with work items again.

Listing 33.11 shows the public interface functions of this class. Note that this class might go away after a generalized Boost thread pool has become available.

Listing 33.11: Public interface functions of the `GThreadPool` class

```
class GThreadPool :private boost::noncopyable // prevent copying of pool
{
public:
        GThreadPool();
        GThreadPool(const std::size_t&); // Initialization with number of threads
        ~GThreadPool();

        void setNThreads(std::size_t);
        std::size_t getNThreads() const;

        bool wait(); // Block until all jobs have finished

        bool hasErrors() const; // Check whether errors have occurred
        void getErrors(std::vector<std::string>&); // Extract errors
        void clearErrors(); // Clear error logs

        // Submits the task to Boost.ASIO's io_service.
        // This function will return immediately.
        template <typename F> void async_schedule(F f) { /* code not shown */ }

private:
        // Private functions not shown
};
```

33.9. The Plot Designer

In the course of performing optimizations, you will often come across the need to graphically represent some of the data coming out of the program. Be it that you need to visualize the progress of the optimization or that you want to plot one or more parameters of your individuals, in one- or two-dimensional plots.

Geneva already uses the services of the ROOT analysis framework (compare appendix C) in many places. ROOT uses C++ as a *scripting language*. Plots are thus described using C++ classes and functions. However, ROOT can have quite a steep learning curve, and letting your optimization programs automatically create ROOT scripts can be a bit challenging. For this reason we have included a facility in the Geneva library collection that let you create these scripts with relative ease. By the same token, the library can be easily extended to fit your particular design needs. Geneva contains a demo with the name `GPlotDesignerTest` that shows the usage of this facility. Figure 33.1 shows a plot created with this demo. Listing 33.12 shows the code that was used to create this plot.

The logic of creating plots with this facility is straight forward. A plot, referenced through the `GPlotDesigner` class, may contain sub-canvases. The layout is specified by providing the constructor with the number of sub-canvases in x- and y-direction. The user then creates canvas objects, such as `GGraph2D` (a two-dimensional plot, in our example using the `SCATTER` mode), `GFunctionPlotter1D` or `GFunctionPlotter2D`. Many other canvas objects are available (compare the reference documentation). Each canvas resides in a `boost::shared_ptr`. Once it has been filled with data, it is registered with the `GPlotDesigner` object, which can then be instructed to write the resulting ROOT script to disc, where its contents can then be visualized offline.

Listing 33.12: The code of the GPlotDesigner example

```
using namespace Gem::Common;

int main(int argc, char** argv) {
        boost::tuple<double,double> minMaxX(-M_PI,M_PI);
        boost::tuple<double,double> minMaxY(-M_PI,M_PI);

        boost::shared_ptr<GGraph2D> gsin_ptr(new GGraph2D());
        gsin_ptr->setPlotMode(Gem::Common::SCATTER);
        gsin_ptr->setPlotLabel("A sine function, plotted through TGraph");
        gsin_ptr->setXAxisLabel("x");
        gsin_ptr->setYAxisLabel("sin(x)");

        boost::shared_ptr<GGraph2D> gcos_ptr(new GGraph2D());
        gcos_ptr->setPlotMode(Gem::Common::SCATTER);
        gcos_ptr->setPlotLabel("A cosine function, plotted through TGraph");
        gcos_ptr->setXAxisLabel("x");
        gcos_ptr->setYAxisLabel("cos(x)");

        for(std::size_t i=0; i<1000; i++) {
                double x = 2*M_PI*double(i)/1000. - M_PI;

                (*gsin_ptr) & boost::tuple<double, double>(x, sin(x));
                (*gcos_ptr) & boost::tuple<double, double>(x, cos(x));
        }

        boost::shared_ptr<GFunctionPlotter1D>
            gsin_plotter_1D_ptr(new GFunctionPlotter1D("sin(x)", minMaxX));
        gsin_plotter_1D_ptr->setPlotLabel("A sine function, plotted through TF1");
        gsin_plotter_1D_ptr->setXAxisLabel("x");
        gsin_plotter_1D_ptr->setYAxisLabel("sin(x)");

        boost::shared_ptr<GFunctionPlotter1D>
            gcos_plotter_1D_ptr(new GFunctionPlotter1D("cos(x)", minMaxX));
        gcos_plotter_1D_ptr->setPlotLabel("A cosine function, plotted through TF1");
        gcos_plotter_1D_ptr->setXAxisLabel("x");
        gcos_plotter_1D_ptr->setYAxisLabel("cos(x)");

        boost::shared_ptr<GFunctionPlotter2D>
            schwefel_plotter_2D_ptr(new GFunctionPlotter2D("-0.5*(x*sin(sqrt(abs(x)))
                                 + y*sin(sqrt(abs(y))))", minMaxX, minMaxY));
        schwefel_plotter_2D_ptr->setPlotLabel("The Schwefel function");
        schwefel_plotter_2D_ptr->setXAxisLabel("x");
        schwefel_plotter_2D_ptr->setYAxisLabel("y");
        schwefel_plotter_2D_ptr->setYAxisLabel("Schwefel function");
        schwefel_plotter_2D_ptr->setDrawingArguments("surf1");

        boost::shared_ptr<GFunctionPlotter2D>
            noisyParabola_plotter_2D_ptr(new GFunctionPlotter2D("(cos(x^2+y^2) +
                                        2)*(x^2+y^2)", minMaxX, minMaxY));

        noisyParabola_plotter_2D_ptr->setPlotLabel("The noisy parabola");
        noisyParabola_plotter_2D_ptr->setXAxisLabel("x");
        noisyParabola_plotter_2D_ptr->setYAxisLabel("y");
        noisyParabola_plotter_2D_ptr->setYAxisLabel("Noisy parabola");
        noisyParabola_plotter_2D_ptr->setDrawingArguments("surf1");

        GPlotDesigner gpd("Sine and cosine and 2D-function", 2,3);

        gpd.setCanvasDimensions(1200,1400);
        gpd.registerPlotter(gsin_ptr);
        gpd.registerPlotter(gcos_ptr);
        gpd.registerPlotter(gsin_plotter_1D_ptr);
        gpd.registerPlotter(gcos_plotter_1D_ptr);
        gpd.registerPlotter(schwefel_plotter_2D_ptr);
        gpd.registerPlotter(noisyParabola_plotter_2D_ptr);

        gpd.writeToFile("result.C");
```

```
}
```

33.10. Parsing Formulas

In the context of the handling of inter-parameter constraints (compare chapter 16), the Gemfony team has integrated the possibility to parse and evaluate simple C++ -style forumulas into Geneva, using the `GFormulaParserT<fp_type>` class template. The class accepts formulas of the type $sqrt(sin(pi)*cos(3.)/5.)$, but can also handle variables. Formulas then have the form $sqrt(sin(\{\{x\}\})*cos(\{\{y\}\})/5.)$, where $\{\{x\}\}$ and $\{\{y\}\}$ indicate variable names.

Listing 33.13: Parsing and evaluating a simple formula with variables

```
std::string formula("sin({{x}})/{{y}}");

std::map<std::string, std::vector<double> > parameterValues;
std::vector<double> list0 = boost::assign::list_of(4.34343434343434);
std::vector<double> list1 = boost::assign::list_of(8.98989898989899);
parameterValues["x"] = list0;
parameterValues["y"] = list1;

GFormulaParserT<double> f(formula);
double parse_val = f(parameterValues);
```

Listing 33.13 shows an example of the usage of this parser together with variables. In a first step, a simple formula with place holders is specified as a `std::string`. Note that this formula might also be taken from an external source, such as the command line or a configuration file.

Next, a `std::map` is filled with parameter names and values. Note that the values are themselves stored in a `std::vector<double>`, so for the sake of simplicity we use a Boost function to create the `vector` and fill it in the same step.

Finally we create the parser, giving it the formula as argument, and pass the value map to the resulting object for evaluation[7]. Note that the evaluation step currently happens as a textual replacement of the place holders with parameter values before parsing, so performance may not be high.

The reason for using `std::vector<double>` as the second `map`-parameter may become apparent in listing 33.14.

Listing 33.14: Parsing and evaluating a simple formula with variables

```
std::string formula("sin({{x[2]}})/{{y}}");

std::map<std::string, std::vector<double> > parameterValues;
std::vector<double> list0 = boost::assign::list_of(1.5)(2.5)(3.5);
std::vector<double> list1 = boost::assign::list_of(8.98989898989899);

```

[7]The parser constructor may also accept a map of text patterns and values to allow the usage of user-defined constants, which is not shown here.

```
parameterValues["x"] = list0;
parameterValues["y"] = list1;

GFormulaParserT<double> f(formula);
double parse_val = f(parameterValues);
```

With `GFormularParserT` it is not only possible to pass parameter names such as $\{\{x\}\}$ and $\{\{y\}\}$, but also a kind of vector notation, such as $\{\{x[2]\}\}$, referring to the third value in the associated vector.

The use case is of course Geneva-related: Geneva comes both with individual parameter types, such as `GConstrainedDoubleObject`, and with parameter collections such as `GConstrainedDoubleCollection`. Such objects may be assigned a name. We can then extract their values (there may be more than one for some parameter types) and pass their name and value to the formula parser.

`GFormulaParserT` is only capable of dealing with floating point parameters at the moment, but does support most common C++ math functions. Common math errors, such as division by 0, will result in an exception being thrown.

So far, the following exceptions are implemented:

- `division_by_0` with an obvious meaning
- `acos_invalid_range<fp_type>` and `asin_invalid_range<fp_type>` to flag arguments outside of the range $[-1:1]$
- `log_negative_value<fp_type>` and `log10_negative_value<fp_type>` to indicate function arguments ≤ 0
- `sqrt_negative_value<fp_type>` to flag function arguments < 0

`fp_type` will be either `double` or `float`, depending on which parameter type you have chosen for `GFormulaParserT`.

All of these exceptions derive from `math_logic_error`, so you can just catch this exception type (or alterative the parent class `gemfony_error_condition`).

Further information on the `GFormularParserT` class is available in the `GFormulaParserTest` code in the Geneva distribution.

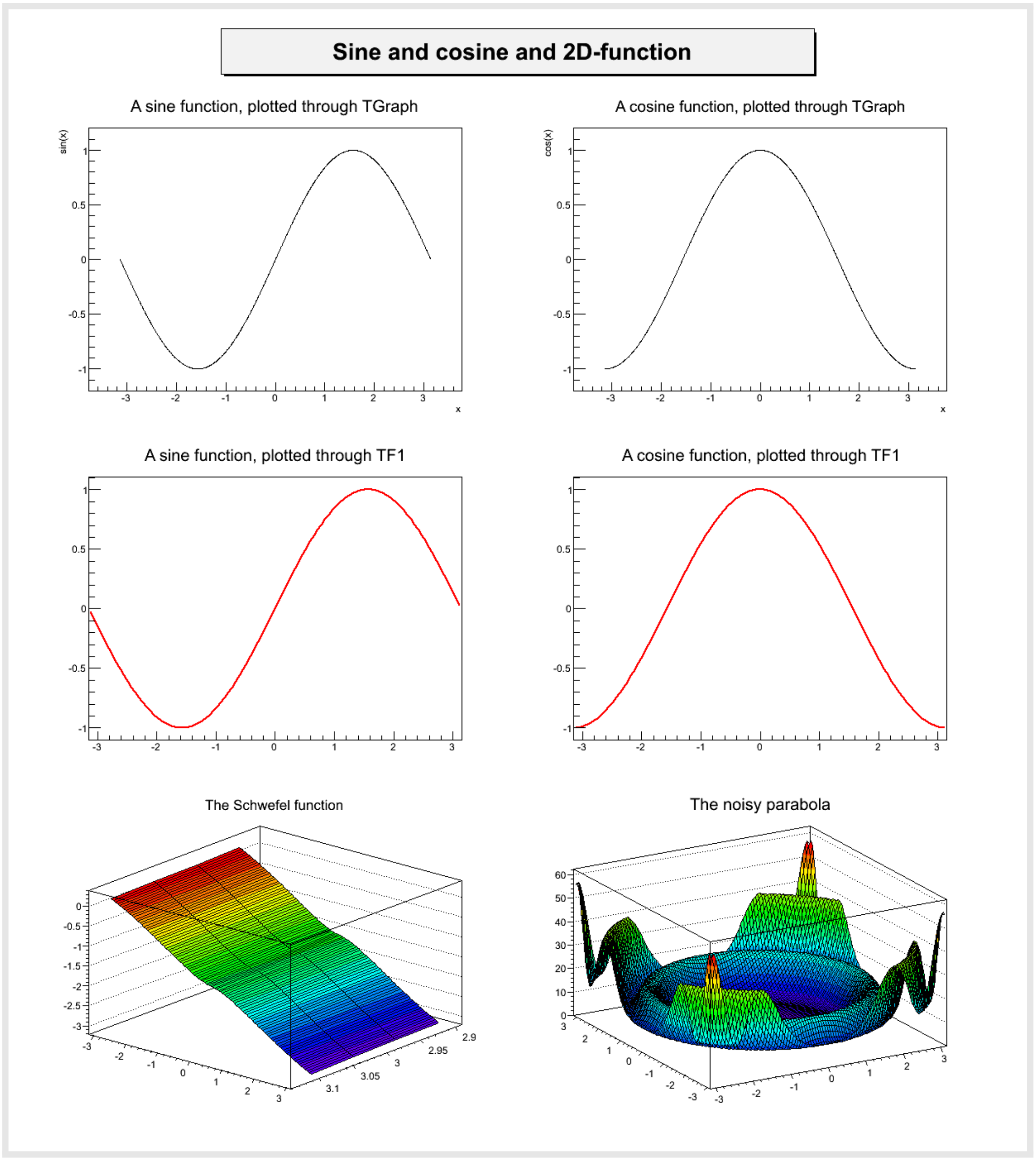


Figure 33.1.: *A sample plot created with Geneva's `GPlotDesigner` class, as demonstrated in the `GPlotDesignerTest` example delivered with the Geneva library.*

Part V.

