Mathematics for Engineers IV

Numerics

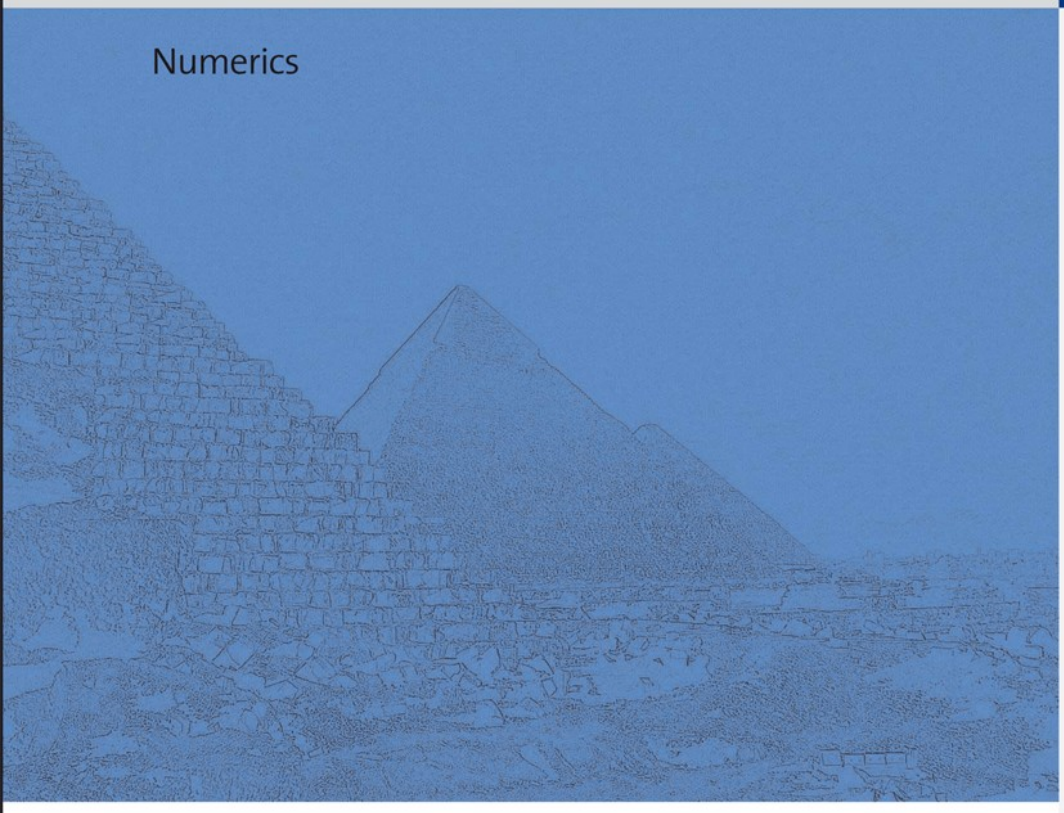
Bearbeitet von
Gerd Baumann

1. Auflage 2009. Buch. XII, 345 S.
ISBN 978 3 486 59042 5
Format (B x L): 17 x 24 cm
Gewicht: 6990 g

Gerd Baumann

Mathematics for Engineers IV

Numerics



With CD-ROM

Oldenbourg

Mathematics for Engineers IV

Numerics

by
Gerd Baumann

Oldenbourg Verlag München

Prof. Dr. Gerd Baumann is head of Mathematics Department at German University Cairo (GUC). Before he was Professor at the Department of Mathematical Physics at the University of Ulm.

Rosenheimer Straße 145, D-81671 München
Telefon: (089) 45051-0
oldenbourg.de

Editor: Kathrin Mönch
Producer: Anna Grosser
Cover design: Kochan & Partner, München
Printed on acid-free and chlorine-free paper
Printing: Druckhaus „Thomas Müntzer“ GmbH, Bad Langensalza

ISBN 978-3-486-59042-5

Preface

Theory without Practice is empty,
Practice without Theory is blind.

The current text *Mathematics for Engineers* is a collection of four volumes covering the first three up to the fifth terms in undergraduate education. The text is mainly written for engineers but might be useful for students of applied mathematics and mathematical physics, too.

Students and lecturers will find more material in the volumes than a traditional lecture will be able to cover. The organization of each of the volumes is done in a systematic way so that students will find an approach to mathematics. Lecturers will select their own material for their needs and purposes to conduct their lecture to students.

For students the volumes are helpful for their studies at home and for their preparation for exams. In addition the books may be also useful for private study and continuing education in mathematics. The large number of examples, applications, and comments should help the students to strengthen their knowledge.

The volumes are organized as follows: Volume I treats basic calculus with differential and integral calculus of single valued functions. We use a systematic approach following a bottom-up strategy to introduce the different terms needed. Volume II covers series and sequences and first order differential equations as a calculus part. The second part of the volume is related to linear algebra. Volume III treats vector calculus and differential equations of higher order. In Volume IV we use the material of the previous volumes in numerical applications; it is related to numerical methods and practical calculations. Each of the volumes is accompanied by a CD containing the *Mathematica* notebooks of the book.

As prerequisites we assume that students had the basic high school education in algebra and geometry. However, the presentation of the material starts with the very elementary subjects like numbers and introduces in a systematic way step by step the concepts for functions. This allows us to repeat most of

the material known from high school in a systematic way, and in a broader frame. This way the reader will be able to use and categorize his knowledge and extend his old frame work to a new one.

The numerous examples from engineering and science stress on the applications in engineering. The idea behind the text concept is summarized in a three step process:

$$\text{Theory} \rightarrow \text{Examples} \rightarrow \text{Applications}$$

When examples are discussed in connection with the theory then it turns out that the theory is not only valid for this specific example but useful for a broader application. In fact, usually theorems or a collection of theorems can even handle whole classes of problems. These classes are sometimes completely separated from this introductory example; e.g. the calculation of areas to motivate integration or the calculation of the power of an engine, the maximal height of a satellite in space, the moment of inertia of a wheel, or the probability of failure of an electronic component. All these problems are solvable by one and the same method, integration.

However, the three-step process is not a feature which is always used. Some times we have to introduce mathematical terms which are used later on to extend our mathematical frame. This means that the text is not organized in a historic sequence of facts as traditional mathematics texts. We introduce definitions, theorems, and corollaries in a way which is useful to create progress in the understanding of relations. This way of organizing the material allows us to use the complete set of volumes as a reference book for further studies.

The present text uses *Mathematica* as a tool to discuss and to solve examples from mathematics. The intention of this book is to demonstrate the usefulness of *Mathematica* in everyday applications and calculations. We will not give a complete description of its syntax but demonstrate by examples the use of its language. In particular, we show how this modern tool is used to solve classical problems and to represent mathematical terms.

We hope that we have created a coherent way of a first approach to mathematics for engineers.

Acknowledgments Since the first version of this text, many students made valuable suggestions. Because the number of responses are numerous, I give my thanks to all who contributed by remarks and enhancements to the text. Concerning the historical pictures used in the text, I acknowledge the support of the http://www-gapdcs.st-and.ac.uk/~history/ webserver of the University of St Andrews, Scotland. The author deeply appreciates the understanding and support of his wife, Carin, and daughter, Andrea, during the preparation of the books.

Cairo

Gerd Baumann

To Carin and Andrea

Contents

1
Outline

1.1. Introduction

We have compiled this material for a sequence of courses on the application of numerical approximation techniques. The text is designed primarily for undergraduate students from engineering who have completed the basic courses of calculus. Familiarity with the fundamentals of linear algebra and differential equations is helpful, but adequate introductory material on these topics is presented in the text so that those courses will be refreshed at the certain point.

Our main objective with this book is to provide an introduction to modern approximation techniques; to explain how, why, and when they can be expected to work; and to provide a firm basis for future study of numerical analysis and scientific computing.

The book contains material for a single term of study, but we expect many readers to use the text not only for a single term course but as a basic reference for their future studies. In such a course, students learn to identify the types of problems that require numerical techniques for their solution and see examples of the error propagation that can occur when numerical methods are applied. They accurately approximate the solutions of problems that cannot be solved symbolically in an exact way and learn techniques for estimating error bounds for the approximations.

Mathematics is for engineers a tool. As for all other engineering applications working with tools you must know how they act and react in applications. The same is true for mathematics. If you know how a mathematical procedure (tool) works and how the components of this tool are connected by each other you will understand its application. Mathematical tools consist as engineering tools of components. Each component is usually divisible into other components until the basic components (elements) are found. The same idea is used in mathematics there are basic elements you should know as an engineer. Combining these basic elements we are able to set up a mathematical frame which

incorporates all those elements which are needed to solve a problem. In other words, we use always basic ideas to derive advanced structures. All mathematical thinking follows a simple track which tries to apply fundamental ideas used to handle more complicated situations. If you remember this simple concept you will be able to understand advanced concepts in mathematics as well as in engineering.

1.2. Concept of the Text

Every concept in the text is illustrated by examples, and we included more than 1,000 tested exercises for assignments, class work and home work ranging from elementary applications of methods and algorithms to generalizations and extensions of the theory. In addition, we included many applied problems from diverse areas of engineering. The applications chosen demonstrate concisely how numerical methods can be, and often must be, applied in real life situations.

