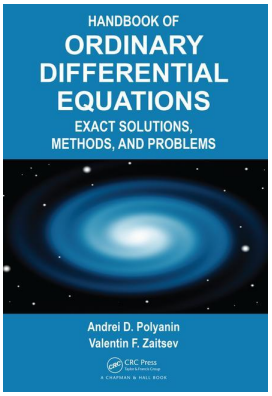

Handbook of Ordinary Differential Equations Exact Solutions, Methods, and Problems

Andrei D. Polyanin, Valentin F. Zaitsev

Chapter 21: Symbolic and Numerical Solutions of ODEs with MATLAB

Publication details
https://test.routledgehandbooks.com/doi/10.1201/9781315117638-21
Andrei D. Polyanin, Valentin F. Zaitsev
Published online on: 03 Nov 2017

Chapter 21

Symbolic and Numerical Solutions of ODEs with MATLAB

21.1 Introduction

21.1.1 Preliminary Remarks

In the previous two chapters, we paid special attention to analytical solutions of ordinary differential equations and systems owing to the availability of the computer algebra systems Maple and Mathematica in modern mathematics.

Frequently, the functions and data in ODE problems are defined at discrete points and equations are too complicated, so it is not possible to construct analytical solutions. Therefore, we have to study and develop numerical approximation methods for ordinary differential equations [e.g., see Gear (1971), Shampine and Gordon (1975), Forsythe et al. (1977), Conte and de Boor (1980), Fox and Mayers (1987), Kahaner et al. (1989), Shampine (1994), Ascher et al. (1995), Shampine and Reichelt (1997), Shampine et al. (2003), Lee and Schiesser (2004)].

Following the most important ideas and methods, we apply and develop numerical methods to obtain numerical and graphical solutions for studying ordinary differential equations.

Nowadays, for this purpose one can use computers and supercomputers extensively applying convenient and powerful computational software, e.g., an interactive programming environment for scientific computing, MATLAB®, which provides integrated symbolic and numerical computation and graphics visualization in a high-level programming language. Additionally, MATLAB's excellent graphics capabilities can help one understand the results and analyze the solution properties.

In this chapter, we turn our attention to numerical methods for solving ordinary differential equations using MATLAB. Since numerical and analytical methods are complementary techniques for investigating solutions of differential equations, we will also consider some essential analytical tools provided in MATLAB.

MATLAB has an extensive library of predefined functions for solving ordinary differential equations. We compute symbolic and numerical solutions using MATLAB's predefined functions (which implement known methods for solving ordinary differential equations)

and develop new MATLAB procedures for constructing symbolic and numerical solutions.

Remark 21.1. The numerical methods embedded in MATLAB can solve only first-order ODEs or systems of first-order ODEs. To obtain solutions of nth-order ODEs (where the order is $n > 1$) by applying predefined functions, we have to rewrite higher-order ODEs as an equivalent system of first-order ODEs.

21.1.2 Brief Introduction to MATLAB

► **MATLAB's conventions and terminology.**

In this chapter, we use the following conventions introduced in MATLAB:

- `Cn` ($n = 1, 2, \ldots$), for arbitrary constants
- the letter `D`, the differential operator (should not be used for symbolic variables)
- `Dc`, for a dependent variable (in differential equations), where `c` is any character
- the letter `t`, the independent variable (by default) for the predefined function `dsolve`

Also we introduce the following notation for the MATLAB solutions:

- `Eqn`, for equations ($n = 1, 2, \ldots$)
- `ODEn`, for ODEs
- `Soln`, for solutions
- `Exprn`, for expressions
- `Strn`, for string expressions
- `ODESysn`, for systems of ODEs
- `ICn`, `BCn`, for initial and boundary conditions
- `IVPn`, `BVPn`, for initial and boundary value problems
- `Ln`, for lists of expressions
- `Gn`, for graphs of solutions
- `ops`, `val`, for various optional arguments in predefined functions and their values
- `vars`, for independent variables
- `funcs`, for dependent variables (indeterminate functions)

► **Basic description.**

MATLAB (short for "matrix laboratory") is not a general purpose programming language as Maple and Mathematica. MATLAB is an interactive programming environment that provides powerful high-performance numerical computing, excellent graphics visualization, symbolic computing capabilities, and capabilities for writing new software programs using a high-level programming language.

The *Symbolic Math Toolbox* (Ver. ≥ 4.9), based on the muPAD symbolic kernel, provides symbolic computations and variable-precision arithmetic. Earlier versions of the *Symbolic Math Toolbox* are based on the Maple symbolic kernel.

Simulink (short for "simulation and link"), also included in MATLAB, offers modeling, simulation, and analysis of dynamical systems (e.g., signal processing, control, communications, etc.) under a *graphical user interface* (GUI) environment.

The first concept of MATLAB and its original version (written in Fortran) was developed by Prof. Cleve Moler at the University of New Mexico in the late 1970s to provide his students with a simple interactive access (without having to learn Fortran) to LINPACK

and EISPACK software.* Over the next several years, this original version of MATLAB had spread within the applied mathematics community. In early 1983, Jack Little (an engineer), together with Cleve Moler and Steve Bangert, developed a professional version of MATLAB (written in C and integrated with graphics). The company MathWorks was created in 1984 and headquartered in Natick, Massachusetts, to continue its development.

► Most important features.

The most important features of MATLAB are as follows:

- interactive user interface;
- a combination of comprehensive mathematical and graphics functions with a powerful high-level language in an easy-to-use environment;
- fast numerical computation and visualization, especially for performing matrix operations [e.g., see Higham (2008)];
- easy usability and great flexibility in data manipulation;
- symbolic computing capabilities via the Symbolic Math Toolbox (Ver. < 4.9 or Ver. ≥ 4.9), based on the Maple or muPAD symbolic kernel, respectively;
- the basic data element is an array that does not require dimensioning;
- a large library of functions for a wide range of applications;
- it is easy to incorporate new user-defined capabilities (toolboxes consisted of M-files and written for specific applications);
- understandable and available for almost all operating systems;
- powerful programming language, intuitive and concise syntax, and easy debugging;
- Simulink, as an integral part of MATLAB, provides modeling, simulation, and analysis of dynamical systems;
- free resources, such as MathWorks Web Site (`www.mathworks.com`), MathWorks Education Web Site (`www.mathworks.com/education`), MATLAB newsgroup (`comp.soft-sys.matlab`), etc.

► Basic parts.

MATLAB consists of five parts:

- The *Development Environment*, a set of tools that facilitate using MATLAB functions and files (e.g., graphical user interfaces and the workspace).
- The *Mathematical Function Library*, a vast collection of computational algorithms.
- The *MATLAB language*, a high-level matrix/array language (with flow control statements, functions, data structures, input/output, and object-oriented programming features).
- The *MATLAB graphics system*, which includes high-level functions (for 2D/3D data visualization, image processing, animation, etc.) and low-level functions (for fully customizing the graphics appearance and constructing complete graphical user interfaces).
- The *Application Program Interface* (API), a library for writing C and Fortran programs that interact with MATLAB.

*LINPACK and EISPACK is a collection of Fortran subroutines, developed by Cleve Moler and his several colleagues, for solving linear equations and eigenvalue problems, respectively.

► **Basic concepts.**

The *prompt symbol* >> indicates where to type a MATLAB command; typing a statement and pressing `Return` or `Enter` at the end starts the evaluation of the command, displays the result, and inserts a new prompt; the semicolon (`;`) symbol at the end of the command tells MATLAB to evaluate the command but not display any result.

In MATLAB, the cursor cannot be moved to the desired line (unlike Maple and Mathematica) but, for simple problems, corrections can be made by pressing the `up` or `down arrow` key to scroll through the list of (recently used) functions and then the `left` or `right arrow` key to change the text. Also, corrections can be made using `copy/paste` of the previous lines located in the `Command Window` or `Command History`.

The *previous result* (during a session) can be referred to with the variable `ans` (the last result). MATLAB prints the *answer* and assigns the value to `ans`, which can be used for further calculations.

MATLAB has many forms of help: a complete *online help system* with tutorials and reference information for all functions; the command-line help system, which can be accessed by using the `Help` menu, pressing `F1`, selecting `Help->Demos`, or entering `Help` and selecting `Functions->Alphabetical List` or `Index`, `Search`, `MATLAB->Mathematics`; or by typing `helpbrowser`, `lookfor` (e.g., `lookfor plot`) or `help FunctionName`, `doc FunctionName`, etc.

In MATLAB (Ver. 7), a new feature for correctly typing function names has been added. One can type only the first few letters of the function and then press the `TAB` key (to see all available functions and complete typing the function).

MATLAB *desktop* appears, containing tools (graphical user interfaces) for managing files, variables, and applications. The default configuration of desktop includes various tools, e.g., `Command Window`, `Command History`, `Workspace`, `Find Files`, `Current Directory` (for more details, see `demo MATLAB desktop`), etc. One can modify the arrangement of tools and documents.

For a new problem, it is best to begin with the statement `clear all` for cleaning all variables from MATLAB's memory. All examples and problems in the book assume that they begin with `clear all`.

A MATLAB program can be typed at the prompt >> or, alternatively (e.g., for more complicated problems), by creating an *M-file* (with `.m` extension) using MATLAB *editor* (or using another text editor). MATLAB editor is invoked by typing `edit` at the prompt.

M-files are files that contain code in the MATLAB language. There are two kinds of M-files: *script M-files* (which do not accept input arguments or return output data) and *function M-files* (which can accept input arguments and return output arguments).

In the process of working with various M-files, it is necessary to define the path, which can be done by selecting `File->Set Path->Add Folder` or via the `cd` function.

The *structure* of a MATLAB program or source code is as follows: the *main program* or *script* and the necessary *user-defined functions*. The execution starts by typing the file name of the main program.

Incorrect response. If you get no response or an incorrect response, you may have entered or executed the function incorrectly. Correct the function or interrupt the computation by entering debug mode and setting breakpoints: select the following on the Desktop menu:

```
Debug->Open M-files when Debugging
Debug->Stop if Errors/Warnings
```

Also, one can detect erroneous or unexpected behavior in a program with the aid of MATLAB functions, e.g., `break`, `warning`, and `error`.

Palettes can be used, e.g., for building or editing graphs (`Figure Palette`), displaying the names of the GUI components (`Component Palette`), etc.

MATLAB *graphical user interface development environment* (GUIDE) provides a set of tools for creating graphical user interfaces (GUIs). These tools greatly simplify the construction of GUIs, e.g., layout the GUI components (panels, buttons, menus, etc.) and program the GUI.

MATLAB consists of a family of add-on *toolboxes*, which are collections of functions (M-files) and extend the MATLAB environment to solve particular classes of problems.

The toolboxes can be *standard* or *specialized* (see `Contents` in `Help`). Nowadays, many specialized toolboxes are available. MATLAB can be augmented by a number of toolboxes consisting of M-files and written for specific applications.

21.1.3 MATLAB Language

MATLAB language is a high-level procedural dynamic and imperative programming language (similar to Fortran 77, C, and C++), with powerful matrix/array operations, control statements, functions, data structures, input/output, and object-oriented programming features. In addition, it is an interpreted language, similar to Maple and Mathematica [e.g., see Shingareva and Lizárraga-Celaya (2009)]; i.e., the instructions are translated into machine language and executed in real time (one at a time). MATLAB language allows programming-in-the-small (coding or creating programs for performing small-scale tasks) and programming-in-the-large (creating complete large and complex application programs). It supports a large collection of data structures or MATLAB classes and operations among these classes.

In linear algebra, there exist two types of operations with vectors/matrices: operations based on the mathematical structure of vector spaces and element-by-element operations on vectors/matrices as in data arrays. This difference can be made in the name of the operation or the name of the data structure. In MATLAB, separate operations are defined (for matrix and array manipulation), but the data structures `array` and `vector/matrix` are the same. But, for example, in Maple the situation is opposite: the operations are the same, but the data structures are different.

Arithmetic operators: scalar operators (`+ - * / ^`), matrix multiplication/power (`* ^`), array multiplication/power (`.* .^`), matrix left/right division (`\ /`), and array division (`./`).

Logical operators: and (`&`), or (`|`), exclusive or (`xor`), and not (`~`).

Relational operators: less/greater than (`< >`), less/greater than or equal to (`<= >=`), and equal/not equal (`== ~=`).

A variable name is a character string of letters, digits, and underscores such that it begins with a letter and its length is bounded by `N=namelengthmax` (e.g., $N = 63$). Punctuation marks are not allowed (see `genvarname` function). Variable declaration is not necessary in MATLAB, but all variables must be given initial values; e.g., `a12_new=9`. A variable can change in the calculation process, e.g., from integer to real (and vice versa).

MATLAB is case sensitive, and there is a difference between lowercase and uppercase letters, e.g., `pi` and `Pi`.

Various reserved keywords, symbols, names, and functions, for example, reserved keywords and function names, cannot be used as variable names (see `isvarname`, `which -all`, `isreserved`, `iskeyword`).

A string variable is enclosed by single quotes and belongs to the `char` class (e.g., `x='string'`), and the function `sin(x)` is invalid. Strings can be used with converting, formatting, and parsing functions (e.g., see `cellstr`, `char`, `sprintf`, `fprintf`, `strfind`, `findstr`).

MATLAB provides three basic types of variables: *local variables*, *global variables*, and *persistent variables*.

The operator "set equal to" (=). A variable in MATLAB (in contrast to Maple and Mathematica) cannot be "free" (with no assigned value) and must be assigned any initial value by the operator "set equal to" (=).

The difference between the operators "set equal to" (=) and "equal" (==) is that the operator `var=val` is used to assign `val` to the variable `var`, while `val1==val2` compares two values; e.g., `A=3; B=3; A==B`.

Statements are input instructions from the keyboard that are executed by MATLAB (e.g., `for i=1:N s=s+i*2; end`). A MATLAB statement may begin at any position in a line and may continue indefinitely in the same line, or may continue in the next line, by typing by three dots (`...`) at the end of the current line. White spaces between words in a statement are ignored; a number cannot be split into two pieces separated by a space.

The statement separator semicolon (;). The result of a statement followed with a semicolon (;) will not be displayed. If the semicolon is omitted, the results will be printed on the screen; e.g., `x=-pi:pi/3:pi;` and `x=-pi:pi/3:pi`.

Multiple statements in a line: two or more statements may be written in the same line if they are separated with semicolons.

Comments can be included with the percentage sign `%` and all characters following it up to the end of a line. Comments at the start of a code have a special significance: they are used by MATLAB to provide the entry for the help manual for a particular script. The block comment operators, `%{ %}`, can be used for writing comments that require more than one line.

An expression is a valid statement and is formed as a combination of numbers, variables, operators, and functions. The arithmetic operators have different precedences (increasing precedence `+ - * / ^`). Precedence is altered by parentheses (expressions within parentheses are evaluated before expressions outside parentheses).