Appendix and Bibliography

Appendix A.

Frequently Used Test Functions

This appendix introduces a number of test functions that are typically used to test the ability of optimization algorithms to find the global optimum. These functions have been implemented as part of the `GFunctionIndividual`, as introduced in chapter 26.

A.1. Parabola

An n-dimensional parabola is arguably the easiest useful test function, as it only has a single optimum. Thus optimization algorithms cannot get stuck in local optima. It *is* possible, however, to use this test function in order to measure, how well a given algorithm can pinpoint the exact location of the optimum. An n-dimensional parabola can be defined by equation A.1.

$$f_n(x_1,x_2,\ldots,x_n)=\sum_{i=1}^{n}x_i^2=x_1^2+x_2^2+\ldots+x_n^2 \tag{A.1}$$

A 2-D version of this function is visualized in figure A.1.

A.2. Berlich Noisy Parabola

The "noisy" parabola was invented as a test function for the Geneva library. Its general shape is that of a parabola, albeit with a large amount of concentric local optima around the global optimum, whose frequency and amplitude increases with increasing distance from 0. It is defined by equation A.2.

$$f_n(x_1,x_2,\ldots,x_n)=\left(cos\left(\sum_{i=1}^{n}x_i^2\right)+2\right)*\sum_{i=1}^{n}x_i^2 \tag{A.2}$$

Figure A.2 visualizes this function in two dimensions.

A.3. Rosenbrock Function

The Rosenbrock function is particularly interesting, as it has un un-obvious global optimum at (0.0) in an otherwise flat region. It is defined in two dimensions as

$$f_2(x,y) = (1-x)^2 + 100\left(y - x^2\right)^2 \tag{A.3}$$

and in n dimensions (where n must be an even number) as

$$f_n(x_1,x_2,\ldots,x_n) = \sum_{i=1}^{n/2}\left((1-x_{2i-1})^2 + 100\left(x_{2i} - x_{2i-1}^2\right)^2\right) \tag{A.4}$$

Figure A.3 visualizes this function in two dimensions.

A.4. Ackley Function

The Ackley function features a huge number of local optima. In two dimensions, it also has *two* global optima. It is defined by:

$$f_n(x_1,x_2,\ldots,x_n) = \sum_{i=1}^{n-1}\left(e^{-0.2}\sqrt{{x_i}^2 + {x_{i+1}}^2} + 3\left(\cos(2x_i) + \sin(2x_{i+1})\right)\right) \tag{A.5}$$

Figure A.4 visualizes this function in two dimensions.

A.5. Rastrigin Function

The Rastrigin function resembles the noisy parabola (compare figure A.2) in that local optima are overlaid to a parabola, and thar the global optimum can be found in $(0,0)$ in two dimensions. It is defined by equation A.6.

$$f_n(x_1,x_2,\ldots,x_n) = 10n + \sum_{i=1}^{n}\left({x_i}^2 - 10\cos(2\pi x_i)\right) \tag{A.6}$$

Figure A.5 visualizes this function in two dimensions.

A.6. Schwefel Function

The Schwefel function has a global optimum at $(420.969, 420.969)$ in two dimensions, when being restricted to $(-512, 512)$. The function is defined by equation A.7.

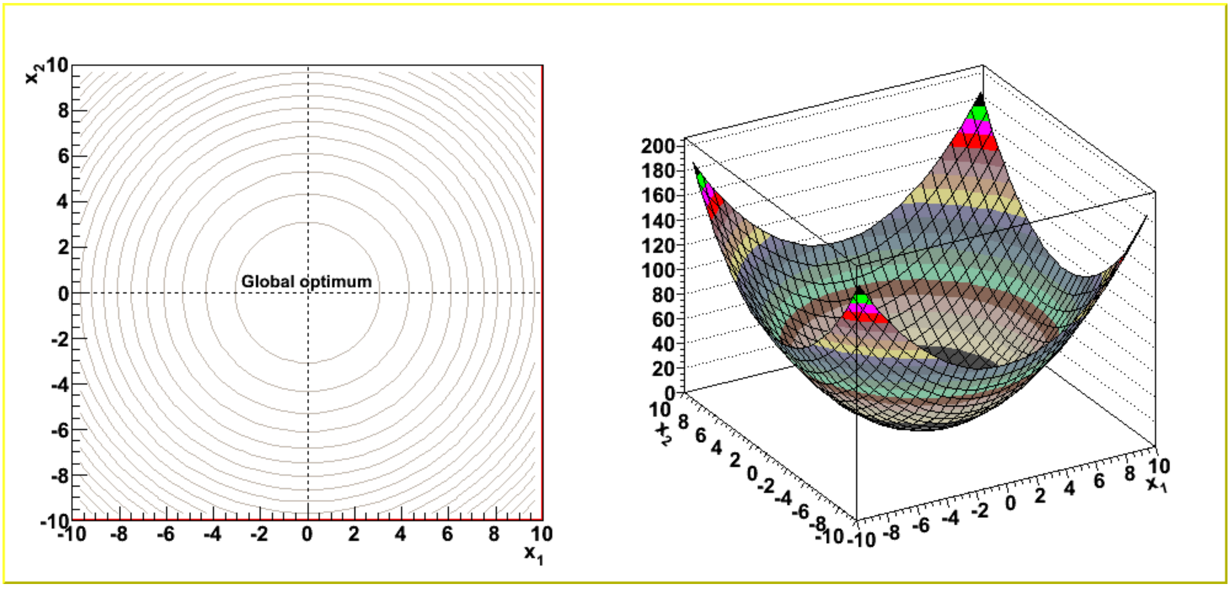


Figure A.1.: *Two views of a two-dimensional paraboloid. Left: contour lines; right: three-dimensional view with function values.*

$$f_n(x_1,x_2,\dots,x_n) = -\frac{1}{n}\sum_{i=1}^{n}\left(x_i\, sin\left(\sqrt{|x_i|}\right)\right) \tag{A.7}$$

Figure A.6 visualizes this function in two dimensions.

A.7. Salomon Function

The Salomon function again resembles the noisy parabola (compare figure A.2), albeit with fewer local optima. The global optimum in two dimensions can be found in the center of the coordinate system, at $(0,0)$. The function is defined by equation A.8.

$$f_n(x_1,x_2,\dots,x_n) = 1 + 0.1\sqrt{\sum_{i=1}^{n} x_i^2} - cos\left(2\pi\sqrt{\sum_{i=1}^{n} x_i^2}\right) \tag{A.8}$$

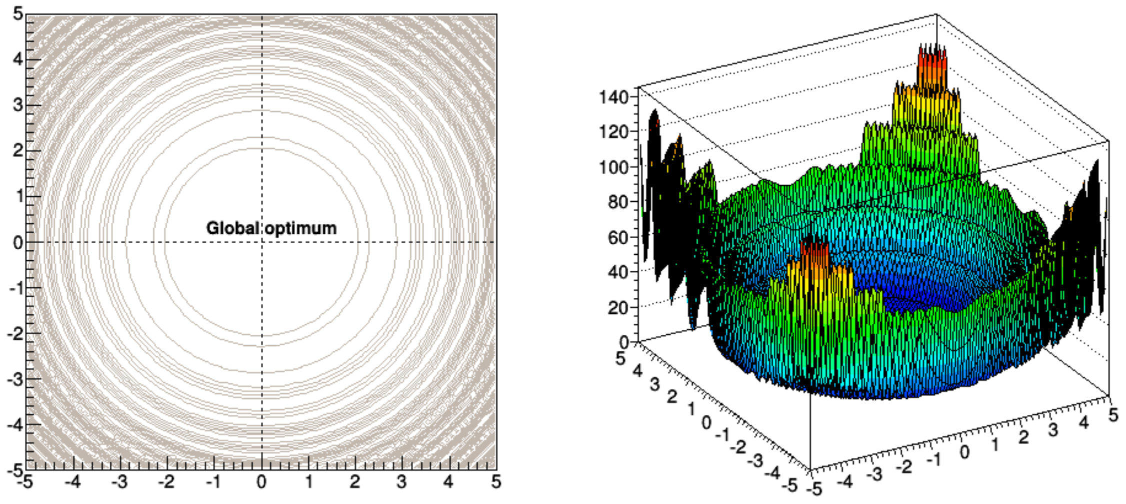


Figure A.2.: *Two views of the "noisy" parabola in two dimensions. Left: contour lines; right: three-dimensional view with function values. Note that, due to the very high number of local optima, the plot program is unable to entirely represent this function, even in the chosen small value range.*

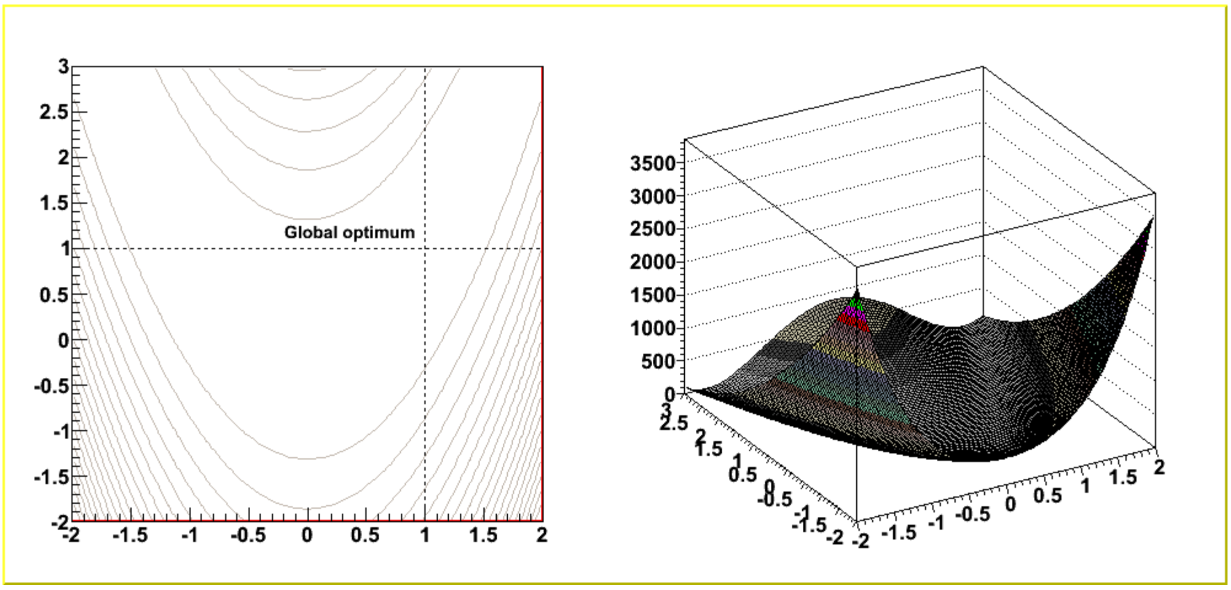


Figure A.3.: *The Rosenbrock function is particularly interesting, as it has a less obvious global optimum at* (0.0) *in an otherwise flat region.*

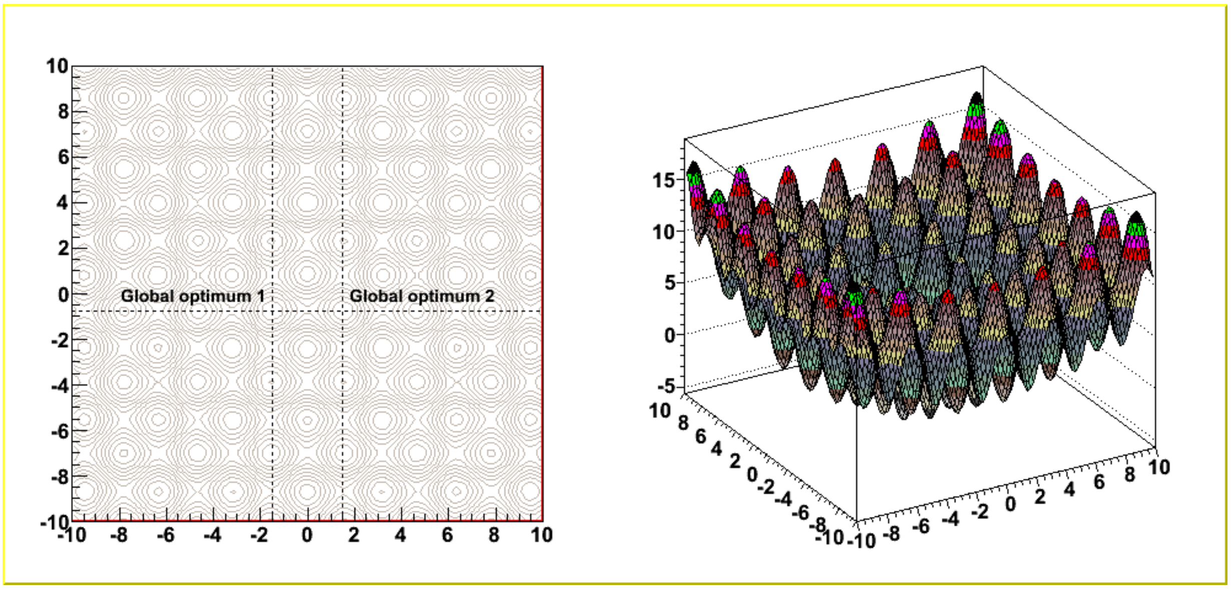


Figure A.4.: *The Ackley function features a huge number of local optima. In two dimensions, it also has* two *global optima.*

