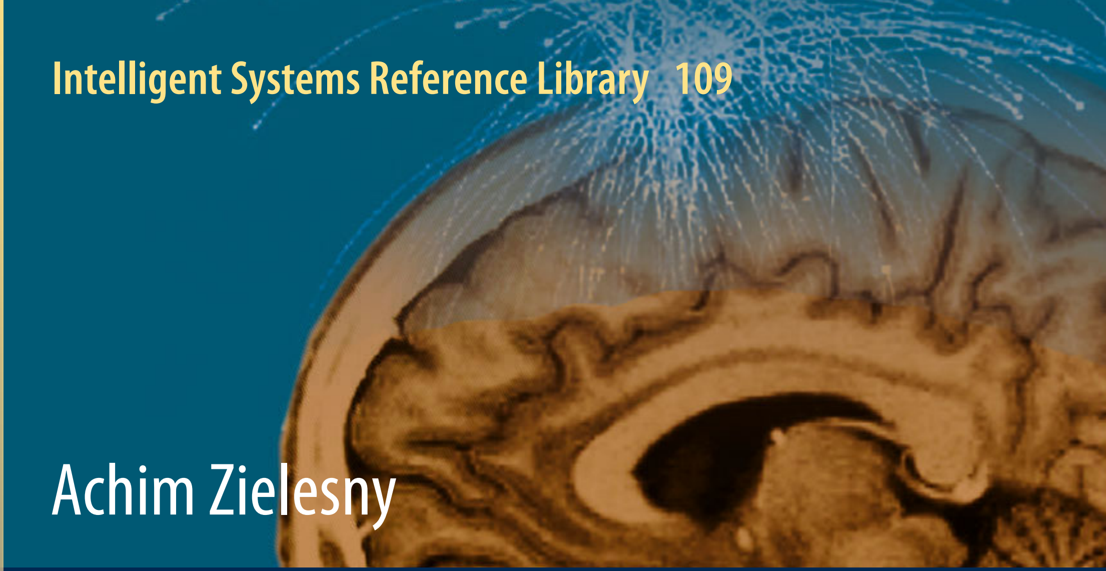
Intelligent Systems Reference Library 109



Achim Zielesny

From Curve Fitting to Machine Learning

An Illustrative Guide to Scientific Data Analysis and Computational Intelligence

Second Edition

Springer

Intelligent Systems Reference Library

Volume 109

Achim Zielesny

From Curve Fitting to Machine Learning

An Illustrative Guide to Scientific Data Analysis and Computational Intelligence

Second Edition

Springer

Achim Zielesny
Institut für biologische und chemische
Informatik
Westfälische Hochschule
Recklinghausen
Germany

ISSN 1868-4394 ISSN 1868-4408 (electronic)
Intelligent Systems Reference Library
ISBN 978-3-319-32544-6 ISBN 978-3-319-32545-3 (eBook)
DOI 10.1007/978-3-319-32545-3

Library of Congress Control Number: 2016936957

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer International Publishing AG Switzerland

To my parents

Preface

Preface to the first edition

The analysis of experimental data is at heart of science from its beginnings. But it was the advent of digital computers in the second half of the 20th century that revolutionized scientific data analysis twofold: Tedious pencil and paper work could be successively transferred to the emerging software applications so sweat and tears turned into automated routines. In accordance with automation the manageable data volumes could be dramatically increased due to the exponential growth of computational memory and speed. Moreover highly non-linear and complex data analysis problems came within reach that were completely unfeasible before. Non-linear curve fitting, clustering and machine learning belong to these modern techniques that entered the agenda and considerably widened the range of scientific data analysis applications. Last but not least they are a further step towards computational intelligence.

The goal of this book is to provide an interactive and illustrative guide to these topics. It concentrates on the road from two-dimensional curve fitting to multidimensional clustering and machine learning with neural networks or support vector machines. Along the way topics like mathematical optimization or evolutionary algorithms are touched. All concepts and ideas are outlined in a clear cut manner with graphically depicted plausibility arguments and a little elementary mathematics. Difficult mathematical and algorithmic details are consequently banned for the sake of simplicity but are accessible by the referred literature. The major topics are extensively outlined with exploratory examples and applications. The primary goal is to be as illustrative as possible without hiding problems and pitfalls but to address them. The character of an illustrative cookbook is complemented with specific sections that address more fundamental questions like the relation between machine learning and human intelligence. These sections may be skipped without affecting the main road but they will open up possibly interesting insights beyond the mere data massage.

All topics are completely demonstrated with the aid of the computing platform Mathematica and the Computational Intelligence Packages (CIP), a high-level function library developed with Mathematica's programming language on top of Mathematica's algorithms. CIP is open-source so the detailed code of every method is freely accessible. All examples and applications shown throughout the book may be used and customized by the reader without any restrictions. This leads to an interactive environment which allows individual manipulations like the rotation of 3D graphics or the evaluation of different settings up to tailored enhancements for specific functionality.

The book tries to be as introductory as possible calling only for a basic mathematical background of the reader - a level that is typically taught in the first year of scientific education. The target readerships are students of (computer) science and engineering as well as scientific practitioners in industry and academia who deserve an illustrative introduction to these topics. Readers with programming skills may easily port and customize the provided code. The majority of the examples and applications originate from teaching efforts or solution providing. The outline of the book is as follows:

- The introductory **chapter 1** provides necessary basics that underlie the discussions of the following chapters like an initial motivation for the interplay of data and models with respect to the molecular sciences, mathematical optimization methods or data structures. The chapter may be skipped at first sight but should be consulted if things become unclear in a subsequent chapter.
- The main chapters that describe the road from curve fitting to machine learning are chapters 2 to 4. The curve fitting **chapter 2** outlines the various aspects of adjusting linear and non-linear model functions to experimental data. A section about mere data smoothing with cubic splines complements the fitting discussions.
- The clustering **chapter 3** sketches the problems of assigning data to different groups in an unsupervised manner with clustering methods. Unsupervised clustering may be viewed as a logical first step towards supervised machine learning - and may be able to construct predictive systems on its own. Machine learning methods may also need clustered data to produce successful results.
- The machine learning **chapter 4** comprises supervised learning techniques, in particular multiple linear regression, three-layer feed-forward neural networks and support vector machines. Adequate data preprocessing and their use for regression and classification tasks as well as the recurring pitfalls and problems are introduced and thoroughly discussed.
- The discussions **chapter 5** supplements the topics of the main road. It collects some open issues neglected in the previous chapters and opens up the scope with more general sections about the possible discovery of new knowledge or the emergence of computational intelligence.

The scientific fields touched in the present book are extensive and in addition constantly and progressively refined. Therefore it is inevitable to neglect an awful lot of important topics and aspects. The concrete selection always mirrors an author's preferences as well as his personal knowledge and overview. Since the missing parts unfortunately exceed the selected ones and people always have strong feelings about what is of importance the final statement has to be a request for indulgence.

Recklinghausen, April 2011 *Achim Zielesny*

Preface to the second edition

The first edition was friendly reviewed as a useful introductory cookbook for the novice reader. The second edition tries to keep this character and resists the temptation to heavily expand topics or lift the discussion to more subtle academic levels. Besides numerous minor additions and corrections throughout the whole book (together with the unavoidable introduction of some new errors) the only substantial extension of the second edition is the addition of Multiple Polynomial Regression (MPR) in order to support the discussions concerning the method crossover from linear and near-linear up to highly non-linear machine learning approaches. As a consequence several examples and applications have been reworked to improve readability and line of reasoning. Also the construction of minimal predictive models is outlined in an updated and more comprehensible manner.

The second edition is based on the extended version 2.0 of the Computational Intelligence Packages (CIP) which now allows parallelized calculations that lead to an often considerably improved performance with multiple (or multicore) processors. Specific parallelization notes are given throughout the book, the description of CIP is accordingly extended and reworked examples and applications make now use of the new functionality.

With this second edition the book hopefully strengthens its original intent to provide a clear and straight introduction to the fascinating road from curve fitting to machine learning.

Recklinghausen, February 2016 *Achim Zielesny*

Acknowledgements

Certain authors, speaking of their works, say, "My book", "My commentary", "My history", etc. They resemble middle-class people who have a house of their own, and always have "My house" on their tongue. They would do better to say, "Our book", "Our commentary", "Our history", etc., because there is in them usually more of other people's than their own.

Pascal

Acknowledgements to the first edition

I would like to thank Lhoussaine Belkoura, Manfred L. Ristig and Dietrich Woermann who kindled my interest for data analysis and machine learning in chemistry and physics a long time ago.

My mathematical colleagues Heinrich Brinck and Soeren W. Perrey contributed a lot - may it be in deep canyons, remote jungles or at our institute's coffee kitchen. To them and my IBCI collaborators Mirco Daniel and Rebecca Schultz as well as the GNWI team with Stefan Neumann, Jan-Niklas Schäfer, Holger Schulte and Thomas Kuhn I am deeply thankful.

The cooperation with Christoph Steinbeck was very fruitful and an exceptional pleasure: I owe a lot to his support and kindness.

Karina van den Broek, Mareike Dörrenberg, Saskia Faassen, Jenny Grote, Jennifer Makalowski, Stefanie Kleiber and Andreas Truszkowski corrected the manuscript with benevolence and strong commitment: Many thanks to all of them.

Last but not least I want to express deep gratitude and love to my companion Daniela Beisser who not only had to bear an overworked book writer but supported all stages of the book and its contents with great passion.

Every book is a piece of collaborative work but all mistakes and errors are of course mine.

Acknowledgements to the second edition

Kolja Berger implemented the code for parallelized CIP calculations with strong commitment and Marie Theiß carefully rechecked the updated CIP version 2.0 commands throughout the whole book: Many thanks to both of them.

For additional support and helpful discussions I am deeply grateful to Karina van den Broek, Karolin Kleemann, Christoph Steinbeck, Andreas Truszkowski, my mathematical colleagues Heinrich Brinck and Soeren W. Perrey, and my companion Daniela Beisser who again accompanied all efforts with great encouragement and passion.

As a final remark I simply have to repeat myself: Every book is a piece of collaborative work but all mistakes and errors are again mine.

Contents

Chapter 1
Introduction

This chapter discusses introductory topics which are helpful for a basic understanding of the concepts, definitions and methods outlined in the following chapters. It may be skipped for the sake of a faster passage to the more appealing issues or only browsed for a short impression. But if things appear dubious in later chapters this one should be consulted again.