A Boolean or *logical expression* is formed with *logical* and *relational operators*; e.g., `x>0`. Logical expressions are used in `if`, `switch`, and `while` statements. The logical values, *true* and *false*, are represented by numerical values, *1* and *0*, respectively.

A regular expression is a string of characters that defines a *pattern* (for details, see `help pattern`). For example, `'Math?e\w*'`. Regular and dynamic expressions can be used to search text for a group of words that matches the pattern (e.g., for parsing or replacing a subset of characters within text).

MATLAB is sensitive to types of brackets and quotes (for details, see `help paren`, `help punct`).

Types of brackets:

Square brackets, `[ ]`, for constructing vectors and matrices, for example, `A1=[1 2 3]`, `A2=[1, 2, 3]`, `A3=[1, 2; 4,5]`. For multiple assignment statements, for example `A4=[1,5;2,6]`, `[L,U]=lu(A4)`.

Parentheses, `( )`, for grouping expressions, `(5+9)*3`, for delimiting the arguments of functions, `sin(5)`, for vector and matrix elements, `A1(2)`, `A3(1,1)`, `A2([1 2])`; in logical expressions, `A1(A1>2)`.

Curly brackets, { }, for working with cell arrays; e.g., `C1={int8(3) 2.59 'A'}`, `C1{1}`, `X(2,1)={[1 3; 4 6]}`.

Dot-parentheses, `.( )`, for working with a structure via a dynamic field name; e.g., `S.F1=1; S.F2=2; F='F1'; val1=S.(F)`.

Quotes:

Forward-quotes, `(' ')`, for creating strings, for example, `T='the name=7;' k=5; disp('the value of k is'); disp(k)`,

A single forward-quote and dot single forward-quote, `(' .')`, for matrix transposition (the complex conjugate/nonconjugate transpose of a matrix), `A1=[1+i,i;-i,1-i]; A1'; A.'`.

Types of numbers. Numbers are stored (by default) as double-precision floating point (class `double`). To operate with integers, it is necessary to convert from double to the integer type (e.g., classes `int8`, `int16`, `int32`), `x=int16(12.3)`, `str='MATLAB'`, `int8(str)`. Mathematical operations that involve integers and floating-point numbers result in an integer data type. Real numbers can be stored as *double-precision floating point* (by default) or *single-precision floating point*; e.g., `x1=3.25`, `x2=single(x1)`, `x3=double(x2)` (for details, see `whos`, `isfloat`, `class`). Complex numbers can be created as `z1=1+2*i`, `z2=complex(1,2)`. Rational numbers can be formed by setting the `format` to `rational`; e.g., `x=3.25; format rational x format`. To check the current format setting, we type `get(0,'format')`.

Predefined constants: symbols for definitions of commonly used mathematical constants; e.g., `true`, `false`, `pi`, `i`, `j`, `Inf`, `inf`, `NaN` (not a number), `exp(1)`, the Euler constant γ, `-psi(1)`, `eps`.

In MATLAB, there are *predefined functions* and *user-defined functions*. Predefined functions are divided into *built-in functions* and *library functions*:

- *Built-in functions* are precompiled executable programs and run much more efficiently (see `help elfun`, `help elmat`).
- *Library functions* are stored as M-files (in the libraries or toolboxes), which are available in readable form (see `which`, `type`, `exist`). MATLAB can be complemented with locally user-developed M-files and toolboxes.

Many functions are overloaded (i.e., have an additional implementation of an existing function) so that they handle different classes (e.g., `which -all plot`).

Numerous *special functions* are defined; e.g., `help bessel`, `help specfun`.

User-defined functions can be created as M-files (see `help 'function'`) or as anonymous functions.

A *User-defined function* written in an M-file (with the extension `.m`) must contain only one function. It is best to have the same name for the *function name* and the *file name*. The

process of creating functions is as follows: create and save an M-file using a text editor, then call the function in the main program (or in `Command Window`).

Functions written in M-files have the following forms:

`function OArg=FunName(IArg); FunBody; end`, or

```
function [OArg1,OArg2,...]=FunName(IArg1,IArg2,...);
FunBody; end
```

where `OArg` and `IArg` are the output arguments and the input arguments, respectively. For example, the function $y = \sin x$ is defined as follows:

```
function f=SinFun(x); f=sin(x); end
```

Evaluation of functions: `FunName(Args)`.

For example, for the sine function we have `cd('c:/mypath'); SinFun(pi/2); type SinFun`.

Anonymous functions create simple functions without storing functions to files. Anonymous functions can be constructed either in the `Command Window` or in any function or script; e.g., the function $f(x) = \sin x$ is defined as `f=@(x) sin(x); f(pi/2)`.

A *function handle*, `@`, is one of the standard MATLAB data types that provides calling functions indirectly, e.g., to call a subfunction when outside the file that defines that function (see class `function_handle`).

Nested functions are allowed in MATLAB; i.e., one or more functions or *subfunctions* within another function can be defined in MATLAB. In this case, the `end` statements are necessary.

MATLAB language has the following *control structures*: the selection structures `if`, `switch`, `try` and the repetition structures `for`, `while`.

MATLAB does not have a *module system* in the traditional form: it has a system based on storing *scripts* and *functions* in M-files and placing them into *directories* (see `cd` function for changing the current directory, `help ..`).

MATLAB *data structures* or *classes*, vectors, matrices, and arrays, are used to represent more complicated data. There are 15 fundamental classes, which are in the form of a matrix or array: `double`, `single`, `int8`, `uint8`, `int16`, `uint16`, `int32`, `uint32`, `int64`, `uint64`, `char`, `logical`, `function_handle`, `struct`, and `cell`. The numerical values are represented (by default) as floating-point double precision (`float double`). One can construct various *composite data types* (e.g., sequences, lists, sets, tables, etc.) using the classes `struct` and `cell`.

Vectors are ordered lists of numbers separated by commas or spaces inside `[ ]`; no dimensioning is required. But *vector and array indices* can only be positive and nonzero. The notation `X=[1:0.1:9]` stands for a vector of numbers from 1 to 9 in increments of 0.1 (see `help colon`).

Matrices are rectangular arrays of numbers (row/column vectors are special cases of matrices).

⊙ *Literature for Section .1*: C. W. Gear (1971), L. F. Shampine and M. K. Gordon (1975), G. E. Forsythe, M. A. Malcolm, and C. B. Moler (1977), S. D. Conte and C. de Boor (1980), L. Fox and D. F. Mayers (1987), D. Kahaner, C. B. Moler, and S. Nash (1989), L. F. Shampine (1994), U. M. Ascher, R. M. M. Mattheij, and R. D. Russell (1995), L. F. Shampine and M. W. Reichelt (1997), L. F. Shampine, I. Gladwell, and S. Thompson (2003), H. J. Lee and W. E. Schiesser (2004), N. J. Higham (2008), I. K. Shingareva and C. Lizárraga-Celaya (2009).

21.2 Analytical Solutions and Their Visualizations

21.2.1 Analytical Solutions in Terms of Predefined Functions

The `Symbolic Toolbox` provides various predefined functions for solving, plotting, and manipulating symbolic mathematical equations. If we solve ordinary differential equations, we can obtain explicit or implicit exact solutions [e.g., see Murphy (1960), Kamke (1977), Zwillinger (1997), Polyanin and Manzhirov (2007)]. Consider the most relevant related functions for finding analytical solutions of a given ODE problem.

```
syms y(x); Sol1=dsolve(ODE)        Sol2=dsolve('ODE','var')
Sol3=dsolve(ODE,ICs)            Sol4=dsolve(ODE,ICs,ops,val)
```

- `x=sym('x')`, `y=sym('y')`, declaring symbolic objects (one at a time)
- `syms y(x)`, declaring symbolic objects (all at once)
- `dsolve`, finding closed-form solutions for a single `ODE`, where `ODE` is a symbolic equation containing `diff` or a string with the letter `D` (for the derivatives); for more details, see `help dsolve`
- `dsolve`, `ODE`, `ICs`, solving an ODE with given initial or boundary conditions
- `dsolve`, `ODE`, `ICs`, `ops`, `val`, specifying additional options and their values for solving ODEs

► **Verification of exact solutions.**

Let us assume that we have obtained exact solutions and we wish to verify whether these solutions are exact solutions of given ODEs.

Example 21.1. *First-order nonlinear ODE. Special Riccati equation. Verification of solutions.* For the first-order nonlinear ODE, the *special Riccati equation*

$$y'_x = ay^2 + bx^n,$$

we can verify that the solutions

$$y(x) = -\frac{1}{a}\frac{w'_x}{w},$$

where

$$w(x) = \sqrt{x}\left[C_1 J_v\left(\frac{\sqrt{ab}}{k}x^k\right) + C_2 Y_v\left(\frac{\sqrt{ab}}{k}x^k\right)\right], \qquad k = \frac{1}{2}(n+2), \quad v = \frac{1}{2k},$$

are exact solutions of the special Riccati equation as follows:

```
syms x y w k n v q a b C1 C2; k=(n+2)/2; v=1/(2*k); q=1/k*sqrt(a*b);
w=sqrt(x)*(C1*besselj(v,q*x^k)+C2*bessely(v,q*x^k));
y=-1/a*diff(w,x)/w; Test1=simplify(diff(y,x)-a*y^2-b*x^n)
```

Here $a, b, n \in \mathbb{R}$ ($ab \neq 0$, $n \neq -2$) are real parameters, $J_v(x)$ and $Y_v(x)$ are the Bessel functions, and C_1 and C_2 are arbitrary constants.

► **Finding and verification of exact solutions.**

Let us find exact solutions and verify whether these solutions are exact solutions of given ODEs.

Example 21.2. *First-order linear ODE. Finding and verification of the general solution.*
For the first-order linear ODE

$$g(x)y'_x = f_1(x)y + f_0(x),$$

we can find and verify that the solution

$$Sol1 = \mathrm{e}^{\int \frac{\mathrm{f1}(x)}{g(x)}\,\mathrm{d}x}\left(\int \frac{\mathrm{e}^{-\int \frac{\mathrm{f1}(x)}{g(x)}\,\mathrm{d}x}\,\mathrm{f0}(x)}{g(x)}\,\mathrm{d}x\right) + \mathrm{C3}\,\mathrm{e}^{\int \frac{\mathrm{f1}(x)}{g(x)}\,\mathrm{d}x}$$

presented here as the MATLAB result (for `Sol1`) is the general solution of this ODE as follows:

```
syms x g(x) y(x) f1(x) f0(x) Sol1(x);
ODE1='g(x)*Dy-f1(x)*y-f0(x)==0';
Sol1=expand(dsolve(ODE1,'x'))
pretty(Sol1)
Test1=simplify(g(x)*diff(Sol1,x)-f1(x)*Sol1-f0(x))
latex(Sol1)
```

Here $f0(x)$, $f1(x)$, and $g(x)$ are arbitrary functions, and $C3$ is an arbitrary constant.

Remark 21.2. It should be noted that in this example and in what follows, when we solve a differential equation using the predefined function `dsolve` (without specifying initial or boundary conditions), we obtain the solution with an *arbitrary parameter name* (in this case, `C3`). Since the solution of this problem has just one parameter, the name of the arbitrary constant should be `C1` (according to standard mathematical notation). We think this is an example of stylistic negligence and should be corrected in the future.

Example 21.3. *Clairaut's equation. Finding and verifying solutions.*
For Clairaut's equation

$$y = xy'_x + f(y'_x),$$

we can find and verify that

$$y(x) = Cx + f(C)$$

is the general solution of this equation as follows:

```
syms x y(x) f(x) Sol1(x);
ODE1='y-x*Dy-f(Dy)==0';
Sol1=expand(dsolve(ODE1,'x'))
Test1=simplify(Sol1-x*diff(Sol1,x)-f(diff(Sol1,x))==0)
```

Here $f(x)$ is an arbitrary function and C is an arbitrary constant.

Example 21.4. *Linear ODE of the second order. Exact explicit solution.*
The exact explicit solution

$$y(x)=\frac{C_3 M_{k,\mu}(z)\left(\frac{a-2b}{4a},-\frac{1}{4},ax-\frac{a}{2}-\frac{ax^2}{2}\right)}{\sqrt{e^{\frac{ax(x-2)}{2}}}\sqrt{x-1}}+\frac{C_4 W_{k,\mu}(z)\left(\frac{a-2b}{4a},-\frac{1}{4},ax-\frac{a}{2}-\frac{ax^2}{2}\right)}{\sqrt{e^{\frac{ax(x-2)}{2}}}\sqrt{x-1}}$$

of the second-order linear ODE

$$y''_{xx} + a(x-1)y'_x + by = 0 \qquad (a, b \in \mathbb{R})$$

can be found and tested as follows:

```
syms a b x y(x) f(x) Sol1(x);
ODE1='D2y+a*(x-1)*Dy+b*y==0';
Sol1=dsolve(ODE1,'x')
Test1=simplify(diff(Sol1,x,x)+a*(x-1)*diff(Sol1,x)+b*Sol1==0)
```

Here $M_{k,\mu}(z)$ and $W_{k,\mu}(z)$ are the Whittaker M and W functions, denoted by $\mathrm{whittakerM}(k, \mu, z)$ and $\mathrm{whittakerW}(k, \mu, z)$, respectively, in MATLAB.

Remark 21.3. In this example, we have the arbitrary constants `C3` and `C4` instead of `C1` and `C2`; for details, see Remark 21.2 (stylistic negligence of MATLAB).

► Graphical solutions.

Consider the most relevant related functions for plotting solutions of ordinary differential equations.

```
x=linspace(x1,x2,n); Y= eval(vectorize(y)); plot(x,Y,ops);
ezplot(func);                                ezplot(func,[x1,x2]);
ezplot(func,[x1,x2,y1,y2]);    ezplot(funcX,funcY,[t1,t2]);
                              fcontour(func,[x1,x2,y1,y2],ops);
```

- `linspace`, generating a linear space vector
- `vectorize`, converting symbolic objects into strings
- `eval`, evaluating strings (character arrays and symbolic objects)
- `plot`, constructing a 2-D line plot of the data in `Y` versus the corresponding values in `X`
- `ezplot`, constructing plots of the expression `func(x)` over the default domain $-2\pi < x < 2\pi$, where `func(x)` is an explicit function of x

Example 21.5. *Linear ODE of the first order. Graphical solutions.*
Graphical solutions of the linear first-order ODE

$$y'_x = y + \cos(x)x^2$$

can be generated as follows:

```
clear all; close all; echo on; format long;
syms x y(x); ODE1='Dy==y+cos(x)*x^2'; Sol1=dsolve(ODE1,'x')
x = 0:0.01:2; Y = [];
for i=-2:2 C3=i+i*2;
 Y=[Y;C3*exp(x)+((x+1).*(cos(x)+sin(x)-x.*cos(x)+x.*sin(x)))/2]; end
plot(x,Y(1,:),'k-',x,Y(2,:),'k-.',x,Y(3,:),'k--',...
 x,Y(4,:),'k.',x,Y(5,:),'k:','LineWidth',1);
grid on; xlabel('x'); ylabel('y'); title('Solutions of the
first-order linear ODE');
legend('C3=-2','C3=-1','C3=0','C3=1','C3=2');
set(gca,'FontSize',12); set(gca,'FontName','Arial');
set(gca,'LineWidth',1); shg
```

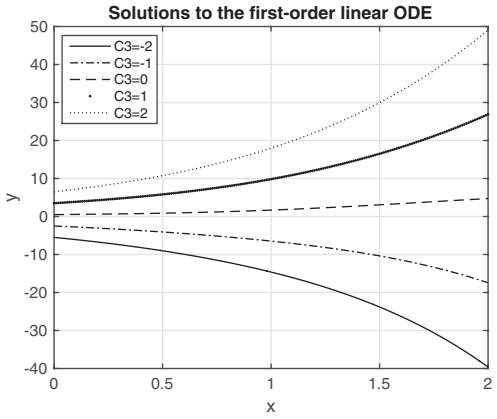

Figure 21.1: Graphical solutions of the linear equation $y'_x = y + \cos(x)x^2$.

