Chapter 4
Numerical Methods

Abstract The purpose of this chapter is to provide a brief introduction as to how a first- or second-order differential equation may be solved to the desired precision by using numerical methods like Euler's method and fourth-order Runge–Kutta method. Emphasis is placed on the difference between an analytical and a numerical solution. Movement of a pendulum for an arbitrary amplitude is calculated numerically to exemplify how easily some problems can be solved by numerical methods. Methods for solving partial differential equations are also described, but are not used until a later chapter. The importance of testing, reproducibility and documentation of successive program versions are discussed. Specimen programs are given at the end of the chapter.

4.1 Introductory Remarks

During my student days (1969–1974), Norway's largest computer had a memory capacity (RAM) of 250 kB and it filled a whole room. We made programs by punching holes in a card, one card for each line (see Fig. 4.1). The pack of cards was carried carefully to a separate building; Abel's House (it was a disaster to drop the pack). A waiting period of a few hours up to a whole day passed before we could collect the result in the form of a printout on perforated pages. A punching error meant that a card had been punched again so that the wrong card in the stack could be exchanged with the new card. This was followed by a new submission and another waiting period. Guess if debugging a program took an eternity! Today, the situation is totally different. Everyone owns a computer. Program development is incomparably easier and far less time-consuming than in earlier times. And numerical methods have become a tool as natural as analytical mathematics.

But all tools have one thing in common: training is needed in how they are to be used. In this chapter, our primary concern will be to see how the equation of motion for an oscillating system and the wave equation can be solved in a satisfactory manner. It is not enough to read how things can be done. Practice is needed for acquiring the requisite skills and mastering the routine.

A. I. Vistnes, *Physics of Oscillations and Waves*, Undergraduate Texts in Physics,
https://doi.org/10.1007/978-3-319-72314-3_4

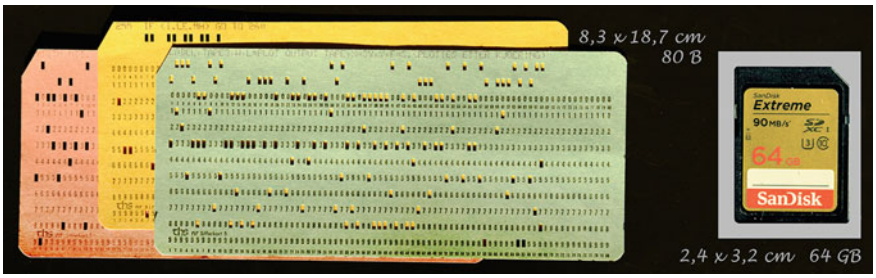

Fig. 4.1 Examples of punch cards, along with a modern memory device (sizes indicated) with storage capacity equivalent to 800 million punch cards (which would have weighed 1900 tons!). The memory device weighs about 0.5 g

Parts of the chapter were written by David Skålid Amundsen as a summer job for CSE 2008. Amundsen's text has since been revised and expanded several times by Arnt Inge Vistnes.

4.2 Introduction

When in the "old days" (i.e. more than 30 years ago), we investigated the motion of a mathematical or physical pendulum in a lower-level physics course, we had to be content with "small amplitudes". At that time, with only the rudiments of analytical mathematics in our toolkit, we could only proceed by imposing the approximation of small displacements, which implied that the movement is a simple harmonic motion. Larger amplitudes are much more difficult to handle analytically, and if we consider complicated friction as well, there is simply no analytical solution to the problem.

Once we have learned to use numerical methods of solution, it is often almost as easy to use a realistic, nonsimplified description of a moving system as an idealized simplified description.

This book is based on the premise that the reader already knows something about solving, for example, differential equations with the aid of numerical methods. Nevertheless, we make a quick survey of some of the simplest solution methods so that those who have no previous experience with numerical methods would nonetheless be able to keep pace with the rest. After the quick review of some simple methods, we spend a little more time on a more robust alternative. Additionally, we will say a little about how these methods can be generalized to solve partial differential equations.

It should be mentioned here that the simplest numerical methods are often good enough for calculating, for example, the motion of a projectile, even in the presence of air resistance. However, the simplest methods often accumulate errors and give quite a bad result for oscillatory motion. In other words, it is often necessary to use some advanced numerical methods in dealing with oscillations and waves.

This chapter is structured along the following lines:

First, a quick review of the simplest numerical methods used for solving differential equations is given. Secondly, the fourth-order Runge–Kutta's method is presented. This first part of the chapter is rather mathematical. Then comes a practical example, and finally, we will include examples of program codes that can be used for solving the problems given in later chapters.

4.3 Basic Idea Behind Numerical Methods

In many parts of physics, we come across the second-order ordinary differential equations:

$$\ddot{x} \equiv \frac{\mathrm{d}^2 x}{\mathrm{d}t^2} = f\big(x(t), \dot{x}(t), t\big) \,. \tag{4.1}$$

with the initial conditions $x(t_0) = x_0$ and $\dot{x}(t_0) = \dot{x}_0$. The symbol $f\big(x(t), \dot{x}(t), t\big)$ means that f (for the case when x is the position variable and t the time) is a function of time, position and velocity.