During the last 25 years a number of symbolic software packages have been developed to provide symbolic mathematical computations on a computer. The standard packages widely used in academic applications are *Mathematica*®, Maple® and Derive®. The last one is a package which is used for basic calculus while the two other programs are able to handle high sophisticated calculations. Both *Mathematica* and Maple have nearly the same mathematical functionality and are very useful in symbolic and numeric calculations. This is a major advantage if we develop algorithms and implement them to generate numerical results. This approach is quite different from the traditional approach where algorithms are developed by pencil and paper and afterwards implemented in packages like MATLAB or in a programming language like FORTRAN or C. Our approach in using *Mathematica* is a combination of symbolic and numeric approach, a so called hybrid approach, allowing the design and implementation in the same environment. However the author's preference is *Mathematica* because the experience over the last 25 years showed that *Mathematica*'s concepts are more stable than Maple's one. The author used both of the programs and it turned out during the years that programs written in *Mathematica* 25 years ago still work with the latest version of *Mathematica* but not with Maple. Therefore the book and its calculations are based on a package which is sustainable for the future.

Having a symbolic computer algebra program available can be very useful in the study of approximation techniques. The results in most of our examples and exercises have been generated using problems for which exact values can be determined, since this permits the performance of the approximation method to be monitored. Exact solutions can often be obtained quite easily using symbolic computation. In addition, for many numerical techniques the error analysis requires bounding a higher ordinary or partial derivative of a function, which can be a tedious task and one that is not particularly instructive once the techniques of calculus have been mastered. Derivatives can be quickly obtained symbolically, and a little insight often permits a symbolic computation to aid in the bounding process as well.

We have chosen *Mathematica* as our standard package because of its wide distribution and reliability. Examples and exercises have been added whenever we felt that a computer algebra system would be of

significant benefit, and we have discussed the approximation methods that *Mathematica* employs when it is unable to solve a problem exactly.

1.3. Organization of the Text

The book is organized in chapters which will cover the first steps in *Numerical Methods* demonstrating when numeric is useful and how it can be applied to specific problems in simulation. We go on with the representation of numbers and the finite representation of numbers on a computer. Having introduced numbers and errors for a finite number representation we discuss how we can represent functions and their approximation by interpolation. Numerical interpolation is the basis of different other approaches in numeric such as integration and solution of differential equations. Related to the representation of functions are the methods of root finding for a function of a single or several variables. After the discussion of the procedures for numerical root finding we examine linear systems of equations. In this field the classical solution approaches are discussed as well as iterative methods to solve linear equations. The final topic is related to the solution of initial and boundary value problems. We will also examine the finite element method applied to ordinary differential equations.

The whole material is organized in seven chapters where the first part of this chapter is the current introduction. In Chapter 2 we will deal with object oriented concepts of simulation; it is an introductory chapter to demonstrate where and how numeric methods is applied. In Chapter 3, we deal with concepts of numbers and the representation of numbers in finite formats. We also discuss the consequences of a finite representation and discuss the basic errors occurring in such representation. In Chapter 4 we use different approaches to approximate functions by polynomials. We also discuss how polynomials are represented in a numerically stable way. Chapter 5 discusses non polynomial functions and their roots. Different approaches are discussed to find roots of a single valued function. The classic approaches like bisection, Regula Falsi, and Newton's method are discussed. In addition, we use Banach's fixed point theorem to show that for a certain class of functions this approach is quite efficient. Chapter 6 deals with linear systems of equations. We discuss the methods by Gauß and give estimations on the number of operations needed to carry out this approach. For large systems of equations we introduce iterative solution procedures allowing us to treat also sparse equations. An important point in engineering are eigenvalues and eigenvectors, discussed in connection with the power method approach. Finally, in Chapter 7, we discuss the solution of ordinary differential equations. For ordinary differential equations the standard solver like Euler, Taylor, and Runge-Kutta's method are discussed. In addition to the solution of initial value problems, we discuss also the solution of boundary value problems especially two point boundary methods. The approaches used to solve boundary problems are shooting and optimization methods and finite differences and collocation methods. The finite element method and its basic principles are described for ordinary differential equations.

1.4. Presentation of the Material

Throughout the book I will use the traditional presentation of mathematical terms using symbols, formulas, definitions, theorems, etc. to set up the working frame. This representation is the classical mathematical part. In addition to these traditional presentation tools we will use *Mathematica* as a symbolic, numeric, and graphic tool. *Mathematica* is a computer algebra system allowing us to carry out hybrid calculations. This means calculations on a computer are either symbolic or/and numeric. *Mathematica* is a calculation tool allowing us to do automatic calculations on a computer. Before you use such kind of tool it is important to understand the mathematical concepts. The use of *Mathematica* allows you to minimize the calculations but you should be aware that you will only understand the concepts if you do your own calculations by pencil and paper. Once you have understood the way how to avoid errors in calculations and in concepts you are ready to use the symbolic calculations offered by *Mathematica*. It is important for your understanding that you make errors and derive an improved understanding from these errors. You will never reach a higher level of understanding if you apply the functionality of *Mathematica* as a black box solver of your problems. Therefore I recommend to you first try to understand by using pencil and paper calculations and then switch to the symbolic algebra system if you have understood the concepts.

You can get a trial version of *Mathematica* directly from Wolfram Research by requesting a download address from where you can download the trial version of *Mathematica*. The corresponding web address to get *Mathematica* for free is:

http://www.wolfram.com/products/mathematica/experience/request.cgi

2
Simulation Methods

2.1 Introduction

Today we are in a situation which allows us to set up different kind of calculations ranging from simple algebraic calculations up to the complicated integration procedure for nonlinear partial differential equations. Since the 40ties of the last century there started a development of computers and software which supports the ideas of numerical calculations in an ever increasing way in efficiency. The new approach to mathematics by means of numerical calculations was dramatically changed and pushed in one direction by inventing and developing personal computers (PCs). PCs are now used by everybody who has to calculate or to simulate some kind of models. The PC somehow revolutionized the way how calculations are carried out today.

In the past, numerical calculations were not the tool of the best choice but a burden for those which had to carry out such kind of calculations. For example, the calculations related to the now famous Kepler laws took more than 10 years of laborious calculations by Kepler himself. These calculations examining the paths of planets actually were numerical calculations done by pencil and paper. Later on, mechanical calculators and recently pocket calculators replaced ruler based calculators in engineering which was somehow a great step forward to make numerical calculations more efficient. Today, we use pocket calculators and computers to carry out this kind of calculations. However, numerical calculations are necessary to boil down a theory to a number; the theory itself can today be generated symbolically on computers.

Since the beginning of mathematics there were always two branches of calculations; a symbolic and a numeric one. The symbolic calculations were used to establish the theoretical framework of mathematics while numerical calculations were used for specific practical applications. Up to now, these two branches remained in mathematics as a symbiosis of two ways to consistently formulate theoretical ideas and to derive practical information from it. Today, some of the mathematicians and

engineers make a sharp distinction between numeric and symbolic and even some practically oriented engineers believe that numeric is the only way how calculations can be carried out today. This point of view ignores that today we have strong symbolic programs available which also run on computers and generate much more useful information than just a single number. The symbolic approach generates general formulas which allows us to describe the system or model under general conditions for the parameters. The symbolic approach to calculations is thus more useful for the theoretical work. It allows for example to formulate the theoretical background needed in numerical calculations. Another advantage of symbolic calculations is that formulas can be generated on a computer which are the basis of numerical calculations. This brings us to the point where we have to distinguish between the formulation of a model or equation and the evaluation of a model or equation.

The formulation of a mathematical expression is solely related to the symbolic creation of the expression. The evaluation of an expression is related to the numerical evaluation and derivation of a number. To derive numbers from a symbolic expression, we have to specify the values of each symbol in the expression and use the algebraic rules we agreed on. The evaluation of an expression assumes that we know the symbols of this expressions and the numerical values for these symbols. On the other hand, if we know the symbolic representation of the expression then we usually distinguish between important symbols which can change their values and unimportant symbols which keep their values. This kind of classification allows us to introduce so called major variables and parameters in an expression, respectively. The major variables or model variables are those variables which can be changed continuously in the model. The parameters of the expression or model are those quantities which remain constant for the time of evaluation. Thus a model or an expression consists of two kinds of symbols: parameters and model variables. Model variables determine the basic structure of the model while parameters represent the influence of different quantities on the model. The basic definition of a model can be formulated as follows:

Definition 2.1. *Model and Parameters*

A mathematical model is symbolically represented by the methods and the parameters. Methods define the structure of the model and thus the calculation while parameters define the influences onto the model. ■

This kind of definition is used since the beginning of mathematics and becomes nowadays a practical application in software engineering. The link between the definition given and the application in generating calculation or simulation programs is the style of programming.

Since the beginning of numerical calculations it was the case that between the tool (machine, pocket calculator, PC) there was always a link between the symbolic formulation and the numerical result. This link is known today as programming language allowing us to represent the original symbolic calculations in different steps appropriate for the calculation tool used to carry out the calculation. For mechanical calculations this programming language consists of how the steps of the calculations are applied to the machine while for computers the steps are collected in programs. During the past decades, different programming styles were developed such as sequential, functional, and object oriented programming. It turned out that the last programming style is most appropriate to establish a

one-to-one correspondence between the mathematical model and the programming style on a computer. This correspondence between the program and the theoretical background does not dramatically influence the numerical evaluation but helps to avoid difficult formulations.

The following sections will introduce the ideas used in simulation and the methods to generate software from these ideas.

2.2 Simulation

Simulation is the subject in engineering which generates the link between the reality and the theoretical description (model) of a situation. Simulation uses numerical and symbolic procedures to generate results which are related to the questions under discussion.

2.2.1 Structure of Models

As discussed before, a model consists of two components. These two components are: the methods on which the model is based on and the parameters determining the properties of the model. The methods are the laws governing the model while the parameters are the interface of the method to the environment. Thus the parameters can be seen as the input quantities which determine the behavior of the model. The methods or laws of the model determine the structure of the model. This means that the methods determine the output or results in a specific way. These results are influenced by the parameters. The following diagram shows the schematic structure of a model.