Since we obtain the analytical solution (`Sol1`)

```
Sol1 =
C3*exp(x) + ((x + 1)*(cos(x) + sin(x) - x*cos(x) + x*sin(x)))/2,
```

where `C3` is an arbitrary constant, we generate several graphical solutions of this ODE and present them in Fig. 21.1.

Example 21.6. *Linear second-order ODE with constant coefficients. Graphical solutions.*

Graphical solutions of the linear second-order ODE with constant coefficients and with the initial conditions

$$y''_{xx} - 9y'_x + 5y = \sin x, \qquad y(0) = 0, \quad y'_x(0) = -1$$

can be generated (by using the predefined functions `plot` and `ezplot`) as follows:

```
clear all; close all; echo on; format long; syms x y(x);
ODE1='D2y-9*Dy+5*y==sin(x)'; ICs='y(0)=0,Dy(0)=-1';
Sol1=dsolve(ODE1,ICs,'x') x=linspace(0,1,40);
Y=eval(vectorize(Sol1)); figure(1); plot(x,Y) figure(2);
ezplot(Sol1,[0,1]) title('Solution of the second-order linear ODE')
```

► Constructing exact explicit and implicit solutions.

If an exact solution is given as a function of the independent variable, then the solution is said to be *explicit*. For some differential equations, explicit solutions cannot be determined; however, we can obtain an *implicit form* of the solution, i.e., an equation that involves no derivatives and relates the dependent and independent variables.

```
dsolve(ODE1,'x'); ezplot(impSol==0,[x1,x2],ops);
ezcontour(func,[x1,x2,y1,y2],name,value);
```

Example 21.7. *First-order separable ODE. Exact implicit solutions. Graphical solutions.*

For the first-order separable ODE

$$y'_x + \frac{x^2}{y} = 0, \tag{21.2.1.1}$$

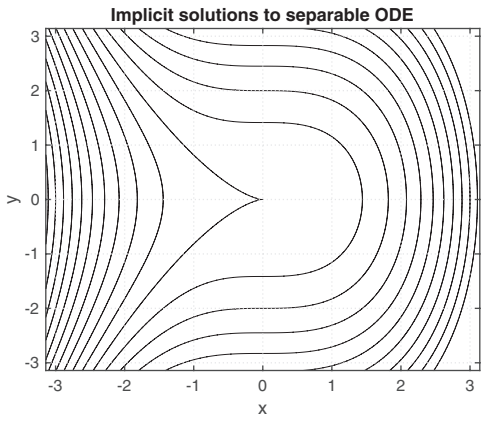

Figure 21.2: Implicit solutions of the first-order separable ODE (21.2.1.1).

we can construct the explicit (`Sol1`) and implicit (`Sol2`) solutions

$$y = \pm\sqrt{2}\sqrt{-\frac{x^3}{3} + C4}, \qquad y^2 + \frac{2}{3}x^3 - 2C4 = 0,$$

respectively, and plot the graphs of the implicit solution as follows:

```
clear all; close all; echo on; format long; syms x y(x) z; Y=[];
ODE1='Dy+x^2/y==0'; Sol1=dsolve(ODE1,'x') Sol2=z^2==Sol1(1)^2 for
i=-10:10 C4=i; Y=[Y;subs(Sol2)]; end for i=1:21
h=ezplot(Y(i),[-pi,pi]); hold on; set(h,'color',[0 0 0]); end grid
on; xlabel('x'); ylabel('y'); title('Implicit solutions of
separable ODE'); set(gca,'FontSize',12);
set(gca,'FontName','Arial'); set(gca,'LineWidth',1); shg; hold off
```

Here `C4` is an arbitrary constant.

Example 21.8. *First-order nonlinear ODE. Exact implicit solutions. Graphical solutions.*
For the first-order nonlinear ODE

$$y'_x(1 + y^2) = \sin x,$$

we can construct the implicit solution (`Sol1`)

$$Sol1 = \mathrm{RootOf}(z^3 + 3z - 3C_4 + 3\cos(x), z),$$

where the function `RootOf` represents the symbolic set of roots of the expression $z^3 + 3z - 3C_4 + 3\cos(x)$ with respect to the variable z. Also, we plot the graphs of the implicit solution (see Fig. 21.2) as follows:

```
clear all; close all; echo on; format long; syms x y(x) z; Y = [];
ODE1='Dy*(1+y^2)=sin(x)'; Sol1=dsolve(ODE1,'x') for i=-10:10 C4=i;
Y=[Y;subs(z^3+3*z-3*C4+3*cos(x)==0)]; end for i=1:21
ezplot(Y(i),[-pi,pi]); hold on; end grid on; xlabel('x');
ylabel('y'); title('Implicit solutions of nonlinear ODE'); shg;
hold off
```

Here the arbitrary constant is `C4` (instead of `C1`); see Remark 21.2. In this case (the predefined function `dsolve`), we have the warning: "Explicit solution could not be found; implicit solution returned." This result means that the solution in implicit form reads

$$y^3 + 3y - 3C4 + 3\cos x = 0.$$

► **Constructing exact solutions of higher-order ODEs.**

One can construct exact solutions of higher-order ordinary differential equations by applying the predefined function `dsolve`.

Example 21.9. *Higher-order linear homogeneous ODEs with constant coefficients.*
For the fourth-order linear homogeneous ODE with constant coefficients

$$y_x'''' + a_1 y_x''' + a_2 y_x'' + a_3 y' + a_4 y = 0,$$

where the constant coefficients are $a_1 = 1$, $a_2 = -1$, $a_3 = 5$, and $a_4 = -2$ and all solutions are of exponential form, we can determine the general solution (`Sol1`)

$$y(x) = C3e^{x/2}\cos\Big(\frac{\sqrt{7}}{2}x\Big) + C4e^{x/2}\sin\Big(\frac{\sqrt{7}}{2}x\Big) + C5e^{(\sqrt{2}-1)x} + C6e^{-(\sqrt{2}+1)x}$$

as follows:

```
clear all; close all; echo on; format long;
syms x y(x); ODE1='D4y+D3y-D2y+5*Dy-2*y==0';
Sol1=dsolve(ODE1,'x'); pretty(Sol1)
```

Example 21.10. *Higher-order linear homogeneous ODEs with nonconstant coefficients.*
For the fourth-order linear homogeneous ODE with nonconstant coefficients, the *Euler equation*

$$a_1 x^4 y_x'''' + a_2 x^3 y_x''' + a_3 x^2 y_x'' + a_4 x y' + a_5 y = 0,$$

where $a_1 = 1$, $a_2 = 14$, $a_3 = 55$, $a_4 = 65$, and $a_5 = 16$, we can determine the general solution (`Sol1`)

$$y(x) = \frac{C3\ln(x)^2}{x^2} + \frac{C4\ln(x)^3}{x^2} + \frac{C5\ln(x)}{x^2} + \frac{C6}{x^2}$$

as follows:

```
clear all; close all; echo on; format long; syms x y(x);
ODE1='x^4*D4y+14*x^3*D3y+55*x^2*D2y+65*x*Dy+16*y==0';
Sol1=dsolve(ODE1,'x'); pretty(Sol1)
```

The general solution $y = y(x)$ of a nonhomogeneous linear ODE can be written as the sum of a particular solution $y_{\rm p}(x)$ of the nonhomogeneous equation and the general solution of the corresponding homogeneous equation. The general solution of the homogeneous equation is a linear combination of the solutions in a fundamental set of solutions. The general solution of the nth-order nonhomogeneous linear ODE has the form

$$y = y_{\rm p}(x) + \sum_{i=1}^{n} C_i y_i(x), \tag{21.2.1.2}$$

where $y_i(x)$ $(i = 1, \ldots, n)$ is a fundamental set of solutions and C_i are arbitrary constants.

Example 21.11. *Higher-order linear nonhomogeneous ODEs with constant coefficients.*
Consider the fourth-order linear nonhomogeneous ODE with constant coefficients

$$y_x'''' + a_1 y_x''' + a_2 y_x'' + a_3 y' + a_4 y = \sin(x),$$

where the constant coefficients are $a_1 = 1$, $a_2 = -1$, $a_3 = 5$, $a_4 = -2$.

First, we determine the general solution of the homogeneous ODE (`SolGenHom`). Then we write out a particular solution of the nonhomogeneous equation (`SolPartNonHom`) and form the general solution of the nonhomogeneous ODE (`SolGenNonHom`) according to Eq.(21.2.1.2),

$$y(x) = C3e^{x/2}\cos\Big(\frac{\sqrt{7}}{2}x\Big) + C4e^{x/2}\sin\Big(\frac{\sqrt{7}}{2}x\Big) + C5e^{(\sqrt{2}-1)x} + C6e^{-(\sqrt{2}+1)x} - \frac{1}{4}\cos(x),$$

as follows:

```
clear all; close all; echo on; format long; syms x y(x);
ODE1='D4y+D3y-D2y+5*Dy-2*y==0';
ODE2='D4y+D3y-D2y+5*Dy-2*y==sin(x)';
SolGenHom=dsolve(ODE1,'x'); pretty(SolGenHom)
SolPartNonHom=-cos(x)/4;
SolGenNonHom=SolGenHom+SolPartNonHom; pretty(SolGenNonHom)
```

Finally, we find the general solution of the given ODE and compare the solution `SolGenNonHom` (as a result of our construction procedure) to the solution `SolGenNonHom1` (as a result of `dsolve`). It should be noted that these solutions (`SolGenNonHom` and `SolGenNonHom1`) are the same:

```
SolGenNonHom1=simplify(dsolve(ODE2,'x')); pretty(SolGenNonHom1)
```

21.2.2 Analytical Solutions of Mathematical Problems

► **Initial value problems.**

In many applications, it is required to solve an *initial value problem* or a *Cauchy problem*, i.e., a problem consisting of a differential equation supplemented with one or more initial conditions (which must be satisfied by the solutions). The number of conditions equals the order of the equation. Therefore, we have to determine a *particular solution* that satisfies the given initial conditions.

Consider some initial value problems that model various processes and phenomena [see Lin and Segel (1998)].

Example 21.12. *Malthus model. Cauchy problem. Analytical and graphical solutions.*
A basic model for population growth consists of a first-order linear ODE and an initial condition and has the form

$$y_t' = ky, \quad y(0) = y_0 \quad (k > 0),$$

where k ($k > 0$) is a constant representing the rate of growth (the difference between the birth rate and the death rate). The increase in the population is proportional to the total number of people.

We can obtain the particular solution

$$y(t) = y_0 e^{kt}$$

of this mathematical problem, which predicts exponential growth of the population, as follows:

```
clear all; close all; echo on; format long; syms k t y(t);
ODE1='Dy==k*y'; IC1='y(0)==y0';
Sol1=dsolve(ODE1,IC1,'t')
Test1=simplify(diff(Sol1,t)-k*Sol1)
```

Example 21.13. *First-order linear ODE with nonconstant coefficients. Cauchy problem.*
For the first-order linear ODE with nonconstant coefficients and with the initial condition

$$y'_x - 2y = 3x, \qquad y(0) = n, \tag{21.2.2.1}$$

we can determine the particular analytical solution (`Sol1`)

$$y(x) = -\frac{3}{2}x - \frac{3}{4} + e^{2x}\Big(n + \frac{3}{4}\Big)$$

and construct it for various values of the parameter n as follows:

```
clear all; close all; echo on; format long; syms n x y(x);
N=7; ODE1='Dy-2*y==3*x'; IC1='y(0)==n';
Sol1=dsolve(ODE1,IC1,'x')
for i=1:N n=-3+(i-1); Sols(i)=subs(Sol1); end
Sols
for i=1:N h=ezplot(eval(vectorize(Sols(i))),[0,2.5]); hold on;
 set(h,'color',[0 0 0]);
end grid on; xlabel('x'); ylabel('y'); title('Analytical solutions
of Cauchy problem'); set(gca,'FontSize',12);
set(gca,'FontName','Arial'); set(gca,'LineWidth',1); shg; hold off
```

► **Boundary value problems.**

Consider the *two-point linear boundary value problems* that consist of the second-order ODE

$$\mathcal{F}(x, y, y'_x, y''_{xx}) = 0$$

and boundary conditions at the two endpoints of an interval $[a, b]$ [e.g., see Bailey et al. (1968)]. Some (simple) boundary value problems can be solved (with the aid of MATLAB) analytically as initial value problems except that the value of the function and its derivatives are given at two values of x (the independent variable) rather than one. Note that an initial value problem has a unique solution, while a boundary value problem may have more than one solution or no solution at all.

Boundary conditions can be *homogeneous* (if the prescribed values are zero) and *nonhomogeneous* (otherwise) and can be divided into three classes, the *Dirichlet conditions*, the *Neumann boundary conditions*, and the *Robin boundary conditions*.