Chapter 1 starts with an overview about the interplay between data and models and the challenges of scientific practice especially in the molecular sciences to motivate all further efforts (section 1.1). The mathematical machinery that plays the most important role behind the scenes is dedicated to the field of optimization, i.e. the determination of the global minimum or maximum of a mathematical function. Basic problems and solution approaches are briefly sketched and illustrated (section 1.2). Since model functions play a major role in the main topics they are categorized in an useful manner that will ease further discussions (section 1.3). Data need to be organized in a defined way to be correctly treated by corresponding algorithms: A dedicated section describes the fundamental data structures that will be used throughout the book (section 1.4). A more technical issue is the adequate scaling of data: This is performed automatically by all clustering and machine learning methods but may be an issue for curve fitting tasks (section 1.5). Experimental data experience different sources of error in contrast to simulated data which are only artificially biased by true statistical errors. Errors are the basis for a proper statistical analysis of curve fitting results as well as for the assessment of machine learning outcomes. Therefore the different sources of error and corresponding conventions are briefly described (section 1.6). Machine learning methods may be used for regression or classification tasks: Whereas regression tasks demand a precise calculation of the desired output values a classification task requires only the correct assignment of an input to a desired output class. Within this book classification tasks are tackled as adequately coded regression tasks which is sketched in a specific section (1.7). The Computational Intelligence Packages (CIP) offer a largely unified structure for different types of calculations which is summarized in a following section to make their use more intuitive and less subtle. In addition a short description of Mathematica's top-down programming and proper initialization is provided (section 1.8). This chapter ends with a note on the reproducibility of calculations reported throughout the book (section 1.9).

1.1 Motivation: Data, models and molecular sciences

Essentially, all models are wrong, but some are useful.

G.E.P. Box

Science is an endeavor to understand and describe the real world out there to (at best) alleviate and enrich human existence. But the structures and dynamics of the real world are very intricate and complex. A humble chemical reaction in the laboratory may already *involve perhaps* 10^{20} *molecules surrounded by* 10^{24} *solvent molecules, in contact with a glass surface and interacting with gases ... in the atmosphere. The whole system will be exposed to a flux of photons of different frequency (light) and a magnetic field (from the earth), and possibly also a temperature gradient from external heating. The dynamics of all the particles (nuclei and electrons) is determined by relativistic quantum mechanics, and the interaction between particles is governed by quantum electrodynamics. In principle the gravitational and strong (nuclear) forces should also be considered. For chemical reactions in biological systems, the number of different chemical components will be large, involving various ions and assemblies of molecules behaving intermediately between solution and solid state (e.g. lipids in cell walls)* [Jensen 2007]. Thus, to describe nature, there is the inevitable necessity to set up limitations and approximations in form of simplifying and idealized models - based on the known laws of nature. Adequate models neglect almost everything (i.e. they are, strictly speaking, wrong) but they may keep some of those essential real world features that are of specific interest (i.e. they may be useful).

The dialectical interplay of experiment and theory is a key driving force of modern science. Experimental data do only have meaning in the light of a particular model or at least a theoretical background. Reversely theoretical considerations may be logically consistent as well as intellectually elegant: Without experimental evidence they are a mere exercise of thought no matter how difficult they are. Data analysis is a connector between experiment and theory: Its techniques advise possibilities of model extraction as well as model testing with experimental data.

Model functions have several practical advantages in comparison to mere enumerated data: They are a comprehensive representation of the relation between the quantities of interest which may be stored in a database in a very compact manner with minimum memory consumption. A good model allows interpolating or extrapolating calculations to generate new data and thus may support (up to replace) expensive lab work. Last but not least a suitable model may be heuristically used to explore interesting optimum properties (i.e. minima or maxima of the model function) which could otherwise be missed. Within a market economy a good model is simply a competitive advantage.

The ultimate goal of all sciences is to arrive at quantitative models that describe nature with a sufficient accuracy - or to put it short: to calculate nature. These calculations have the general form

$$\text{answer} = f(\text{question}) \text{ or } \text{output} = f(\text{input})$$

where input denotes a question and output the corresponding answer generated by a model function *f*. Unfortunately the number of interesting quantities which can be directly calculated by application of theoretical ab-initio techniques solely based on the known laws of nature is rather limited (although expanding). For the overwhelming number of questions about nature the model functions f are unknown or too difficult to be evaluated. This is the daily trouble of chemists, material's scientists, engineers or biologists who want to ask questions like the biological effect of a new molecular entity or the properties of a new material's composition. So in current science there are three situations that may be sensibly distinguished due to our knowledge of nature:

- **Situation 1:** The model function f is theoretically or empirically known. Then the output quantity of interest may be calculated directly.
- **Situation 2:** The structural form of the function f is known but not the values of its parameters. Then these parameter values may be statistically estimated on the basis of experimental data by curve fitting methods.
- **Situation 3:** Even the structural form of the function f is unknown. As an approximation the function f may be modelled by a machine learning technique on the basis of experimental data.

A simple example for situation 2 is the case that the relation between input and output is known to be linear. If there is only one input variable of interest, denoted *x,* and one output variable of interest, denoted *y,* the structural form of the function f is a straight line

$$y = f(x) = a_1 + a_2 x$$

where a_1 and a_2 are the unknown parameters of the function which may be statistically estimated by curve fitting of experimental data. In situation 3 it is not only the values of the parameters that are unknown but in addition the structural form of the model function f itself. This is obviously the worst possible case which is addressed by data smoothing or machine learning approaches that try to construct a model function with experimental data only.

Situations 1 to 3 are widely encountered by the contemporary molecular sciences. Since the scientific revolution of the early 20th century the molecular sciences have a thorough theoretical basis in modern physics: Quantum theory is able to (at least in principle) quantitatively explain and calculate the structure, stability and reactivity of matter. It provides a fundamental understanding of chemical bonding and molecular interactions. This foundational feat was summarized in 1929 by Paul A. M. Dirac with famous words: *The underlying physical laws necessary for the mathematical theory of a large part of physics and the whole of chemistry are thus completely known* ... it became possible to submit molecular research and development (R&D) problems to a theoretical framework to achieve correct and satisfactory solutions - but unfortunately Dirac had to continue ... *and the difficulty is only that the exact application of these laws leads to equations much too complicated to be soluble.* The humble "only" means a severe practical restriction: It is in fact only the smallest quantum-mechanical systems like the hydrogen atom with one single proton in the nucleus and one single electron in the surrounding shell that can be treated by pure analytical means to come to an exact mathematical solution, i.e. by solving the Schroedinger equation of this mechanical system with pencil and paper. Nonetheless Dirac added an optimistic prospect: *It therefore becomes desirable that approximate practical methods of applying quantum mechanics should be developed, which can lead to an explanation of the main features of complex atomic systems without too much computation* [Dirac 1929]. A few decades later this hope begun to turn into reality with the emergence of digital computers and their exponentially increasing computational speed: Iterative methods were developed that allowed an approximate quantum-mechanical treatment of molecules and molecular ensembles with growing size (see [Leach 2001], [Frenkel 2002] or [Jensen 2007]). The methods which are ab-initio approximations to the true solution of the Schroedinger equation (i.e. they only use the experimental values of natural constants) are still very limited in applicability so they are restricted to chemical ensembles with just a few hundred atoms to stay within tolerable calculation periods. If these methods are combined with experimental data in a suitable manner so that they become semi-empirical the range of applicability can be extended to molecular systems with several thousands of atoms (up to more than a hundred thousand atoms by the writing of this book [Clark 2010/2015]). The size of the molecular systems and the time frames for their simulation can be even further expanded by orders of magnitude with mechanical force fields that are constructed to mimic the quantum-mechanical molecular interactions so that an atomistic description of matter exceeds the million-atoms threshold. In 1998 and 2013 the Royal Swedish Academy of Sciences honored these scientific achievements by awarding the Nobel prize in chemistry with the prudent comment in 1998 that *Chemistry is no longer a purely experimental science* (see [Nobel Prize 1998/2013]). This atomistic theory-based treatment of molecular R&D problems corresponds to situation 1 where a theoretical technique provides a model function f to "simply calculate" the desired solution in a direct manner.

Despite these impressive improvements (and more is to come) the overwhelming majority of molecular R&D problems is (and will be) out of scope of these atomistic computational methods due to their complexity in space and time. This is especially true for the life and the nano sciences that deal with the most complex natural and artificial systems known today - with the human brain at the top. Thus the molecular sciences are mainly faced with situations 2 and 3: They are a predominant area of application of the methods to be discussed on the road from curve fitting to machine learning. Theory-loaded and model-driven research areas like physical chemistry or biophysics often prefer situation 2: A scientific quantity of interest is studied in dependence of another quantity where the structural form of a model function f that describes the desired dependency is known but not the values of its parameters. In general the parameters may be purely empirical or may have a theoretically well-defined meaning. An example of the latter is usually encountered in chemical kinetics where phenomenological rate equations are used to describe the temporal progress of the chemical reactions but the values of the rate constants - the crucial information - are unknown and may not be calculated by a more fundamental theoretical treatment [Grant 1998]. In this case experimental measurements are indispensable that lead to xy-error data triples (x_i, y_i, σ_i) with an argument value x_i, the corresponding dependent value y_i and the statistical error σ_i of the y_i value (compare below). Then optimum estimates of the unknown parameter values can be statistically deduced on the basis of these data triples by curve fitting methods. In practice a successful model function may at first be only empirically constructed like the quantitative description of the temperature dependence of a liquid's viscosity (illustrated in chapter 2) and then later be motivated by more theoretical lines of argument. Or curve fitting is used to validate the value of a specific theoretical model parameter by experiment (like the critical exponents in chapter 2). Last but not least curve fitting may play a pure support role: The energy values of the potential energy surface of hydrogen fluoride could be directly calculated by a quantum-chemical ab-initio method for every distance between the two atoms. But a restriction to a limited number of distinct calculated values that span the range of interest in combination with the construction of a suitable smoothing function for interpolation (shown in chapter 2) may save considerable time and enhance practical usability without any relevant loss of precision.