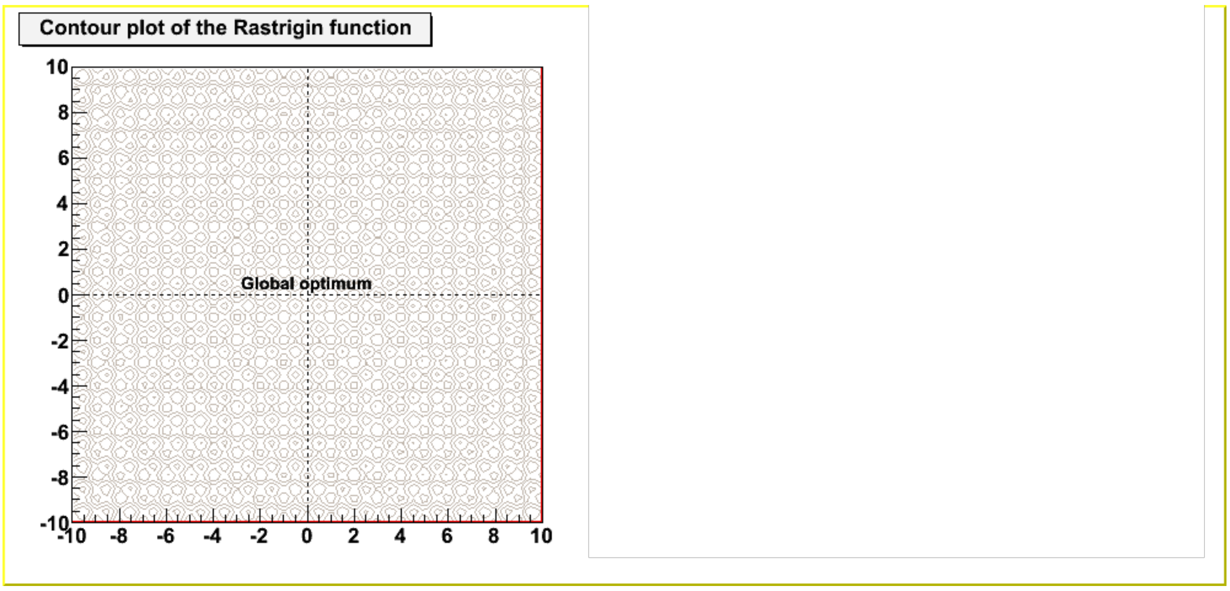


Figure A.5.: *The Rastrigin function resembles the noisy parabola (compare figure A.2) in that local optima are overlaid to a parabola, and that the global optimum can be found in* $(0,0)$ *in two dimensions.*

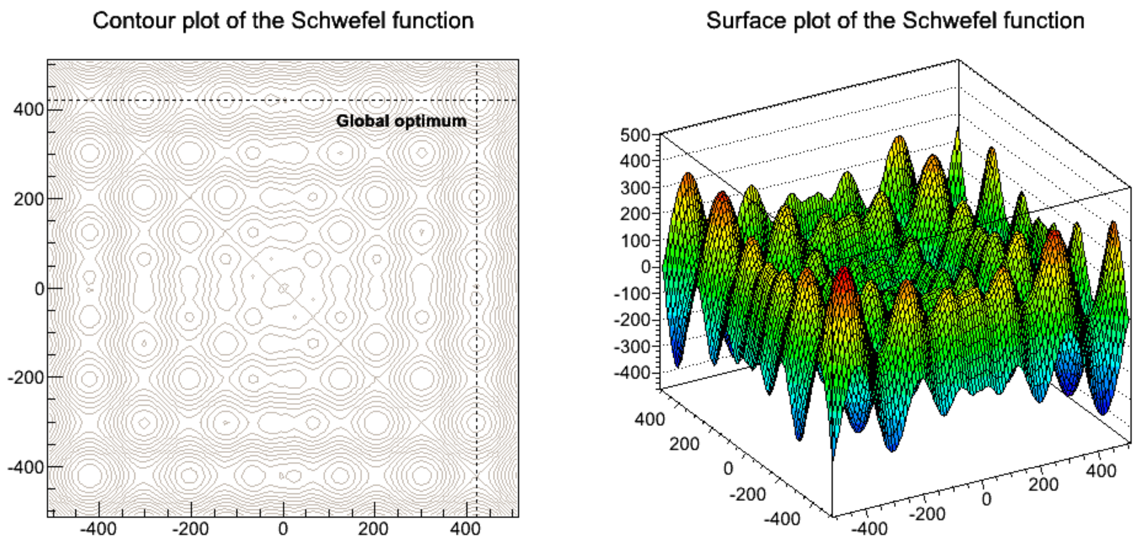


Figure A.6.: *The Schwefel function has a global optimum at* $(420.969, 420.969)$ *in two dimensions.*

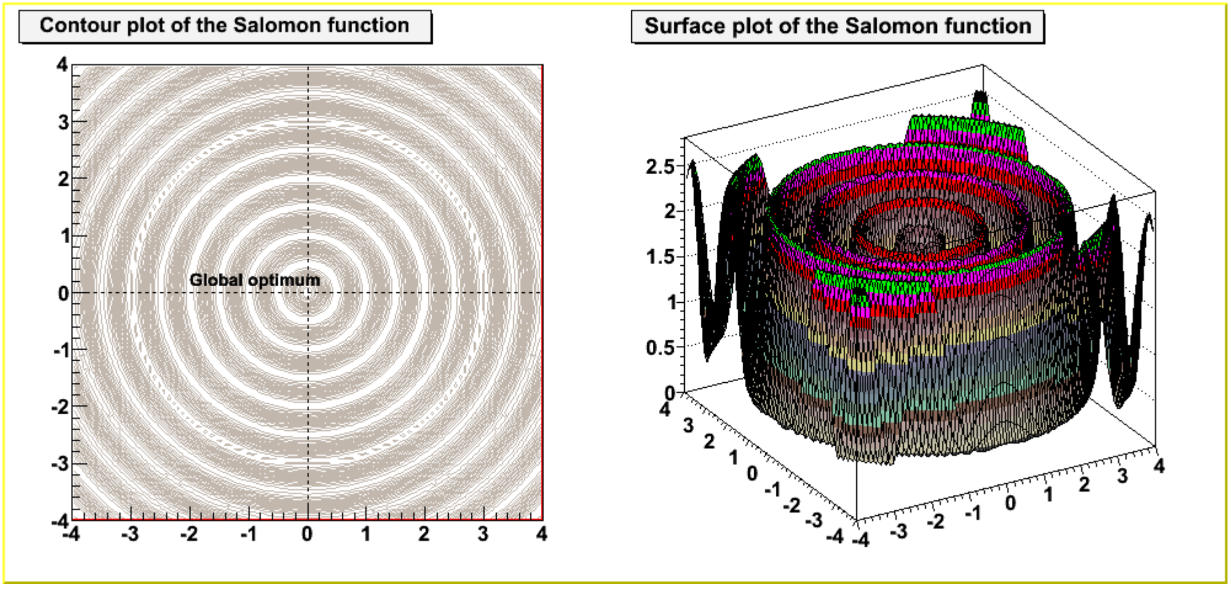


Figure A.7.: *The Schwefel function has a global optimum at* (0.0)*, local optima form concentric rings around the center. The function resembles the "noisy parabola" in figure A.2.*

Figure A.7 visualizes this function in two dimensions.

Appendix B.

The Boost Library Collection

Boost[72] is a volunteer effort of many of the brightest minds of the C++ ecosystem. A large number of Boost libraries has even been submitted to and accepted into the new C++11 standard. Boost libraries are peer-reviewed and undergo thorough, automated testing in nightly builds. New versions are released roughly every three months. This appendix does not want to replicate existing books and tutorials. See e.g. `http://en.highscore.de/cpp/boost/` for an excellent online introduction. Instead, we only want to shortly introduce a number of components of particular relevance to Geneva here. Further information will be added to this chapter over time, as the Geneva manual evolves and grows.

B.1. Smart Pointers

The new C++11 standard defines a `std::shared_ptr<>`, which allows access to objects very similar to standard pointers. However, using the RAII technique and reference counting, "pointed-to" objects get automatically out of scope as soon as the last `std::shared_ptr<>` gets out of scope and is reclaimed by the run-time environment. This technique thus has a similar effect for the user as a garbage collector, making it unnecessary to explicitly `delete` dynamically allocated objects. This is particularly useful with factory classes, or when "shipping" objects via Geneva's broker to remote locations.

`std::shared_ptr<>` has been modelled after the corresponding `boost::shared_ptr<>`, so that the technique is also available with "old" C++03 compilers. `boost::shared_ptr<>` objects are used throughout Geneva, wherever user-supplied objects need to be registered or added to other objects.

B.2. Serialization

Serialization of objects is not typically the domain of C++ – you'd rather find this feature with its stepbrothers and sisters C# and Java. However, Boost contains the Boost.Serialization library. The Geneva library collection makes heavy use of this library, particularly when running in networked mode, but also for check-pointing. Within Boost.Serialization, code for most standard constructs is readily available, so they can be serialized with virtually no effort.

User-defined classes can be serialized essentially by listing all variables and objects to be serialized inside of a private `serialize()` function. Objects to be serialized as part of the surrounding class also need to follow this convention. Geneva's optimization-related classes are all pre-configured so they can be subject to serialization. User-defined classes, such as individuals, can thus concentrate on their own, local data. Geneva handles the rest.

B.3. Threads

Boost has made available a thread implementation that allows to create concurrent programs across amyn different platforms. The new `std::thread` implementation of C++11 has also been crafted after Boost.Thread. Geneva uses this facility wherever threads are used, so that Geneva programs are as portable as possible.

Appendix C.

The ROOT Analysis Framework

ROOT stands for **R**OOT **O**bject **O**riented **T**oolkit – i.e. it is a recursive acronym in the style of GNU (**G**NU is **N**ot **U**nix).

ROOT is a comprehensive, C++ *interpreter*-based framework which is primarily used in particle physics. It is the standard tool for performing analysis at the CERN Large Hadron Collider experiments.

ROOT comprises histogramming and visualization, statistics, matrix algebra and many other components used in mathematics and physics, as well as distributed computing.

Geneva uses ROOT in many places to output results, e.g. in the context of optimization monitors. Listing C.1 shows a simple root script that was automatically generated by Geneva.

Listing C.1: A simple root script that was automatically created by a Geneva optimization monitor

```
{
  gROOT->Reset();
  gStyle->SetOptTitle(0);
  TCanvas *cc = new TCanvas("cc","cc",0,0,1024,768);

  std::vector<long> iteration;
  std::vector<double> sigma;

  // Fill with results
  iteration.push_back(0);
  sigma.push_back(2);
  iteration.push_back(1);
  sigma.push_back(1.07169);

  // [...]

    iteration.push_back(998);
  sigma.push_back(0.000728762);
  iteration.push_back(999);
  sigma.push_back(0.000728762);

  // Transfer the results into a TGraph object
  double iteration_arr[iteration.size()];
  double sigma_arr[sigma.size()];
```

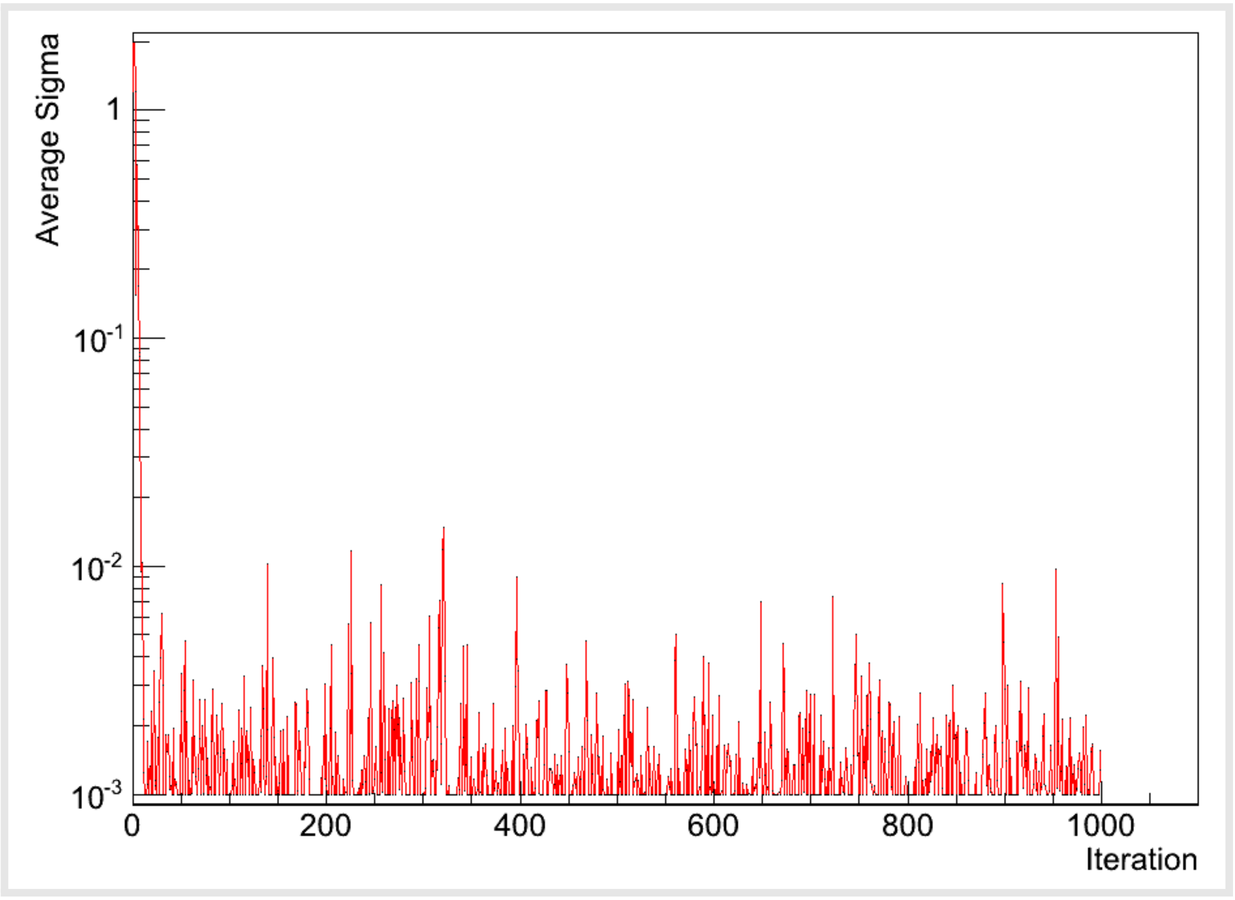


Figure C.1.: *A picture that was created with the help of an automatically generated ROOT script*

```

for(std::size_t i=0; i<iteration.size(); i++) {
    iteration_arr[i] = (double)iteration[i];      sigma_arr[i] = sigma[i];
}

// Create a TGraph object
TGraph *sGraph = new TGraph(sigma.size(), iteration_arr, sigma_arr);

// Set the axis titles
sGraph->GetXaxis()->SetTitle("Iteration");
sGraph->GetYaxis()->SetTitleOffset(1.1);
sGraph->GetYaxis()->SetTitle("Average Sigma");

// Set the line color to red  sGraph->SetLineColor(2);

// Set the y-axis to a logarithmic scale
cc->SetLogy();
// Do the actual drawing
sGraph->Draw("ALP");
```

```
}
```

Figure C.1 shows an example of a picture that was created using the above script (although with different input values).