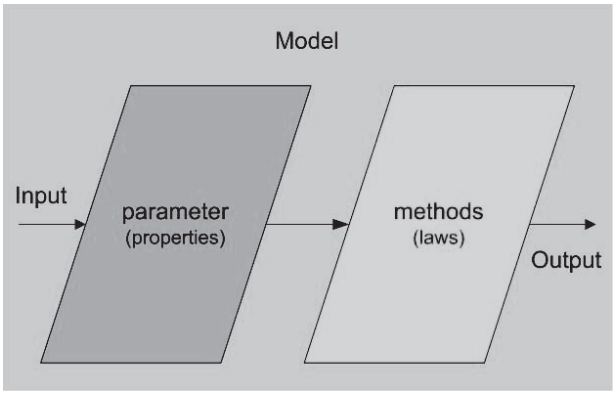


Figure 2.1. Classification of a model by its parameters and methods.

To clarify the structure of a model let us discuss the most prominent model in engineering and physics. This model is Newton's third law which verbally is stated as to every action there is an equal and opposite reaction.

Example 2.1. Newton's Law

Newton's law was established by Sir Isak Newton in 1660s. He uses this model to describe his observations regarding the action and reaction of forces. Let us assume we are looking on a ball which is falling from a tower of height H. We know from Newton's second law that this fall can be described by the formula

$$m\,a = F \tag{2.1}$$

where m is the mass of the ball, a the acceleration which the particle experiences, and F the force acting on the ball. The model here is a simple relation connecting physical (model) quantities like acceleration a and force F with model parameters like mass m. The distinction between parameters and model variables here is very simple because the only property the particle has is its mass. The model is generated by Newton's law combining acceleration with force. For the falling particle we know that the acting force is the gravitation force of earth. In this case $F = -m\,g$ where g is the gravitational acceleration which in fact is also a constant. The model thus simplifies to

$$m\,a = -m\,g \tag{2.2}$$

or

$$a = -g. \tag{2.3}$$

Here the model reduces to a simple algebraic equation where the left hand side is the model variable and the right hand side represents a model parameter. Thus model parameters and model variables are the same. This situation changes if we assume that acceleration is a quantity which changes with time.▲

The first example was used to introduce the notation and terminology in modelling and simulation processes. However, the final relation was a very simple algebraic relation which does not need much mathematical efforts to determine the precise meaning of this relation and to evaluate the relation. Since the value of the terrestrial acceleration is known as $g = 9.81\,m/s^2$, there is no need to use numerical procedure to find a reliable value for a. However the question arises how accurate the value for g can be found in measurements or tables. The accuracy and precision will be discussed in a later section where we will define these terms.

To see how models are created and how models are evaluated we will discuss here the same example with a different aim. Our goal is to introduce the time dependence of the model and to demonstrate how this time dependence changes the reliability of the results.

Example 2.2. Newton's Law as a time-dependent relation

If we assume that acceleration is defined due to the temporal changes of the velocity then we can replace acceleration in Newton's equation by the derivative of the velocity; i.e. $a = \mathrm{d}v/\mathrm{d}t$. Thus Newton's law for a falling ball can be written as

$$m\,a = -m\,g \tag{2.4}$$

which is equivalent to

$$m \frac{\mathrm{dv}}{\mathrm{dt}} = -m\,g \tag{2.5}$$

or

$$\frac{\mathrm{dv}}{\mathrm{dt}} = -g \tag{2.6}$$

This is a simple first order differential equation for the velocity with constant right hand side. To solve this equation we can use either numerical or symbolic methods to derive a solution. Let us first use our mathematical knowledge to solve this equation. The equation is a separate equation which can be written in separated variables as

$$\mathrm{dv} = -g\,\mathrm{dt} \tag{2.7}$$

Integrating this equation results to

$$\int \mathrm{dv} = -g \int \mathrm{dt} \tag{2.8}$$

which will deliver the solution as a function of time

$$v(t) = -g\,t + C \tag{2.9}$$

where C is an integration constant. The result gained is a linear function in time t. The parameters of the model are here g the terrestrial acceleration g and some integration constant C which is determined by the initial velocity of the ball. Let us assume that the initial velocity is zero and thus the constant C vanishes. This assumption will simplify the result to

$$v(t) = -g\,t \tag{2.10}$$

which is an exact result for the solution.

On the other hand we can solve this first order ordinary differential equation by means of numerical methods. This means that we approximate the continuous description by a discrete one; i.e.

$$\frac{\mathrm{dv}}{\mathrm{dt}} \approx \frac{\Delta \mathrm{v}}{\Delta \mathrm{t}} = \frac{v(t_{i+1}) - v(t_i)}{t_{i+1} - t_i} = -g \tag{2.11}$$

which is equivalent to

$$v(t_{i+1}) = -g(t_{i+1} - t_i) + v(t_i) \tag{2.12}$$

This equation represents a formula which allows us to evaluate the velocity at a later time (t_{i+1}) using the information we have for the velocity from previous times (t_i). This formula can be used to iteratively solve the velocities. Any iteration procedure starts with an initial value at the starting time t_0. Let us assume that the starting point for our iteration is $t_0 = 0$. At the starting time the velocity is zero so that we start with $v(t_0) = 0$. The next we have to choose is the time step we are interested in. Let's assume that we are interested in an interval of one second; i.e. the time step spans 1 second

away from the initial one. So $t_1 = 1$ and thus $t_1 - t_0 = 1$. If we use the value for the terrestrial acceleration with $g = 9.81$ we will gain the velocity at $t_1 = 1$ by

```
v1 = 0 – 9.81 (1 + 0)
```

-9.81

If we continue the iteration for $t_2 = 2$ seconds, we iterate the formula by using the above formulas which delivers

```
v2 = v1 – 9.81 (2 – 1)
```

-19.62

The next iteration is gained by the step

```
v3 = v2 – 9.81 (3 – 2)
```

-29.43

The next step is to automatically evaluate this kind of formula up to a certain time. This iteration can be carried out by a program which can be used by a computer. Such kind of program is contained in the next line.

```
vi = 0;
tv = Table[{i + 1, vi = vi – 9.81 ((i + 1) – i)}, {i, 0, 10}];
tv = Prepend[tv, {0, 0}]
```

$$\begin{pmatrix} 0 & 0 \\ 1 & -9.81 \\ 2 & -19.62 \\ 3 & -29.43 \\ 4 & -39.24 \\ 5 & -49.05 \\ 6 & -58.86 \\ 7 & -68.67 \\ 8 & -78.48 \\ 9 & -88.29 \\ 10 & -98.1 \\ 11 & -107.91 \end{pmatrix}$$

The results can be graphically represented in a plot, demonstrating the linear behavior of the solution.

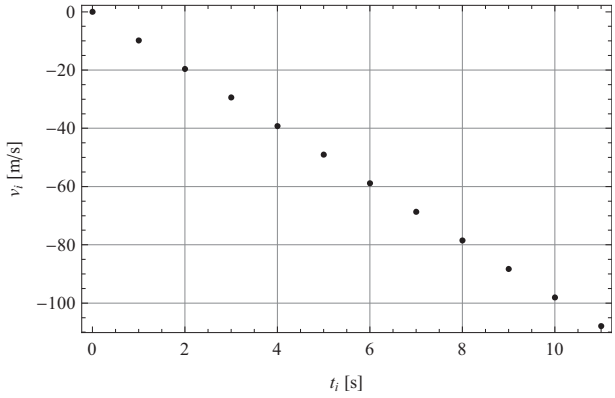


Figure 2.2. Simple numerical solution for a motion using fixed time steps Δt.

The theoretical result can also be used in a graph which is shown in the next plot:

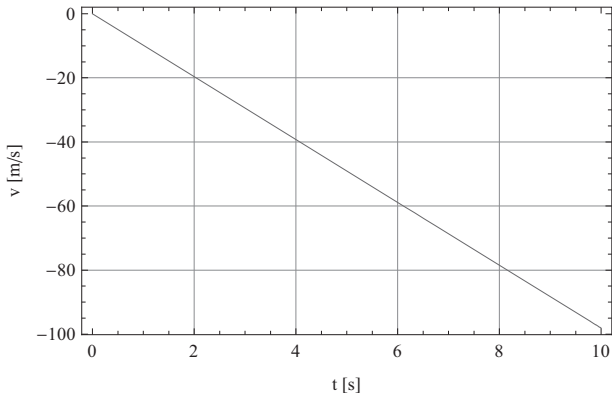


Figure 2.3. Exact solution for the motion using the analytic expression, is given by $v(t) = -g\,t$.

If we now plot both the theoretical and numerical result in common we see that the two solutions derived correspond. The following plot shows this result.

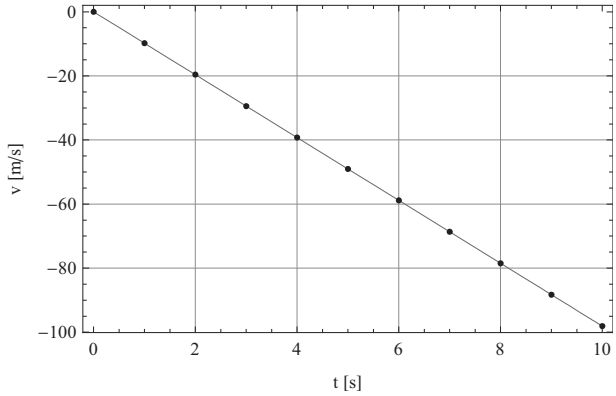


Figure 2.4. Comparison of numerical and exact solution of the motion.