With increasing complexity of the natural system under investigation a quantitative theoretical treatment becomes more and more difficult. As already mentioned a quantitative theory-based prediction of a biological effect of a new molecular entity or the properties of a new material's composition are in general out of scope of current science. Thus situation 3 takes over where a model function f is simply unknown or too complex. To still achieve at least an approximate quantitative description of the relationships in question a model function may be tried to be solely constructed with the available data only - a task that is at heart of machine learning. Especially quantitative relationships between chemical structures and their biological activities or physico-chemical and material's properties draw a lot of attention: Thus QSAR (Quantitative Structure Activity Relationship) and QSPR (Quantitative Structure Property Relationship) studies are active fields of research in the life, material's and nano sciences (see [Zupan 1999], [Gasteiger 2003], [Leach 2007] or [Schneider 2008]). Cheminformatics and structural bioinformatics provide a bunch of possibilities to represent a chemical structure in form of a list of numbers (which mathematically form a vector or an input in terms of machine learning, see below). Each number or sequence of numbers is a specific structural descriptor that describes a specific feature of a chemical structure in question, e.g. its molecular weight, its topological connections and branches or electronic properties like its dipole moments or its correlation of surface charges. These structure-representing inputs alone may be analyzed by clustering methods (discussed in chapter 3) for their chemical diversity. The results may be used to generate a reduced but representative subset of structures with a similar chemical diversity in comparison to the original larger set (e.g. to be used in combinatorial chemistry approaches for a targeted structure library design). Alternatively different sets of structures could be compared in terms of their similarity or dissimilarity as well as their mutual white spots (these topics are discussed in chapter 3). A structural descriptor based QSAR/QSPR approach takes the form

$$\text{activity}/\text{property} = f(\text{descriptor1}, \text{descriptor2}, \text{descriptor3}, ...)$$

with the model function f as the final target to become able to make model-based predictions (the methods used for the construction of an approximate model function f are outlined in chapter 4). The extensive volume of data that is necessary for this line of research is often obtained by modern high-throughput (HT) techniques like the biological assay-based high-throughput screening (HTS) of thousands of chemical compounds in the pharmaceutical industry or HT approaches in materials science all performed with automated robotic lab systems. Among others these HT methods lead to the so called BioTech data explosion that may be thoroughly exploited for model construction. In fact HT experiments and model construction via machine learning are mutually dependent on each other: Models deserve data for their creation as well as the mere heaps of data produced by HT methods deserve models for their comprehension.

With these few statements about the needs of the molecular sciences in mind the motivation of this book is to show how situations 2 (model function f known, its parameters unknown) and 3 (model function f itself unknown) may be tackled on the road from curve fitting to machine learning: How can we proceed from experimental data to models? What conceptual and technical problems occur along this path? What new insights can we expect?

1.2 Optimization

```
Clear["Global`*"];
<<CIP`Graphics`
```

At the beginning of each section or subsection the global Clear command clears all earlier variables and definitions and thus cares for a proper initialization. Then the necessary CIP packages are loaded, e.g. the Graphics package for this section. A proper initialization prevents possible code interferences due to earlier definitions. Note that Mathematica has a top-down programming style: Once a variable is assigned it keeps its value.

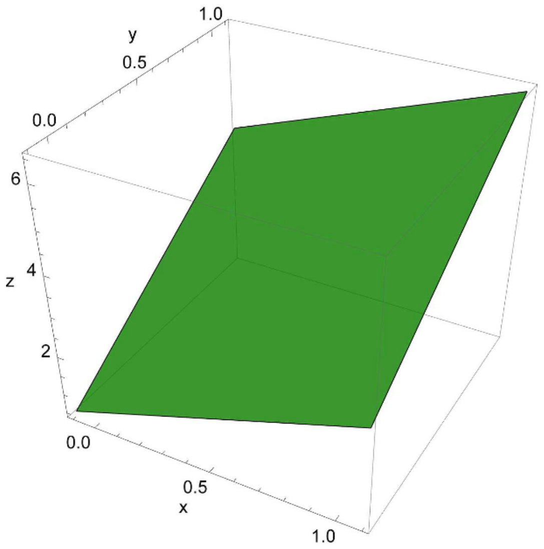
Optimization means a process that tries to determine the optima, i.e. the minima and maxima of a mathematical function. A plethora of important scientific problems can be traced back to an issue of optimization so they are essentially optimization problems. Optimization tasks also lie at heart of the road from curve fitting to machine learning: The methods discussed in later chapters will predominantly use mathematical optimization techniques to do their job. It should be noticed that the following optimization strategies are also utilized for the (common) research situation where no direct path to success can be advised and a kind of educated trial and error is the only way to progress.

A mathematical function may contain ...

- ... **no optimum at all**. An example is a 2D straight line, a 3D plane (illustrated below) or a hyperplane in many dimension. But also non-linear functions like the exponential function may not contain any optimum.

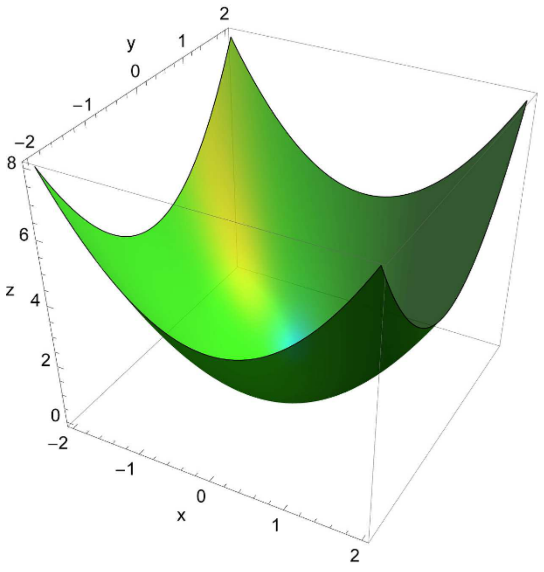
```
pureFunction=Function[{x,y},1.0+2.0*x+3.0*y];
xRange={-0.1,1.1};
yRange={-0.1,1.1};
labels={"x","y","z"};
CIP`Graphics`Plot3dFunction[pureFunction,xRange,yRange,labels]
```

All CIP based calculations are scripted as shown above: First all variables are defined with intuitive names and then passed to specific CIP functions to calculate results or create graphical illustrations. All variables remain valid until the next global Clear command. Note that Mathematica allows the definition of pure functions which may be used like normal variables. If a specific function definition is to be passed to a CIP method a pure function is commonly used. The CIP methods internally use pure functions for distinct function value evaluations. Pure functions are a powerful functional programming feature of the Mathematica computing platform to simplify many operations in an elegant and efficient manner.



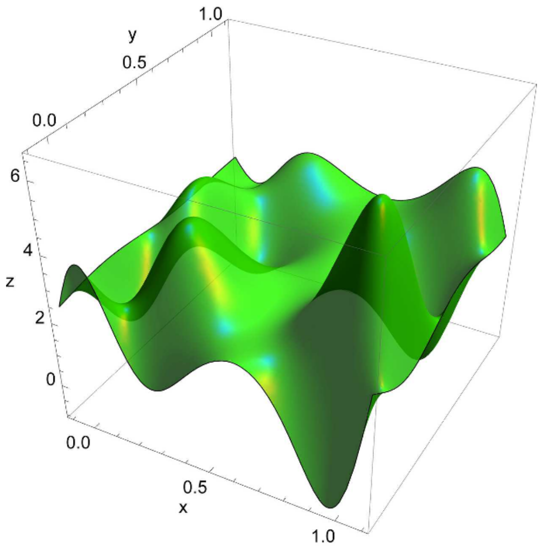
- ... **exactly one optimum**, e.g. a 2D quadratic parabola, a 3D parabolic surface (illustrated below) or a parabolic hyper surface in many dimensions.

```
pureFunction=Function[{x,y},x^2+y^2];
xRange={-2.0,2.0};
yRange={-2.0,2.0};
CIP`Graphics`Plot3dFunction[pureFunction,xRange,yRange,labels]
```



- ... **multiple up to an infinite number of optima** like a 2D sine function, a curved 3D surface (illustrated below) or a curved hyper surface in multiple dimensions.

```
pureFunction=Function[
 {x,y},1.9*(1.35+Exp[x]*Sin[13.0*(x-0.6)^2]*Exp[-y]*Sin[7.0*y])];
xRange={-0.1,1.1};
yRange={-0.1,1.1};
CIP`Graphics`Plot3dFunction[pureFunction,xRange,yRange,labels]
```



The sketched categorization holds for functions with one argument

$$y = f(x)$$

as well as functions with multiple arguments

$$y = f(x_1, x_2, ..., x_M) = f(\underline{x}) \text{ with } \underline{x} = (x_1, x_2, ..., x_M)$$

i.e. from 2D curves $f(x)$ up to M-dimensional hyper surfaces $f(x_1, x_2, ..., x_M)$. If no optimum exists there is obviously nothing to optimize. For a curve or hyper surface that contains exactly one optimum the optimization problem is usually successfully solvable by analytical methods which are able to calculate the optimum position directly. It is the last category of non-linear functions with multiple optima that cause severe problems - and unfortunately the overwhelming majority of practical applications belong to this drama: The following sections try to reveal some of its tragedy and ways to hold forth a hope again.

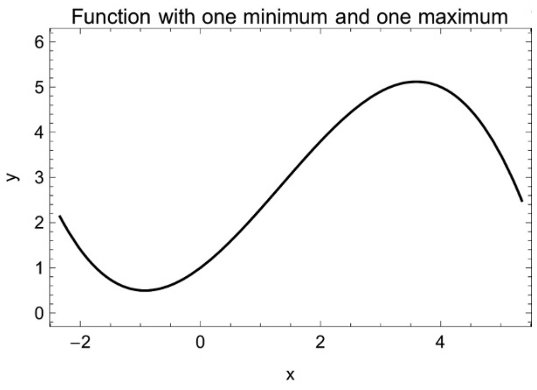
1.2.1 Calculus

```
Clear["Global`*"];
<<CIP`Graphics`
```

The standard analytical procedure to determine optima is known from calculus: An example function of the form $y = f(x)$ with one argument x may contain one minimum and one maximum:

```
function=1.0+1.0*x+0.4*x^2-0.1*x^3;
pureFunction=Function[argument,function/.x -> argument];
argumentRange={-2.0,5.0};
functionValueRange={0.0,6.0};
labels={"x","y","Function with one minimum and one maximum"};
CIP`Graphics`Plot2dFunction[pureFunction,argumentRange,
 functionValueRange,labels]
```

Note that the function is defined twice for different purposes: First as a normal symbolic function and in addition as a pure function. The normal function is used in subsequent calculations, the pure function as an argument of the CIP method Plot2dFunction.



To calculate the positions of the optima the first derivative

```
firstDerivative=D[function,x]
```

$1. + 0.8x - 0.3x^2$

D is Mathematica's operator for partial differentiation to a specified variable which is x in this case.

and their (two) roots are determined:

```
roots=Solve[firstDerivative==0,x]
```

$\{\{x \to -0.927443\}, \{x \to 3.59411\}\}$

Solve is Mathematica's command to solve (systems of) equations. The Solve command returns a list in curly brackets with two rules (also in curly brackets) for setting the x value to solve the equation in question, i.e.

assigning -0.927443 or 3.59411 to x solves the equation. Also note that the number of digits of the result values is a standard output only: A higher precision could be obtained on demand and is used for internal calculations (usually the machine precision supported by the hardware).