Example 21.14. *Second-order linear homogeneous ODE. Boundary value problem.*
For the second-order linear homogeneous ODE with constant coefficients and with the boundary conditions (the nonhomogeneous Dirichlet conditions)

$$y''_{xx} + a_1 y = 0, \qquad y(a) = g_1, \quad y(b) = g_2, \tag{21.2.2.2}$$

where $a_1 = 2$, $a = 0$, $b = \pi$, $g_1 = 1$, and $g_2 = 0$, we can determine the particular analytical solution (`Sol1`)

$$y(x) = -\frac{\cos(\sqrt{2}\pi)\sin(\sqrt{2}x)}{\sin(\sqrt{2}\pi)} + \cos(\sqrt{2}x)$$

and construct the graphical solution as follows:

```
clear all; close all; echo on; format long; syms x y(x);
ODE1='D2y+2*y==0'; BC1='y(0)==1,y(pi)==0';
Sol1=dsolve(ODE1,BC1,'x') h1=ezplot(Sol1,[0,pi]) set(h1,'color',[0
0 0]); grid on; xlabel('x'); ylabel('y'); title({'Analytical
solution of boundary value problem.',...
       'Dirichlet boundary conditions.'});
set(gca,'FontSize',12); set(gca,'FontName','Arial');
set(gca,'LineWidth',1); shg;
```

Modifying the boundary conditions (the nonhomogeneous Neumann conditions), we obtain the following:

$$y''_{xx} + a_1 y = 0, \qquad y'_x(a) = g_1, \quad y'_x(b) = g_2, \tag{21.2.2.3}$$

where $a_1 = 2$, $a = 0$, $b = \pi$, $g_1 = 1$, and $g_2 = 0$, and the particular analytical solution (`Sol2`)

$$y(x) = \frac{1}{2}\sqrt{2}\sin(\sqrt{2}x) + \frac{1}{2}\frac{\sqrt{2}\cos(\sqrt{2}\pi)\cos(\sqrt{2}x)}{\sin(\sqrt{2}\pi)}$$

can be constructed as follows:

```
clear all; close all; echo on; format long; syms x y(x);
ODE2='D2y+2*y==0'; BC2='Dy(0)==1,Dy(pi)==0';
Sol2=dsolve(ODE2,BC2,'x') h2=ezplot(Sol2,[0,pi]) set(h2,'color',[0
0 0]); grid on; xlabel('x'); ylabel('y'); title({'Analytical
solution of boundary value problem.',...
       'Neumann boundary conditions.'});
set(gca,'FontSize',12); set(gca,'FontName','Arial');
set(gca,'LineWidth',1); shg;
```

For solving more complicated boundary value problems, we can follow the numerical approach (see Section 21.3.3).

21.2.3 Analytical Solutions of Systems of ODEs

One can find analytical solutions of a given ODE system by applying the predefined function `dsolve`:

```
Y=dsolve(ODESys)                    Y=dsolve(ODESys,ICs,ops,val)
[y1,...,yN]=dsolve(ODESys)   [y1,...,yN]=dsolve(ODESys,ICs)
[y1,...,yN]=dsolve(ODESys,ICs,ops,val)
```

- `Y=dsolve(ODESys)`, solving a system of ODEs with the result being a structure array that contains the solutions
- `Y=dsolve(ODESys,ICs,ops,val)`, solving a system of ODEs with initial (or boundary) conditions and additional options (specified by one or more pair arguments `ops`, `val`)
- `[y1,...,yN]=dsolve(ODESys)`, solving a system of ODEs and assigning the solutions to the variables `y1,...,yN`
- `[y1,...,yN]=dsolve(ODESys,ICs,ops,val)`, solving a system of ODEs with initial (or boundary) conditions and additional options (specified by one or more pair arguments `ops`, `val`)

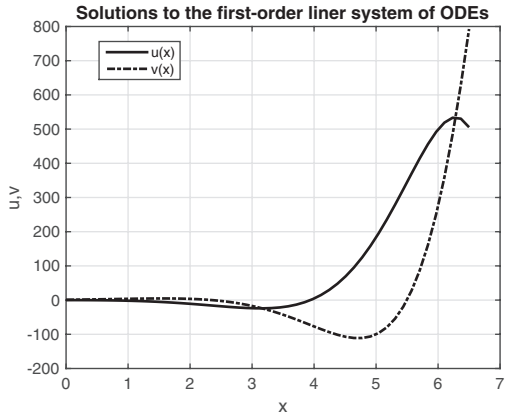

Figure 21.3: Exact solutions of the first-order linear system of ODEs (21.2.3.1).

► Linear systems of ODEs.

For first-order linear systems of ODEs, one can find the general solution and the particular solution for any initial condition (with the aid of the predefined function `dsolve`). For higher-order linear ODEs or systems of ODEs, one can convert them to a system of first-order ODEs and then solve them.

Example 21.15. *First-order two-dimensional linear system of ODEs. Analytical solution.*

Consider the general first-order two-dimensional linear system of ODEs with constant coefficients

$$u'_x = a_0 + a_1 u + a_2 v, \quad v'_x = b_0 + b_1 u + b_2 v, \tag{21.2.3.1}$$

where $u(x)$ and $v(x)$ are unknown functions and the coefficients are $a_0 = 1$, $a_1 = 1$, $a_2 = -1$, $b_0 = 1$, $b_1 = 1$, and $b_2 = 1$.

By applying the predefined function `dsolve`, we find the general solution

$$u\,(x) = -1 + \mathrm{e}^x\,(C2\,\cos(x) + C3\,\sin(x))\,,$$
$$v\,(x) = -\mathrm{e}^x\,(C3\,\cos(x) - C2\,\sin(x))$$

of this linear system; then we verify and plot it for certain values of the parameters $C2$ and $C3$ (see Fig. 21.3) as follows:

```
clear all; close all; echo on; format long; syms x u(x) v(x);
ODE1='Du==1+u-v', ODE2='Dv==1+u+v' [uS,vS]=dsolve(ODE1,ODE2,'x')
Test1=simplify(diff(uS,x)-1-uS+vS)
Test2=simplify(diff(vS,x)-1-uS-vS) C2=1; C3=-1; uSP=subs(uS),
vSP=subs(vS) x=linspace(0,6.5,50); ux=eval(vectorize(uSP));
vx=eval(vectorize(vSP)); plot(x,ux,'k-',x,vx,'k-.','LineWidth',2)
grid on; xlabel('x'); ylabel('u,v'); title('Solutions of the
first-order linear system of ODEs') legend('u(x)','v(x)');
set(gca,'FontSize',12); set(gca,'FontName','Arial');
set(gca,'LineWidth',1); shg
```

Example 21.16. *First-order two-dimensional linear system of ODEs. Cauchy problem.*

Consider the following first-order two-dimensional linear system of ODEs with initial conditions:

$$\begin{aligned} u'_x = a_0 + a_1 u + a_2 v, \quad v'_x = b_0 + b_1 u + b_2 v, \\ u(x_0) = u_0, \quad v(x_0) = v_0, \end{aligned} \tag{21.2.3.2}$$

where $u(x)$ and $v(x)$ are unknown functions and the coefficients are $a_0 = -1$, $a_1 = 1$, $a_2 = -1$, $b_0 = 1$, $b_1 = -1$, and $b_2 = 1$. For a first-order two-dimensional system in $u(x)$ and $v(x)$, each initial condition can be specified in the form `IC`$= \{u(x_0) = u_0,\ v(x_0) = v_0\}$ (e.g., $u(0) = 0$, $v(0) = 1$). One solution curve is generated for each initial condition. The solution of the initial value problem (`IVP1`) can be found as follows:

```
clear all; close all; echo on; format long; syms x u(x) v(x);
ODE1='Du==-1+u-v', ODE2='Dv==1-u+v' [uS,vS]=dsolve(ODE1,ODE2,'x')
Test1=simplify(diff(uS,x)-1-uS+vS)
Test2=simplify(diff(vS,x)-1-uS-vS) IC='u(0)==0,v(0)==1';
[uC,vC]=dsolve(ODE1,ODE2,IC,'x') simplify(uC), simplify(vC)
x=linspace(0,6.5,50); ux=eval(vectorize(uC));
vx=eval(vectorize(vC)); plot(x,ux,'k-',x,vx,'k-.','LineWidth',2)
grid on; xlabel('x'); ylabel('u,v'); title('Exact solutions of the
Cauchy problem for ODE system') legend('u(x)','v(x)');
set(gca,'FontSize',12); set(gca,'FontName','Arial');
set(gca,'LineWidth',1); shg
```

⊙ *Literature for Section .2*: G. M. Murphy (1960), P. B. Bailey, L. F. Shampine, and P. E. Waltman (1968), C. C.Lin and L. A. Segel (1998), D. Zwillinger (1997), A. D. Polyanin and A. V. Manzhirov (2007).

21.3 Numerical Solutions of ODEs

Since 2000, MATLAB has become one of the most important *problem-solving environments* (PSEs) for scientists, professors, and students.

The first implementation of numerical methods for solving ODEs, `RKF45` [see Shampine and Watts (1977, 1979)], was a FORTRAN program (based on the explicit Runge–Kutta formulas $F(4, 5)$ of Fehlberg), which is widely used in *general scientific computation* (GSC). It is the foundation of the predefined functions for solving initial value problems `rkf45` (in Maple), `NDSolve` (in Mathematica), and `ode45` (in MATLAB).

`MATLAB ODE Suite` was then developed [Shampine & Reichelt (1997)] with further evolutions [Shampine et al. (1999), Kierzenka & Shampine (2001), Shampine & Thompson (2001)]. `MATLAB ODE Suite` (replacing `ode45` since *Ver*. 5) is different in many aspects; e.g., it is based on the *explicit Runge–Kutta* $(4, 5)$ formulas, the Dormand–Prince pair.

Frequently, it is not possible to solve nonlinear (or complicated) systems of ODEs arising in realistic problems by applying analytical solution methods. In this section, we consider various numerical approximation methods for initial value problems, boundary value problems, and eigenvalue problems for ordinary differential equations.

21.3.1 Numerical Solutions via Predefined Functions

MATLAB has several predefined functions for finding numerical solutions of a given ODE problem. These predefined functions are very effective, and they have a common syntax (i.e., it is easy to use them).

► **Numerical methods embedded in MATLAB for initial value problems.**

Let us refer to numerical methods embedded in MATLAB or predefined functions for solving some type of problems as *solvers*.

The syntax, common to all solvers (predefined functions for solving initial value problems), is as follows:

```
[outputs]=SolverName(inputs)
[IndVar,DepVar]=SolverName(ODEfun,InInteg,ICs,ops)
```

- `ODEfun`, a given function containing the derivatives (specified as a scalar or vector function)
- `ICs`, initial conditions (specified as a scalar or vector)
- `InInteg`, the interval of integration (specified as a vector)
- `ops`, option structure (specified as a structure array)
- `IndVar`, evaluation points (specified as a column vector)
- `DepVar`, numerical solution (specified as an array)
- `SolverName`, one of the numerical methods embedded in MATLAB

The solvers for initial value problems implement a variety of methods. All the solvers for initial value problems of MATLAB require first-order ODEs or systems of first-order ODEs. More detailed information about numerical methods for initial value problems is presented in Table 21.1 (for variable-step solvers embedded in MATLAB) and in Table 21.2 (for fixed-step solvers available in Simulink or on the internet).

Remark 21.4. The following abbreviations in Tables 21.1–21.3 are adopted: `IVP`, initial value problem; `BVP`, boundary value problem; `BDF`, backward-differentiation formula; `IVP–DAE`, initial value problem for differential-algebraic equations; `IVP–DDE`, initial value problem for delay differential equations.

Fixed-step numerical methods for solving initial value problems are available in Simulink (for modeling and generating code for real-time systems) or on the internet.

Example 21.17. *Cauchy problem with several initial conditions.*
For the Cauchy problem (with several initial conditions)

$$y'_x = y + x^2, \qquad \{y(0) = 0,\ y(0) = 0.5,\ y(0) = 1\} \tag{21.3.1.1}$$

on the interval $[a, b]$ ($a = 0, b = 2$), we find the numerical and graphical solutions (see Fig. 21.4) as follows:

```
clear all; close all; echo on; format long; InInteg=[0 2]; y01=0;
y02=0.5; y03=1; [x,y1]=ode45(@(x,y) y+x^2,InInteg,y01);
[x,y2]=ode45(@(x,y) y+x^2,InInteg,y02); [x,y3]=ode45(@(x,y)
y+x^2,InInteg,y03);
plot(x,y1,'k-o',x,y2,'k-.',x,y3,'k--','LineWidth',1) grid on;
xlabel('x'); ylabel('y'); title({'Numerical solutions of Cauchy
problem.',...
       'Several initial conditions.'});
legend('IC1=0','IC2=0.5','IC3=1')
set(gca,'FontSize',12); set(gca,'FontName','Arial');
set(gca,'LineWidth',1); shg
```

Table 21.1.
Variable-step numerical methods for initial value problems embedded in MATLAB with brief description and some references

Numerical method	Brief description	References
`ode45`	Explicit one-step Runge–Kutta $(4, 5)$ formula, the Dormand–Prince pair. Variable step. Method for nonstiff IVP. Order of accuracy: medium (4-5). Apply as a "first step" for most problems.	Enright et al. (1986) Fehlberg (1970) Shampine and Corless (2000)
`ode23`	Explicit one-step Runge–Kutta $(2, 3)$ formula, the Bogacki–Shampine pair. Variable step. Method for nonstiff IVP. Order of accuracy: low (2-3).	Enright et al. (1986) Cash and Karp (1990) Forsythe et al. (1977)
`ode113`	Multistep Adams–Bashforth–Moulton method. Variable step. Method for nonstiff IVP. Order of accuracy: low to high (1-13).	Hairer and Wanner (1996) Shampine and Corless (2000) Forsythe et al. (1977)
`ode15s`	Implicit multistep BDF formulas (the Gear method). Method for stiff IVP (and IVP-DAEs). Variable step. Order of accuracy: low to medium (1-5). Apply if `ode45` fails (or inefficient).	Enright (1989) Verner (1978) Forsythe et al. (1977)
`ode23s`	Implicit one-step method. The modified Rosenbrock formula. Method for stiff IVP. Order of accuracy: low (2).	Hindmarsh (1983) Forsythe et al. (1977) Shampine and Corless (2000)
`ode23t`	Implicit one-step method. Method for stiff IVP. The trapezoidal rule using a "free" interpolant. Order of accuracy: low (2). IVP-DAEs can be solved with `ode23t`.	Hosea and Shampine (1996) Shampine et. al (1999)
`ode23tb`	Implicit Runge–Kutta formula with 2 stages (TR–BDF2). The first stage: trapezoidal rule (TR), the second stage: BDF of order 2 (BDF2). Method for moderately stiff IVP. Order of accuracy: low (2).	Barton et al. (1971) Forsythe et al. (1977) Shampine and Corless (2000)
`ode15i`	Order: variable (1–5). for fully implicit problems $f(x, y, y'_x) = 0$, for IVP-DAE of index 1. Order of accuracy: low.	Boyce and DiPrima (2004) Conte and de Boor (1980) Fox and Mayers (1987)

► **Numerical methods embedded in MATLAB for boundary value problems.**

One can obtain a solution of a given boundary value problem of the form

$$y'_x = f(x, y), \quad g\big(a, b, y(a), y(b)\big) = 0, \quad \text{or}$$
$$y'_x = f(x, y, p), \quad g\big(a, b, y(a), y(b), p\big) = 0,$$

where p is the vector of unknown parameters and f is a continuous function on $[a, b]$ and a Lipschitz function in y and has a continuous first derivative there.