However, this picture changes if we change the time step $\Delta t = (t_{i+1} - t_i)$ by increasing the difference by a factor 2. If we generate the date again and plot the results as before

```
vi = 0;
tv = Table[{i + 2, vi = vi − 9.81 ((i + 1) − i)}, {i, 0, 10, 2}];
tv = Prepend[tv, {0, 0}]
```

$$\begin{pmatrix} 0 & 0 \\ 2 & -9.81 \\ 4 & -19.62 \\ 6 & -29.43 \\ 8 & -39.24 \\ 10 & -49.05 \\ 12 & -58.86 \end{pmatrix}$$

we observe that the slope of the line changes.

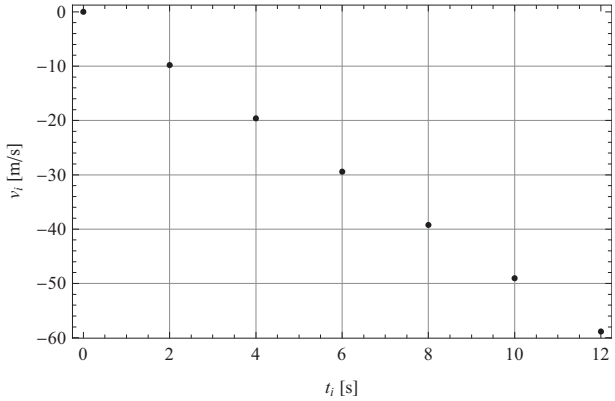


Figure 2.5. Simple numerical solution for a motion using an increased fixed time step Δt.

This becomes obvious when we plot the numerical result and the theoretical result in the same plot.

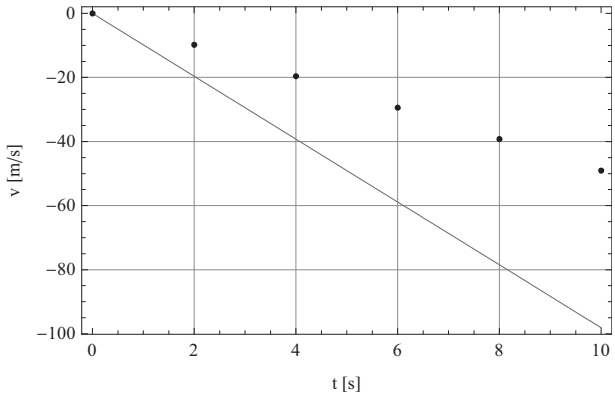


Figure 2.6. Numerical and theoretical solution in a common plot. The time step Δt was increase by a factor 2.

The plot above shows one common characteristic of all numeric solutions generated by an iterative process. If the time step in the iteration is chosen the wrong way then the long term behavior of the numerical solutions deviates dramatically from the theoretical one. This phenomenon is always true for numerical calculations. Thus, the question raises how to guarantee that a numerical solution represents the correct figures of the calculation. This question is the central question in all numerical calculations.▲

These observations led us to the following question. Which requirements in a simulation are necessary to guarantee that the numerical results represent the true results? At the moment, we do not know how to

define a true numerical representation and how to make sure that a related program generates the correct results.

2.2.2 Requirements on a Modeling Process

Models and especially programs should satisfy some requirements to guarantee a quality standard for the calculations. Most of the programs related to models are currently written in a way to just solve a single and specific task. This means that programs are written in a way that they are only useful for a specific application. However, the development and testing of a program and thus a model needs a certain amount of time till the program is working and another period to make sure that the results are reliable. This observation on the development process consequently requires that a model respectively a program should be reusable. This means that the model or components of a model should be designed in such a way that in another setup these parts can be used again. Since the development of models are time and cost intensive a model should be scalable. This means that the development costs are gained not only once but several times. So if the model is scalable and the related program is useful for others too, then the return of investment should be a multiple of the development costs. To achieve this economic goal it is necessary to design the model in such a way that it can be easily used. This means that beside a complete documentation of the model there should be an easy way to use and apply the model. Usually, models need some fundamental theoretical description which are necessary for the development of the model but not for the application. However, the user should know what is the frame of the theoretical assumptions and where are the limits of these assumptions. Only if we know the limits and side conditions of a model we can make a judgment on the efficiency and reliability of an application. These few ideas concerning the general design process of a model and thus a program are summarized in the following list.

The requirements on programs in practical applications are:

1. a program has to deliver reliable results
2. a program should be reusable
3. an application must be scalable
4. the use of the software should be as easy as possible
5. a program should be accompanied by a complete documentation.

So far we discussed the basic requirements for a modelling process but did not describe how a model is generated and which steps are necessary to setup a model for symbolic or numeric calculations. The following section will briefly introduce the way how a model is generally developed and which steps are necessary to derive a model.

2.2.3 Modelling Process

The creation of a model is usually a process which consists of different steps. These steps are basically related to the original physical system which exists in reality. The modeling process consists typically of an abstraction of the complicated real word incorporating only the most important features to describe the system. The abstraction is governed by efficiency and necessary arguments guaranteeing the basic properties of the real system. Thus, the abstraction is somehow a short hand description of the reality taking into account only the essential features of the system, compare Figure 2.7. If such an abstraction is derived from the reality it is necessary to create a mathematical formulation. The mathematical formulation usually consists of equations, differential equations, and boundary as well as initial conditions. The mathematical model again uses some abstract ways to describe the real system. This means that depending on the mathematical description the model can be simple or very complicated depending on the representation of the model. If the mathematical description is available, complicated or not the last step which must be carried out is to solve this model. At this point, numeric and numerical mathematics comes into play. The solution step of a model usually cannot be carried out in a pure symbolic way. Only special models or parts of a model are generally accessible by symbolic solution methods. Thus the standard approach to solve a mathematical model is numeric. However, numeric delivers a solution only under certain boundary conditions. This means that a numeric solution depends on the choice of some numeric parameters determining the solution strategy for a model. Thus, the solution derived is only valid under the validity of the numerical procedure. Our goal here is to clarify which of the assumptions are critical (if we apply numerical solution procedures) and which are not.

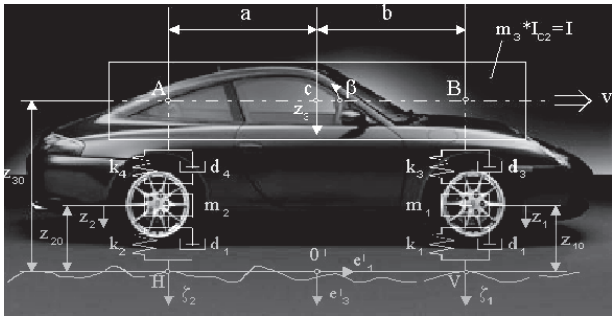


Figure 2.7. Simplified model for a car using springs and dampers in a simple mechanical model allowing elongations and rotations.

To summarize the steps used in a modelling process, we collected the different steps in the following list.

Three step process:

1. the physical model (hardware, reality)
2. the mathematical formulation, and
3. the solution of the problem.

The modeling process divides the real system into components and the components are divided into properties and methods to mathematically describe the system. The division of the real system into parts should be carried out in such a way that physical components are related to mathematical models or methods. This means that we look right from the beginning at parts and laws governing these components which combined generate the total system.

This physical ideas have counterparts not only in the mathematical description but also in the programming style.The different programming styles also influence the efficiency of the calculations and especially the design of the model. In general there are the following ways to generate a program:

- procedural
- recursive
- functional
- object oriented

These different types of program development allow to generate different kinds of programs which may be usable only once or they are reusable if programmed in an object-oriented way. The older programing approaches based on procedural programming languages like FORTRAN, MATLAB, C, etc. allow to generate programs in a direct way. However, the generated program using subprograms have a sequential structure. This means that there exists a chain of dependencies on the different parts of the program which makes it difficult to reuse parts of it. Recursive and functional programming generates also a dependent structure which cannot be efficiently divided in independent parts. The object oriented programming style allows to extract parts of the programs in such a way that these components can be reused in other programs. In addition, the object-oriented programming style is directly related to the hardware components, because the physical components are directly related to the classes and objects of the programs. How this works out will be discussed in more detail in the following section.

2.2.4 Object Oriented Approach Using Classes

Object oriented programming has the advantage that classes consisting of properties and methods are directly related to the physical objects of the system. The methods are related to the mathematical model while the properties are the model parameters determining the systems property. An object-oriented approach in programming assumes that the total system can be divided into components which are described by specific formulas and parameters. Formulas correspond to methods and parameters correspond to properties of the class concept. The concept of a class is shown in Figure 2.8.

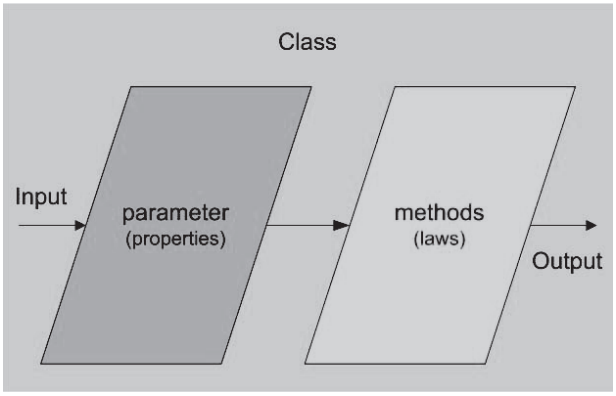


Figure 2.8. Representation of a class by parameters and methods.