Then the second derivative

```
secondDerivative=D[function,{x,2}]
```

$0.8 - 0.6x$

D may be told to calculate higher derivatives, i.e. the second derivative in this case.

is used to analyze the type of the two detected optima:

```
secondDerivative/.roots[[1]]
```

1.35647

roots[[1]] denotes the first expression of the roots list above, i.e. the rule $\{x \to \textit{-0.927443}\}$*: This means that the value -0.927443 is to be assigned to x. The /. notation applies this rule to the secondDerivative expression before, i.e. the x in secondDerivative gets the value -0.927443 and then secondDerivative is numerically evaluated to 1.35647. These Mathematica specific notations seem to be a bit puzzling at first but they become convenient and powerful with increased usage.*

A value larger zero indicates a minimum at the first optimum position and

```
secondDerivative/.roots[[2]]
```

-1.35647

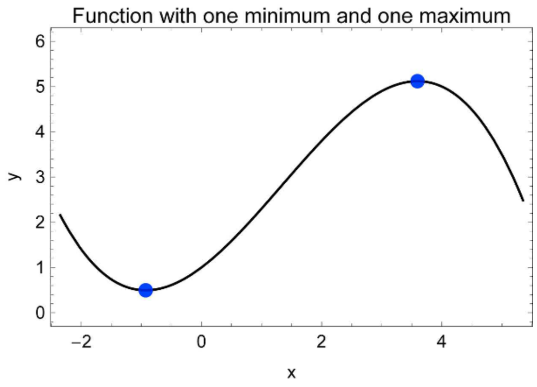
a value smaller zero a maximum at the second optimum position. The determined minimum and maximum points

```
minimumPoint={x/.roots[[1]],function/.roots[[1]]};
maximumPoint={x/.roots[[2]],function/.roots[[2]]};
```

may be displayed for visual validation:

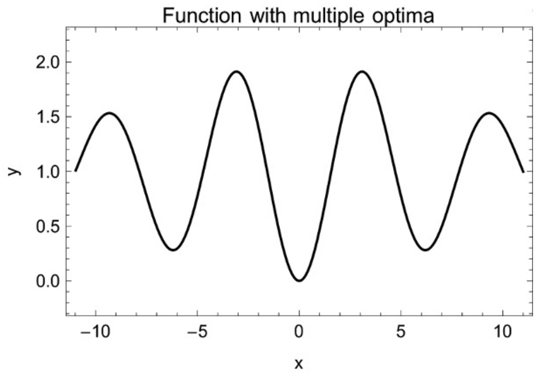
```
points2D={minimumPoint,maximumPoint};
CIP`Graphics`Plot2dPointsAboveFunction[points2D,pureFunction,labels,
 GraphicsOptionArgumentRange2D -> argumentRange,
 GraphicsOptionFunctionValueRange2D -> functionValueRange]
```

Method signatures may contain variables and options. Options are set with an arrow as shown in the Plot2dPointsAboveFunction method above. In contrast to variables the options must not be specified: Then their default values are used.



Unfortunately this analytical procedure fails in general. Lets take a somewhat more difficult function with multiple (or more precise: an infinite number of) optima:

```
function=1.0-Cos[x]/(1.0+0.01*x^2);
pureFunction=Function[argument,function/.x -> argument];
argumentRange={-10.0,10.0};
functionValueRange={-0.2,2.2};
labels={"x","y","Function with multiple optima"};
CIP'Graphics'Plot2dFunction[pureFunction,argumentRange,
 functionValueRange,labels]
```



The first derivative may still be obtained

```
firstDerivative=D[function,x]
```

$$-\frac{0.02x\mathrm{Cos}[x]}{\left(1.+0.01x^2\right)^2}+\frac{\mathrm{Sin}[x]}{1.+0.01x^2}$$

but the determination of the roots fails

```
roots=Solve[firstDerivative==0,x]
```

The equations appear to involve the variables to be solved for in an essentially non-algebraic way.

$$\text{Solve}\left[\frac{0.02x\text{Cos}[x]}{\left(1.+0.01x^2\right)^2}+\frac{\text{Sin}[x]}{1.+0.01x^2}==0,x\right]$$

since this non-linear equation can no longer be solved by analytical means. This problem becomes even worse with functions that contain multiple arguments

$$y = f(x_1, x_2, ..., x_M) = f(\underline{x})$$

i.e. with M-dimensional curved hyper surfaces. The necessary condition for an optimum of a M-dimensional hyper surface y is that all partial derivatives become zero:

$$\frac{\partial f(x_1, x_2, ..., x_M)}{\partial x_i} = 0 \; ; \; i = 1, ..., M$$

Whereas the partial derivatives may be successfully evaluated in most cases the resulting system of M (usually non-linear) equations may again not be solvable by analytical means in general. So the calculus-based analytical optimization is restricted to only simple non-linear special cases (linear functions are out of question since they do not contain optima at all). Since these special cases are usually taught extensively at schools and universities (they are ideal for examinations) there is the ongoing impression that the calculus-based solution of optimization problems also achieves success in practice. But the opposite is true: The overwhelming majority of scientific optimization problems is far too difficult for a successful calculus-based treatment. That is one reason why digital computers revolutionized science: With their exponentially growing calculation speed (known as Moore's law which - successfully - predicts a doubling of calculation speed every 18 months) they opened up the perspective for iterative search-based approaches to at least approximate optima in these more difficult and practically relevant cases - a procedure that is simply not feasible with pencil and paper in a man's lifetime.

1.2.2 Iterative optimization

```
Clear["Global`*"];
<<CIP`Graphics`
```

In general the optima of curves and hyper surfaces may only be approximated by iterative step-by-step search procedures - but without any guarantee of success! There are two basic types of iterative optimization strategies:

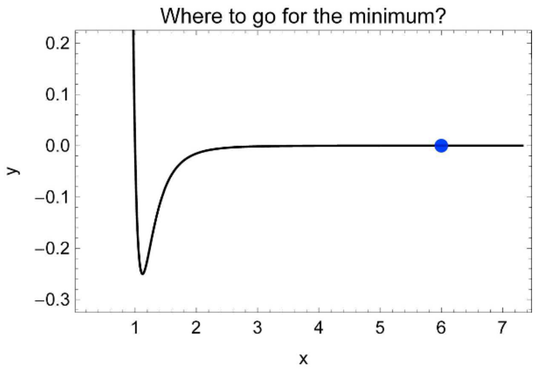
- **Local optimization:** Beginning at a start position the iterative search method tries to find at least a local optimum (which may not necessarily be the next neighbored optimum to the start position). This local optimum is in general different from the global optimum, i.e. the lowest minimum or the highest maximum of the function.
- **Global optimization:** The iterative search method tries to find the global optimum inside an a priori defined search space.

Global iterative optimization is usually far more computational demanding than local optimization and therefore slower. Both optimization strategies may fail due to two sources of problems:

- **Function related problems:** The function itself to optimize may not contain any optima (e.g. a straight line or a hyperplane) or may otherwise be ill-shaped.
- **Iterative search related problems:** The search algorithm may encounter numerical problems (like division by zero) or simply not find an optimum of required precision within the allowed maximum number of iterations. Whereas in the latter case an increase of the number of iterations should help this solution would fail if the search algorithm is trapped in oscillations around the optimum. Problems are often caused by an inappropriate start position or search space, e.g. if the search algorithm relies on second derivative information but the curvature of the function to be optimized is effectively zero in the search region.

As an example for an unfavorable start position for a minimum detection consider the following situation:

```
function=1.0/x^12-1/x^6;
pureFunction=Function[argument,function/.x -> argument];
xStart=6.0;
startPointForOptimization={xStart,pureFunction[xStart]};
points2D={startPointForOptimization};
argumentRange={0.5,7.0};
functionValueRange={-0.3,0.2};
labels={"x","y","Where to go for the minimum?"};
CIP`Graphics`Plot2dPointsAboveFunction[points2D,pureFunction,labels,
 GraphicsOptionArgumentRange2D -> argumentRange,
 GraphicsOptionFunctionValueRange2D -> functionValueRange]
```



The start position (point) is fairly outside the interesting region that contains the minimum: Its slope (first derivative)

```
D[function,x]/.x -> xStart
```

0.0000214326

and its curvature (second derivative)

```
D[function,{x,2}]/.x -> xStart
```

−0.0000250037

are nearly zero with the function value itself being nearly constant. In this situation it is difficult for any iterative algorithm to devise a path to the minimum and it is likely for the search algorithm to simply run aground without converging to the minimum.

In practice it is often hard to recognize what went wrong if an optimization failure occurs. And although there are numerous parameters to tune local and global optimization methods for specific optimization problems that does not guarantee to always solve these issues in general. And it becomes clear that any a priori knowledge about the location of an optimum from theoretical considerations or practical experience may play a crucial role. Throughout the later chapters a number of standard problems are discussed and strategies for their circumvention are described.

1.2.3 Iterative local optimization

```
Clear["Global`*"];
```

```
<<CIP`Graphics`
```

Iterative local optimization (or just minimization since maximizing a function f is identical to minimizing $-f$ or f^{-1}) is in principle a simple issue: From a given start position just move downhill as fast as possible by appropriate steps until a local minimum is reached within a desired precision. Thus local optimization methods differ only in the amount of functional information they evaluate to set their step sizes along their chosen downhill directions (see [Press 2007] for details). The evaluation part determines the computational costs of each iteration whereas the directional part determines the convergence speed towards a local minimum where both parts often oppose each other: The more functional information is evaluated the slower a single iteration is performed but the number of iterative steps may be reduced due to more appropriate step sizes and directions.

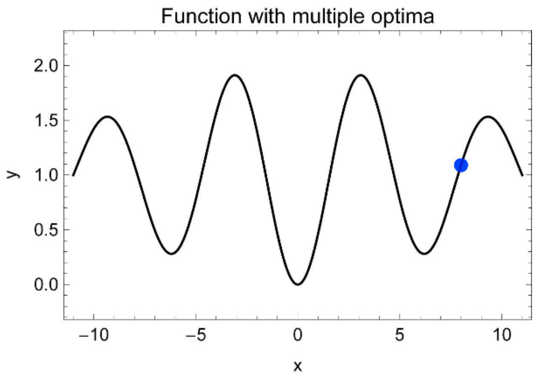
- Some methods do **only use function value evaluations** at different positions to recognize more or less intelligent downhill paths with adaptive step sizes, e.g. the Simplex method.
- More advanced methods use (first derivative) **slope/gradient information** in addition to function values which allows steepest descent orientations: The so called Gradient method and the more elaborate Conjugate-Gradient and Quasi-Newton methods belong to this type of minimization techniques: The latter two families of methods can find the (one and global) minimum of a M-dimensional parabolic hyper surface with at most M steps (note that this statement just describes a characteristic feature of these algorithms since the optimum of a parabolic hyper surface may simply be calculated with second derivative information by analytical means).
- Also (second derivative) **curvature information** of the function to be minimized may be utilized for a faster convergence near a local minimum as implemented by the so called Newton methods (which were already invented by the grand old father of modern science). If a parabolic hyper surface is under investigation a Newton step leads directly to the minimum, i.e. the Newton method converges to this minimum in one single step (in fact each Newton step assumes a hyper surface to be parabolic and thus calculates the position of its supposed minimum analytically. This assumption is the more accurate the nearer the minimum is located. Since a Newton method has to evaluate an awful lot of functional information for each iterative step which takes its time it is only effective in the proximity of a minimum).