In MATLAB, there exist two predefined functions for solving boundary value problems, `bvp4c` and `bvp5c`. These solvers have been developed by Kierzenka and Shampine [see Kierzenka and Shampine (2001)] and require first-order ODEs or systems of first-order ODEs.

The solvers `bvp4c` and `bvp4c` can solve boundary value problems with unknown parameters, multi-point boundary value problems, and a class of singular boundary value

Table 21.2.
Fixed-step numerical methods for initial value problems
with brief description and some references

Numerical method	Brief description	References
`ode1`	Explicit one-step Euler's method, Euler1. Method for nonstiff IVP. Order of accuracy: 1. Fixed-step method.	Enright et al. (1986) Fehlberg (1970) Shampine and Corless (2000)
`ode2`	Explicit one-step Heun's method, Euler2. Method for nonstiff IVP. Order of accuracy: 2. Fixed-step method.	Enright et al. (1986) Cash and Karp (1990) Forsythe et al. (1977)
`ode3`	Explicit one-step Bogacki–Shampine formula, RK3. Method for nonstiff IVP. Order of accuracy: 3. Fixed-step method.	Hairer and Wanner (1996) Shampine and Corless (2000) Forsythe et al. (1977)
`ode4`	Explicit fourth-order Runge–Kutta formula, RK4. Method for nonstiff IVP. Order of accuracy: 4. Fixed-step method.	Enright (1989) Verner (1978) Forsythe et al. (1977)
`ode5`	Explicit one-step Dormand–Prince formula, RK5. Method for nonstiff IVP. Order of accuracy: 5. Fixed-step method.	Hindmarsh (1983) Forsythe et al. (1977) Shampine and Corless (2000)
`ode87`	Explicit one-step Dormand–Prince formula, RK8(7). Method for nonstiff IVP. Order of accuracy: 8. Fixed-step method.	Hosea and Shampine (1996) Shampine et. al (1999)
`ode14x`	Implicit one-step Newton's method with extrapolation. Method for stiff IVP. Order of accuracy: variable. Fixed-step method.	Lubich (1989) Deuflhrd et al. (1987) Hairer and Wanner (1996)

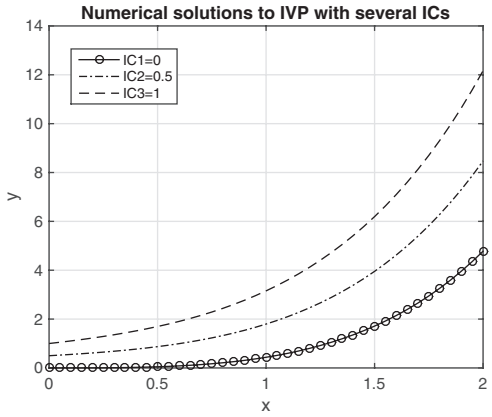

Figure 21.4: Numerical solutions of the initial value problem (21.3.1.1) with several initial conditions (`IC1`, `IC2`, `IC3`).

problems. The difference between the solvers `bvp4c` and `bvp5c` is in the meaning of error tolerances. The function `bvp5c` controls the true error $|y(x) - Y(x)|^*$ directly, and the function `bvp4c` controls the true error indirectly; i.e., it controls the discrepancy $|Y'_x - f(x, Y(x))|$.

*$Y(x)$ is an approximate solution.

Table 21.3.
Numerical methods for boundary value problems embedded in MATLAB with brief description and some references

Numerical method	Brief description	References
`bvp4c`	Implicit three-stage Lobatto IIIa formula. Order of accuracy: 4. Mesh selection, error control are based on the discrepancy. Collocation polynomials provide $C^1[a,b]$-continuous solutions.	Enright et al. (1986) Fehlberg (1970) Shampine and Corless (2000)
`bvp5c`	Implicit four-stage Lobatto IIIa formula. Order of accuracy: 5. Mesh selection, error control are based on the discrepancy. Collocation polynomials provide $C^1[a,b]$-continuous solutions.	Enright et al. (1986) Cash and Karp (1990) Forsythe et al. (1977)

More detailed information about numerical methods for boundary value problems embedded in MATLAB is presented in Table 21.3.

The syntax, common to these predefined functions for solving boundary value problems, is as follows:

```
Sol=SolverName(ODEfun,BCfun,SolIG,ops)
SolIG=bvpinit(x,yIG,params)                yk=deval(Sol,xk)
```

- `ODEfun` is a given function $f(x,y,p)$ (specified as a scalar or vector function); it can include unknown parameters p (specified as a scalar or a vector).
- `BCfun` is a function that computes the *discrepancy in the boundary conditions* (BCs). For example, for two-point boundary conditions of the form $g\big(y(a),y(b),p\big)=0$, `BCfun` can have the form `Res=BCfun(ya,yb,params)`, where `ya` and `yb` are column vectors corresponding to $y(a)$ and $y(b)$ and `Res` is a column vector.
- `IG` is a structure containing the *initial guess* for the numerical solution, where `IG.x` are ordered nodes of the initial mesh, `IG.y` is the initial guess for the solution, and `IG.parameters` is a vector for specifying the initial guess for unknown parameters. The boundary conditions are `a=IG.x(1)` and `b=IG.x(end)`. A guess for the solution at the node `IG.x(i)` is `IG.y(:,i)`. It can be formed by using the function `bvpinit` (for specifying the boundary points).
- `ops` is the option structure (specified as a structure array); it can be formed by using the function `bvpset`.
- `Sol` is the numerical solution structure, where `Sol.x` is a mesh (selected by the solver), `Sol.y` is an approximation to $y(x)$ at the mesh points, `Sol.yp` is an approximation to y'_x at the mesh points, and `Sol.parameters` are the resulting values for the unknown parameters.
- `deval` evaluates the solution at specific points `xk` on the interval $[a,b]$.
- `SolverName` is one of the numerical methods for solving boundary value problems.

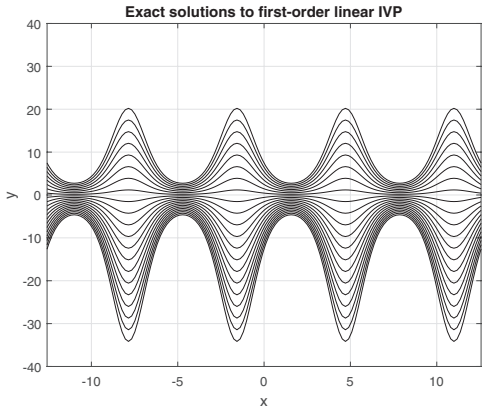

Figure 21.5: Several exact solutions of equation (21.3.2.1).

21.3.2 Initial Value Problems: Examples of Numerical Solutions

Consider some examples of initial value problems.

► Linear initial value problems.

Example 21.18. *First-order linear Cauchy problem. Analytical, numerical, graphical solutions.* For the first-order linear initial value problem

$$y'_x = -y\cos(x), \qquad y(0) = 1 \tag{21.3.2.1}$$

on the interval $[a, b]$ ($a = 0$, $b = 4\pi$), we find infinitely many solutions (`Sols`) of the ordinary differential equation and plot some of them (see Fig. 21.5). Then we obtain the unique exact solution (`Sol1`) and an approximate numerical solution (with `ode23` solver) of the Cauchy problem and plot them (see the first graph in Fig. 21.6) as follows:

```
clear all; close all; echo on; format long;
syms x y(x) z; Z=[]; ODE1='Dy==-y*cos(x)'; IC1='y(0)==1';
Sols=dsolve(ODE1,'x')
for i=-4*pi:4*pi C3=i; Z=[Z;subs(Sols)]; end
figure(1); K=21; for i=1:K h=ezplot(Z(i),[-4*pi,4*pi,-40,40]); hold on;
 set(h,'color',[0 0 0]); end
grid on; xlabel('x'); ylabel('y'); title('Exact solutions of
first-order linear initial value problem');
Sol1=dsolve(ODE1,IC1,'x') figure(2); ezplot(Sol1,[0,4*pi])
title('Exact solution of linear initial value problem'); N=46;
x=linspace(0,4*pi,N); Y=eval(vectorize(Sol1)); InInteg=[0 4*pi];
y0=1; [x,y]=ode23(@(x,y) -y*cos(x),InInteg,y0); figure(3);
plot(x,Y,'k-',x,y,'k-o'); grid on; xlabel('x'); ylabel('y');
title('Exact and numerical solutions of Cauchy problem');
legend('Exact','Numerical'); set(gca,'FontSize',12);
set(gca,'FontName','Arial'); set(gca,'LineWidth',1); shg
```

Remark 21.5. Here the MATLAB notation `'k-'` and `'k-o'` (for the predefined function `plot`) denotes the line styles of the two solutions, the solid line (`-`) of black color (`k`) for the exact solution, and the line with marker type (`-o`) of black color (`k`) for the numerical solution.

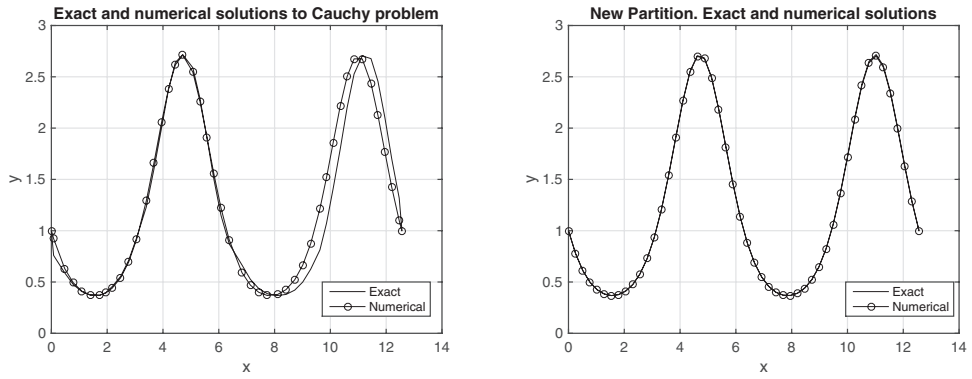

Figure 21.6: New partition. Exact and numerical solutions of the Cauchy problem (21.3.2.1).

Each numerical solver has a certain partition of the interval $[a, b]$, and we obtain a value of y at each point in this partition. For this problem, we choose the solver `ode23`, the interval of integration is $[0, 4\pi]$, and the number of points for this solver is $N = 46$. To plot the numerical and analytical solutions, we have to use the same number of points.

If we would like to increase the accuracy of our approximate solution (see the second graph in Fig. 21.6), we can specify the partition of values, e.g., $N = 50$, as follows:

```
N=50;  x=linspace(0,4*pi,N); Y=eval(vectorize(Sol1));
InInteg=0:(4/49*pi):(4*pi); y0=1;
[x,y]=ode23(@(x,y) -y*cos(x),InInteg,y0);
figure(4); plot(x,Y,'k-',x,y,'k-o'); grid on; xlabel('x'); ylabel('y');
title('New Partition. Exact and numerical solutions');
legend('Exact','Numerical'); set(gca,'FontSize',12);
set(gca,'FontName','Arial'); set(gca,'LineWidth',1); shg
```

► Nonlinear initial value problems.

Example 21.19. *First-order nonlinear Cauchy problem. Numerical and graphical solutions.* For the nonlinear initial value problem

$$y'_x = -e^{xy}\cos(x^2), \qquad y(0) = p$$

on the interval $[a, b]$ ($a = 0, b = 4\pi$), we find the numerical and graphical solutions of the problem for various initial conditions $y(0) = p$, where $p = 0.1\,i$ ($i = 1, 2, ..., 5$), as follows:

```
clear all; close all; echo on; format long;
ICs=[0.1:0.1:0.5]; InInteg=[0 4*pi];
for n=1:5 [x,Y]=ode45(@(x,y) -exp(y*x)*cos(x^2),InInteg,ICs); end
for i=1:5
 h=plot(x,Y(:,i),'k-'); hold on; set(h,'color',[0 0 0]);
end grid on; xlabel('x'); ylabel('y'); title('Numerical solutions
of nonlinear Cauchy problem'); set(gca,'FontSize',12);
set(gca,'FontName','Arial'); set(gca,'LineWidth',1); shg; hold off
```

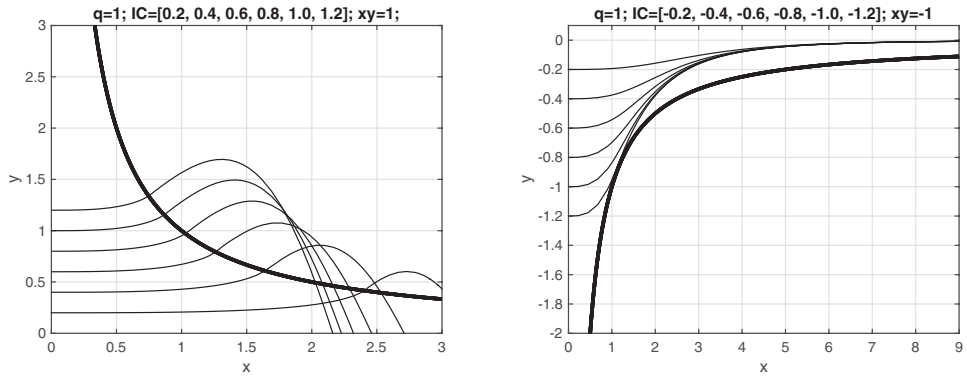

Figure 21.7: Numerical solutions of the Cauchy problem (21.3.2.2) for $q = 1$, $p > 0$ (left) and $p < 0$ (right).

Example 21.20. *First-order nonlinear Cauchy problem. Numerical and graphical solutions.* Consider the initial value problem for the nonlinear differential equation

$$y'_x = 1 - \sqrt{1 - qx^2y^2}, \qquad y(0) = p, \tag{21.3.2.2}$$

where $p \in \mathbb{R}$ and $q > 0$.

The existence domain for the solutions of this differential equation with $q > 0$ is given by the inequality $x^2y^2 \le 1/q$.

The differential equation in the Cauchy problem (21.3.2.2) has the equilibrium point $y = 0$. The solutions of the Cauchy problem for this equation with the initial condition $y(0) = p$ behave differently depending on the sign of p.

If $p < 0$, then the solutions are infinitely extendible to the right. If $p > 0$, then the solutions approach the boundary of the existence domain at some x (that is, they are not infinitely extendible to the right). Therefore, the equilibrium position $y = 0$ is unstable, because in any neighborhood of $y = 0$ there exist solutions that are not infinitely extendible.

For $q = 1$, several numerical solutions of the Cauchy problem (21.3.2.2) for various values of p are presented in Fig. 21.7 (left) for $p > 0$ and in Fig. 21.7 (right) for $p < 0$.