A class basically consists of input values and output quantities. The input and output are related by methods and parameters which interact with each other. The methods are laws or formulas used in the calculations. The parameters are quantities which influence the formulas with their values. This influence is not only static but can be used in a dynamic way. The methods and parameters are used to derive an object from this class which is an entity determined by the choice of the parameters. Different parameter sets will define different objects. For each object the same methods are used to represent the laws determining the mathematical and physical behavior. The dynamic use of parameters allows us to connect different classes or different objects. Generating different objects allows us to generate a tool box and thus generate different models.

The properties of a class can be summarized in two topics:

- classes depend on parameters determining the properties of the system
- classes are based on calculation methods

There are different approaches to generate classes. The most recent way to generate programs based on classes is JAVA. Other programming languages are Simula, Smalltalk, C++, eclipse, ... An environment in *Mathematica* was designed by myself to have this kind of programing style not only available for numerical calculations but also for symbolic ones.

The package within *Mathematica* allowing this style of programming is called *Elements*. *Elements* uses elementary data structures to represent a class. The most elementary structure in *Mathematica* is a list. Thus classes in *Elements* consist of lists containing the parameters and methods in different sublists. The classes are defined by the function Class[]. This function uses four arguments which define the class name, references to other classes, a list of properties, and a list of methods. The class name defines the name of the class by a string. The second argument is used to refer to other classes. If the class is based on the basic elements in *Elements* then the class Class["*Elements*"] should appear in the second slot. If the class we are going to define is based on another class than *Elements,* then this class name should occur at the second slot of the function. The third and fourth slot of the class function is

used to define properties and methods used in this class. The following example demonstrates how Newton's equation can be represented in this class concept.

Example 2.3. Class Concept

Lets assume we need Newton's equation in a general form in our models.

Solution 2.3. Newton's equation is given in its general form by

$$m \frac{d^2 x(t)}{dt^2} = F \tag{2.13}$$

In words: mass multiplied by the second derivative of the dependent variable with respect to the independent variable equals the acting force. These terms can be directly implemented in a class. The following lines show how this works.

```
newton = Class[ "Newton", Class["Element"],
  (* --- parameter section --- *)
  {description = "Classe Newton defining Newton's equation",
   {Mass = 10, Description → "Mass of the body", Dimension -> kg},
   {Force = 0, Description → "Force acting on the body",
    Dimension -> kg m / s^2},
   {dependentVariable = x, Description →
     "dependent variable used in Newton's equation", Dimension -> m},
   {independentVariable = t, Description →
     "independent variable used in Newton's equation", Dimension -> s}},
  (* --- method section --- *)
   {
   {equationOfMotion[] := Block[{},
      Mass ∂independentVariable,independentVariable dependentVariable[
           independentVariable] == Force],
    Description → "equation of motion"},
   {(*--- Coordinates --- *)
    getVariables[] := {{dependentVariable}, independentVariable},
    Description → "get dependent and independent variables"}

  } ]
<Class Newton>
```

The class Newton allows to define the parameters mass m, force F, the dependent and independent variable x and t, respectively. The method used is the equation of motion defined in (2.13). Choosing the parameters of the class in different ways we are able to generate different types of objects. The different objects represent different models and thus we can generate a large variety of equations. The objects are generated by using the functionality of *Elements* which is contained in an object generator ◦ and represented by the function new[]. These functions allow us to define objects and

assign them to variables. The following line is an example for this procedure using the predefined values for the parameters.

```
newton001 = newton∘new[]
```

```
<Object of Newton>
```

In *Elements* there are functions available which allow us to derive the properties of an object. The following line demonstrates this

```
GetProperties[newton001]
```

```
{{dependentVariable, x},
 {description, Classe Newton defining Newton's equation},
 {Force, 0}, {independentVariable, t}, {Mass, 10}}
```

Another function of *Elements* is SetProperties allowing us to change or set properties of an object

```
SetProperties[newton001, {Mass -> 100}]
```

To check the new values of the object we use again GetProperties

```
GetProperties[newton001]
```

```
{{dependentVariable, x},
 {description, Classe Newton defining Newton's equation},
 {Force, 0}, {independentVariable, t}, {Mass, 100}}
```

The attributes attached to the properties can be derived by using the function GetAttributes[]

```
GetAttributes[newton001, Force]
```

$$\left\{\text{Description} \to \text{Force of the body, Dimension} \to \frac{\text{kg m}}{\text{s}^2}\right\}$$

Attributes of the methods are derivable the same way, demonstrated in the next line

```
GetAttributes[newton001, equationOfMotion]
```

```
{Description → equation of motion}
```

The methods are used by using the object and the function defined in the method's section combined with the object operator ∘. The following line shows the derivation of the equation of motion

```
newton001∘equationOfMotion[]
```

```
100 x''[t] == 0
```

Having the class Newton available we are now able to define different models by changing the properties of an object. Let us assume that a particle encounters a periodic external force with amplitude f_0 and driving frequency ω. So we define a new object incorporating these properties of the force

```
newton002 = newton∘new[{Mass -> m, Force -> f0 Sin[ω t]}]
```

```
<Object of Newton>
```

A second model uses a harmonic force due to a spring and in addition instead of the sine function a cosine function to represent the external force with amplitude A_0 and driving frequency ω. The external force is here given by $A_0 \cos(\omega t)$

```
newton003 = newton∘new[
    {unabhaengigeVariable -> s, Mass -> 20, Force -> -k x[s] + A0 Cos[ω s]}]
```

```
<Object of Newton>
```

The two different equations of motion are now available by

```
newton002∘equationOfMotion[]
```

```
m x''[t] == f0 Sin[t ω]
```

and

```
newton003∘equationOfMotion[]
```

```
20 x''[t] == A0 Cos[s ω] - k x[s]
```

The use of classes and objects allows us to derive a huge variety of models by changing the properties of an object. This variation is not possible by using standard programming approaches like in procedural programs. The derivation of equations is one of the important steps in modelling a system. Also important is the last step of a simulation process which is related to the solution of the equations.

To get the methods of a class *Elements* provides another function which allows to identify the methods in a class. This function is GetMethodsOfClass[className]. The following line demonstrates the application of this function.

```
GetMethodsOfClass[newton]
```

```
{equationOfMotion, getVariables}
```

Analogously, the properties of a class can be identified by using the function Properties[className]. For our example this is done by

```
Properties[newton]
```

```
{dependentVariable, description, Force, independentVariable, Mass}
```

The discussed example demonstrates that an object oriented approach to formulate a model is a natural approach and allows to setup different models by using a single class.▲

2.2.5 Tests and Exercises

The following two subsections serve to test your understanding of the last section. Work first on the test examples and then try to solve the exercises.

2.2.5.1 Test Problems

T1. What is a model? Give examples.

T2. What is a class? Give examples.

T3. How are classes and models related?

What is an object oriented representation? Give examples.

2.2.5.2 Exercises

E1. Set up a model for a ball thrown in a base ball game.

E2. Define a class structure for a falling ball incorporating air friction.

E3. Implement in *Mathematica* a model for a falling particle with friction.

2.3 Car on a Bumpy Road

The problem considered in this section is a practical application originating from the early phase of a car design process. Engineers in the automotive business have to know in advance how a new car will react under certain environmental conditions. In the sequel, we discuss a procedure for building a model that provides basic information on the behavior of a car at the very beginning of its design. In this early stage of the process, very little information on the new car is available and thus it is sufficient to reduce the model to the most essential properties of the car. These properties can include the mass distribution on a chassis, that is, the location of the engine and other heavy components like the fuel tank, batteries, gear box, and so on. Another essential feature in this early design phase is the undercarriage. In our model, these components are combined into a multibody system interacting by forces with each other and with the environment. Knowing the preliminary behavior of the new car, the next step is to improve the model over several iterations to incorporate more details about the interacting components. Here, we restrict our considerations to the very beginning of the design process and demonstrate how an object-oriented approach can be used to model different components in a transparent way.

2.3.1 Modelling Steps

2.3.1.1 Hardware

Figure 2.9 shows the simplified one-dimensional mechanical model of a car moving on a bumpy road. In principle, the car consists of three main components: the two axles and the car body. We assume that the two axles are coupled to the chassis by a dash pot and a spring. In addition to the translation, we allow a narrow rotation of the body around the center of gravity. The wheels are in contact with the road (this interaction is modeled by introducing springs and dampers representing the two tires). The symbols shown in Figure 2.9 have the following meaning: k_i, d_i with $(i = 1 - 4)$, m_k with $(k = 1, 2, 3)$, a, and b are the physical and geometric parameters of the system. The k_i and d_i denote force and damping constants, m_k are the wheel and chassis masses, and a and b determine the center of the mass of the chassis. $J_{C2} = I$ is the inertia moment of the chassis. The dynamical coordinates denoted by z_1, z_2, z_3, and β describe the vertical and angular motion of the car. The acting external forces on the car are due to bumps in the road and gravity. The whole system can be described by Newton's equations.

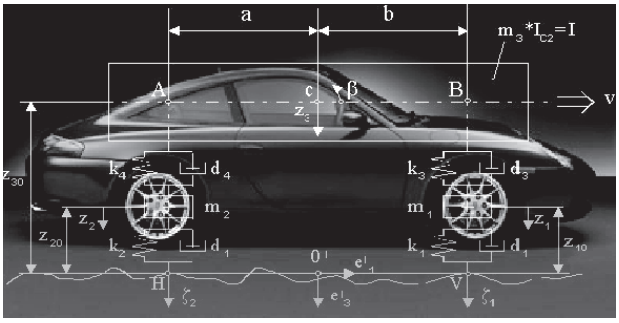


Figure 2.9. Simplified model for a car on a bumpy road.