For special types of functions to be minimized like a sum of squares specific combination methods like Levenberg-Marquardt are helpful that try to switch between gradient steps (far from a minimum) and Newton steps (near a minimum) in an effective manner. And besides these general iterative local minimization techniques there are numerous specific solutions for specific optimization tasks that try to take advantage of their specific characteristics. But note that in general there is nothing like the best iterative local optimization method: Being the most effective and therefore fastest method for one minimization problem does not mean to be necessarily superior for another. As a rule of thumb Conjugate-Gradient and Quasi-Newton methods have shown to exert a good compromise between computational costs (function and first derivatives evaluations) and local minimum convergence speed for many practical minimization problems. For the already used multiple optima function

```
function=1.0-Cos[x]/(1.0+0.01*x^2);
pureFunction=Function[argument,function/.x -> argument];
argumentRange={-10.0,10.0};
functionValueRange={-0.2,2.2};

startPosition=8.0;
startPoint={startPosition,function/.x -> startPosition};
points2D={startPoint};

labels={"x","y","Function with multiple optima"};
CIP`Graphics`Plot2dPointsAboveFunction[points2D,pureFunction,labels,
 GraphicsOptionArgumentRange2D -> argumentRange,
 GraphicsOptionFunctionValueRange2D -> functionValueRange]
```



a local minimum may be found from the specified start position (indicated point) with Mathematica's FindMinimum command that provides a unified access to different local iterative search methods (FindMinimum uses a variant of the Quasi-Newton methods by default, see comments on [FindMinimum/FindMaximum] in the references):

```
localMinimum=FindMinimum[function,{x,startPosition}]
```

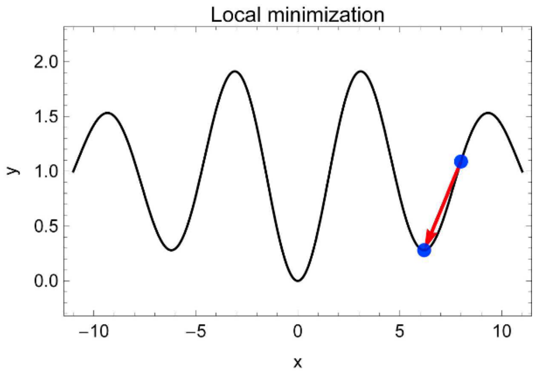
$\{0.28015, \{x \to 6.19389\}\}$

FindMinimum returns a list with the function value at the detected local minimum and the rule(s) for the argument value(s) at this minimum

Start point and approximated minimum may be visualized (the arrow indicates the minimization path):

```
minimumPoint={x/.localMinimum[[2]],localMinimum[[1]]};
points2D={startPoint,minimumPoint};
labels={"x","y","Local minimization"};
arrowGraphics=Graphics[{Thick,Red,{Arrowheads[Medium],
 Arrow[{startPoint,minimumPoint}]}}];
functionGraphics=CIP`Graphics`Plot2dPointsAboveFunction[points2D,
 pureFunction,labels,
 GraphicsOptionArgumentRange2D -> argumentRange,
 GraphicsOptionFunctionValueRange2D -> functionValueRange];
Show[functionGraphics,arrowGraphics]
```

Mathematica's Show command allows the overlay of different graphics which are automatically aligned.



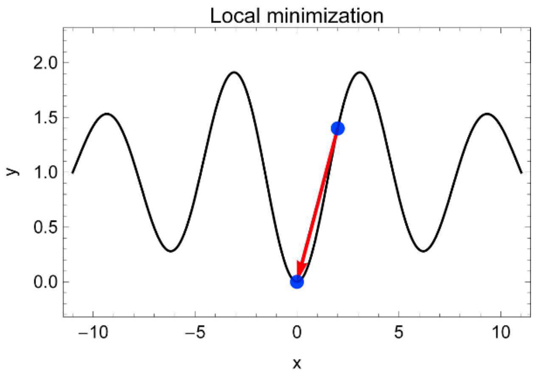
From a different start position a different minimum is found

```
startPosition=2.0;
localMinimum=FindMinimum[function,{x,startPosition}]
```

$\{0.,\{x \to 9.64816 \times 10^{-12}\}\}$

again illustrated as before:

```
startPoint={startPosition,function/.x -> startPosition};
minimumPoint={x/.localMinimum[[2]],localMinimum[[1]]};
points2D={startPoint,minimumPoint};
arrowGraphics=Graphics[{Thick,Red,{Arrowheads[Medium],
 Arrow[{startPoint,minimumPoint}]}}];
functionGraphics=CIP`Graphics`Plot2dPointsAboveFunction[points2D,
 pureFunction,labels,
 GraphicsOptionArgumentRange2D -> argumentRange,
 GraphicsOptionFunctionValueRange2D -> functionValueRange];
Show[functionGraphics,arrowGraphics]
```



In the last case the approximated minimum is accidentally the global minimum since the start position was near this global optimum. But in general local optimization leads to local optima only.

1.2.4 *Iterative global optimization*

```
Clear["Global`*"];
<<CIP`Graphics`
```

An optimization of a function usually targets the global optimum of the scientifically relevant argument space. An iterative local search may find the global optimum but is usually only trapped in a local optimum near its start position as demonstrated above. Global optimization strategies try to circumvent this problem by sampling a whole a priori defined search space: They need a set of min/max values for each argument $x_1, x_2, ..., x_M$ of the function $f(x_1, x_2, ..., x_M)$ to be globally optimized where it is assumed that the global optimum lies within the search space that is spanned by these M min/max intervals $[x_{1,\text{min}}, x_{1,\text{max}}]$ to $[x_{M,\text{min}}, x_{M,\text{max}}]$. The most straightforward method to achieve this goal seams to be a systematic grid search where the function values are evaluated at equally spaced grid points inside the a priori defined argument search space and then compared to each other to detect the optimum. This grid search procedure is illustrated for an approximation of the global maximum of the curved surface $f(x,y)$ already sketched above

```
function=1.9*(1.35+Exp[x]*Sin[13.0*(x-0.6)^2]*Exp[-y]* Sin[7.0*y]);
pureFunction=
 Function[{argument1,argument2},
  function/.{x -> argument1,y -> argument2}];
```

with a search space of the arguments x and y to be their [0, 1] intervals

```
xMinBorderOfSearchSpace=0.0;
xMaxBorderOfSearchSpace=1.0;
yMinBorderOfSearchSpace=0.0;
yMaxBorderOfSearchSpace=1.0;
```

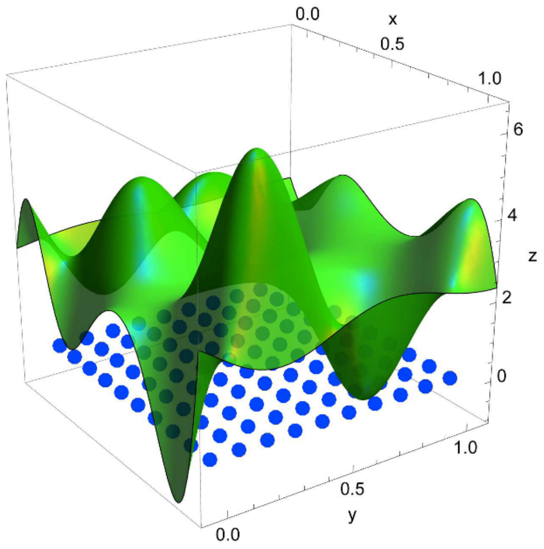
and 100 equally spaced grid points at $z = 0$ inside this search space (100 grid points means a 10×10 grid, i.e. 10 grid points per dimension):

```
numberOfGridPointsPerDimension=10.0;
gridPoints3D={};
Do[
 Do[
  AppendTo[gridPoints3D,{x,y,0.0}],
   {x,xMinBorderOfSearchSpace,xMaxBorderOfSearchSpace,
   (xMaxBorderOfSearchSpace-xMinBorderOfSearchSpace)/
   (numberOfGridPointsPerDimension-1.0)}
 ],
 {y,yMinBorderOfSearchSpace,yMaxBorderOfSearchSpace,
 (yMaxBorderOfSearchSpace-yMinBorderOfSearchSpace)/
 (numberOfGridPointsPerDimension-1.0)}
];
```

The grid points are calculated with nested Do loops in the xy plane.

This setup can be illustrated as follows (with the grid points located at $z = 0$):

```
xRange={-0.1,1.1};
yRange={-0.1,1.1};
labels={"x","y","z"};
viewPoint3D={3.5,-2.4,1.8};
CIP`Graphics`Plot3dPointsWithFunction[gridPoints3D,pureFunction,
 labels,
 GraphicsOptionArgument1Range3D -> xRange,
 GraphicsOptionArgument2Range3D -> yRange,
 GraphicsOptionViewPoint3D -> viewPoint3D]
```



The function values at these grid points are then evaluated and compared

```
winnerGridPoint3D={};
maximumFunctionValue=-Infinity;
Do[
 functionValue=pureFunction[gridPoints3D[[i, 1]],
  gridPoints3D[[i, 2]]];
 If[functionValue>maximumFunctionValue,
  maximumFunctionValue=functionValue;
  winnerGridPoint3D={gridPoints3D[[i, 1]],gridPoints3D[[i, 2]],
   maximumFunctionValue}
 ],
 {i,Length[gridPoints3D]}
];
```

to evaluate the winner grid point

```
winnerGridPoint3D
```

```
{1.,0.222222,6.17551}
```

that corresponds to the maximum detected function value

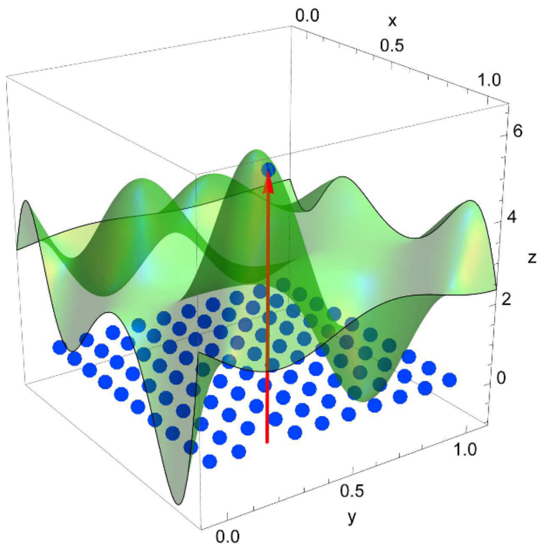
```
maximumFunctionValue
```

```
6.17551
```

which may be visually validated (with the winner grid point raised to its function value indicated by the arrow and all other grid points still located at $z = 0$):

```
Do[
 If[gridPoints3D[[i,1]] == winnerGridPoint3D[[1]] &&
  gridPoints3D[[i,2]] == winnerGridPoint3D[[2]],
  gridPoints3D[[i]] = winnerGridPoint3D
 ],
 {i,Length[gridPoints3D]}
];

arrowStartPoint={winnerGridPoint3D[[1]],winnerGridPoint3D[[2]],0.0};
arrowGraphics3D=Graphics3D[{Thick,Red,{Arrowheads[Medium],
 Arrow[{arrowStartPoint,winnerGridPoint3D}]}}];
plotStyle3D=Directive[Green,Specularity[White,40],Opacity[0.4]];
functionGraphics3D=CIP`Graphics`Plot3dPointsWithFunction[
 gridPoints3D,pureFunction,labels,
 GraphicsOptionArgument1Range3D -> xRange,
 GraphicsOptionArgument2Range3D -> yRange,
 GraphicsOptionViewPoint3D -> viewPoint3D,
 GraphicsOptionPlotStyle3D -> plotStyle3D];
Show[functionGraphics3D,arrowGraphics3D]
```