Appendix D.

Important Open Source Licenses

This appendix lists a number of important Open Source licenses that are used in the Geneva library collection.

D.1. The GNU Affero General Public License

Most of Geneva is covered by the Affero GPL v3. We encourage you to read this license in its entirety in order to understand your obligations. At the time of writing, the entire license is available from `http://www.gnu.org/licenses/agpl-3.0.html`. It is quoted verbatim below.

an appropriate copyright notice, and (2) tells the user that there is no warranty for the work (except to the extent that warranties are provided), that licensees may convey the work under this License, and how to view a copy of this License. If the interface presents a list of user commands or options, such as a menu, a prominent item in the list meets this criterion.

1. Source Code.

The "source code" for a work means the preferred form of the work for making modifications to it. "Object code" means any non-source form of a work.

A "Standard Interface" means an interface that either is an official standard defined by a recognized standards body, or, in the case of interfaces specified for a particular programming language, one that is widely used among developers working in that language.

The "System Libraries" of an executable work include anything, other than the work as a whole, that (a) is included in the normal form of packaging a Major Component, but which is not part of that Major Component, and (b) serves only to enable use of the work with that Major Component, or to implement a Standard Interface for which an implementation is available to the public in source code form. A "Major Component", in this context, means a major essential component (kernel, window system, and so on) of the specific operating system (if any) on which the executable work runs, or a compiler used to produce the work, or an object code interpreter used to run it.

The "Corresponding Source" for a work in object code form means all the source code needed to generate, install, and (for an executable work) run the object code and to modify the work, including scripts to control those activities. However, it does not include the work's System Libraries, or general-purpose tools or generally available free programs which are used unmodified in performing those activities but which are not part of the work. For example, Corresponding Source includes interface definition files associated with source files for the work, and the source code for shared libraries and dynamically linked subprograms that the work is specifically designed to require, such as by intimate data communication or control flow between those subprograms and other parts of the work.

The Corresponding Source need not include anything that users can regenerate automatically from other parts of the Corresponding Source.

The Corresponding Source for a work in source code form is that same work.

2. Basic Permissions.

All rights granted under this License are granted for the term of copyright on the Program, and are irrevocable provided the stated conditions are met. This License explicitly affirms your unlimited permission to run the unmodified Program. The output from running a covered work is covered by this License only if the output, given its content, constitutes a covered work. This License acknowledges your rights of fair use or other equivalent, as provided by copyright law.

You may make, run and propagate covered works that you do not convey, without conditions so long as your license otherwise remains in force. You may convey covered works to others for the sole purpose of having them make modifications exclusively for you, or provide you with facilities for running those works, provided that you comply with the terms of this License in conveying all material for which you do not control copyright. Those thus making or running the covered works for you must do so exclusively on your behalf, under your direction and control, on terms that prohibit them from making any copies of your copyrighted material outside their relationship with you.

Conveying under any other circumstances is permitted solely under the conditions stated below. Sublicensing is not allowed; section 10 makes it unnecessary.

3. Protecting Users' Legal Rights From Anti-Circumvention Law.

No covered work shall be deemed part of an effective technological measure under any applicable law fulfilling obligations under article 11 of the WIPO copyright treaty adopted on 20 December 1996, or similar laws prohibiting or restricting circumvention of such measures.

When you convey a covered work, you waive any legal power to forbid circumvention of technological measures to the extent such circumvention is effected by exercising rights under this License with respect to the covered work, and you disclaim any intention to limit operation or modification of the work as a means of enforcing, against the work's users, your or third parties' legal rights to forbid circumvention of technological measures.

4. Conveying Verbatim Copies.

You may convey verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice; keep intact all notices stating that this License and any non-permissive terms added in accord with section 7 apply to the code; keep intact all notices of the absence of any warranty; and give all recipients a copy of this License along with the Program.

You may charge any price or no price for each copy that you convey, and you may offer support or warranty protection for a fee.

5. Conveying Modified Source Versions.

You may convey a work based on the Program, or the modifications to produce it from the Program, in the form of source code under the terms of section 4, provided that you also meet all of these conditions:

a) The work must carry prominent notices stating that you modified it, and giving a relevant date.

operation of the network or violates the rules and protocols for communication across the network.

Corresponding Source conveyed, and Installation Information provided, in accord with this section must be in a format that is publicly documented (and with an implementation available to the public in source code form), and must require no special password or key for unpacking, reading or copying.

7. Additional Terms.

"Additional permissions" are terms that supplement the terms of this License by making exceptions from one or more of its conditions. Additional permissions that are applicable to the entire Program shall be treated as though they were included in this License, to the extent that they are valid under applicable law. If additional permissions apply only to part of the Program, that part may be used separately under those permissions, but the entire Program remains governed by this License without regard to the additional permissions.

When you convey a copy of a covered work, you may at your option remove any additional permissions from that copy, or from any part of it. (Additional permissions may be written to require their own removal in certain cases when you modify the work.) You may place additional permissions on material, added by you to a covered work, for which you have or can give appropriate copyright permission.

Notwithstanding any other provision of this License, for material you add to a covered work, you may (if authorized by the copyright holders of that material) supplement the terms of this License with terms:

a) Disclaiming warranty or limiting liability differently from the terms of sections 15 and 16 of this License; or

b) Requiring preservation of specified reasonable legal notices or author attributions in that material or in the Appropriate Legal Notices displayed by works containing it; or

c) Prohibiting misrepresentation of the origin of that material, or requiring that modified versions of such material be marked in reasonable ways as different from the original version; or

d) Limiting the use for publicity purposes of names of licensors or authors of the material; or

e) Declining to grant rights under trademark law for use of some trade names, trademarks, or service marks; or

f) Requiring indemnification of licensors and authors of that material by anyone who conveys the material (or modified versions of it) with contractual assumptions of liability to the recipient, for any liability that these contractual assumptions directly impose on those licensors and authors.

All other non-permissive additional terms are considered "further restrictions" within the meaning of section 10. If the Program as you received it, or any part of it, contains a notice stating that it is governed by this License along with a term that is a further restriction, you may remove that term. If a license document contains a further restriction but permits relicensing or conveying under this License, you may add to a covered work material governed by the terms of that license document, provided that the further restriction does not survive such relicensing or conveying.

If you add terms to a covered work in accord with this section, you must place, in the relevant source files, a statement of the additional terms that apply to those files, or a notice indicating where to find the applicable terms.

Additional terms, permissive or non-permissive, may be stated in the form of a separately written license, or stated as exceptions; the above requirements apply either way.

8. Termination.

You may not propagate or modify a covered work except as expressly provided under this License. Any attempt otherwise to propagate or modify it is void, and will automatically terminate your rights under this License (including any patent licenses granted under the third paragraph of section 11).

However, if you cease all violation of this License, then your license from a particular copyright holder is reinstated (a) provisionally, unless and until the copyright holder explicitly and finally terminates your license, and (b) permanently, if the copyright holder fails to notify you of the violation by some reasonable means prior to 60 days after the cessation.

Moreover, your license from a particular copyright holder is reinstated permanently if the copyright holder notifies you of the violation by some reasonable means, this is the first time you have received notice of violation of this License (for any work) from that copyright holder, and you cure the violation prior to 30 days after your receipt of the notice.

Termination of your rights under this section does not terminate the licenses of parties who have received copies or rights from you under this License. If your rights have been terminated and not permanently reinstated, you do not qualify to receive new licenses for the same material under section 10.

9. Acceptance Not Required for Having Copies.

You are not required to accept this License in order to receive or run a copy of the Program. Ancillary propagation of a covered work occurring solely as a consequence of using peer-to-peer transmission to receive a copy likewise does not require acceptance. However, nothing other than this License grants you permission to propagate or modify any covered work. These actions infringe copyright if you do not accept this License. Therefore, by modifying or propagating a covered work, you indicate your acceptance of this License to do so.

```
10. Automatic Licensing of Downstream Recipients.

Each time you convey a covered work, the recipient automatically receives a
license from the original licensors, to run, modify and propagate that work,
subject to this License. You are not responsible for enforcing compliance by
third parties with this License.

An "entity transaction" is a transaction transferring control of an
organization, or substantially all assets of one, or subdividing an
organization, or merging organizations. If propagation of a covered work results
from an entity transaction, each party to that transaction who receives a copy
of the work also receives whatever licenses to the work the party's predecessor
in interest had or could give under the previous paragraph, plus a right to
possession of the Corresponding Source of the work from the predecessor in
interest, if the predecessor has it or can get it with reasonable efforts.

You may not impose any further restrictions on the exercise of the rights
granted or affirmed under this License. For example, you may not impose a
license fee, royalty, or other charge for exercise of rights granted under this
License, and you may not initiate litigation (including a cross-claim or
counterclaim in a lawsuit) alleging that any patent claim is infringed by
making, using, selling, offering for sale, or importing the Program or any
portion of it.

11. Patents.

A "contributor" is a copyright holder who authorizes use under this License of
the Program or a work on which the Program is based. The work thus licensed is
called the contributor's "contributor version".

A contributor's "essential patent claims" are all patent claims owned or
controlled by the contributor, whether already acquired or hereafter acquired,
that would be infringed by some manner, permitted by this License, of making,
using, or selling its contributor version, but do not include claims that would
be infringed only as a consequence of further modification of the contributor
version. For purposes of this definition, "control" includes the right to grant
patent sublicenses in a manner consistent with the requirements of this License.

Each contributor grants you a non-exclusive, worldwide, royalty-free patent
license under the contributor's essential patent claims, to make, use, sell,
offer for sale, import and otherwise run, modify and propagate the contents of
its contributor version.

In the following three paragraphs, a "patent license" is any express agreement
or commitment, however denominated, not to enforce a patent (such as an express
permission to practice a patent or covenant not to sue for patent infringement).
To "grant" such a patent license to a party means to make such an agreement or
commitment not to enforce a patent against the party.

If you convey a covered work, knowingly relying on a patent license, and the
```

```
Corresponding Source of the work is not available for anyone to copy, free of
charge and under the terms of this License, through a publicly available network
server or other readily accessible means, then you must either (1) cause the
Corresponding Source to be so available, or (2) arrange to deprive yourself of
the benefit of the patent license for this particular work, or (3) arrange, in a
manner consistent with the requirements of this License, to extend the patent
license to downstream recipients. "Knowingly relying" means you have actual
knowledge that, but for the patent license, your conveying the covered work in a
country, or your recipient's use of the covered work in a country, would
infringe one or more identifiable patents in that country that you have reason
to believe are valid.

If, pursuant to or in connection with a single transaction or arrangement, you
convey, or propagate by procuring conveyance of, a covered work, and grant a
patent license to some of the parties receiving the covered work authorizing
them to use, propagate, modify or convey a specific copy of the covered work,
then the patent license you grant is automatically extended to all recipients of
the covered work and works based on it.

A patent license is "discriminatory" if it does not include within the scope of
its coverage, prohibits the exercise of, or is conditioned on the non-exercise
of one or more of the rights that are specifically granted under this License.
You may not convey a covered work if you are a party to an arrangement with a
third party that is in the business of distributing software, under which you
make payment to the third party based on the extent of your activity of
conveying the work, and under which the third party grants, to any of the
parties who would receive the covered work from you, a discriminatory patent
license (a) in connection with copies of the covered work conveyed by you (or
copies made from those copies), or (b) primarily for and in connection with
specific products or compilations that contain the covered work, unless you
entered into that arrangement, or that patent license was granted, prior to 28
March 2007.

Nothing in this License shall be construed as excluding or limiting any implied
license or other defenses to infringement that may otherwise be available to you
under applicable patent law.

12. No Surrender of Others' Freedom.

If conditions are imposed on you (whether by court order, agreement or
otherwise) that contradict the conditions of this License, they do not excuse
you from the conditions of this License. If you cannot convey a covered work so
as to satisfy simultaneously your obligations under this License and any other
pertinent obligations, then as a consequence you may not convey it at all. For
example, if you agree to terms that obligate you to collect a royalty for
further conveying from those to whom you convey the Program, the only way you
could satisfy both those terms and this License would be to refrain entirely
from conveying the Program.

13. Remote Network Interaction; Use with the GNU General Public License.
```

Notwithstanding any other provision of this License, if you modify the Program, your modified version must prominently offer all users interacting with it remotely through a computer network (if your version supports such interaction) an opportunity to receive the Corresponding Source of your version by providing access to the Corresponding Source from a network server at no charge, through some standard or customary means of facilitating copying of software. This Corresponding Source shall include the Corresponding Source for any work covered by version 3 of the GNU General Public License that is incorporated pursuant to the following paragraph.