For example, for $p > 0$ we take the values $0.2, 0.4, 0.6, 0.8, 1.0, 1.2$, and for $p < 0$ we take the values $-0.2, -0.4, -0.6, -0.8, -1.0, -1.2$. The solutions are valid for $x \ge 0$ and are presented on the interval $[a, b]$, where $a = 0$ and $b = 3$ or $b = 9$. In these figures, we also draw the boundary $xy = \pm 1$ of the existence domain of solutions.

To generate Fig. 21.7 (left), where $q = 1$ and $p = 0.2, 0.4, 0.6, 0.8, 1.0, 1.2$, we can write the following program:

```
clear all; close all; echo on; format long;
a=0; b=3; IC=[0.2:0.2:1.2]; InInteg=[a b]; c=0; d=3; Lstyle=['-'];
for n=1:6
 [x,Y]=ode45(@(x,y) (1.-sqrt(1.-(1.).*x.^2.*y.^2)),InInteg,IC);
end
for i=1:6
  h=plot(x,Y(:,i),Lstyle);  hold on; axis([a b c d]);
  set(h,'color',[0 0 0],'linewidth',1);
    z=ezplot('1/x',[a,b]); set(z,'color',[0 0 0],'linewidth',3);
  hold on; axis([a b c d]);
end
grid on; xlabel('x'); ylabel('y');
```

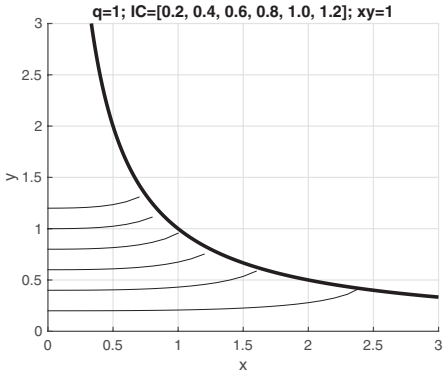

Figure 21.8: Real numerical solutions of the Cauchy problem (21.3.2.2) for $q = 1, p > 0$.

```
title('q=1; IC=[0.2, 0.4, 0.6, 0.8, 1.0, 1.2]; xy=1;');
set(gca,'FontSize',12); set(gca,'FontName','Arial');
set(gca,'LineWidth',1); shg; hold off
```

However, the solutions presented in Fig. 21.7 (left) are wrong, since the real solutions do not exist if $xy > 1$. The correct solutions for this case are presented in Fig. 21.8. There is no possibility of simple correction of this situation for all solvers for initial value problems embedded in MATLAB. Therefore, we have to write another program for this case, for example, as follows:

```
clear all; close all; echo on; format long;
a=0; b=3; IC=[0.2:0.2:1.2];  c=0; d=3; hold on;
b1=2.4028792; b2=1.6394157; b3=1.2629138; b4=1.0282819;
b5=.86598559; b6=.74669500;
IC1=0.2; IC2=0.4; IC3=0.6; IC4=0.8; IC5=1.0; IC6=1.2;
InInteg1=0:0.1:b1; InInteg2=0:0.1:b2; InInteg3=0:0.1:b3;
InInteg4=0:0.1:b4; InInteg5=0:0.1:b5; InInteg6=0:0.1:b6;
g=@(x,y) 1.-sqrt(1.-(1.).*x.^2.*y.^2);
[x1,Y1]=ode15s(g,InInteg1,IC1); [x2,Y2]=ode15s(g,InInteg2,IC2);
[x3,Y3]=ode15s(g,InInteg3,IC3); [x4,Y4]=ode15s(g,InInteg4,IC4);
[x5,Y5]=ode15s(g,InInteg5,IC5); [x6,Y6]=ode15s(g,InInteg6,IC6);
h1=plot(x1,Y1,'k-'); h2=plot(x2,Y2,'k-'); h3=plot(x3,Y3,'k-');
h4=plot(x4,Y4,'k-'); h5=plot(x5,Y5,'k-'); h6=plot(x6,Y6,'k-');
z1=ezplot('1/x',[a,b]); set(z1,'color',[0 0 0],'linewidth',3);
axis([a b c d]); grid on; xlabel('x'); ylabel('y');
title('q=1; IC=[0.2, 0.4, 0.6, 0.8, 1.0, 1.2]; xy=1');
set(gca,'FontSize',12); set(gca,'FontName','Arial');
set(gca,'LineWidth',1); shg; hold off
```

With the aid of Maple, we have evaluated the values b_i $(i = 1, \ldots, 6)$ to the right of which the solution becomes complex (see Chapter 18).

To generate Fig. 21.7 (right), where $q = 1$ and $p = -0.2, -0.4, -0.6, -0.8, -1.0, -1.2$, we can write the following program:

```
clear all; close all; echo on; format long;
a=0; b=9; IC=[-2/10 -4/10 -6/10 -8/10, -1, -12/10];
InInteg=[a b]; c=-2; d=0.1;
for n=1:6
```

```
 [x,Y]=ode45(@(x,y) (1-sqrt(1-(1)*(x.*y).^2)),InInteg,IC);
end
for i=1:6
  h=plot(x,Y(:,i),'-');  hold on; axis([a b c d]);
  set(h,'color',[0 0 0],'linewidth',1);
      z=ezplot('-1/x',[a,b]); set(z,'color',[0 0 0],'linewidth',3);
  hold on; axis([a b c d]);
end
grid on; xlabel('x'); ylabel('y');
title('q=1; IC=[-0.2, -0.4, -0.6, -0.8, -1.0, -1.2]; xy=-1');
set(gca,'FontSize',12); set(gca,'FontName','Arial');
set(gca,'LineWidth',1); shg; hold off
```

21.3.3 Boundary Value Problems: Examples of Numerical Solutions

Let us numerically solve *two-point boundary value problems*. A two-point boundary value problem includes an ODE (of order ≥ 2) and the values of the solution at two distinct points.

Consider some examples of boundary value problems applying embedded methods and constructing step-by-step solutions.

► **Linear boundary value problems.**

Example 21.21. *Second-order linear nonhomogeneous ODE with nonconstant coefficients.*

Consider a second-order linear nonhomogeneous ODE with nonconstant coefficients and with the boundary conditions

$$y''_{xx} + xy'_x + y = \cos(x), \qquad y(a) = 0, \quad y(b) = 1, \tag{21.3.3.1}$$

where $a = 0$ and $b = 2$. Numerical and graphical solutions (`solN`, `figure(1)`, and `figure(2)`) can be constructed as follows:

1. We rewrite the boundary value problem (21.3.3.1) as the first-order system

$$(y_1)'_x = y_2, \quad (y_2)'_x = \cos x - xy_2 - y_1,$$

where $y_1 = y$ and $y_2 = y'_x$, and define this system in the M-file (`bvp1.m`) as follows:

```
function dydx=bvp1(x,y); dydx=[y(2); cos(x)-x*y(2)-y(1)]; end
```

2. We write the boundary conditions in the M-file (`bc1.m`) as the residues of the boundary conditions. For the boundary conditions $y(a) = 0$ and $y(b) = 1$, the residues are $ya(1)$ and $yb(1)$. The variables ya and yb represent the solution at $x = a$ and $x = b$ respectively. The symbol 1 in parentheses indicates the first component of the vector (e.g., if we have the boundary condition $y'_x(a) = 1$, we have to write $ya(2) - 1$).

```
function res=bc1(ya,yb); res=[ya(1); yb(1)-1]; end
```

3. We solve the boundary value problem by specifying an initial guess $[y(0), y'_x(0)]$ (where $y(0)$ is known and $y'_x(0)$ is a guess) for the initial value problem and a grid of x values. Thus, a family of initial value problems is solved such that the boundary conditions are satisfied. Finally, we find the numerical solution (see Fig. 21.9) of the boundary value problem with the solver `bvp4c` as follows:

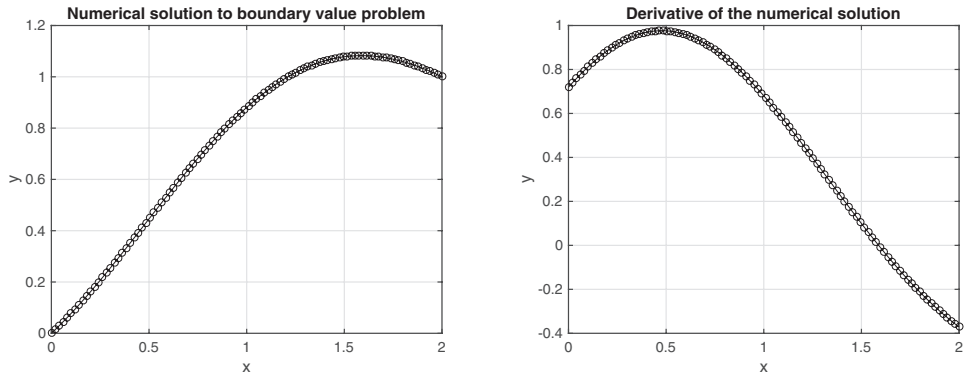

Figure 21.9: Numerical solution of the boundary value problem (21.3.3.1) and its derivative.

```
clear all; close all; echo on; format long;
solInit1=bvpinit(linspace(0,2,5),[0 0]);
solN=bvp4c(@bvp1,@bc1,solInit1); x=linspace(0,2,100);
y=deval(solN,x); figure(1); plot(x,y(1,:),'k-o'); grid on;
xlabel('x'); ylabel('y'); title('Numerical solution of boundary
value problem'); set(gca,'FontSize',12);
set(gca,'FontName','Arial'); set(gca,'LineWidth',1); shg;
figure(2); plot(x,y(2,:),'k.'); grid on; xlabel('x'); ylabel('y');
title('Derivative of the numerical solution');
set(gca,'FontSize',12); set(gca,'FontName','Arial');
set(gca,'LineWidth',1); shg;
```

Example 21.22. *Second-order linear ODE. Boundary value problem. No solution.*

Solving a boundary value problem for the second-order linear homogeneous ODE with constant coefficients

$$y''_{xx} + \pi^2 y = 0, \qquad y(a) = \alpha, \quad y(b) = \beta, \tag{21.3.3.2}$$

where $a = 0$, $b = 1$, $\alpha = 1$, and $\beta = 1$, we can find the general solution. However, the boundary conditions cannot be satisfied for any choice of the constants (see Chapter 18). Therefore, there exists no solution of this boundary value problem. In MATLAB, this can be observed with the following functions (M-files `bvp2.m` and `bc2.m`):

```
function dydx=bvp2(x,y); dydx=[y(2); -pi^2*y(1)]; end
function res=bc2(ya,yb); res=[ya(1)-1; yb(1)-1]; end
```

and the main program:

```
clear all; close all; echo on; format long;
solInit2=bvpinit(linspace(0,1,5),[0 0]);
solN=bvp4c(@bvp2,@bc2,solInit2); x=linspace(0,1,100);
solN.x, solN.y(1,:), solN.y(2,:)
```

Note that the solution is written as a structure whose first component `sol.x` gives the x values, and the second component `sol.y` of the structure is a matrix, where the first row contains the values of $y(x)$ (at the x grid points) and the second row contains the values of y'_x.

The results `solN.y(1,:)` tell us that the solution $y(x)$ is wrong (with unexpected behavior):

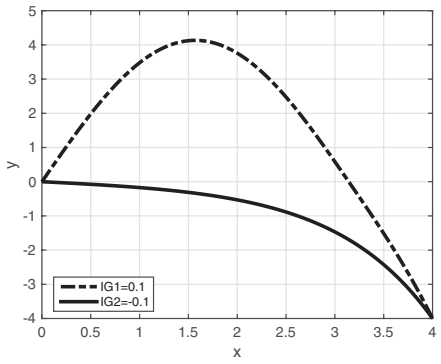

Figure 21.10: Nonuniqueness of numerical solutions of the nonlinear boundary value problem (21.3.3.3).

```
ans = 1.0e+04 *
Columns 1 through 5
0.000100000000000   0.714160289360580   1.400775911995384   2.588166088502601
3.381556897420243
Columns 6 through 10
3.589840740907997   3.660169584376011   3.589840740907997   3.381556897420243
2.588166088502601
Columns 11 through 13
1.400775911995384   0.714160289360580   0.000100000000000
```

► **Nonlinear boundary value problems.**

In addition to the nonlinear boundary value problem

$$y''_{xx} = f(x, y, y'_x), \qquad y(a) = \alpha, \quad y(b) = \beta,$$

consider the initial value problem

$$y''_{xx} = f(x, y, y'_x), \qquad y(a) = \alpha, \quad y'_x(a) = s,$$

where $x \in [a, b]$. The real parameter s describes the initial slope of the solution curve.

Let $f(x, y, u)$ be a continuous function satisfying the Lipschitz condition with respect to y and u. Then, by the Picard–Lindelöf theorem, for each s there exists a unique solution $y(x, s)$ of the above initial value problem.

To find a solution of the nonlinear boundary value problem, we choose a value of the parameter s such that $y(b, s) = \beta$; i.e., we have to solve the nonlinear equation $F(s) = y(b, s) - \beta = 0$ by applying one of the known numerical methods.

Example 21.23. *Second-order nonlinear ODE. Boundary value problem. Nonuniqueness.* Solving a boundary value problem for the second-order nonlinear ODE

$$y''_{xx} + k|y| = 0, \qquad y(a) = \alpha, \quad y(b) = \beta, \tag{21.3.3.3}$$

where $a = 0$, $b = 4$, $\alpha = 0$, $\beta = -4$, and $k = 1$, we can find two numerical solutions by applying the `bvp4c` solvers twice with distinct guess functions (`solInit1`, `solInit1`), where `IG1`$= 0.1$ and

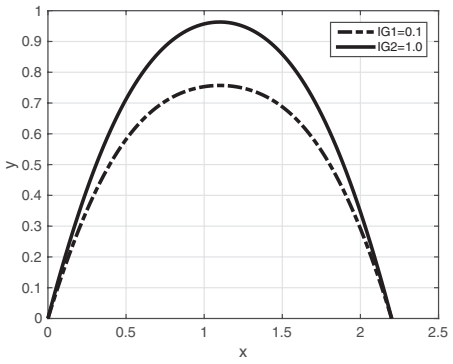

Figure 21.11: Nonuniqueness of numerical solutions of the nonlinear boundary value problem (21.3.3.4).