2.3.1.2 Formulation

These equations of motion (Newton's equations) are given for the coordinates by

$$m_1 z_1'' = -F_1 + F_3 \tag{2.14}$$

$$m_2 z_2'' = -F_2 + F_4 \tag{2.15}$$

$$m_3 z_3'' = F_3 \tag{2.16}$$

$$I \beta'' = b F_3 - a F_4, \tag{2.17}$$

where F_i are the acting forces. One of the main problems in car design is to estimate how the total system will react under different forces, different damping parameters, different velocities, different environment conditions, and so on. These questions are partially answered in the following subsections. The complete examination of all these inquiries would exceed the space available in this section. Hence, we will restrict our consideration to a few examples.

To attack the physical and mathematical aspects we first identify classes that would capture the essence of this problem. If we look at the equations of motion (2.14-17) and at Figure 2.9, it seems natural to divide the car into three main classes: the body, the axles, and the environment.

2.3.2 Class Definitions

The problem can be split into the following classes (see Figure 2.10): the general setup class, the car body, the axles, the car itself, and some tools needed to carry out the simulation. The simulation class that provides graphical output refers to other classes that serve as a basis for the computation required to produce the output. The definition of classes here is mainly guided by the hardware.

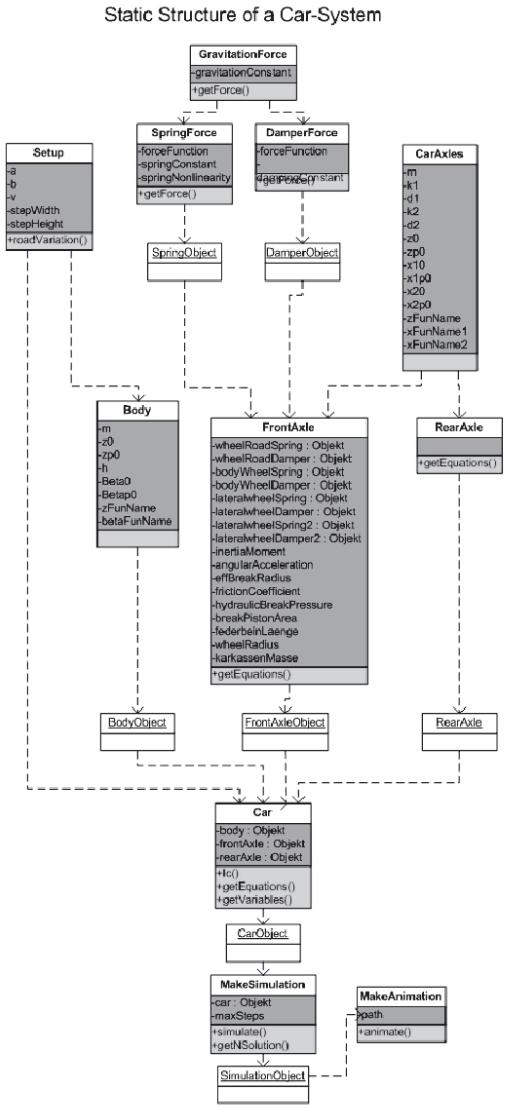


Figure 2.10. Connections of classes and objects for the simple car model.

In Figure 2.10, the static class structure of the car model is shown. Basic components are springs dampers, gravitation, axles, and the road. With these components a higher class for a car is created. The

car class is used in the simulation and animation class. The arrows show how the classes are interconnected and where objects of classes are involved. Objects are the interfaces of the model and allow a modular composition of the simulation.

The car model can be split into the following classes which are discussed in the following subsections: the general setup class, the car body, the axles, the car itself, and some tools needed to carry out the simulation. The simulation class provides graphical output. This class refers to all other classes that serve as a basis for the computation required to produce the output.

2.3.2.1 Class for Setup

The general setup is concerned with geometric properties and the influence of the environment, such as the external force given here by four bumps in the road.

```
Setup = Class["Setup", Class["Element"],
 (* --- Parameters --- *)
 {{a = 1.2, Description -> "distance front axle", Dimension -> m},
  {b = 1.2, Description -> "distance rear axle", Dimension -> m},
  {v = 5, Description -> "velocity of the car", Dimension -> m / s},
  {stepWidth = 0.1`, Description -> "step width", Dimension -> m},
  {stepHeight = 0.07, Description -> "step height", Dimension -> m}},
 (* --- Methods --- *)
 {extForce[t_] :=
  Fold[Plus, 0, Table[
       (Tanh[100 * v * (t - n stepWidth)] - Tanh[100 * (v * (t - n stepWidth) -
      stepWidth)]) * stepHeight / 2, {n, 1, 4}]]}
 ]
<Class Setup>
```

2.3.2.2 Class Body

The class ***Body*** consists solely of geometric and physical parameters of the car body. This class includes also the initial conditions for displacement, angle, and velocity.

```
Body = Class["Body", Setup,
 (* --- Parameters --- *)
 {description = "Body",
  {m = 1200, Description -> "mass", Dimension -> kg},
  {z0 = 0, Description -> "initial displacement", Dimension -> m},
  {zp0 = 0, Description -> "initial velocity", Dimension -> m / s},
  {h = 0.75, Description -> "geometric quantity", Dimension -> m},
  {β0 = 0, Description -> "initial value for β", Dimension -> rad},
    {βp0 = 0, Description -> "initial value for β prime",
     Dimension -> rad / s},
  {zFunName = z, Description -> "name of the z function",
     Dimension -> ""},
  {betaFunName = β , Description -> "name of the β function",
     Dimension -> ""}},
 (* --- Methods --- *)
 {}]
```

```
<Class Body>
```

2.3.2.3 Classes for Axles

The common properties of the two axles are collected in the class ***CarAxle***. This class contains only parameters - force and damping constants together with initial conditions for the equations of motion.

```
CarAxle = Class["CarAxle", Setup,
 (* --- Parameters --- *)
 {description = "Axle",
  {m = 42.5, Description -> "mass of the axle"},
  {k1 = 150 000, Description -> "wheel-body force constant"},
  {d1 = 700, Description -> "wheel-body damping constant"},
  {k2 = 4000, Description -> "wheel-road force constant"},
  {d2 = 1800, Description -> "wheel-road damping constant"},
  {z0 = 0, Description -> "initial displacement"},
  {zp0 = 0, Description -> "initial velocity"},
  {zFunName = z, Description -> "name of the function z"}},
 (* --- Methods --- *)
 {}]
```

```
<Class CarAxle>
```

In the next step we define classes for the front and the rear axle. The front axle consist solely of a method generating the equation of motion for the axle by considering the acting forces.

```
FrontAxle = Class["FrontAxle", CarAxle,
 (* --- Parameters --- *)
 {},
 (* --- Methods --- *)
 {getEquations[body_ /; ObjectOfClassQ[body, Body]] :=
  Block[{f1 = k1 * (zFunName[t] - extForce[t]) +
         d1 * (zFunName'[t] - D[extForce[t], t]),
   f3 = k2 * ((body ∘ zFunName)[t] - b * (body ∘ betaFunName)[t]
      - zFunName[t]) + d2 * (D[(body ∘ zFunName)[t] -
               b * (body ∘ betaFunName)[t], t] - zFunName'[t])},
   {m * zFunName''[t] + (-f1 + f3) * -1 == 0,
   zFunName[0] == z0, zFunName'[0] == zp0}]
 }
]
```

```
<Class FrontAxle>
```

The rear axle is generated in the same way by taking the relations specific for this component into account.

```
RearAxle = Class["RearAxle", CarAxle,
 (* --- Parameters --- *)
 {},
 (* --- Methods --- *)
 {getEquations[body_ /; ObjectOfClassQ[body, Body]] :=
    Block[{f2 = k1 * (zFunName[t] - extForce[t - (a + b) / v]) +
         d1 * ((body ∘ zFunName)'[t] - D[extForce[t - (a + b) / v], t]),
      f4 = k2 * ((body ∘ zFunName)[t] + a * (body ∘ betaFunName)[t]
     - zFunName[t]) + d2 * (D[(body ∘ zFunName)[t] +
               a * (body ∘ betaFunName)[t], t] - zFunName'[t])},
  {m * zFunName''[t] + (-f2 + f4) * -1 == 0,
   zFunName[0] == z0, zFunName'[0] == zp0}]
 }
]
```

```
<Class RearAxle>
```

2.3.2.4 Class Car

Finally, after setting up classes for the body and axles, we define a class to describe the car. This class incorporates parameters as well as methods to combine information from different components and deliver the equations of motion.

```
Car = Class["Car", Setup,
 (* --- Parameters --- *)
 {description = "Car",
 {body = Null, Description -> "body"},
 {frontAxle = Null, Description -> "front axle"},
 {rearAxle = Null, Description -> "rear axle"}},
 (* --- Methods --- *)

  {Ic[a0_, b0_, h0_, m03_] := Block[{}, m03 / 12 * ((a0 + b0)^2 + h0^2)],
 getEquations[] := Block[
  {f3 = frontAxle∘k2 *
    ((body∘zFunName)[t] -
            b * (body∘betaFunName)[t] - (frontAxle∘zFunName)[t]) +
   frontAxle∘d2 *
    (D[(body∘zFunName)[t] - b * (body∘betaFunName)[t], t] -
            (frontAxle∘zFunName)'[t]),
  f4 = rearAxle∘k2 *
    ((body∘zFunName)[t] +
            a * (body∘betaFunName)[t] - (rearAxle∘zFunName)[t]) +
   rearAxle∘d2 *
    (D[(body∘zFunName)[t] + a * (body∘betaFunName)[t], t] -
            (rearAxle∘zFunName)'[t])},
     Union[{body∘m * (body∘zFunName)''[t] + 2 f3 == 0,
        (body∘zFunName)[0] == body∘z0,
   (body∘zFunName)'[0] == body∘zp0, Ic[a, b, body∘h, body∘m] *
          (body∘betaFunName)''[t] + (b * f3 - a * f4) * -1 == 0,
   (body∘betaFunName)[0] == body∘β0, (body∘betaFunName)'[0] ==
        body∘βp0},
   frontAxle∘getEquations[body],
   rearAxle∘getEquations[body]]
  ],
 getVariables[] := {frontAxle∘zFunName, rearAxle∘zFunName,
  body∘zFunName, body∘betaFunName}}
]
<Class Car>
```