Notwithstanding any other provision of this License, you have permission to link or combine any covered work with a work licensed under version 3 of the GNU General Public License into a single combined work, and to convey the resulting work. The terms of this License will continue to apply to the part which is the covered work, but the work with which it is combined will remain governed by version 3 of the GNU General Public License.

14. Revised Versions of this License.

The Free Software Foundation may publish revised and/or new versions of the GNU Affero General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Program specifies that a certain numbered version of the GNU Affero General Public License "or any later version" applies to it, you have the option of following the terms and conditions either of that numbered version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of the GNU Affero General Public License, you may choose any version ever published by the Free Software Foundation.

If the Program specifies that a proxy can decide which future versions of the GNU Affero General Public License can be used, that proxy’s public statement of acceptance of a version permanently authorizes you to choose that version for the Program.

Later license versions may give you additional or different permissions. However, no additional obligations are imposed on any author or copyright holder as a result of your choosing to follow a later version.

15. Disclaimer of Warranty.

THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

D.2. The Boost Software License, v 1.0

Some of Geneva's classes are covered by the Boost Software License (v 1.0). They typically represent code that has been derived from code available in the Boost library collection. At the time of writing, you can find the license at `http://www.boost.org/users/license.html`. It is quoted verbatim below.

Appendix E.

Glossary

API stands for **A**pplication **P**rogramming **I**nterface

A **Bit** is a binary state, only capable of assuming the value 0 and 1

A **Byte** are 8 **Bits**

Bandwidth refers to the amount of data leaving one end of a network per time unit

A **Batch Submission System** handles the allocation of cluster resources

Cooling Schedule is a measure for the degradation of the “temperature” in the Simulated Annealing optimization algorithm.

CUDA is a standard for performing general purpose calculations particularly on NVIDIA graphics cards

DirectX is a standard for computer graphics

EGEE stands for **E**nabling **G**rids for **E**-Scienc**E**

EGI stands for the **E**uropean **G**rid **I**nfrastructure

In the context of this manual, a **Feature Vector** describes the properties of an optimization problem. See also `http://en.wikipedia.org/wiki/Feature_vector` for other uses.

A **Function Object** refers to an object of a class that defines an `operator()`.

A **Gigabyte** are 1024 **Megabytes**

GPGPU stands for **General Purpose G**raphics **P**rocessing **U**nits

GPU stands for **G**raphics **P**rocessing **U**nit

Hyperthreading refers to the doubling of registers of a processor which makes it look like having twice the number of cores

Individual denotes the definition of a parameter set and a numeric evaluation. Sometimes individuals are also called “candidate solution” in this document.

Invariant Mass is a way of calculating the likely mass of elementary particles from the properties of two or more “tracks” emanating from its decay point.

JSON is the Java Script Object Notation. It is used in Geneva for configuration files.

A **Kilobyte** are 1024 **Bytes**

Latency refers to the amount of time data needs to travel from one end of a connection to another.

LHC stands for the **L**arge **H**adron **C**ollider

A **Manipulator** is a simple class that modifies the output provided through a stream. In the Geneva library, this facility is used for logging and raising of exceptions (compare e.g. sections 33.2.2 and 33.2.3).

A **Megabyte** are 1024 **Kilobytes**

MPI stands for **M**essage **P**assing **Interface**

A **Mutex** is a type of variable whose value can be switched atomically. It is used for locking in multi-threaded applications.

OpenCL is a C-based language for performing general-purpose calculations on modern graphics cards.

OpenGL is a C-based general purpose language for the creation of graphics.

A **Petabyte** are 1024 **Terabytes**

POD stands for **P**lain **O**ld **D**ata

PSO stands for **P**article **S**warm **O**ptimization criterion, expressed through Geneva's classes and interfaces

In the context of the Geneva library collection, a **Quantization Effect** refers to the fact that higher numbers of individuals do not necessarily lead to a speed up. There are "steps" in the speedup graph as a function of the number of individuals.

The term **reentrant** refers to functions that can run in parallel to each other in a multithreaded environment.

ROOT is the **R**OOT **O**bject **O**riented **T**oolkit

Serialization allows to transform binary objects into a format such as XML that can be stored on disk or transferred over a network. The opposite process is called **De-Serialization** and transforms serialized data into binary objects.

SIMD means **S**ingle **I**nstruction **M**ultiple **D**ata

Evolutionary Algorithms are a representative of **Stochastic Optimization**

The term **Solver** stands for the evaluation function used to rate candidate solutions.

A **Terabyte** are 1024 **Gigabytes**

A **Teraflop** is the equivalent of 10^{12} single precision floating point operations per second

Travelling Salesman refers to a problem where a virtual salesman needs to find the shortest route, travelling through a number of predefined points, possibly involving constraints.

A **Web Service** is a means of running or accessing distributed applications over a defined interface.

WLCG stands for the **W**orldwide **L**HC **C**omputing **G**rid

XML stands for Extensible Markup Language

Appendix F.

References

Printed Resources

Books

[9] Rdiger Berlich. *Visualisierung hadronischer Splitoffs und ihre Erkennung mit neuronalen Netzen. Diploma Thesis, Ruhr-Universitochum*. German. University of Bochum (Germany), 1995.

[10] Christian Blum and Daniel Merkle. *Swarm Intelligence. Introduction and Applications*. Heidelberg: Springer, 2008. isbn: 978-3-540-74088-9.

[12] Nicholas Carr. *The Big Switch. Rewiring the world, from Edison to Google*. New York: W. W. Norton & Company, Inc., 2008. isbn: 978-0-393-06228-1.

[23] Ian Foster and Carl Kesselman, eds. *The Grid. Blueprint for a new Computing Infrastructure*. San Francisco: Morgan Kauffmann Publishers, Inc., 1998. isbn: 1-55860-475-8.

[29] Ingrid Gerdes, Frank Klawonn, and Rudolf Kruse. *Evolution Algorithmen*. German. Wiesbaden: Vieweg, 2004. isbn: 3-528-05570-7.

[36] John H. Holland. *Adaptation in natural and artificial systems. An introductory analysis with applications to biology, control, and artificial intelligence*. MIT Press, 2001. isbn: 0-262-58111-6.

[43] Bjrn Karlsson. *Beyond the C++ Standard Library*. Addison-Wesley Professional, 2005. isbn: 978-0321133540.

[44] Bernd Kost. *Optimierung mit Evolutionsstrategien*. German. Frankfurt am Main: Wissenschaftlicher Verlag Harri Deutsch, 2003. isbn: 3-8171-1699-3.

[52] Pawel Plaszczak and Richard Wellner Jr. *Grid Computing. The savvy manager's guide*. San Francisco: Morgan Kaufmann Publishers, 2006. isbn: 0-12-742503-9.

[53] Bogdan Povh et al. *Teilchen und Kerne. Eine Einfhrung in die physikalischen Konzepte*. German. Berlin: Springer Verlag, 2001. isbn: 3-540-65928-5.

[63] Ingo Rechenberg. *Evolutionsstrategie. Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. German. Frommann-Holzboog, 1973. isbn: 3-7728-0374-1.

[64] Ingo Rechenberg. *Evolutionsstrategie ’94*. German. Frommann-Holzboog, 1994. isbn: 3-7728-1642-8.

[71] *The Boost C++ Libraries*. XML Press, 2011. isbn: 978-0982219195.

[144] Anthony Williams. *Concurrency in Action. Practical Multithreading*. Manning, 2012.

Articles

[4] Dr. Christian Baun. “Tonangebend. Grid-, Cloud-, Cluster- und Meta-Computing”. German. In: *c’t Magazin fr Computer Technik* 21 (2008).

[6] Dr. Rdiger Berlich. “Bewegliche Ziele. Anspruch und Wirklichkeit des Grid Computing”. German. In: *c’t Magazin fr Computer Technik* 21 (2008).

[8] Rdiger Berlich. “Training feedforward neural networks using evolutionary strategies. Proceedings of AIHENP 95”. In: *New Computing Techniques in Physics Research IV* (1995), 521ff.

[22] George I. Evers and Mounir Ben Ghalia. “Regrouping Particle Swarm Optimization: A New Global Optimization Algorithm with Improved Performance Consistency Across Benchmarks”. In: *Proceedings of 2009 IEEE International Conference on Systems, Man, and Cybernetics* (2009). url: `http://www.georgeevers.org/publications.htm` (visited on 07/05/2011).

[38] Ralph Hlsenbusch. “Hinter den Wolken. Cloud Computing auf altbekannten Wegen”. German. In: *Magazin fr professionelle Informationstechnik* 12 (2008).

[41] J.Kennedy and R.C.Eberhart. “Particle Swarm Optimization”. In: *Proceedings of the 1995 IEEE International Conference on Neural Networks* IV (1995), pp. 1942–1948.

[66] Bernhard Schott. “Weather forecast: Will it rain Grids and Clouds?” In: *Science, Technology and Innovation Projects* (2008). issn: 1758-2369.

[83] “Towards building a cloud for scientific applications”. In: *Advances in Engineering Software* 42(9) (2011), pp. 714–722.

Online Resources

Articles

[17] Thomas A. DeFanti et al. “Overview of the I-WAY: Wide Area Visual Supercomputing”. In: *Online presence of the Mathematics and Computer Science division at Argonne National Lab* (1995). url: `www-unix.mcs.anl.gov/fl/publications/iway-ijsa.pdf` (visited on 11/26/2008).

[24] Ian Foster, Carl Kesselman, and Steven Tuecke. “The Anatomy of the Grid. Enabling Scalable Virtual Organizations”. In: *International Journal of Supercomputing Applications* (2001). url: `www.globus.org/alliance/publications/papers/anatomy.pdf` (visited on 11/15/2008).

[25] Ian Foster et al. “The Physiology of the Grid. An Open Grid Services Architecture for Distributed Systems Integration”. In: *Online publication of the Globus alliance* (2001). url: `http://www.globus.org/alliance/publications/papers/ogsa.pdf` (visited on 11/15/2008).

[28] Wolfgang Gentzsch. “Grids are Dead! Or are they?” In: *On-Demand Enterprise. Peer Reviewed Journal on the Internet* (June 2008). url: `http://www.gridtoday.com/grid/2381106.html` (visited on 10/04/2008).

[46] Tim Berners Lee. “Information Management: A Proposal”. In: *Online presence of the World Wide Web Consortium* (Mar. 1989). url: `http://www.w3.org/History/1989/proposal.html` (visited on 11/26/2008).

[47] Barry M. Leiner et al. “A Brief History of the Internet, Part I”. In: *Online publication of the Internet Society* (May 1997). url: `http://www.isoc.org/oti/articles/0597/leiner.html` (visited on 11/26/2008).

[48] Barry M. Leiner et al. “A Brief History of the Internet, Part II”. In: *Online publication of the Internet Society* (July 1997). url: `http://www.isoc.org/oti/articles/0797/leiner.html` (visited on 11/26/2008).

[62] Eric S. Raymond. “The Cathedral and the Bazaar”. In: *First Monday. Peer Reviewed Journal on the Internet* (1998). url: `http://www.firstmonday.org/issues/issue3_3/raymond/` (visited on 05/12/2008).

Technical Reports

[7] Rdiger Berlich. “Application of Evolutionary Strategies to Automated Parametric Optimization Studies in Physics Research”. PhD thesis. Ruhr-Universitochum, 2003.

[16] Charles Darwin. *On the origin of species by means of natural selection*. Nov. 1859. url: `http://www.gutenberg.org/ebooks/1228` (visited on 12/29/2010).

[31] Rishab Aiyer Ghosh et al. *Study on the Economic impact of open source software on innovation and the competitiveness of the Information and Communication Technologies (ICT) sector in the EU*. UNU-MERIT, the Netherlands et al., Nov. 2006. url: `http://ec.europa.eu/idabc/servlets/Doc?id=27255` (visited on 04/25/2008).

Wikipedia Resources

[30] *German Wikipedia entry for the simulated annealing optimization algorithm*. German. url: `http://de.wikipedia.org/wiki/Simulierte_Abk%C3%BChlung` (visited on 01/11/2011).

[130] *Wikipedia entry for Amdahl’s Law*. url: `http://en.wikipedia.org/wiki/Amdahls_Law` (visited on 01/10/2011).

[131] *Wikipedia entry for "Cloud Computing"*. url: http://en.wikipedia.org/wiki/Cloud_Computing (visited on 11/30/2008).

[132] *Wikipedia entry for hard disk drives*. url: http://en.wikipedia.org/wiki/Hard_disk (visited on 11/22/2008).

[133] *Wikipedia entry for "Infrastructure-as-a-Service" (IaaS)*. url: http://en.wikipedia.org/wiki/Infrastructure_as_a_Service (visited on 11/21/2008).

[134] *Wikipedia entry for particle swarm optimization*. url: http://en.wikipedia.org/wiki/Particle_Swarm_Optimization (visited on 01/11/2011).