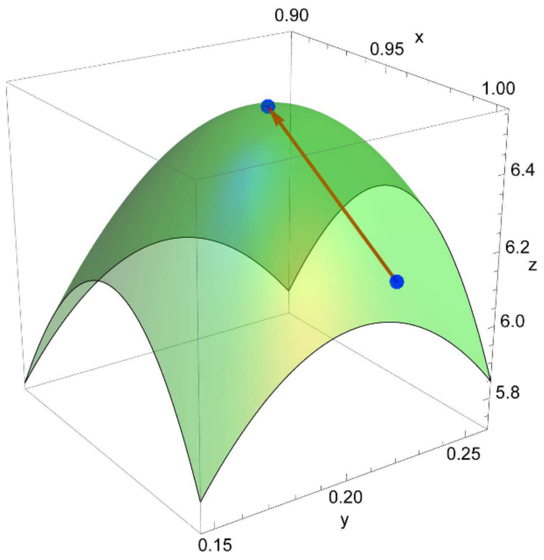
The winner grid point of the global grid search does only approximate the global optimum with an error corresponding to the defined grid spacing. To refine the approximate grid search maximum it may be used as a start point for a following local search since the grid search maximum should be near the global maximum which means that the local search can be expected to converge to the global maximum (but note that there is no guarantee for this proximity and the following convergence in general). Thus the approximate grid search maximum is passed to Mathematica's FindMaximum command (the sister of the FindMinimum command sketched above which utilizes the same algorithms) as a start point for the post-processing local search

```
globalMaximum=FindMaximum[function,{{x,winnerGridPoint3D[[1]]},
 {y,winnerGridPoint3D[[2]]}}]
```

$\{6.54443, \{x \to 0.959215, y \to 0.204128\}\}$

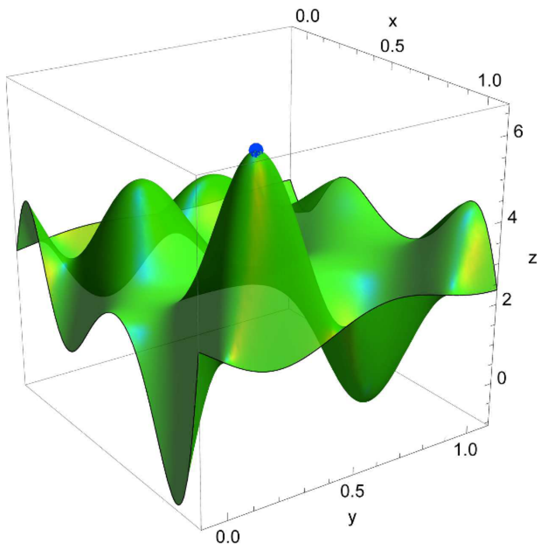
to determine the global maximum with sufficient precision. The improvement obtained by the local refinement process may be inspected (the arrow indicates the maximization path from the winner grid point to the maximum point detected by the post-processing local search in a zoomed view)

```
globalMaximumPoint3D={x/.globalMaximum[[2,1]],
 y/.globalMaximum[[2,2]],globalMaximum[[1]]};
xRange={0.90,1.005};
yRange={0.145,0.26};
arrowGraphics3D=Graphics3D[{Thick,Red,{Arrowheads[Medium],
 Arrow[{winnerGridPoint3D,globalMaximumPoint3D}]}}];
points3D={winnerGridPoint3D,globalMaximumPoint3D};
functionGraphics3D=CIP`Graphics`Plot3dPointsWithFunction[points3D,
 pureFunction,labels,
 GraphicsOptionArgument1Range3D -> xRange,
 GraphicsOptionArgument2Range3D -> yRange,
 GraphicsOptionViewPoint3D -> viewPoint3D,
 GraphicsOptionPlotStyle3D -> plotStyle3D];
Show[functionGraphics3D,arrowGraphics3D]
```



and finally the detected global maximum (point in diagram below) may be visually validated:

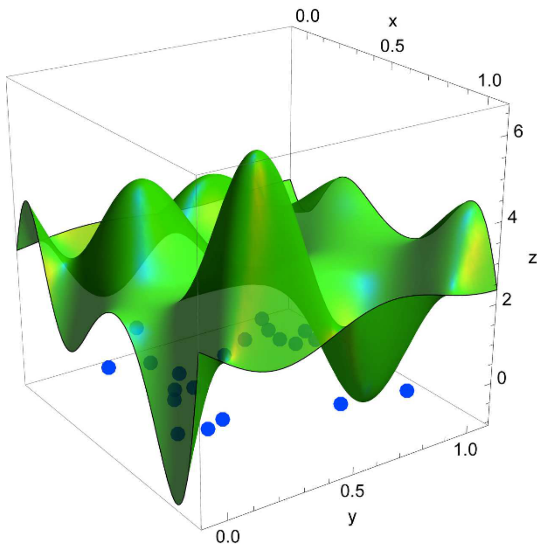
```
xRange={-0.1,1.1};
yRange={-0.1,1.1};
points3D={globalMaximumPoint3D};
CIP`Graphics`Plot3dPointsWithFunction[points3D,pureFunction,labels,
 GraphicsOptionArgument1Range3D -> xRange,
 GraphicsOptionArgument2Range3D -> yRange,
 GraphicsOptionViewPoint3D -> viewPoint3D]
```



Although a grid search seams to be a rational approach to global optimization it is only an acceptable choice for low-dimensional grids, i.e. global optimization problems with only a small number of function arguments as the example above. This is due to the fact that the number of grid points to evaluate explodes (i.e. grows exponentially) with an increasing number of arguments: The number of grid point is equal to N^M with N to be number of grid points per argument and M the number of arguments. For 12 arguments $x_1, x_2, ..., x_{12}$ with only 10 grid points per argument the grid would already contain one trillion $\left(10^{12}\right)$ points so with an increasing number of arguments the necessary function value evaluations at the grid points would become quickly far too slow to be explored in a man's lifetime. As an alternative the number of argument values in the search space to be tested could be confined to a manageable quantity. A rational choice would be randomly selected test points because there is no a priori knowledge about any preferred part of the search space. Note that this random search space exploration would be comparable to a grid search if the number of random test points would equal the number of systematic grid points before (although not looking as tidy). For the current example 20 random test points could be chosen instead of the grid with 100 points:

```
SeedRandom[1];
randomPoints3D=
 Table[
  {RandomReal[{xMinBorderOfSearchSpace,xMaxBorderOfSearchSpace}],
  RandomReal[{yMinBorderOfSearchSpace,yMaxBorderOfSearchSpace}],
  0.0},
  {20}
 ];
CIP`Graphics`Plot3dPointsWithFunction[randomPoints3D,pureFunction,
 labels,
 GraphicsOptionArgument1Range3D -> xRange,
 GraphicsOptionArgument2Range3D -> yRange,
 GraphicsOptionViewPoint3D -> viewPoint3D]
```

The generation of random points can be made deterministic (i.e. always the same sequence of random points is generated) by setting a distinct seed value which is done by the SeedRandom[1] command.



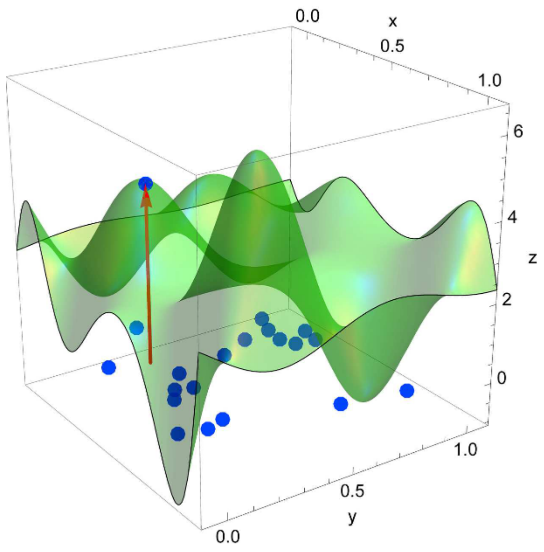
The winner random point is evaluated

```
winnerRandomPoint3D={};
maximumFunctionValue=-Infinity;
Do[
 functionValue=pureFunction[randomPoints3D[[i, 1]],
  randomPoints3D[[i, 2]]];
 If[functionValue>maximumFunctionValue,
  maximumFunctionValue=functionValue;
  winnerRandomPoint3D={randomPoints3D[[i, 1]],randomPoints3D[[i, 2]],
   maximumFunctionValue}
 ],
 {i,Length[randomPoints3D]}
];
```

and visualized (with only the winner random point shown raised to its functions value indicated by the arrow):

```
Do[
 If[randomPoints3D[[i,1]] == winnerRandomPoint3D[[1]] &&
  randomPoints3D[[i,2]] == winnerRandomPoint3D[[2]],
  randomPoints3D[[i]] = winnerRandomPoint3D
 ],
 {i,Length[randomPoints3D]}
];

arrowStartPoint={winnerRandomPoint3D[[1]],winnerRandomPoint3D[[2]],
 0.0};
arrowGraphics3D=Graphics3D[{Thick,Red,{Arrowheads[Medium],
 Arrow[{arrowStartPoint,winnerRandomPoint3D}]}}];
plotStyle3D=Directive[Green,Specularity[White,40],Opacity[0.4]];
functionGraphics3D=CIP`Graphics`Plot3dPointsWithFunction[
 randomPoints3D,pureFunction,labels,
 GraphicsOptionArgument1Range3D -> xRange,
 GraphicsOptionArgument2Range3D -> yRange,
 GraphicsOptionViewPoint3D -> viewPoint3D,
 GraphicsOptionPlotStyle3D -> plotStyle3D];
Show[functionGraphics3D,arrowGraphics3D]
```