2.3.2.5 Class Simulation

Having the classes for all physical components available, we need a tool to carry out the simulation. The following classes provide an efficient simulation of the model and generate a graphical representation of the results.

```
MakeSimulation = Class[{"MakeSimulation", Class["Element"],
 (* --- Parameters --- *)
 {description = "Simulation of the motion",
 {car = Null, Description -> "Car"},
 {maxSteps = 100 000,
    Description -> "Maximum number of steps in NDSolve"}},
 (* --- Methods --- *)
 {simulate[tstart_, tend_] :=
  Block[{nsol = getNSolution[tstart, tend]},
   $TextStyle = {FontFamily -> "Arial", FontSize -> 12};
   MapThread[Plot[Evaluate[{{car∘extForce[t]}, #1 /. nsol}],
         {t, tstart, tend}, PlotRange -> All,
         PlotStyle -> {RGBColor[0.25098, 0, 0.25098],
           {RGBColor[0.9, 0, 0], AbsoluteThickness[3]}},
         AxesLabel -> {"t [s]", " "}, PlotLabel -> #2,
         GridLines -> Automatic, Frame -> True] &,
       {Map[Apply[#, {t}] &, car∘getVariables[]], {"Front Axle",
         "Rear Axle", "Body Movement", "Rotation Angle"}}]],

 getNSolution[tstart_, tend_] :=
  NDSolve[car∘getEquations[],
      car∘getVariables[], {t, tstart, tend},
   MaxSteps -> maxSteps]}
]
<Class MakeSimulation>
```

If we wish to see the motion of the car, it is convenient to declare a special class for animations.

```
MakeAnimation = Class[{"MakeAnimation", Class["Element"],
 (* --- Parameters --- *)
 {description = "Animate the motion of the car"
  {path = "C:\\Documents and Settings\\Gerd.Baumann\\My
          Documents\\Mma\\Books\\Engineering\\Vol_IV_Numerics\\
          Simulation_Methods\\Generic",
       Description -> "Path of the car model"}},
 (* --- Methods --- *)
 {animate[sim1_, tstart_, tend_, δt_] :=
    Block[{names, l1, tire1, tire2, window1, window2, window3, body,
   sol = sim1∘getNSolution[tstart, tend]},
   SetDirectory[path];
   names = FileNames["*.dat"];
```

```
l1 = Map[ReadList[#, Number, RecordLists -> True,
  RecordSeparators ->
   {If[StringMatchQ[$OperatingSystem, "Windows*"],
          "\n", "\r"]}] &, names];
tire1 = {GrayLevel[0.752941], Polygon[l1[[1]] /.
        {x_, y_} -> {x, y + z1[t]}]};
tire2 = {GrayLevel[0.752941], Polygon[l1[[2]] /.
        {x_, y_} -> {x, y + z2[t]}]};
window1 = {GrayLevel[0.752941], Polygon[l1[[3]] /.
        {x_, y_} -> {x Cos[-β[t]] + y Sin[-β[t]],
          y Cos[-β[t]] - x Sin[-β[t]] + z3[t]}]};
window2 = {GrayLevel[0.752941], Polygon[l1[[4]] /.
        {x_, y_} -> {x Cos[-β[t]] + y Sin[-β[t]],
          y Cos[-β[t]] - x Sin[-β[t]] + z3[t]}]};
window3 = {GrayLevel[0.752941], Polygon[l1[[5]] /.
        {x_, y_} -> {x Cos[-β[t]] + y Sin[-β[t]],
          y Cos[-β[t]] - x Sin[-β[t]] + z3[t]}]};
body = {RGBColor[0, 0, 0.25098], Polygon[l1[[6]] /.
        {x_, y_} -> {x Cos[-β[t]] + y Sin[-β[t]],
          y Cos[-β[t]] - x Sin[-β[t]] + z3[t]}]};
ListAnimate[Table[Show[Graphics[{body, tire1, tire2,
          window1, window2, window3} /. sol],
       AspectRatio -> Automatic, Axes -> False, Frame -> True,
       PlotRange -> {{-.1, 4.4}, {0, 1.4}},
       FrameTicks -> None], {t, tstart, tend, δt}]]
 ]}
]
<Class MakeAnimation>
```

2.3.3 Objects

Up to this point, nothing more than a framework was defined in which the computation can be carried out. The given classes represent essentially only templates for creating the calculation objects. In order to setup the simulation, we have to generate objects and specify their properties. The car is generated by a modular process incorporating the distinguished objects. At this point, it is apparent that we have a great advantage in designing a specific car compared to traditional approaches in simulation. For example, we can define different objects for axles and use them as in different simulations as parameters of the underlying model. Components (objects) of the same type can be exchanged without causing any harm to the system. In the sequel, we demonstrate how this process is carried out.

2.3.3.1 Body Object

First, we create the body by applying *Elements* method ***new[]*** to the class ***Body***. All properties except ***zFunName*** and ***betaFunName*** will have their default valued specified in the class declaration.

```
b = Body ∘ new[{zFunName → z3, betaFunName → β}]
```

```
<Object of Body>
```

The actual properties of the body can be checked by

```
GetPropertiesForm[b]
```

Property	Value
description	Body
a	1.2
b	1.2
betaFunName	β
h	0.75
m	1200
stepHeight	0.07
stepWidth	0.1
v	5
z0	0
zFunName	z3
zp0	0
β0	0
βp0	0

2.3.3.2 Axle Object

Next, objects for the axles are created. The front axle is defined with a mass of 45.5 kg, the function representing the displacement of the center of the wheel attached to this axle is denoted by z_1.

```
fa = FrontAxle ∘ new[{m -> 45.5, zFunName → z1}]
```

```
<Object of FrontAxle>
```

The properties of the axle are recalled by:

```
GetPropertiesForm[fa]
```

Property	Value
description	Axle
a	1.2
b	1.2
d1	700
d2	1800
k1	150 000
k2	4000
m	45.5
stepHeight	0.07
stepWidth	0.1
v	5
z0	0
zFunName	z1
zp0	0

Since no values for the damping and the force constant were given in the statement above to ***new[]***, these properties are initialized with values specified in the class declaration itself.

A second version of the front axle with a more stiffer damping is generated in the following line. We will use this object later in our simulations to compare the influence of choosing stiffer springs in a design. Here, k_1 and k_2 are set to the specific values.

```
fas = FrontAxle∘new[{m → 45.5,
     d1 → 5000, d2 → 5000,
     k1 → 150 000, k2 → 150 000,
     zFunName → z1}]
```

```
<Object of FrontAxle>
```

For the rear axle we just use the default settings. The coordinate of elongation is denoted by z_2.

```
ra = RearAxle∘new[{zFunName → z2}]
```

```
<Object of RearAxle>
```

A check of parameters shows the values.

```
GetPropertiesForm[ra]
```

Property	Value
description	Axle
a	1.2
b	1.2
d1	700
d2	1800
k1	150 000
k2	4000
m	42.5
stepHeight	0.07
stepWidth	0.1
v	5
z0	0
zFunName	z2
zp0	0

2.3.3.3 Car Object

Having all components needed available, it is easy to generate the car by incorporating the defined objects into a common model. The car object is defined by

```
c = Car ∘ new[{frontAxle → fas, rearAxle → ra, body → b}];
```

It consist of the front axle, the rear axle, and the body. We note that an object-oriented approach allows an easy change of the behavior of the model by just changing different components or their properties, respectively. Hence, if an engineer would have access to a collection of different bodies, axles and springs, he would be able to create and test many different car designs without the need to adapt the model.

2.3.3.4 Simulation Object

A simulation object is created from the class ***MakeSimulation***. The simulation can be performed with any object of the class ***Car***.

```
sim = MakeSimulation ∘ new[{car → c}]
<Object of MakeSimulation>
```

Finally, the actual simulation is initiated by calling the ***simulate[]*** method of the class ***MakeSimulation***. The input for this function are just the starting point and the end point of the simulation interval. The result are four plots for coordinates of axles, the body, and the rotation angle of the body. All quantities are plotted depending on the simulation time.

```
GraphicsGrid[Partition[sim∘simulate[0, 2], 2]]
```

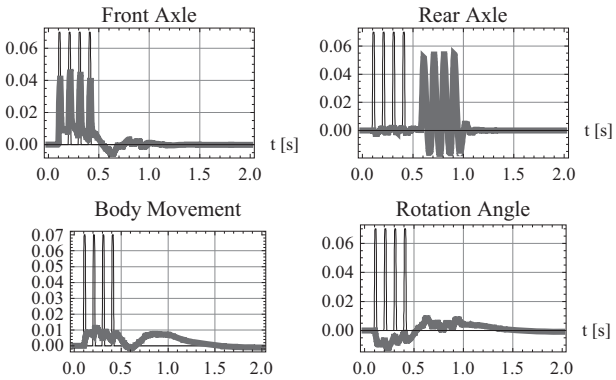


Figure 2.11. Simulation results for the different components, front and rear axle, body motion, and rotation about the center of mass.