[135] *Wikipedia entry for the brute force search*. url: http://en.wikipedia.org/wiki/Brute_force_search (visited on 01/11/2011).

[136] *Wikipedia entry for the Gray code*. url: http://en.wikipedia.org/wiki/Gray_code (visited on 01/06/2011).

[137] *Wikipedia entry for the PCI bus*. url: http://en.wikipedia.org/wiki/PCI_Local_Bus (visited on 11/22/2008).

[138] *Wikipedia entry for the simulated annealing optimization algorithm*. url: http://en.wikipedia.org/wiki/Simulated_annealing (visited on 01/11/2011).

[139] *Wikipedia entry listing the bandwidths of different devices*. url: http://en.wikipedia.org/wiki/List_of_device_bandwidths (visited on 11/22/2008).

[140] *Wikipedia entry on permissive licenses*. url: http://en.wikipedia.org/wiki/Permissive_license (visited on 06/04/2008).

[141] *Wikipedia entry on the Box-Mller transform*. url: http://en.wikipedia.org/wiki/Box-Muller_transform (visited on 01/03/2011).

[142] *Wikipedia entry on the performance and characteristics of AMD graphics processing units*. url: http://en.wikipedia.org/wiki/Comparison_of_AMD_graphics_processing_units (visited on 01/09/2011).

[143] *Wikiquote entry for Victor Hugo*. url: http://en.wikiquote.org/wiki/Victor_Hugo (visited on 11/30/2008).

General Web Links

[1] *A simulated Higgs event at the CMS detector*. url: http://en.wikipedia.org/wiki/Image:CMS_Higgs-event.jpg (visited on 11/21/2008).

[3] *Article in the Cloud Computing Journal listing companies active in Cloud Computing*. url: http://cloudcomputing.sys-con.com/node/665165 (visited on 11/30/2008).

[5] John Maddock Beman Dawes Jens Mauer. *Boost Standard Integer Types*. url: http://www.boost.org/doc/libs/1_43_0/libs/integer/doc/html/boost_integer/cstdint.html (visited on 05/15/2010).

[11] *Boost's reference-counted smart pointers (version 1.43)*. url: http://www.boost.org/doc/libs/1_43_0/libs/smart_ptr/smart_ptr.htm (visited on 12/05/2010).

[13] *CERN's structure*. url: http://public.web.cern.ch/public/en/About/Structure-en.html (visited on 11/21/2008).

[14] *Conclusion of Victor Hugo's "Histoire dun crime"*. url: http://fr.wikisource.org/wiki/Histoire_d%C3%A2%C2%80%C2%99un_crime_-_Conclusion#X. (visited on 11/30/2008).

[15] *CUDA: NVIDIA's parallel computing architecture for GPUs*. url: http://www.nvidia.co.uk/object/cuda_home_new_uk.html (visited on 11/09/2011).

[18] *Description of Google's "App Engine"*. url: http://code.google.com/appengine/docs/whatisgoogleappengine.html (visited on 11/30/2008).

[19] *Distance from Hamburg/Germany to Adelaide/Australia*. url: http://www.citycomparator.com/compare/6_adelaide_vs_119_hamburg.html (visited on 11/23/2008).

[20] *EGEE's training efforts*. url: http://training.eu-egee.org/index.php?id=237 (visited on 11/27/2008).

[26] *Frequently Asked Questions about the GNU Licenses*. url: http://www.gnu.org/licenses/gpl-faq.html (visited on 01/23/2011).

[27] Jeff Garland. *The Boost.DateTime Library*. url: http://www.boost.org/doc/libs/1_47_0/doc/html/date_time.html (visited on 11/05/2011).

[32] *Glossary of terms related to Web services*. url: http://www.w3.org/TR/ws-gloss/ (visited on 11/30/2008).

[33] *Google Trends*. url: http://www.google.de/trends (visited on 11/15/2008).

[34] *Google's application offerings*. url: http://www.google.com/apps/intl/en/business/index.html (visited on 11/23/2008).

[35] *Google's free mail offerings*. url: http://mail.google.com (visited on 11/23/2008).

[37] *HP's IT outsourcing offers*. url: http://h20219.www2.hp.com/services/cache/575706-0-0-225-121.html (visited on 11/23/2008).

[39] *IBM's Blue Gene supercomputers*. url: http://domino.research.ibm.com/comm/research_projects.nsf/pages/bluegene.index.html (visited on 11/21/2008).

[40] *IBM's IT outsourcing and hosting offers*. url: http://www-935.ibm.com/services/us/index.wss/itservice/so/a1000414 (visited on 11/23/2008).

[42] *JMol: an open-source Java viewer for chemical structures in 3D*. url: http://jmol.sourceforge.net (visited on 01/23/2011).

[49] *Official Website of the Python programming language*. url: http://www.python.org/ (visited on 11/30/2008).

[50] *OpenBabel: The Open Source Chemistry Toolbox*. url: `http://www.openbabel.org` (visited on 01/23/2011).

[51] *OpenCL: An open standard for parallel programming of heterogeneous devices*. url: `http://www.khronos.org/opencl/` (visited on 01/09/2011).

[54] *Presentation covering the causes of the LHC start-up failure as well as likely times for the restart*. url: `http://indico.cern.ch/getFile.py/access?contribId=92&sessionId=6&resId=1&materialId=slides&confId=22937` (visited on 11/27/2008).

[55] *Press release covering IBMs European Cloud Computing Hub in Dublin*. Mar. 19, 2008. url: `http://www-03.ibm.com/press/us/en/pressrelease/23710.wss` (visited on 11/30/2008).

[56] *Press release covering re-start of LHC in 2009*. url: `http://press.web.cern.ch/press/PressReleases/Releases2008/PR10.08E.html` (visited on 11/21/2008).

[57] *Press release for IBM's Blue Cloud*. Nov. 15, 2007. url: `http://www-03.ibm.com/press/us/en/pressrelease/22613.wss` (visited on 11/30/2008).

[58] *Press release on the occasion of the start of the third EGEE project phase*. url: `http://press.eu-egee.org/fileadmin/documents/press_release/egee_III_press_release_final.pdf` (visited on 10/04/2008).

[59] *Product offering for streaming Internet TV by the German Telekom*. url: `https://eki-pi.t-home.de/pma-integration/entertain-comfort/4520002` (visited on 11/23/2008).

[61] Robert Ramey. *The Boost Serialization Library*. url: `http://www.boost.org/doc/libs/1_48_0/libs/serialization/doc/index.html` (visited on 12/30/2011).

[65] *Sam Johnston on the denial of Dell's cloud computing trademark application*. url: `http://samj.net/2008/08/dell-denied-cloud-computing-both.html` (visited on 10/04/2008).

[67] Herb Sutter. *A Pragmatic Look at Exception Specifications*. url: `http://www.gotw.ca/publications/mill22.htm` (visited on 05/15/2010).

[68] *The author's web page*. url: `http://ruediger.berlich.com` (visited on 06/04/2008).

[69] *The Boost C++ Libraries*. url: `http://en.highscore.de/cpp/boost/` (visited on 10/19/2011).

[70] *The Boost C++ Libraries*. url: `http://en.highscore.de/cpp/boost/index.html` (visited on 12/05/2010).

[72] *The Boost library collection*. url: `http://www.boost.org` (visited on 12/05/2010).

[73] *The Boost online documentation*. url: `http://www.boost.org/doc/` (visited on 12/05/2010).

[74] *The CMS detector*. url: http://cms-project-cmsinfo.web.cern.ch/cms-project-cmsinfo/Detector/index.html (visited on 11/21/2008).

[75] *The crypto law survey*. url: http://rechten.uvt.nl/koops/cryptolaw/ (visited on 11/27/2008).

[76] *The Doxygen source code documentation generator tool*. url: http://www.doxygen.org (visited on 05/15/2010).

[77] *The Gilda Grid training testbed*. url: https://gilda.ct.infn.it/ (visited on 11/27/2008).

[78] *The GRIA Grid middleware*. url: http://www.gria.org/ (visited on 11/27/2008).

[79] *The LCG realtime monitor of Imperial College London*. url: http://gridportal.hep.ph.ic.ac.uk/rtm/ (visited on 11/26/2008).

[80] *The MONARC study*. url: http://monarc.web.cern.ch/MONARC/ (visited on 11/21/2008).

[81] *The OGSA-DAI middleware*. url: http://www.ogsadai.org.uk (visited on 11/23/2008).

[84] *Web precense of the Open Grid Forum*. url: http://www.ogf.org (visited on 11/15/2008).

[85] *Web presence of Amazon*. url: http://www.amazon.com/ (visited on 11/27/2008).

[86] *Web presence of Amazon's Cloud Front*. url: http://aws.amazon.com/cloudfront/ (visited on 11/30/2008).

[87] *Web presence of Amazon's Elastic Compute Cloud EC2*. url: http://aws.amazon.com/ec2/ (visited on 11/30/2008).

[88] *Web presence of Amazon's Simple Queue Service*. url: http://aws.amazon.com/sqs/ (visited on 11/30/2008).

[89] *Web presence of Amazon's Simple Storage Service S3*. url: http://aws.amazon.com/s3 (visited on 11/30/2008).

[90] *Web presence of Dell*. url: http://www.dell.com (visited on 11/30/2008).

[91] *Web presence of EGEE's gLite middleware*. url: http://glite.web.cern.ch/glite/ (visited on 11/30/2008).

[92] *Web presence of Google*. url: http://www.google.com (visited on 11/30/2008).

[93] *Web presence of Google's "App Engine"*. url: http://appengine.google.com (visited on 11/30/2008).

[94] *Web presence of GridWiseTech*. url: http://www.gridwisetech.com/ (visited on 11/27/2008).

[95] *Web presence of GridWork OpenPBS middleware*. url: www.openpbs.org/ (visited on 11/27/2008).

[96] *Web presence of Hewlett-Packard*. url: http://www.hp.com (visited on 11/30/2008).

[97] *Web presence of Hewlett-Packard Labs*. url: `http://www.hpl.hp.com` (visited on 11/30/2008).

[98] *Web presence of Hewlett-Packard's Dynamic Cloud Services*. url: `http://www.hpl.hp.com/research/cloud.html` (visited on 11/30/2008).

[99] *Web presence of IBM*. url: `http://www.ibm.com` (visited on 11/30/2008).

[100] *Web presence of Intel*. url: `http://www.intel.com` (visited on 11/30/2008).

[101] *Web presence of Karlsruhe Institute of Technology*. German. url: `http://www.kit.edu` (visited on 09/03/2013).

[102] *Web presence of Platform Computing's LSF middleware*. url: `http://www.platform.com/Products/platform-lsf` (visited on 11/27/2008).

[103] *Web presence of Steinbuch Centre for Computing*. German. url: `http://scc.kit.edu` (visited on 09/03/2013).

[104] *Web presence of the Alien2 Grid middleware*. url: `http://alien.cern.ch` (visited on 11/27/2008).

[105] *Web presence of the Andrew filesystem*. url: `http://www.openafs.org` (visited on 11/23/2008).

[106] *Web presence of the Enabling Grids for E-SciencE initiative*. url: `http://www.eu-egee.org` (visited on 10/04/2008).

[107] *Web presence of the Euforia consortium*. url: `http://www.euforia-project.eu/EUFORIA` (visited on 11/27/2008).

[108] *Web presence of the EUGridPMA*. url: `http://www.eugridpma.org/` (visited on 11/27/2008).

[109] *Web presence of the European Data Grid project*. url: `http://eu-datagrid.web.cern.ch/eu-datagrid/` (visited on 10/04/2008).

[110] *Web presence of the European Grid Infrastructure*. url: `http://web.egi.eu/` (visited on 10/12/2011).

[111] *Web presence of the Global Grid User Support*. url: `http://www.ggus.org/` (visited on 11/27/2008).

[112] *Web presence of the Globus Alliance*. url: `http://www.globus.org/` (visited on 11/23/2008).

[113] *Web presence of the Gridipedia project*. url: `http://www.gridipedia.eu` (visited on 11/27/2008).

[114] *Web presence of the GridKa compute centre, including the GermanGrid certificate authority*. German. url: `http://grid.fzk.de/` (visited on 11/27/2008).

[115] *Web presence of the Helmholtz Association of German Research Centres*. url: `http://www.helmholtz.de/en/` (visited on 09/03/2013).

[116] *Web presence of the Infocomm Development Authority in Singapore*. url: http://www.ida.gov.sg (visited on 11/30/2008).

[117] *Web presence of the LHC Computing Grid LCG*. url: http://lcg.web.cern.ch/LCG (visited on 11/30/2008).

[118] *Web presence of the National E-Science Centre in Edinburgh*. url: http://www.nesc.ac.uk/ (visited on 11/27/2008).

[119] *Web presence of the national German Grid initiative D-Grid*. url: http://www.d-grid.org/ (visited on 11/26/2008).

[120] *Web presence of the Nordugrid consortium*. url: http://www.nordugrid.org/middleware/ (visited on 11/27/2008).

[121] *Web presence of the OpenCirrus Cloud Testbed*. url: http://www.cloudtestbed.org/ (visited on 11/30/2008).

[122] *Web presence of the QosCosGrid consortium*. url: http://www.qoscosgrid.eu (visited on 11/27/2008).

[123] *Web presence of the Sun Grid Engine*. url: http://www.sun.com/software/gridware/ (visited on 11/27/2008).

[124] *Web presence of the UNICORE consortium*. url: http://www.unicore.eu/ (visited on 11/27/2008).

[125] *Web presence of the University of Illinois at Urbana Champaign*. url: http://illinois.edu/ (visited on 11/30/2008).

[126] *Web presence of Ulteo*. url: http://www.ulteo.com/home/en/home (visited on 11/23/2008).