In mechanical systems, differential equation often arises when Newton's second law is invoked. In electrical circuitry containing resistors, inductors and capacitors, it is often Kirchhoff's law together with the generalized Ohm's law and complex impedances that are the source of differential equations.

When we solve second-order differential equations numerically, we often consider the equation as a combination of two coupled first-order differential equations. We rename then the first derivative and let this be a new variable:

$$v \equiv \frac{\mathrm{d}x}{\mathrm{d}t} \,.$$

The two coupled first-order differential equations then becomes:

$$\frac{\mathrm{d}x}{\mathrm{d}t} = v\big(x(t), t\big) \,,$$

$$\frac{\mathrm{d}\dot{v}}{\mathrm{d}t} = f\big(x(t), v(t), t\big) \,.$$

We will shortly see some simple examples of this in practice.

4.4 Euler's Method and Its Variants

We can solve a first-order differential equation numerically by specifying a starting value for the solution we are interested in, using our knowledge of the derivative of the function to calculate the solution for a short time Δt afterwards. We then let the new value act as a new initial value to calculate the value that follows Δt after this (that is, at $t = 2\Delta t$). We repeat the process until we have described the solution in as many points n as we are interested in.

The challenge is to find out how we can determine the next value from what we already know. It can be done in a crude or refined method. The easiest method is perhaps Euler's method. It is based on the well-known definition of the derivative:

$$\dot{x}(t) = \lim_{\Delta t \to 0} \frac{x(t + \Delta t) - x(t)}{\Delta t} \ .$$

If Δt is sufficiently small, we can manipulate this expression and write:

$$x(t + \Delta t) \approx x(t) + \Delta t \dot{x}(t) \ .$$

Assume the initial values are given by $(x_n, \dot{x}_n, t_n)$. Then follows the discrete version of our differential equation (named "difference equations"):

$$x_{n+1} = x_n + \dot{x}_n \Delta t \ .$$

By using such an update equation for both $x(t)$ and $\dot{x}(t)$, we get the familiar Euler method (in our context for the solution of second-order differential equation):

$$\begin{aligned} \dot{x}_{n+1} &= \dot{x}_n + \ddot{x}_n \Delta t \\ x_{n+1} &= x_n + \dot{x}_n \Delta t \ . \end{aligned}$$

Thus, we have two coupled difference equations.

Figure 4.2 outlines how the method works. This is the most common way to make such an illustration, but in my view it only gives a superficial understanding. What happens when the discrepancy between the correct solution and the numerical solution becomes bigger and bigger? Here are some details we should know.

Figure 4.3 looks similar to Fig. 4.2, but is illustrating a different message. The mid-blue blue curve (bottom) shows how a projectile thrown obliquely will proceed with an initial velocity of 1.0 m/s in the horizontal direction and 3.0 m/s in the vertical direction. The calculation is based on an analytical solution to this simple problem.

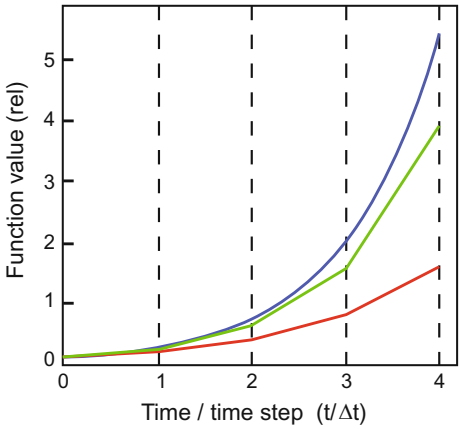

Fig. 4.2 Euler's simple method of calculating a function numerically. The top (blue) curve is the exact analytic solution. The lower (red) curve is calculated using Euler's simple method, while the middle curve is calculated using the midpoint method. The time step is the same in both cases and is chosen very large to accentuate the differences

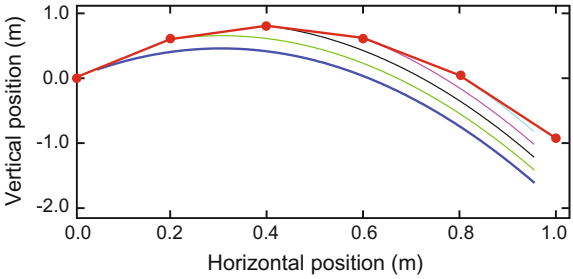

Fig. 4.3 Calculation of an oblique throw using Euler's method with a large time step. Each new point calculated is the starting point for a new solution of the original differential equation. See the text for details

The figure also shows a plot of the solution found by using Euler's method (red curve) with very large time steps (0.2 s). Even after the first step, the calculated new position is quite far from what it should be.

After the first step, new values have been calculated for position and speed in both horizontal and vertical directions. These values are now plugged into the differential equation. If we had calculated the path for exactly these values, we would have got the solution given by a green curve (next to bottom). *This is a different solution of the differential equation than we started with!*

Not even now, we manage to follow this new solution closely since the time step is so big and when we use Euler's method once more, we get a position (and velocity) quite far from the second solution of the differential equation we started with.