`IG2`$= -0.1$ are two distinct initial guesses. In MATLAB, this can be observed with the following functions (M-files `bvp3.m`, `bc3.m`):

```
function dydx=bvp3(x,y); dydx=[y(2); -1*abs(y(1))]; end
function res=bc3(ya,yb); res=[ya(1); yb(1)+4]; end
```

and the main program:

```
clear all; close all; echo on; format long;
a=0; b=4; N=100; IG1=0.1; IG2=-0.1;
solInit1=bvpinit(linspace(a,b,N),[0 IG1]);
solInit2=bvpinit(linspace(a,b,N),[0 IG2]);
solN1=bvp4c(@bvp3,@bc3,solInit1);
solN2=bvp4c(@bvp3,@bc3,solInit2);
x=linspace(a,b,N); y1=deval(solN1,x); y2=deval(solN2,x);
figure(1);
plot(x,y1(1,:),'k-.',x,y2(1,:),'k-','LineWidth',3);
grid on; xlabel('x'); ylabel('y');
legend('IG1=0.1','IG2=-0.1');
set(gca,'FontSize',12); set(gca,'FontName','Arial');
set(gca,'LineWidth',1); shg;
```

The two numerical solutions of this boundary value problem are presented in Fig. 21.10.

Example 21.24. *Second-order nonlinear ODE. Boundary value problem. Nonuniqueness.* Solving a boundary value problem for the second-order nonlinear ODE

$$y''_{xx} + k(1 + y^2) = 0; \qquad y(a) = \alpha, \quad y(b) = \beta, \tag{21.3.3.4}$$

where $a = 0, b = 3, \alpha = 0, \beta = 0$, and $k = 2$, we can find two positive numerical solutions by applying the `bvp5c` solvers twice with distinct guess functions (`solInit1` and `solInit1`), where `IG1`$=$ 0.1 and `IG2`$=$ 1.0 are two distinct initial guesses. In this case, we apply the other MATLAB solver (`bvp5c`), since the solver `bvp4c` does not approach any reasonable accuracy (e.g., $1e-1$). Also, we include some options (see function `odeset`), e.g., `NonNegative`, `RelTol`, and `AbsTol`. In MATLAB, this can be observed with the following functions (M-files `bvp4.m`, `bc4.m`):

```
function dydx=bvp4(x,y); dydx=[y(2); -2*(1+y(1).^2)]; end
function res=bc4(ya,yb); res=[ya(1); yb(1)]; end
```

and the main program:

```
clear all; close all; echo on; format long;
a=0; b=2.2; N=100; IG1=0.1; IG2=1.;
options=odeset('NonNegative',1,'RelTol',1e-1,'AbsTol',1e-1);
solInit1=bvpinit(linspace(a,b,N),[0 IG1]);
solInit2=bvpinit(linspace(a,b,N),[0 IG2]);
solN1=bvp5c(@bvp4,@bc4,solInit1,options);
solN2=bvp5c(@bvp4,@bc4,solInit2,options);
x=linspace(a,b,N); y1=deval(solN1,x); y2=deval(solN2,x);
figure(1);
plot(x,y1(1,:),'k-.',x,y2(1,:),'k-','LineWidth',3);
grid on; xlabel('x'); ylabel('y');
legend('IG1=0.1','IG2=1.0');
set(gca,'FontSize',12); set(gca,'FontName','Arial');
set(gca,'LineWidth',1); shg;
```

The two numerical solutions of this boundary value problem are presented in Fig. 21.11.

21.3.4 Eigenvalue Problems: Examples of Numerical Solutions

Consider *eigenvalue problems*, i.e., boundary value problems that include a real parameter. Discrete values of the parameter that satisfy the ODE are called *eigenvalues* of the problem. For each eigenvalue λ_n, there exists a *nontrivial* solution $y_n(x)$ that satisfies the problem, and it is called the *eigenfunction* associated with λ_n. The set of real eigenvalues is infinite, and the set of eigenfunctions is complete.

Consider the *Sturm–Liouville system* or *Sturm–Liouville eigenvalue problem*, i.e., the second-order linear homogeneous differential equation

$$\big(p(x)y'_x\big)'_x + \big(q(x) + \lambda w(x)\big)y = 0, \qquad a < x < b,$$

together with the boundary conditions

$$\beta_1 y(a) + \beta_2 y'_x(a) = 0, \quad \beta_3 y(b) + \beta_4 y'_x(b) = 0.$$

If $p(x)$, $q(x)$, and $w(x)$ are continuous functions and if both $p(x)$ and $w(x)$ are positive on $[a, b]$, then the *Sturm–Liouville eigenvalue problem* is called *regular*. Let us find the eigenvalues and eigenfunctions for some regular Sturm–Liouville eigenvalue problems.

Example 21.25. *Sturm–Liouville problem. Homogeneous Dirichlet boundary conditions.*
We solve the Sturm–Liouville eigenvalue problem

$$y''_{xx} + \lambda y = 0, \qquad y(a) = 0, \quad y(b) = 0, \tag{21.3.4.1}$$

i.e., a homogeneous linear two-point boundary value problem with a parameter λ and with the homogeneous Dirichlet boundary conditions,* where $a \le x \le b$, $a = 0$, $b = \pi$, $p(x) = 1$, $w(x) = 1$, and $q(x) = 0$, by applying the predefined function `bvp4c`. In MATLAB, this eigenvalue problem can be solved, e.g., for the first two eigenvalues and the corresponding eigenfunctions with the following functions (M-files `evp1.m`, `evp1bc1.m`, `evp1Guess1`, `evp1bc2.m`, and `evp1Guess2`):

*These boundary conditions are called *separated conditions*, for which there exist a complete set of orthogonal eigenfunctions.

```
function dydx=evp1(x,y,lambda); dydx=[y(2);-lambda*y(1)]; end
function res=evp1bc1(ya,yb,lambda); res=[ya(1);yb(1);yb(2)+1]; end
function res=evp1bc2(ya,yb,lambda); res=[ya(1);yb(1);yb(2)-1]; end
function v=evp1Guess1(x); v=[sin(x);cos(x)]; end
function v=evp1Guess2(x); v=[cos(x);sin(x)]; end
```

and the main program:

```
clear all; close all; echo on; format long;
a=0; b=pi; N=10; lambda=0.5;
solInit1=bvpinit(linspace(a,b,N),@evp1Guess1,lambda);
solInit2=bvpinit(linspace(a,b,N),@evp1Guess2,lambda);
solN1=bvp4c(@evp1,@evp1bc1,solInit1);
solN2=bvp4c(@evp1,@evp1bc2,solInit2);
fprintf('The first eigenvalue = %15.6f.\n',solN1.parameters)
fprintf('The second eigenvalue = %15.6f.\n',solN2.parameters)
x=linspace(a,b); y1=deval(solN1,x); y2=deval(solN2,x);
figure(1); plot(x,y1(1,:),'k-','LineWidth',3);
axis([0 pi 0 1]); grid on; xlabel('x'); ylabel('y');
figure(2); plot(x,y2(1,:),'k-','LineWidth',3);
axis([0 pi -0.6 0.6]); grid on; xlabel('x'); ylabel('y');
set(gca,'FontSize',12); set(gca,'FontName','Arial');
set(gca,'LineWidth',1); shg;
```

Note that the predefined function `bvp4c` makes it easy to solve Sturm–Liouville problems involving unknown parameters. If there are unknown parameters, we have to include estimates for them (as the third argument of `bvpinit`). Also, we have to include the vector of unknown parameters (as the third argument of the functions for evaluating the ODEs and the discrepancy in the boundary conditions). If there are unknown parameters, then the solution structure (in our problem, `solN1`, `solN2`) has the *parameters* field, which contains the vector of parameters computed by the solver `bvp4c`.

When solving boundary value problems with `bvp4c`, we have to provide a *guess for the solution*. The guess is included in `bvp4c` as a structure formed by the function `bvpinit`. The first argument of `bvpinit` is a *guess for the mesh*. In our problem, we try 10 equally spaced points in $[0, \pi]$. The second argument is a *guess for the solution* on the specified mesh. In our problem, the solution has two components, $y(x)$ and y'_x, and we try a *function guess*, e.g., $[\sin(x), \cos(x)]$. We guess that λ is about 0.5.

As a result, we have the first eigenvalue $\lambda_1 \approx 1.000028$ and the second eigenvalue $\lambda_2 \approx 4.000130$.

The first two eigenfunctions of this Sturm–Liouville problem are presented in Fig. 21.12.

Example 21.26. *Sturm–Liouville problem. Homogeneous Neumann boundary conditions.*
We solve the Sturm–Liouville eigenvalue problem

$$y''_{xx} + \lambda y = 0, \qquad y'_x(a) = 0, \quad y'_x(b) = 0, \tag{21.3.4.2}$$

i.e., a homogeneous linear two-point boundary value problem with the parameter λ and with the homogeneous Neumann boundary conditions, where $a \le x \le b$, $a = 0$, $b = \pi$, $p(x) = 1$, $w(x) = 1$, and $q(x) = 0$, by applying the predefined function `bvp4c`. In MATLAB, this eigenvalue problem can be solved, e.g., for the second and third eigenvalues and the corresponding eigenfunctions, with the following functions (M-files `evp2.m`, `evp2bc1.m`, `evp2Guess1`, `evp2bc2.m`, and `evp2Guess2`):

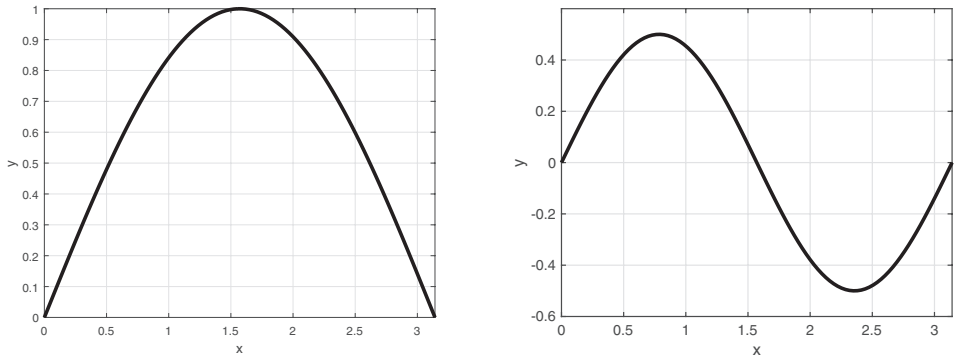

Figure 21.12: The first two eigenfunctions of the Sturm–Liouville problem (21.3.4.1).

```
function dydx=evp2(x,y,lambda); dydx=[y(2);-lambda*y(1)]; end
function res=evp2bc1(ya,yb,lambda); res=[ya(2);yb(2);yb(1)+1]; end
function res=evp2bc2(ya,yb,lambda); res=[ya(2);yb(2);yb(1)-1]; end
function v=evp2Guess1(x); v=[cos(x);sin(x)]; end
function v=evp2Guess2(x); v=[sin(x);cos(x)]; end
```

and the main program:

```
clear all; close all; echo on; format long;
a=0; b=pi; N=10; lambda1=0.5; lambda2=3.5;
solInit1=bvpinit(linspace(a,b,N),@evp2Guess1,lambda1);
solInit2=bvpinit(linspace(a,b,N),@evp2Guess2,lambda2);
solN1=bvp4c(@evp2,@evp2bc1,solInit1);
solN2=bvp4c(@evp2,@evp2bc2,solInit2);
fprintf('The second eigenvalue = %15.6f.\n',solN1.parameters)
fprintf('The third eigenvalue = %15.6f.\n',solN2.parameters)
x=linspace(a,b); y1=deval(solN1,x); y2=deval(solN2,x);
figure(1); plot(x,y1(1,:),'k-','LineWidth',3);
axis([0 pi -1 1]); grid on; xlabel('x'); ylabel('y');
figure(2); plot(x,y2(1,:),'k-','LineWidth',3);
axis([0 pi -1 1]); grid on; xlabel('x'); ylabel('y');
set(gca,'FontSize',12); set(gca,'FontName','Arial');
set(gca,'LineWidth',1); shg;
```

As a result, we have the second eigenvalue $\lambda_2 \approx 1.000028$ and the third eigenvalue $\lambda_3 \approx 4.000130$. The second and third eigenfunctions of this Sturm–Liouville problem are presented in Fig. 21.13.

⊙ *Literature for Section .3*: E. Fehlberg (1970) D. Barton, I. M. Willer, and R. V. M. Zahar (1971) G. E. Forsythe, M. A. Malcolm, and C. B. Moler (1977), L. F. Shampine and H. A. Watts (1977), J. H. Verner (1978), L. F. Shampine and H. A. Watts (1979), S. D. Conte and C. de Boor (1980), A. C. Hindmarsh (1983), H. W. Enright, K. R. Jackson, S. P. Norsett, P. G. Thomsen (1986), L. Fox and D. F. Mayers (1987), P. Deuflhard, B. Fiedler, P. Kunkel (1987), Ch. Lubich (1989), W. H. Enright (1989), J. R. Cash and A. H. Karp (1990), M. E. Hosea and L. F. Shampine (1996), E. Hairer and G. Wanner (1996), L. F. Shampine and M. W. Reichelt (1997), L. F. Shampine et. al (1999), L. F. Shampine, M. W. Reichelt, and J. Kierzenka (1999), L. F. Shampine and R. M. Corless (2000), J. Kierzenka and L. F. Shampine (2001), L. F. Shampine and S. Thompson (2001), W. E. Boyce and R. C. DiPrima (2004).

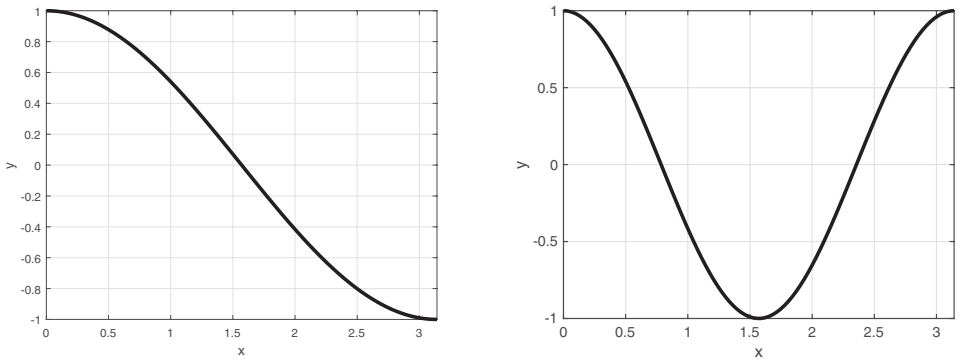

Figure 21.13: The second and third eigenfunctions of the Sturm–Liouville problem (21.3.4.2).

21.4 Numerical Solutions of Systems of ODEs

In this section, we numerically solve initial value problems for systems of differential equations of various classes in MATLAB [e.g., see Murphy (1960), Lapidus et al.(1973), Kamke (1977), MacDonald (1989), Lambert (1991), Zwillinger (1997), Polyanin and Manzhirov (2007)]. We consider the following classes of ODE systems: first-order linear and nonlinear systems of two ODEs, higher-order ODEs with transformations to first-order systems of ODEs, first-order systems of general form, and second-order systems. To this end, we define differential systems in M-files.