But if this global optimization result

```
winnerRandomPoint3D
```

{0.29287,0.208051,4.49892}

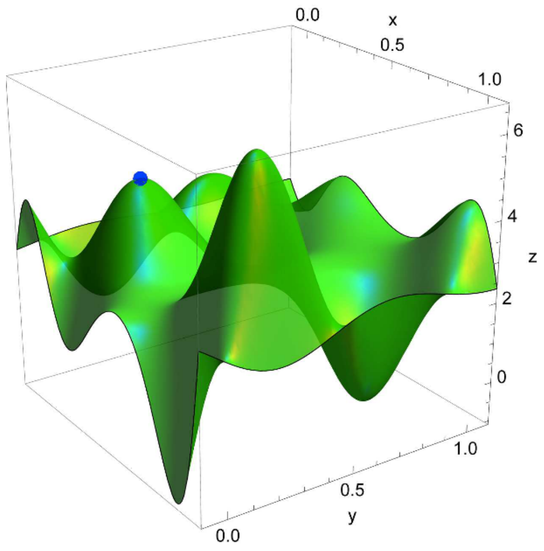
is refined by a post-processing local maximum search starting from the winner random point

```
globalMaximum=FindMaximum[function,
 {{x,winnerRandomPoint3D[[1]]},{y,winnerRandomPoint3D[[2]]}}]
```

$\{4.55146, \{x \to 0.265291, y \to 0.204128\}\}$

only a local maximum is found (point in diagram below) and thus the global maximum is missed:

```
globalMaximumPoint3D={x/.globalMaximum[[2,1]],
 y/.globalMaximum[[2,2]],
 globalMaximum[[1]]};
points3D={globalMaximumPoint3D};
Plot3dPointsWithFunction[points3D,pureFunction,labels,
 GraphicsOptionArgument1Range3D -> xRange,
 GraphicsOptionArgument2Range3D -> yRange,
 GraphicsOptionViewPoint3D -> viewPoint3D]
```



This failure can not be traced to the local optimum search (this worked perfectly from the passed starting position) but must be attributed to an insufficient number of random test points before: If their number is raised the global sampling of the search space would improve and the probability of finding a good test point in the vicinity of the global maximum would increase. But then the same restrictions apply as mentioned with the systematic grid search: With an increasing number of parameters (dimensions) the size of the search space explodes and a random search resembles more and more to be simply looking for a needle in a haystack.

In the face of this desperate situation there was an urgent need for global optimization strategies that are able to tackle difficult search problems in large spaces. As a knight in shining armour a family of so called evolutionary algorithms emerged that rapidly drew a lot of attention. These methods also operate in a basically random manner comparable to a pure random search but in addition they borrow approved refinement strategies from biological evolution to approach the global optimum: These are mutation (random change), crossover or recombination (a kind of random mixing that leads to a directional hopping towards promising search space regions) and selection of the fittest (amplification of the optimal points found so far). The evolution cycles try to speed up the search towards the global optimum by successively composing parts (schemata) of the optimum solution. Mathematica offers an evolutionary-algorithm-based global optimization procedure via the NMinimize and NMaximize commands with the DifferentialEvolution method option (see comments on [NMinimize/NMaximize] for details). The global maximum search

```
globalMaximum=NMaximize[{function,
 {xMinBorderOfSearchSpace<x<xMaxBorderOfSearchSpace,
 yMinBorderOfSearchSpace<y<yMaxBorderOfSearchSpace}},
 {x,y},
 Method -> {"DifferentialEvolution","PostProcess" -> False}]
```

$\{6.54443, \{x \to 0.959215, y \to 0.204128\}\}$

Note the deactivation of the PostProcess in the Method definition: NMaximize automatically applies a local optimization method to refine the result of a global search - the same was done in the grid and random search examples above. The deactivation suppresses this refinement to get the pure result of the evolutionary algorithm.

now directly leads to a result of sufficient precision (compare global maximum location above). But it should be noted that evolutionary algorithms in spite of their popularity belong to the methods of last resort: They may be extremely computationally expensive, i.e. time-consuming. Evolutionary algorithms are regarded to be very effective since they imitate the successful biological evolution. This widespread view neglects the fact that natural evolution needed eons to develop life - and living organisms are by no means optimum solutions. If the evolutionary algorithm is applied to the multiple-optima function already demonstrated above

```
function=1.0-Cos[x]/(1.0+0.01*x^2);
pureFunction=Function[argument,function/.x -> argument];
```

with an appropriate search space (not too small, not too large)

```
xMinBorderOfSearchSpace=-10.0;
xMaxBorderOfSearchSpace=15.0;
```

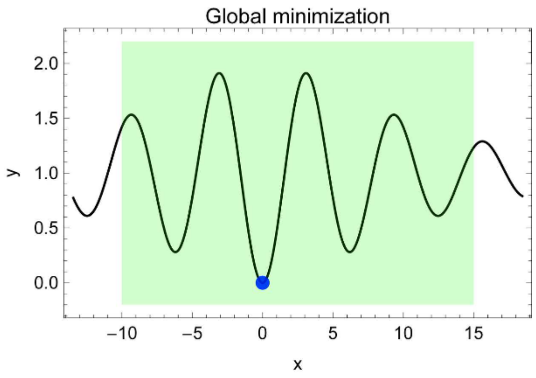
the global minimum (point in diagram below) inside the search space (marked as a background in diagram below)

```
globalMinimum=NMinimize[{function,
 xMinBorderOfSearchSpace<x<xMaxBorderOfSearchSpace},x,
 Method -> {"DifferentialEvolution","PostProcess" -> False}]
```

$\{5.16341 \times 10^{-10}, \{x \to -0.0000318188\}\}$

is also approximated successfully:

```
minimumPoint={x/.globalMinimum[[2]],globalMinimum[[1]]};
points2D={minimumPoint};
argumentRange={-12.0,17.0};
functionValueRange={-0.2,2.2};
labels={"x","y","Global minimization"};
functionGraphics=CIP`Graphics`Plot2dPointsAboveFunction[points2D,
 pureFunction,labels,
 GraphicsOptionArgumentRange2D -> argumentRange,
 GraphicsOptionFunctionValueRange2D -> functionValueRange];
searchSpaceGraphics=Graphics[{RGBColor[0,1,0,0.2],
 Rectangle[{xMinBorderOfSearchSpace,functionValueRange[[1]]},
 {xMaxBorderOfSearchSpace,functionValueRange[[2]]}]}];
Show[functionGraphics,searchSpaceGraphics]
```



But note: If the search space is inadequately chosen (i.e. the global minimum is outside the interval)

```
xMinBorderOfSearchSpace=50.0;
xMaxBorderOfSearchSpace=60.0;
globalMinimum=NMinimize[{function,
 xMinBorderOfSearchSpace<x<xMaxBorderOfSearchSpace},x,
 Method -> {"DifferentialEvolution","PostProcess" -> False}]
```

$\{0.9619, \{x \to 50.2272\}\}$

or the search space is simply to large

```
xMinBorderOfSearchSpace=-100000.0;
xMaxBorderOfSearchSpace=100000.0;
globalMinimum=NMinimize[{function,
 xMinBorderOfSearchSpace<x<xMaxBorderOfSearchSpace},x,
 Method -> {"DifferentialEvolution","PostProcess" -> False}]
```

$\{0.805681, \{x \to 19.2638\}\}$

the global minimum may not be found within the default maximum number of iterations.

1.2.5 Constrained iterative optimization

```
Clear["Global`*"];
<<CIP`Graphics`
```

With the global optimization examples of the previous section the field of constrained optimization was already touched since the a priori defined search space was a constraint of the search (but in fact it was not intended to constrain the optimization procedure: Defining a search space was just a precondition for the global optimization methods to work at all). In general optimization tasks are called unconstrained if they are free from any additional restrictions. If the optimization is subject to one or several constraints the field of constrained optimization is entered. If the function under investigation is not only to be globally minimized but the x value is restricted to lie in an defined interval

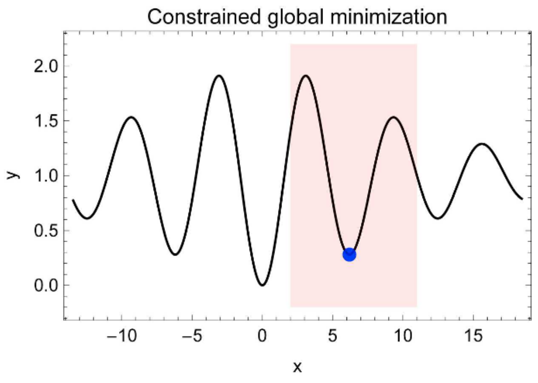
```
function=1.0-Cos[x]/(1.0+0.01*x^2);
pureFunction=Function[argument,function/.x -> argument];
xMinConstraint=2.0;
xMaxConstraint=11.0;
constraint=xMinConstraint<x<xMaxConstraint;
constrainedGlobalMinimum=NMinimize[{function,constraint},x,
 Method -> {"DifferentialEvolution","PostProcess" -> False}]
```

$\{0.28015, \{x \to 6.19386\}\}$

the constrained global minimum (point in diagram below) may differ from the unconstrained one (the constraint is marked as a background in diagram below):

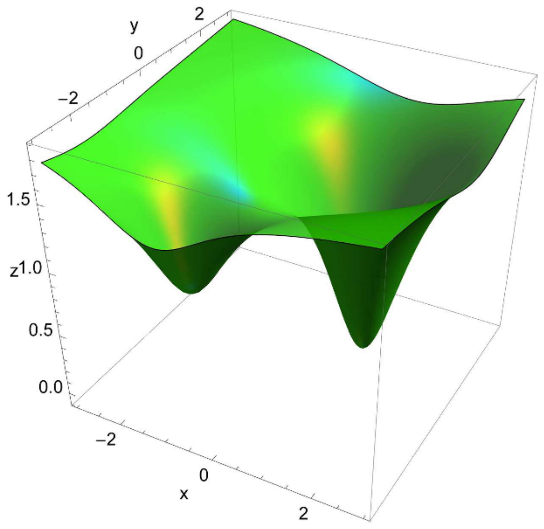
```
constrainedMinimumPoint={x/.constrainedGlobalMinimum[[2]],
 constrainedGlobalMinimum[[1]]};
```

```
points2D={constrainedMinimumPoint};
argumentRange={-12.0,17.0};
functionValueRange={-0.2,2.2};
labels={"x","y","Constrained global minimization"};
functionGraphics=CIP`Graphics`Plot2dPointsAboveFunction[points2D,
 pureFunction,labels,
 GraphicsOptionArgumentRange2D -> argumentRange,
 GraphicsOptionFunctionValueRange2D -> functionValueRange];
constraintGraphics=Graphics[{RGBColor[1,0,0,0.1],
 Rectangle[{xMinConstraint,functionValueRange[[1]]},
 {xMaxConstraint,functionValueRange[[2]]}]}];
Show[functionGraphics,constraintGraphics]
```