[127] *Web presence of Yahoo*. url: http://www.yahoo.com (visited on 11/30/2008).

[128] *Welt der Physik / Der LEP-Beschleuniger bei CERN*. url: http://www.weltderphysik.de/de/3515.php (visited on 11/21/2008).

[129] *What is the Grid ? A Three Point Checklist*. 2002. url: http://www-fp.mcs.anl.gov/~foster/Articles/WhatIsTheGrid.pdf (visited on 11/26/2008).

[145] Anthony Williams. *The Boost.Thread Library*. url: http://www.boost.org/doc/libs/1_43_0/doc/html/thread.html (visited on 01/10/2011).

Presentations

[2] Dr. Torsten Antoni. *The EGEE user support infrastructure*. Presentation given at the EGEE conference 2008 in Istanbul/Turkey. 2008. url: http://indico.cern.ch/materialDisplay.py?contribId=339&sessionId=87&materialId=slides&confId=32220 (visited on 11/30/2008).

[21] Walter Erl. *Vermarktungsmodelle von Open Source-Lsungen*. German. Presentation from the Open Source Meets Business conference (not publicly available). MAX21 Management & Beteiligungen AG. 2008. url: http://www.heise.de/events/2008/open_source_meets_business/ (visited on 04/30/2008).

[45] Dr. Marcel Kunze. *Die Evolution des Rechenzentrums: Die Industrialisierung der IT*. German. 2008. url: http://25dvt.bgc-jena.mpg.de/Z/Abstracts/Kunze.html (visited on 11/30/2008).

[146] Irving Wladawsky-Berger. *Cloud Computing, Grids, and the coming IT Cambrian Explosion*. Irving Wladawsky-Berger is Chairman Emeritus of the IBM Academy of Technology. 2008. url: http://www.ogf.org/OGF22/materials/1137/Irving+Wladawsky-Berger+Keynote.pdf (visited on 10/06/2008).

List of Figures

2.1. Optimizations often act on a model, whose resemblance to reality may vary. 8
2.2. The EVA library (a predecessor of Geneva) was used in this example to optimize the selection of “events” coming from a particle physics experiment. A significant reduction of mis-reconstructed particles is visible (source: own pictures). 9
2.3. Local optima can prevent an optimization algorithm from finding the global optimum . . . 13
2.4. The Pareto Frontier is defined by data points that are not dominated by other solutions . 14
2.5. Boundary condition involving two parameters 15

4.1. Evolutionary Algorithm populations consist of $p >= 1$ parents and $c >= p$ children 22
4.2. The most common mutation operator in Evolution Strategies adds gaussian-distributed random numbers to a given floating point parameter. The success of the optimization procedure crucially depends on the right choice for the width σ of the gaussian. 23
4.3. The Rastrigin function is a common test function for Evolution Strategies. Its major characteristic is a large number of local optima 24
4.4. This picture demonstrates how an Evolution Strategy with a single parent searches for the minimum of the Rastrigin function in two dimensions. The Rastrigin function has a very large number of local optima (compare figure 4.3). Solely “Gauss” mutation is being applied to the parameters, and no recombination schemes are being used. 25
4.5. Using two gaussians instead of one for the mutation of floating point parameters in Evolution Strategies might boost performance, if the current best solution is still far away from the global optimum. 26

6.1. This picture demonstrates how a PSO algorithm searches for the minimum of the Rastrigin function in two dimensions. Note that, in comparison to figure 4.4, the algorithm needs to search a far bigger area for the optimum, as the allowed value range has been increased. 41

7.1. Parameter scans on a regular grid, for a simple parabola (left) and a parabola with overlaid local optima (right). The grid comprises 40 measurements in each direction, for a total of 1600 measurements. The plots were created with the help of pluggable optimization monitors (compare section 25.3). 46

8.1. The GridKa compute cluster is a Tier-1 centre in the academic EGI grid, located at Steinbuch Centre for Computing of Karlsruhe Institute of Technology. At the time of writing it comprised over 10000 CPU cores. (Picture used with permission from Steinbuch Centre for Computing) 50

8.2. Batch submission systems assign incoming processing requests from users to the most suitable resource. Picture courtesy of Martina Hardt (designal • conceptional work by hardt – www.designal.de) 51

8.3. Ian Foster gives a presentation at the Sun booth during Supercomputing 2001 in Denver/USA. It is worth noting the slogan "Sun Powers The Grid" (source: own pictures) . . . 52

8.4. Schematic architecture of a global Grid infrastructure. Picture courtesy of Martina Hardt (designal • conceptional work by hardt – www.designal.de) 53

8.5. The "speed" of a network consists of two components – latency and bandwidth. Picture courtesy of Martina Hardt (designal • conceptional work by hardt – www.designal.de) . . 58

8.6. Response times from servers running in different geographic locations show large variations. Parallel execution in a wide area setting needs to take this into account. 59

8.7. Plot of Amdahl's law: $S = \frac{1}{(1-P)+o(N)+\frac{P}{N}}$ as a function of N, for a fixed value of $o(N)/N$ and different values of P 61

9.1. Semi-transparent triangles can be super-imposed in such a way by optimization algorithms that the amalgamation starts to look like a given target picture. In this example, a clipping from *da Vinci*'s Mona Lisa has been used. The algorithm starts with a random collection of 300 triangles (equivalent to 3000 parameters to be optimized). The target picture is shown in the lower right-hand corner of this figure. 66

9.2. Mapping a star-like structure using the method described in section 9.1. The target image is shown in figure 9.3 67

9.3. Our experience shows that, in Evolutionary Algorithms, a (μ,ν) selection scheme often performs better than a $(\mu+\nu)$ scheme. In both selection schemes, most of the progress during the mapping of the star-like structure is achieved before the first 20% of the optimization run. 68

9.4. This form of the protein ALA12H was obtained by minimizing the potential energy of the molecule through variations of the geometry. The picture was created with the molecular viewer JMol. 70

9.5. A feed-forward neural network calculates the output of a node by applying a sigmoid function to the weighted sums of the output of preceding nodes (minus a threshold) . . . 72

9.6. Input pattern used to train a 2-2-1 network (an even distribution of input values and a distribution confined to the x- and y-axis) and the output of the network after the training in different areas of the canvas (multiplied by 10 and rounded). 73

10.1. Binary packages of the Geneva library are available for some Linux flavours in the dedicated download area of the OpenSUSE Build Service: `https://software.opensuse.org/download/package?project=home:garcia&package=geneva-opt` 78

10.2. In order to download the Geneva library collection, visit `https://launchpad.net/geneva` and click on the green download link on the right side of the page 79

11.1. Geneva makes a hard distinction between the specification of optimization problems (left) and the optimization algorithms (right) trying to provide candidate solutions with increasingly better evaluation. 88
11.2. Parametric optimization with the Geneva library is done in three distinct phases.Phases of parametric optimization 89
11.3. Two views of a two-dimensional paraboloid. Left: contour lines; right: three-dimensional view with function values. 90

12.1. Geneva's optimization-related class hierarchy is rooted in the `GObject` class. The figure shows the innermost classes only. 102

13.1. User-visible parameter-types (marked red) follow an intuitive naming scheme. 115
13.2. Constraints of floating point values are modelled as a mapping from an internal to a user-visible value. This allows to apply modifications of the core value to an unconstrained range, while presenting a constrained value to the user. 116
13.3. This picture shows the real mapping, as achieved for different constraints using the method illustrated in figure 13.2. 117

14.1. Adaptors derive from a common base class whose template parameter determines the types they can be applied to. 134
14.2. This figure shows the internal values of a number of collections of gaussian distributed random numbers with different mean value, as well the externally visible values. The blue curve shows the actual distribution, the curve with red stripes shows the "input" gaussian. It is evident that there is very little distortion close to the outer boundaries of the allowed value range. 143

15.1. Geneva's individuals are derived from the GParameterSet class, which features a `std::vector<>` interface. The class stores smart pointers to `GParameterBase` objects, so that it becomes possible to store different parameter types in the container. 146
15.2. Geneva's individuals can serve as the root of an entire hierarchy of parameters, using the GParameterObjectCollection class. It can be likened to an individual without attached evaluation function and also features a `std::vector<>` interface 147

16.1. A constraint

$$x + y <= 1$$

renders part of the parameter space invalid 156
16.2. Evaluation workflow in the presence of potentially invalid solutions 157
16.3. Valid solutions of a parabola with a "sum-constraint" (left) and an additional "sphere" constraint (right). The solutions were determined with Genevas parameter scan 158
16.4. Quality surface including invalid in the case of the USESIGMOID policy 162
16.5. The effect that the application of a sigmoid function has on a parabola (plotted in blue) . 164

17.1. Optimization algorithms share common features, such as the definition of halt criteria, the main optimization cycle or the basic population structure. They can be implemented in a common base class, called `GOptimizationAlgorithmT` in Geneva. 168

18.1. Evolutionary Algorithm populations consist of $p >= 1$ parents and $c >= p$ children 174

20.1. A swarm population is segmented into neighborhoods. In each iteration, individuals are drawn to a different extent towards their respective personal best-, neighborhood best and globally best solution known so far. 186

21.1. In a gradient-descent population, the individuals representing the current position in each iteration are followed by collections of individuals representing each of the floating point parameters. 192

22.1. Parameter scans of the Rastrigin function (compare appendix A.5), on a regular 60x60 grid (left) and with random scan points (right) 196

26.1. Geneva's default optimization monitor would only write out the progress of the optimization run in ROOT format. The figure shows the progress for a standard 2D parabola, as output by the optimization monitor discussed in this chapter 222

27.1. Choosing a good mutation probability can be important for the success of an evolutionary algorithm. In the left side of the picture, a 5% mutation probability was chosen. Several individuals didn't get updated at all, for some others only some parameters were updated, resulting in a sort of line search. On the right side, a 100% mutation probability was chosen. Except for the mutation probability, all configuration options of the optimization were identical. 231

28.1. Meta optimization, shown here on the example of a two-dimensional "Noisy Parabola", allows to optimize the parameters of optimization algorithms. The figure in the upper left corner represents the best average number of solver calls in each iteration. The other plots show the development of some of the variables being optimized. 240

28.2. Same procedure as in figure 28.1, albeit for the 8-dimensional noisy parabola 241

31.1. Floating-point and integer random numbers can be constrained in their value ranges. . . 265

31.2. Two types of gaussian distributions are availble – a standard gaussian with a user-defined mean and σ, and two gaussians of equal σ superimposed. Both are used as part of mutation operators in Evolutionary Algorithms (compare chapter 4.2.2). 265

31.3. Boolean random numbers can either be produced with a pre-defined probability distribution for `true` and `false` or an even likelihood for both. 266

32.1. Several entities may submit items simultaneously through “buffer ports”, which are plugged into the broker. The broker will deliver items in a round-robin fashion to one or more consumers that ask for work items. Consumers might then deliver the work items to remote sites, or process them locally (depending on the consumer type). Results are shipped back to the original buffer port, from where they can be retrieved by the producer. 274

33.1. A sample plot created with Geneva’s `GPlotDesigner` class, as demonstrated in the `GPlotDesignerTest` example delivered with the Geneva library. 297

A.1. Two views of a two-dimensional paraboloid. Left: contour lines; right: three-dimensional view with function values. 303
A.2. Two views of the “noisy” parabola in two dimensions. Left: contour lines; right: three-dimensional view with function values. Note that, due to the very high number of local optima, the plot program is unable to entirely represent this function, even in the chosen small value range. 304
A.3. The Rosenbrock function is particularly interesting, as it has a less obvious global optimum at (0.0) in an otherwise flat region. 304
A.4. The Ackley function features a huge number of local optima. In two dimensions, it also has *two* global optima. 305
A.5. The Rastrigin function resembles the noisy parabola (compare figure A.2) in that local optima are overlaid to a parabola, and that the global optimum can be found in $(0,0)$ in two dimensions. 305
A.6. The Schwefel function has a global optimum at $(420.969, 420.969)$ in two dimensions. . 306
A.7. The Schwefel function has a global optimum at (0.0), local optima form concentric rings around the center. The function resembles the “noisy parabola” in figure A.2. 306

C.1. A picture that was created with the help of an automatically generated ROOT script . . . 312

Index

/etc/ld.so.conf.d/, 85
05_GHapUsagePatterns, 264

$(\mu+\nu)$, 25
(μ,ν) , 25

Accessing Parameters, 148
Ackley Function, 302
ACO, 43
acos_invalid_range<fp_type>, 296
ACTIVEONLY, 149
adapt(T& val), 103
Adaptors, 103, 109, 133
Adding a Custom Optimization Monitor, 221
Adding Candidate Solutions, 171
Adding optimization algorithms to Go2, 208
Affero General Public License, 315
Affero GPL v3, 315
ALICE, 52
ALLOWREEVALUATION, 165
ALLPARAMETERS, 149
Amdahl's Law, 60
Ant Colony Optimization, 43
asin_invalid_range<fp_type>, 296
ATLAS, 52
Automation of optimization tasks, 7

Bandwidth, 58
bandwidth, 59
batch submission system, 49
Beowulf Cluster, 50
Berlich Noisy Parabola, 301
best accessible solution, 16
best solution, 7
Bi-normal random number distribution, 269
Biological evolution, 21
Bit flip, 30
Boolean Random Numbers, 270
boolean values, 21
Boost, 79, 309
Boost Software License, 326
Boost.Asio, 207
Boost.Thread, 57
Boost.ThreadGroup, 292
boost::shared_ptr<>, 212
BOOST_CLASS_EXPORT, 97
BOOSTROOT, 82
Box-Müller transformation, 28
broker, 273
broker access, 276
broker architecture, 274
broker configuration options, 277
broker work flow, 275
Broker-flooding, 232
Brokered Execution, 200
Brokering, 107
Brute Force, 11
Buffer Ports, 274
Build Environment, 77
building, 82
BUILDMODE, 83
BUILDTESTCODE, 83