Now the hard work for an engineer starts. If he is interested in an optimization of the elongations, he must change the properties of the car in the simulation. How to achieve the best result and how to select the best strategy depends on the experience of the individual engineer. However, within an object oriented simulation environment he has the flexibility to incorporate his thoughts in a quick and transparent way. For example, he can change some parameters of the front axle to achieve a better performance of the car...

```
sim∘car∘frontAxle∘d1 = 700;
sim∘car∘frontAxle∘k1 = 100 000;

GraphicsGrid[Partition[sim∘simulate[0, 2], 2]]
```

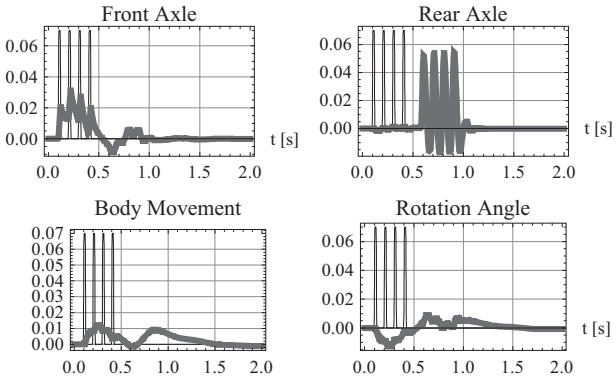


Figure 2.12. Simulation results for changed parameters of the front axle.

...or he can replace the whole front axle by another one with a smoother damper.

```
sim∘car∘frontAxle = fa
```

```
<Object of FrontAxle>
```

```
GraphicsGrid[Partition[sim∘simulate[0, 2], 2]]
```

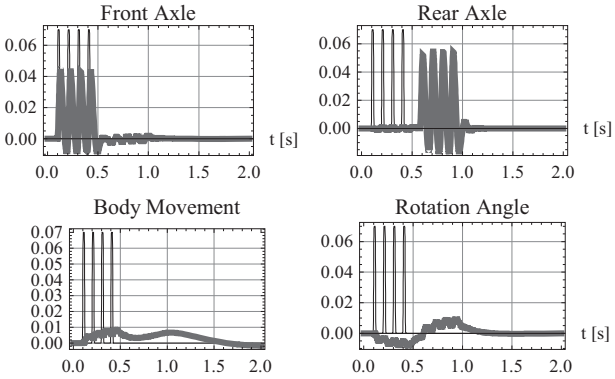


Figure 2.13. Simulation results for the model with a replaced front axle with different properties.

... and so on. However, a clever engineer will resort to some optimization strategies which can be carried out by an additional class in the development environment (not shown here).

Finally, if the results are to be presented at meetings and workshops, the behavior of the optimized car can be shown as an animation using different simulation models or strategies. The generation of an animation object and the animation itself is started with

```
an = MakeAnimation∘new[];
anim = an∘animate[sim, 0, 2, 0.05]
```

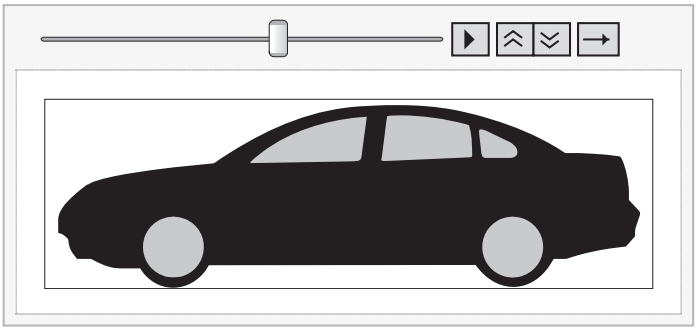


Figure 2.14. Animation of the car motion during a ride over four bumpers.

The flip chart movie is suppressed in the printed version. However, the notebook version of this article shows you the movement of the car body and the wheels.

This example demonstrated that the object-oriented approach to modeling is very efficient and transparent to the user. It divides the process of modeling into two phases. In the first phase, the model and its components are implemented. In the optimization phase, the parameters of the model satisfying certain criteria are found. Both phases benefit from the availability of predefined components (objects). Using the classical approach, minor structural changes (e.g., using a different axle) require modifications by the model builder. This, in turn, may have made a subsequent verification step necessary. In contrast, the object-oriented approach allows this to be done much more easily. A new component (e.g., an axle) can be added or exchanged without affecting the soundness of the model. This fact is of crucial importance in the industrial environment where models contain a huge number of parameters. It is necessary to have a clear and well-founded procedure to handle them. If the process to manage and work with models is clearly specified, results can be obtained more quickly, and they are more predictable, and comprehensible.

References

[1] Abell, M. L. & Braselton, J. P. Mathematica by example. 4th ed. (Elsevier, Amsterdam, 2009).

[2] Atkinson, K. E. An introduction to numerical analysis. 2nd ed. (Wiley, New York, 1989).

[3] Baumann, G. Classical mechanics and nonlinear dynamics. 2 nd ed. (Springer, New York, NY, 2005).

[4] Baumann, G. Electrodynamics, quantum mechanics, general relativity, and fractals. 2 nd ed. (Springer, New York, NY, 2005).

[5] Bradie, B. An introduction to numerical methods (Prentice Hall, 2001).

[6] Burden, R. L. & Faires, J. D. Numerical analysis. 8th ed. (Thomson Brooks/Cole, Belmont CA, 2005).

[7] Conte, S. D. & Boor, C. de. Elementary numerical analysis. An algorithmic approach. 3rd ed. (McGraw-Hill, New York, 1980).

[8] Epperson, J. F. An introduction to numerical methods and analysis (Wiley-Interscience, Hoboken N.J., 2007).

[9] Falknor, V.M., and .Skan, S.W.: Some approximate solutions of the boundary layer equa.
tions. Phil. Mag. 12, 805-896 (1931); ARC RM, 1314 (1930),

[10] Fogiel, M. The Numerical analysis problem solver (REA, New York N.Y., 1983).

[11] Gautschi, W. Numerical analysis. An introduction (Birkhäuser, Boston, 1997).

[12] Gerald, C. F. & Wheatley, P. O. Applied numerical analysis. 7th ed. (Pearson/Addison-Wesley, Boston, 2004).

[13] Hairer, E., Nörsett, S.P., and Wanner, G., Solving Ordinary Differential Equations, 2nd ed., Springer Verlag, New York, 1993-1996.

[14] Hamming, R. W. Introduction to applied numerical analysis (McGraw-Hill, New York, 1971).

[15] Hildebrand, F. B. Introduction to numerical analysis. 2nd ed. (Dover Publications, New York,

1987, 1974).

[16] Householder, A. S. The theory of matrices in numerical analysis (Dover Publications, Mineola N.Y., 2006).

[17] Iserles, A., A First Course in the Numerical Analysis of Differential Equations, Cambridge University Press, Cambridge, 1996.

[18] Kincaid, D. & Cheney, E. W. Numerical analysis. Mathematics of scientific computing. 3rd ed. (Brooks/Cole, Pacific Grove CA, 2002).

[19] Kincaid, D. & Cheney, E. W. Numerical analysis. Mathematics of scientific computing. 3rd ed. (American Mathematical Society, Providence R.I., 2009?).

[20] Leader, J. J. Numerical analysis and scientific computation (Pearson Addison Wesley, Boston 2004).

[21] Moin, P. Fundamentals of engineering numerical analysis (Cambridge University Press, Cambridge UK , New York, 2001).

[22] Pettofrezzo, A. J. Introductory numerical analysis (Dover Publications, Mineola N.Y., 2006).

[23] Phillips, G. M. Theory and applications of numerical analysis. 2nd ed. (Academic Press, London u.a., 1996).

[24] Ralston, A. & Rabinowitz, P. A first course in numerical analysis. 2nd ed. (Dover Publications Mineola NY, 2001).

[25] Ruskeepää, H. Mathematica navigator. Mathematics, statistics, and graphics. 3rd ed (Elsevier/Academic Press, Amsterdam, Boston, 2009).

[26] Sauer, T. Numerical analysis (Pearson Addison Wesley, Boston, 2006).

[27] Schatzman, M. Numerical analysis. A mathematical introduction (Clarendon Press; Oxford Univer sity Press, Oxford , New York, 2002).

[28] Scheid, F. J. Schaum's outline of theory and problems of numerical analysis. 2nd ed. (McGraw-Hill New York, 1989).

[29] Schlichting, Hermann, Boundary-layer theory, McGraw Hill, New York, 1979

[30] Stoer, J. & Bulirsch, R. Introduction to numerical analysis. 3rd ed. (Springer, New York, 2002).

[31] Strang, G., and Fix, G.J., An Analysis of the Finite Element Method, Prentice - Hall, Inc., Engle wood Cliffs, N.J., 1973.

[32] Süli, E. & Mayers, D. F. An introduction to numerical analysis (Cambridge University Press, Cam bridge , New York, 2003).

[33] Temam, R. Numerical analysis (Reidel, Dordrecht, Boston, 1973).

[34] Trott, M. The Mathematica guidebook. concepts, examples and applications (Springer, Berlin u.a 1999).

[35] Trott, M. The Mathematica guidebook. Mathematics in Mathematica (Telos, New York, 2000).

[36] Wellin, P. R., Gaylord, R. J. & Kamin, S. N. An introduction to programming with Mathematica 3rd ed. (Cambridge Univ. Press, Cambridge, 2005).

[37] Wood, A. S. Introduction to numerical analysis (Addison-Wesley, Harlow England, Readin Mass., 1999).

[38] Zienkiewicz, O.C., and Taylor, R.L., The Finite Element Method, 4 th ed., McGraw - Hill, New York, 1989.