But not only may the unconstrained and constrained global optimum differ: The constrained global optimum may in general not be an optimum of the unconstrained optimization problem at all: This can be illustrated with the following example taken from the Mathematica tutorials. The 3D surface

```
function=
 -1.0/((x+1.0)^2+(y+2.0)^2+1)-2.0/((x-1.0)^2+(y-1.0)^2+1)+2.0;
pureFunction=Function[{argument1,argument2},
 function/.{x -> argument1,y -> argument2}];
xRange={-3.0,3.0};
yRange={-3.0,3.0};
labels={"x","y","z"};
CIP`Graphics`Plot3dFunction[pureFunction,xRange,yRange,labels]
```



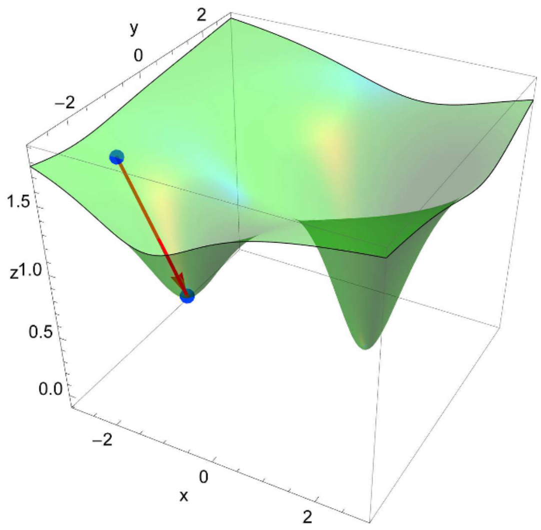
contains two optima: A local and a global minimum. Depending on the start position of the iterative local minimum search method initiated via the FindMinimum command

```
startPosition={-2.5,-1.5};
localMinimum=FindMinimum[function,{{x,startPosition[[1]]},
 {y,startPosition[[2]]}}]
```

$\{0.855748, \{x \to -0.978937, y \to -1.96841\}\}$

the minimization process approximates the local minimum

```
startPoint={startPosition[[1]],startPosition[[2]],
 function/.{x -> startPosition[[1]],y -> startPosition[[2]]}};
minimumPoint={x/.localMinimum[[2,1]],y/.localMinimum[[2,2]],
 localMinimum[[1]]};points3D={startPoint,minimumPoint};
arrowGraphics=Graphics3D[{Thick,Red,{Arrowheads[Medium],
 Arrow[{startPoint,minimumPoint}]}}];
plotStyle3D=Directive[Green,Specularity[White,40],Opacity[0.4]];
functionGraphics=CIP`Graphics`Plot3dPointsWithFunction[points3D,
 pureFunction,labels,
 GraphicsOptionArgument1Range3D -> xRange,
 GraphicsOptionArgument2Range3D -> yRange,
 GraphicsOptionPlotStyle3D -> plotStyle3D];
Show[functionGraphics,arrowGraphics]
```



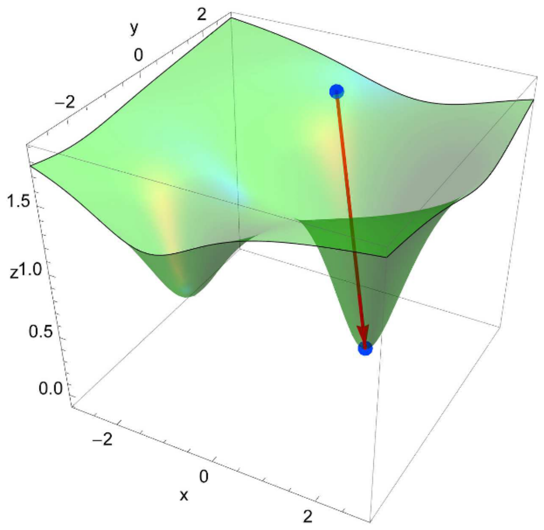
or (with another start point)

```
startPosition={-0.5,2.5};
localMinimum=FindMinimum[function,{{x,startPosition[[1]]},
 {y,startPosition[[2]]}}]
```

$\{-0.071599, \{x \to 0.994861, y \to 0.992292\}\}$

arrives at the global minimum:

```
startPoint={startPosition[[1]],startPosition[[2]],
 function/.{x -> startPosition[[1]],y -> startPosition[[2]]}};
minimumPoint={x/.localMinimum[[2,1]],y/.localMinimum[[2,2]],
 localMinimum[[1]]};points3D={startPoint,minimumPoint};
arrowGraphics=Graphics3D[{Thick,Red,{Arrowheads[Medium],
 Arrow[{startPoint,minimumPoint}]}}];
functionGraphics=CIP`Graphics`Plot3dPointsWithFunction[points3D,
 pureFunction,labels,
 GraphicsOptionArgument1Range3D -> xRange,
 GraphicsOptionArgument2Range3D -> yRange,
 GraphicsOptionPlotStyle3D -> plotStyle3D];
Show[functionGraphics,arrowGraphics]
```



If now the constraint is imposed that

$$x^2 + y^2 > 4.0$$

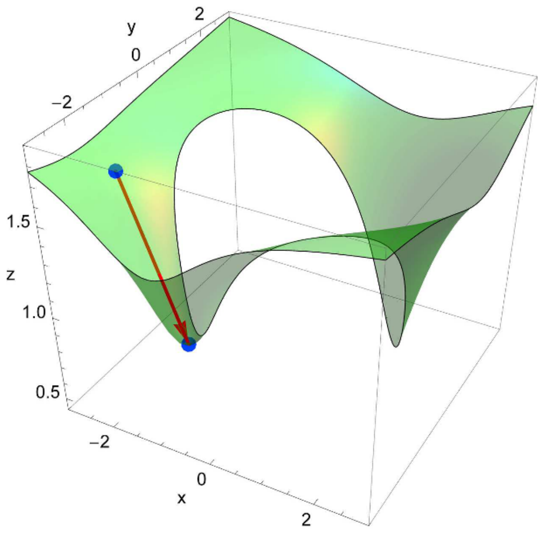
(the constraint removes a circular argument area around the origin (0,0) of the *xy* plane) the constrained local minimization algorithm behind the FindMinimum command is activated (see comments on [FindMinimum/FindMaximum] for details). The constrained local minimization process from the first start position

```
startPosition={-2.5,-1.5};
constraint=x^2+y^2>4.0;
localMinimum=FindMinimum[{function,constraint},
 {{x,startPosition[[1]]},{y,startPosition[[2]]}}]
```

$\{0.855748, \{x \to -0.978937, y \to -1.96841\}\}$

still results in the local minimum of the unconstrained surface

```
startPoint={startPosition[[1]],startPosition[[2]],
 function/.{x -> startPosition[[1]],y -> startPosition[[2]]}};
minimumPoint={x/.localMinimum[[2,1]],y/.localMinimum[[2,2]],
 localMinimum[[1]]};points3D={startPoint,minimumPoint};
regionFunction=Function[{argument1,argument2},
 constraint/.{x -> argument1,y -> argument2}];
arrowGraphics=Graphics3D[{Thick,Red,{Arrowheads[Medium],
 Arrow[{startPoint,minimumPoint}]}}];
functionGraphics=Plot3dPointsWithFunction[points3D,pureFunction,
 labels,
 GraphicsOptionArgument1Range3D -> xRange,
 GraphicsOptionArgument2Range3D -> yRange,
 GraphicsOptionPlotStyle3D -> plotStyle3D,
 GraphicsOptionRegionFunction -> regionFunction];
Show[functionGraphics,arrowGraphics]
```



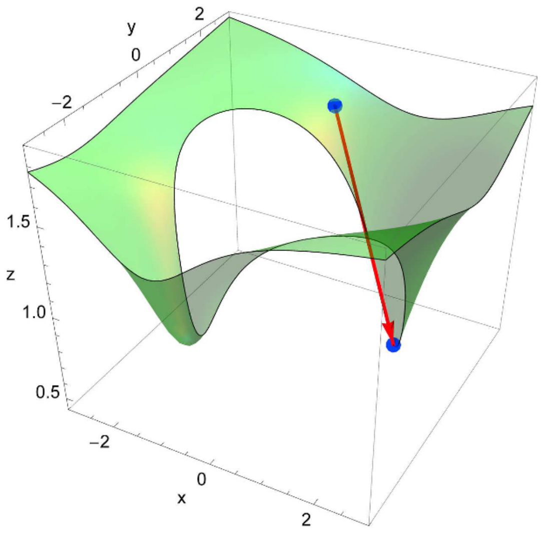
but the second start position

```
startPosition={-0.5,2.5};
localMinimum=FindMinimum[{function,constraint},
 {{x,startPosition[[1]]},{y,startPosition[[2]]}}]
```

$\{0.456856, \{x \to 1.41609, y \to 1.41234\}\}$

leads to a new global minimum since the one of the unconstrained surface is excluded by the constraint:

```
startPoint={startPosition[[1]],startPosition[[2]],
 function/.{x -> startPosition[[1]],y -> startPosition[[2]]}};
minimumPoint={x/.localMinimum[[2,1]],y/.localMinimum[[2,2]],
 localMinimum[[1]]};points3D={startPoint,minimumPoint};
arrowGraphics=Graphics3D[{Thick,Red,{Arrowheads[Medium],
 Arrow[{startPoint,minimumPoint}]}}];
functionGraphics=Plot3dPointsWithFunction[points3D,pureFunction,
 labels,
 GraphicsOptionArgument1Range3D -> xRange,
 GraphicsOptionArgument2Range3D -> yRange,
 GraphicsOptionPlotStyle3D -> plotStyle3D,
 GraphicsOptionRegionFunction -> regionFunction];
Show[functionGraphics,arrowGraphics]
```



An evolutionary-algorithm-based constrained global search in the displayed argument ranges via NMinimize directly approximates the constrained global minimum

```
Off[NMinimize::cvmit]
localMinimum=NMinimize[{function,constraint},
 {{x,xRange[[1]],xRange[[2]]},{y,yRange[[1]],yRange[[2]]}},
 Method -> {"DifferentialEvolution","PostProcess" -> False}]
```

$\{0.456829, \{x \to 1.41637, y \to 1.41203\}\}$

The Off[NMinimize::cvmit] command suppresses an internal message from NMinimize. Internal messages are usually helpful to understand problems and they advise to interpret results with caution. In this particular case the suppression eases readability.

with sufficient precision (compare above).

In general it holds that the more dimensional the non-linear curved hyper surface is and the more constraints are imposed the more difficult it is to approximate a local or even the global optimum with sufficient precision. The specific optimization problems that are related to the road from curve fitting to machine learning will be discussed in the later chapters where they apply.

1.3 Model functions

Since model functions play an important role throughout the book a basic categorization is helpful. A good starting point is the most prominent model function: The straight line.