C. Eberhart, 39
Candidate Solutions, 89
candidate solutions, 11, 22
Caveats and Restrictions, 229
CERN, 52
cGlobal, 188

Charles Darwin, 21
Checkpointing, 172
children, 23
Christian Blum, 42
Class Hierarchy, 101
Class Layout, 167
clone_(), 94
Cloud Computing, 54
Clouds, 47
cluster of clusters, 53
Clusters, 47
CMS, 52
cNeighborhood, 188
Collaborative methods, 39
collaborative methods, 13
Combination of halt criteria, 27
combinatorial problems, 33
combining multiple evaluation criteria, 14
Combining optimization algorithms, 32
combustion, 10
Common library, 281
Common Traits of Optimization Algorithms, 167
Compilation, 78
complex models, 8
compute nodes, 50
Configuration Parameters, 238
Conjugate Gradient, 19
Constraint Handling, 155
constraints, 16
Construction of Evolutionary Algorithm Objects, 174
Construction of Gradient Descent Objects, 192
Construction of Parameter Scan Objects, 195
Construction of PSO Objects, 186
Consumer, 98
continuous parameter value ranges, 12
conversion_iterator, 96, 149
cooling schedule, 33
cooling schemes, 35
Coordinated Resource Sharing, 55
Courtier Library, 107, 273
cPersonal, 188
Creating a Factory, 218
Creating Factories, 290
cross-over, 28
cstdint, 110
CUDA, 57
Custom optimization monitors, 106
cycleInformation(), 212, 213
cycleLogic(), 105

De-activation of Parameters, 114
Debian package, 85
DEFAULTDUPLICATIONSCHEME, 179
Defining the fitness calculation, 104
Definition of Parametric Optimization, 16
difference quotient, 18
differentiable, 10, 18
differential quotient, 18
digital model of reality, 8
Direct access to parameter objects, 150
dirty flag, 161
discontinuities, 10
discrete parameters, 13
distributed system, 58
division_by_0, 296
doInfo(), 169, 211
double, 109
Double gaussian, 30
down-hill, 17
duplication, 21
Duplication Schemes, 178
Dynamic Population Growth, 177

Eclipse, 82
EDG, 53
EGEE, 53
EGI, 53
embarrassingly parallel, 47
Enabling Grids for E-SciencE, 53
European Data Grid, 53
European Grid Infrastructure, 53
Evaluation and Selection, 179
Evaluation Criteria, 88
evolution cycle, 28
Evolution Strategies, 27

Evolution Strategies , 21
Evolutionary Algorithms, 21, 173
EXECMODE_BROKERAGE, 199
EXECMODE_MULTITHREADED, 199
EXECMODE_SERIAL, 199
EXPECTFULLRETURN, 277
extreme values of the quality surface, 12

Fault Tolerance, 62, 275
feature vector, 10
figure of merit, 18
finite variations, 19
firstInformation(), 213
firstInformation(), 212
fitness, 22, 165
fitness(), 96, 103
fitnessCalculation(), 95, 104, 148, 217
Floating Point Accuracy, 229
full definition of an optimization problem, 145

GAdaptorT, 103
Gauss Mutation, 28, 30
gaussian distributed random numbers, 28
GBaseParChildT, 173, 183
GBaseParChildT<T>, 183
GBasePS::setScanRandomly(bool), 197
GBooleanAdaptor, 141
GBooleanCollection, 129
GBooleanObject, 94, 127
GBooleanObjectCollection, 128
GBoundedBufferT, 281
GBoundedBufferWithIdT, 281
GBrokerConnector2T, 273, 277
GBrokerEA, 106
GConsoleLogger, 285
GConstrainedDoubleCollection, 92
GConstrainedDoubleCollection, 122
GConstrainedDoubleObject, 92, 110, 116, 119, 122
GConstrainedDoubleObjectCollection, 119
GConstrainedInt32Object, 93, 94, 123
GConstrainedInt32ObjectCollection, 125
GDoubleBiGaussAdaptor, 136
GDoubleCollection, 93, 112, 121
GDoubleGaussAdaptor, 135
GDoubleObject, 111, 114
GDoubleObjectCollection, 93, 113, 118
Gem::Courtier::submissionReturnMode, 277
Gem::Courtier::GBC_PROCESSED, 278
Gem::Courtier::GBC_UNPROCESSED, 278
Gem::Geneva::ADDINVALID, 158
Gem::Geneva::MULTIPLYINVALID, 158
gemfony_error_condition, 283, 296
gene pool, 21
Genetic Algorithms, 12, 21
GENEVA-EXCEPTION.log, 285
GENEVA-TERMINATION.log, 285
geneva.conf, 85
GENEVA_CONFIG_BASENAME, 289
genevaConfig.gcfg, 82
GEvolutionaryAlgorithmFactory, 173, 174
GEXCEPTION, 285
GFactoryT<>, 290
GFMinIndividual, 215
GFormulaParserT<fp_type>, 295
GFunctionIndividual, 207, 290
GFunctionIndividualFactory, 208
GFunctionMinimizer, 215
GGlobalOptionsT<>, 291
GGradientDescentFactory, 192
Gigabit Ethernet, 49
GInt32Collection, 126
GInt32FlipAdaptor, 140
GInt32GaussAdaptor, 138
GInt32Object, 123
GInt32ObjectCollection, 124
Global optima, 12
Global Options, 291
Globally best individual, 40
glogger, 284
GLOGGING, 285
GMultiThreadedEA, 106
GMultiThreadedGD, 106
GMultiThreadedSwarm, 106
Go2, 98, 173, 183, 215
Go2 client and server mode, 206

GObject, 101, 102, 208
GOptimizableEntity, 96, 103
GOptimizationAlgorithmT, 105, 167, 173, 185, 199
GOptimizationAlgorithmT<>, 211
GOptimizationMonitorT, 106, 211
GOptimizationMonitorT<>, 211
GParaboloid2D, 215
GParaboloidIndividual2D, 91, 98
GParameterBase, 102, 113
GParameterBase::adaptionsActive(), 114
GParameterBase::setAdaptionsActive(), 114
GParameterBase::setAdaptionsInactive(), 114
GParameterObjectCollection, 110, 113, 130
GParameterSet, 91, 104, 145
GParameterT, 111
GParserBuilder, 215, 286
GPersonalityTraits, 105
GPGPU, 47, 57
GPreEvaluationValidityCheckT<ind_type>::setAllowNegative(), 158
Gradient Descent, 12
Gradient Descent Configuration Options, 194
Gradient Descents, 17
Gradient descents, 18
Gradient Methods with Geneva, 191
GRandom, 136, 264
GRandom::uniform_01<double>(), 265
GRandom::uniform_real(const double&), 266
GRandomT.hpp, 264
Gray Code, 31
GridKa, 49
Grids, 47
GRIDtoday, 55
GSerialEA, 106
GSimulatedAnnealingFactory, 183
GSTDERR, 286
GSTDOUT, 286
GStdPtrVectorInterface, 150
GStdSimpleVectorInterface, 112
GSubmissionContainerT, 276
GTERMINATE, 285
GThreadGroup, 292
GThreadPool, 292
GWARNING, 285

Halt Criteria, 26, 170
Hans-Paul Schwefel, 21
Hap library, 263
High multiplicity, 11
histogram, 10
Hybrid feature vectors, 31
hybrid selection scheme, 26

Ian Foster, 52
ideal solution, 16
INACTIVEONLY, 149
INCOMPLETERETURN, 277
Individuals, 88, 103, 145
Infiniband, 49
infinitesimal step, 18
INFOEND, 169, 212
INFOINIT, 169, 212
INFOPROCESSING, 169, 212
informationFunction(), 212
Ingo Rechenberg, 21
Initialization, 22
innermost classes, 101
Installation from source, 81
Installation using packages, 80
INSTALLDIR, 82
Integer random numbers, 270
Internal Architecture of Optimization Monitors, 211
invalidity, 161

J. Kennedy, 39
Jeff Garland, 281
John Holland, 21
JSON, 286

Large Data Sets, 153
Large Hadron Collider, 52
lastInformation(), 213
lastInformation(), 212
Latency, 58
latency, 59

Launchpad, 81
LCG, 52
LD_LIBRARY_PATH, 85
LHC, 52
LHCb, 52
libgeneva, 101
libhap, 108
Linux, 50
Loading Static Data at a Remote Site, 201
Local optima, 12
log10_negative_value<fp_type>, 296
log_negative_value<fp_type>, 296
Logging, 283
long double, 109
loosely coupled, 47
LSF, 49

Main Loop, 169
main(), 90, 205, 215
make, 83
math_logic_error, 296
maximum number of iterations, 26
Merkle, 42
Message Passing Interface, 51
Mixing Parameter Types, 181
Models and Reality, 8
Mona Lisa, 65
MPI, 51, 275
MPI-Consumer, 278
MUCOMMANU_PARETO, 181
MUCOMMANU_SINGLEEVAL, 180
Multi-Criterion Optimization, 13
Multi-Populations, 237
Multicriterion optimization, 150
Multipopulations, 32
Multithreaded Execution, 200
Multithreading, 56
MUNU1PRETAIN_SINGLEEVAL, 180
MUPLUSNU_PARETO, 181
MUPLUSNU_SINGLEEVAL, 179
Mutation, 179
mutation, 21
Mutation Probability, 230
Mutex, 282

Nature as a Role Model, 33
neighborhood, 39
neighborhood-best, 40
networked client, 207
new global evaluation criterion, 14
nicely parallel type, 47
noisy quality surface, 28
non-differentiable functions, 18

Object, 93, 111
OpenCL, 57, 109
OpenSUSE Build Service, 81
optimization, 7
Optimization Algorithm Classes, 105
optimization algorithms, 87
Optimization Monitor, 211
Optimization Monitors: Specifics for the Algorithms, 213
optimum, 7
out of source builds, 81
Overhead of gradient descents, 20
overview, 77

Parabola, 301
paraboloid, 84
Parallelization Modes, 199
Parameter Scans, 45
Parameter Scans with Geneva, 195
Parameter Sets, 104
Parameter Specification, 90
Parameter Types, 109
parents, 22
pareto condition, 15
pareto optimization, 15
Parsing Configuration Files, 286
Particle Swarm Optimization, 12
Particle Swarm Optimization with Geneva, 185
Peforming the actual Optimization, 208
personal best, 40
Personalities, 154
Personality Traits, 104
Phases of parametric optimization, 89

pheromone trail, 43
ping, 59
Pluggable Optimization Monitor, 106
Pluggable optimization monitors, 213
pops, 282
population, 105
Population Layout, 192
prepareBuild.sh, 82
probability, 33
problem definitions, 87
Problem description, 90
Prof. Wolfgang Gentzsch, 55
Protein Folding, 70
PSO, 39
push_back(), 93, 171, 174
pushs, 282

quality, 90
quality surface, 12
quality-based halt-criterion, 27
Quantization Effects, 63

raiseException(), 283
Raising Exceptions, 283
Ramachandran, 156
Random normal distribution, 267
Random Number Creation, 107
Random Number Factory, 263
Random numbers in Geneva, 263
RANDOMDUPLICATIONSCHEME, 178
Rastrigin function, 28, 302
Reality, 8
recombination, 21
Recombination Schemes, 27
RESUBMISSIONAFTERTIMEOUT, 277
right balance, 9
robustness, 20
ROOT, 90, 211, 311
Rosenbrock Function, 302
roundtrip-time, 59
RPM, 85

Salomon Function, 303
saSelect(T, iteration), 36
Schwefel Function, 302
scientific analysis, 11
Security, 63
selection, 21
self adjusting, 20
self-adaption, 29
Serial Execution, 199
Serialization, 80, 90, 151, 309
serialize(), 97
setCGlobal(double), 188
setCNeighborhood(double), 188
setCPersonal(double), 188
setCVelocity(double), 188
setFiniteStep(float fs), 194
setNStartingPoints(std::size_t np), 194
setReportIteration(), 170
setStepSize(float sz), 194
Setting up an individual, 215
setValue(), 93, 111
shared_ptr, 80
Simulated Annealing, 26, 33, 183
simulation, 8
Singletons, 291
Smart Pointers, 309
sorting the population, 24
Speedup, 60
sqrt_negative_value<fp_type>, 296
stall counter based halt criterion, 27
Static nature of Grids, 54
Steepest Descent, 191
steepest descent, 17
Steinbuch Centre for Computing, 49
Streaming, 48
streamline(), 96, 148
streamline(std::vector<par_type>& parVec), 104
Supercomputing, 54
Swarm Intelligence, 39
SWARM_UPDATERULE_CLASSIC, 189
SWARM_UPDATERULE_LINEAR, 189

target directory, 85
temperature, 33
The Go2 class, 205

The Origin of Species, 21
The Population Interface, 171
Thread Group, 292
Thread Pool, 292
Thread-Safe Queue, 281
Threads, 310
Tier-1 centre, 51
Tightly coupled, 47
timeout, 282
Training Feed Forward Neural Networks, 71

Uniform double random numbers, 265
USESIGMOID, 162, 163
USEWORSTCASEFORINVALUD, 162
USEWORSTKNOWNVALIDFORINVALID, 162, 163
Using a Factory Class, 174

validity level, 158
valley floor, 20
Value Ranges, 231
VALUEDUPLICATIONSCHEME, 178
VERBOSEMAKEFILE, 83

WLCG, 51
work items, 276
workOn, 278
Worldwide LHC Computing Grid, 51

Xiadong Li, 42