21.4.1 First-Order Systems of Two Equations

Consider the system of two first-order ordinary differential equations with the initial conditions

$$u'_x = f_1(x,u,v), \quad v'_x = f_2(x,u,v), \qquad u(a) = u_0, \quad v(a) = v_0.$$

The unknown functions are $u(x)$ and $v(x)$, and $x \in [a,b]$.

To obtain numerical solutions, we can apply predefined functions or, alternatively, construct solutions step by step by applying known numerical methods to each equation of the system.

Let us numerically solve some first-order systems of two differential equations (linear and nonlinear).

► **First-order linear systems.**

Example 21.27. *First-order linear system. Exact, numerical, and graphical solutions.*
For the first-order linear system with the initial conditions

$$u'_x = v, \quad v'_x = x - u - 2v, \qquad u(a) = \alpha, \quad v(a) = \beta, \tag{21.4.1.1}$$

where $a \le x \le b$, $a = 0$, $b = 2$, $\alpha = 1$, and $\beta = 1$, we compute the numerical solution `Y` (where `Y(:,1)` and `Y(:,2)` are u and v, respectively) by applying the predefined function `ode45`, com-

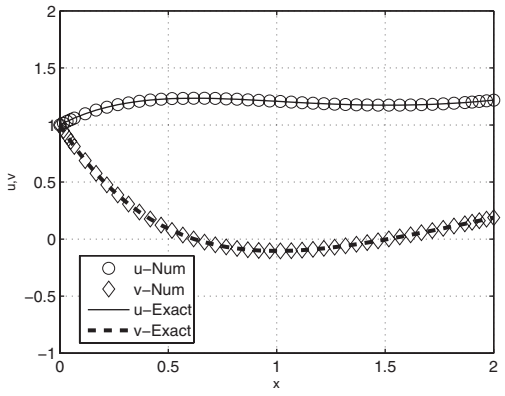

Figure 21.14: The numerical and exact solutions of the linear Cauchy problem (21.4.1.1).

pare the numerical results with the exact solution (`uEx`, `vEx`)*, and plot the exact and numerical solutions. We write the MATLAB M-file containing the differential system (`sys1.m`):

```
function Yprime=sys1(x,Y); Yprime=[Y(2);x-Y(1)-2*Y(2)]; end
```

and the main program:

```
clear all; close all; echo on; format long;
Y0=[1 1]; X=[0,2]; [x,Y]=ode45(@sys1,X,Y0)
plot(x,Y(:,1),'ko','MarkerSize',9);
hold on; grid on; axis([0 2 -1 2]);
plot(x,Y(:,2),'kd','MarkerSize',9);
uEx=x+(3.*x+3).*exp(-x)-2; vEx=-3.*x.*exp(-x)+1;
plot(x,uEx,'k-','LineWidth',1); plot(x,vEx,'k--','LineWidth',3);
grid on; xlabel('x'); ylabel('u,v');
legend('u-Num','v-Num','u-Exact','v-Exact');
set(gca,'FontSize',12); set(gca,'FontName','Arial');
set(gca,'LineWidth',1); shg; hold off;
```

► First-order nonlinear systems.

Example 21.28. *First-order nonlinear system. Numerical and graphical solutions.*
For the first-order nonlinear system with the initial conditions

$$u'_x = uv, \quad v'_x = u + v, \qquad u(a) = \alpha, \quad v(a) = \beta, \tag{21.4.1.2}$$

where $a = 0$, $b = 1$, $\alpha = 1$, and $\beta = 1$, we obtain the numerical solution `Y` (where `Y(:,1)` and `Y(:,2)` are u and v, respectively) by applying the predefined function `ode45` and plot the results. We write the MATLAB M-file containing the differential system (`sys2.m`):

```
function Yprime=sys2(x,Y); Yprime=[Y(1)*Y(2);Y(1)+Y(2)]; end
```

and the main program:

*For more details, see Chapter 18.

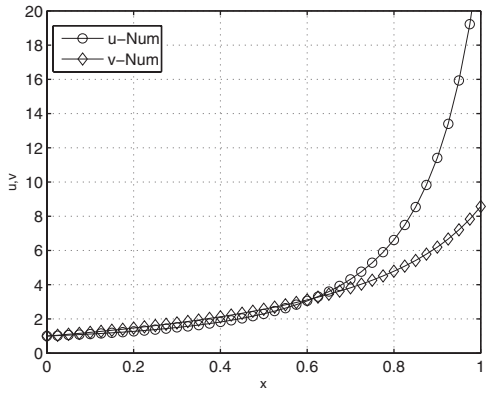

Figure 21.15: The numerical solution of the nonlinear Cauchy problem (21.4.1.2).

```
clear all; close all; echo on; format long;
Y0=[1 1]; X=[0,1]; [x,Y]=ode45(@sys2,X,Y0)
plot(x,Y(:,1),'k-o','MarkerSize',7); hold on;
plot(x,Y(:,2),'k-d','MarkerSize',7); axis([0 1 0 20]);
grid on; xlabel('x'); ylabel('u,v'); legend('u-Num','v-Num');
set(gca,'FontSize',12); set(gca,'FontName','Arial');
set(gca,'LineWidth',1); shg; hold off;
```

► **Higher-order ODEs.**

If we consider an ordinary differential equation of order n ($n > 1$) with n initial conditions

$$\begin{aligned} &y_x^{(n)} = f(x, y, y'_x, \ldots, y_x^{(n-1)}) \qquad (x \in [a, b]), \\ &y(a) = y_0, \quad y'_x(a) = y_1, \quad \ldots, \quad y_x^{(n-1)}(a) = y_n, \end{aligned}$$

then we can always obtain solutions of this higher-order differential equation by transforming it to an equivalent system of n first-order differential equations and by applying an appropriate numerical method (e.g., `ode45`) to this system of differential equations. To this end, we can apply the predefined function `odeToVectorField` and then generate a MATLAB function from the symbolic expression obtained (i.e., the system of first-order differential equations) by applying the predefined function `matlabFunction`:

```
    V=odeToVectorField(eqn1,...,eqnN)
[V,Y]=odeToVectorField(eqn1,...,eqnN)
 mFun=matlabFunction(V,'vars',{'x','Y'})
```

where `V` is a symbolic vector representing the resulting system of first-order differential equations, `Y` is a symbolic vector representing the substitutions made during the transformation, and `M` is a MATLAB function generated from a symbolic expression.

Example 21.29. *Van der Pol equation. Cauchy problem. Numerical and graphical solutions.*

For the van der Pol equation with the initial conditions

$$y''_{xx} + \mu(y^2 - 1)y'_x + y = 0, \quad y(a) = \alpha, \quad y'_x(a) = \beta, \tag{21.4.1.3}$$

where $x \in [a, b]$, $a = 0$, $b = 60$, $\alpha = 1$, $\beta = 0$, and $\mu = \frac{1}{8}$, we can compute a numerical solution as follows:

1. By applying the predefined function `odeToVectorField`, we transform the second-order ODE into an equivalent system of two first-order differential equations (`Sys1`):

```
Sys1=[Y[2]; -Y[1]-Y[2]*(Y[1]^2/8-1/8)].
```

2. By applying the predefined function `matlabFunction`,* we generate a MATLAB function (`mFun`) from this system of first-order differential equations:

```
mFun=@(x,Y)[Y(2); -Y(1)-Y(2).*(Y(1).^2.*(1.0./8.0)-1.0./8.0)].
```

3. By applying the MATLAB numerical solver `ode45`

```
solN=ode45(mFun,[a b],IC)
```

to this system of differential equations, we obtain a numerical solution (`solN`) and graphical solutions, a phase portrait of the solution, and a graph of $u(x)$ and $v(x)$ (see Fig. 21.16) as follows:

```
clear all; close all; echo on; format long;
a=0; b=60; N=500; alpha=1; beta=0; IC=[1 0];
syms x y(x) Sys1
Sys1=odeToVectorField(diff(y,2)+(1/8)*(y^2-1)*diff(y)+y==0)
mFun=matlabFunction(Sys1,'vars',{'x','Y'})
solN=ode45(mFun,[a b],IC);
x=linspace(a,b,N); yN=deval(solN,x);
figure(1);
plot(yN(1,:),yN(2,:),'k-','LineWidth',1);
grid on; xlabel('u'); ylabel('v'); axis([-2.5 2.5 -2.5 2.5]);
figure(2);
plot(x,yN(1,:),'k-','LineWidth',1); hold on;
plot(x,yN(2,:),'k-.','LineWidth',1); axis([a b -2.5 2.5]);
grid on; xlabel('x'); ylabel('u,v'); legend('u(x)','v(x)');
set(gca,'FontSize',12); set(gca,'FontName','Arial');
set(gca,'LineWidth',1); shg; hold off;
```

21.4.2 First-Order Systems of General Form

Consider the first-order system of differential equations of general form with the initial conditions

$$(y'_x)_i = f_i\big(x, y_1, \ldots, y_n\big), \quad y_i(a) = (y_0)_i \quad (i = 1, \ldots, n).$$

The unknown functions are $y_1(x), \ldots, y_n(x)$, and $x \in [a, b]$. To obtain numerical solutions, we can apply predefined functions or, alternatively, construct solutions step by step by applying known numerical methods.

As an example, consider the *Lorenz system*, which is a dissipative chaotic system with a strange attractor. These features can be observed for certain values of the system parameters

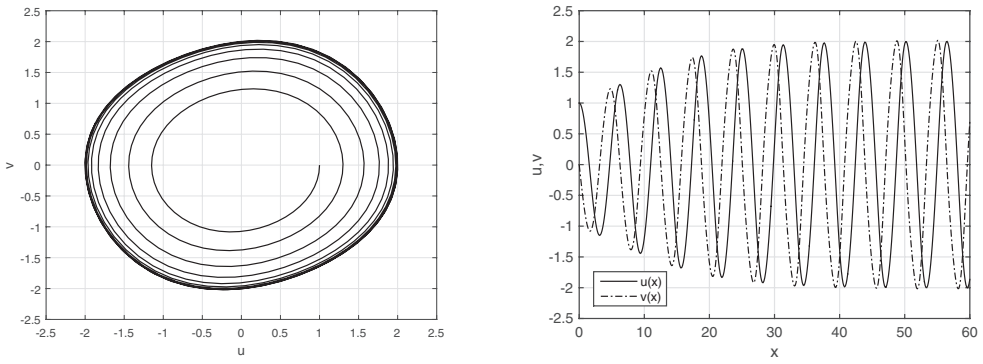

Figure 21.16: Graphical solutions of the van der Pol equation $y''_{xx} + \mu(y^2 - 1)y'_x + y = 0$ (the equivalent system of two first-order ODEs).

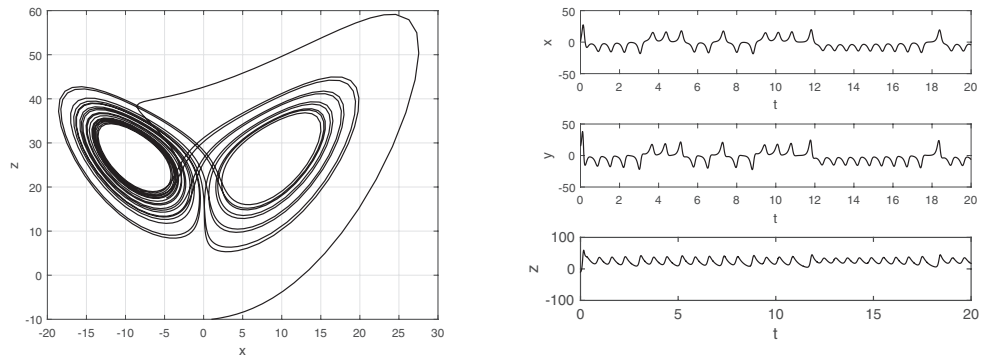

Figure 21.17: Graphical solutions of the Lorenz system (21.4.2.1).

and initial conditions. This system models an unstable thermally convecting fluid (heated from below) and also arises in other simplified models.

Example 21.30. *The Lorenz system. Cauchy problem. Numerical and graphical solutions.* For the nonlinear system of first-order ODEs, e.g., the Lorenz system [see Sparrow (1982)],

$$\begin{gathered} x'_t = \sigma(y - x), \quad y'_t = \rho x - y - xz, \quad z'_t = xy - \beta z, \\ x(0) = 1, \quad y(0) = 15, \quad z(0) = 10, \end{gathered} \tag{21.4.2.1}$$

where σ, ρ, and β are the system parameters, one can investigate the behavior of the system by varying the system parameters σ, ρ, and β and observe the strange attractor.

We obtain the numerical solution `W` (where `W(:,1)`, `W(:,2)`, and `W(:,3)` are $x(t)$, $y(t)$, and $z(t)$), respectively, by applying the predefined function `ode45` and plot the results.
We write the MATLAB M-file containing the differential system (`sys3.m`):

```
function Wprime=sys3(t,W); sigma=15; beta=3; rho=28;
 Wprime=[sigma*(W(2)-W(1);rho*W(1)-W(2)-W(1)*W(3);-beta*W(3)+W(1)*W(2)];
end
```

*The MATLAB predefined functions for solving initial value problems do not accept symbolic expressions (as an input), and so we have to convert the system obtained to a MATLAB function.

and the main program:

```
clear all; close all; echo on; format long;
a=0; b=20; W0=[1 15 -10]; t=[a,b];
[t,W]=ode45(@sys3,t,W0)
figure(1);
plot(W(:,1),W(:,3),'k-','LineWidth',1);
grid on; xlabel('x'); ylabel('z');
figure(2);
subplot(3,1,1); plot(t,W(:,1),'k-','LineWidth',1)
xlabel('t'); ylabel('x');
subplot(3,1,2); plot(t,W(:,2),'k-','LineWidth',1)
xlabel('t'); ylabel('y');
subplot(3,1,3); plot(t,W(:,3),'k-','LineWidth',1)
xlabel('t'); ylabel('z');
set(gca,'FontSize',12); set(gca,'FontName','Arial');
set(gca,'LineWidth',1); shg;
```

The Lorenz strange attractor (the plot of z versus x) and each component of the solution (x, y, z as functions of t) are presented in Fig. 21.17.

Remark 21.6. If a given problem (including a single differential equation or a system of differential equations) is of order 2 or higher, we have to convert this problem into an equivalent system of first-order differential equations and apply an appropriate numerical method (e.g., `ode45`) to this system of differential equations. For this conversion, we can apply the predefined function `odeToVectorField` and then generate a MATLAB function from the symbolic expression obtained (i.e., a system of first-order differential equations) by applying the predefined function `matlabFunction`.

⊙ *Literature for Section .4*: . M. Murphy (1960), L. Lapidus, R. C. Aiken, and Y. A. Liu (1973), E. Kamke (1977), C. Sparrow (1982), N. MacDonald (1989), J. D. Lambert (1991), D. Zwillinger (1997), A. D. Polyanin and A. V. Manzhirov (2007).