We keep going along this route. For each new time step, we get a new solution of differential equation, and in our case, the error, being systematic, becomes bigger and bigger after each time step.

It can be shown that if we reduce the time step significantly (!) compared to that used in Fig. 4.3, the solution will be far better than in the figure. Nevertheless, it is not always enough to only reduce the size of the time step.

First of all, we cannot make the step size so small that we run into trouble with inputting numbers accurately on a computer (without having to use extremely time-consuming techniques). When we calculate $x_{n+1} = x_n + \dot{x}_n \Delta t$, the contribution

$\dot{x}_n \Delta t$ must not always be so small that it can only affect the least significant digit of x_{n+1}.

Another limitation lies in the numerical method itself. If we make systematic errors which accumulate at each time step, no matter how small the time steps are, we also get problems. Then we must use other numerical methods instead of this simplest variant of Euler's method.

An improved version of Euler's method is called the *Euler-Cromer method*. Assume that the starting values are $(x_n, \dot{x}_n, t_n)$. The first step is identical to Euler's simple method:

$$\dot{x}_{n+1} = \dot{x}_n + \ddot{x}_n \Delta t \ .$$

However, the second step in the Euler-Cromer method differs from that in the simpler Euler version: To find x_{n+1}, we use $\dot{x}_{n+1}$ and not $\dot{x}_n$ as we do in Euler's method. It provides the following update equation for x:

$$x_{n+1} = x_n + \dot{x}_{n+1} \Delta t \ .$$

The reason that the Euler-Cromer method works and that it often (but not always) works better than Euler's method is not trivial, and we will not go into this. Euler's method often causes the energy of the modelled system to become an unconserved quantity that slowly but steadily increases. This problem becomes dramatically reduced with the Euler-Cromer method, which in most cases works better.

Another improvement over Euler's method, which is even better than the Euler-Cromer method, is *Euler midpoint method*. Instead of using the gradient *at the beginning* of the step, and using this for the entire interval, we use the gradient *in the middle* of the interval. By using the slope at the midpoint of the interval, we will usually get a more accurate result than using the slope at the beginning of the interval when we are looking for the average growth rate.

In Euler's midpoint method, we first use the gradient at the beginning of the interval, but instead of using this value for the entire interval, we use it for half the interval. Then we calculate the gradient at the middle of the interval and use this for the entire interval. Mathematically, this is done by using the same notation as before:

$$\dot{x}_{n+\frac{1}{2}} = \dot{x}_n + f\left(x_n, \dot{x}_n, t_n\right) \tfrac{1}{2} \Delta t \ ,$$
$$x_{n+\frac{1}{2}} = x_n + \dot{x}_n \tfrac{1}{2} \Delta t \ .$$

Here, $\dot{x}_{n+\frac{1}{2}}$ and $x_{n+\frac{1}{2}}$ are the values of the unknown function and its derivative at the midpoint of the interval. The update equation for the entire range will be as follows:

$$\dot{x}_{n+1} = \dot{x}_n + f\left(x_{n+\frac{1}{2}}, \dot{x}_{n+\frac{1}{2}}, t_{n+\frac{1}{2}}\right)\frac{1}{2}\Delta t \ ,$$
$$x_{n+1} = x_n + \dot{x}_{n+\frac{1}{2}}\,\Delta t \ .$$

4.5 Runge–Kutta Method

In Euler's method, we found the next value by using the slope at the beginning of the chosen step. In Euler's midpoint method, we used the slope in the middle of the chosen step. In either case, it is quite easy to imagine that for some functions we will be able to get a systematic error that will add up to a significant total error after many subsequent calculations have been carried out. It can be shown that the error we make becomes significantly less if we switch to using more refined methods for finding the next value. One of the most popular methods is called the fourth-order Runge–Kutta method. A total of four different estimates of the increase, one at the beginning, two in the middle and one at the end are then used to calculate the average increase in the interval. This makes the Runge–Kutta method much better than Euler's midpoint method, and since it is not much harder to program, this is often used in practice.

Let us see how the fourth-order Runge–Kutta method works and how it can be used to solve a second-order differential equation (At the end of the chapter one will find a pseudocode and the full code for a program that uses the fourth-order Runge–Kutta method.).

4.5.1 Description of the Method

The Runge–Kutta method is not really difficult to understand, but you probably have to read the details that are included twice to see it. We will first provide a mathematical review and then try to summarize the method using a figure (Fig. 4.4). Let us begin with a few words about the mathematical notation. Consider the differential equation given below:

$$\ddot{x}(t) = f\left(x(t), \dot{x}(t), t\right) \ . \tag{4.2}$$

For the damped mass–spring oscillator considered in Chap. 2 (where t does not appear explicitly), this equation will take the following form:

$$\ddot{z}(t) = -\frac{b}{m}\dot{z}(t) - \frac{k}{m}z(t) \ . \tag{4.3}$$

Suppose we are at the point $(x_n, \dot{x}_n, t_n)$ and that the duration of the time step is Δt. In what follows we will find estimates for x_n, $\dot{x}_n$ and $\ddot{x}_n$, and it will be convenient to replace $\dot{x}_n$, and $\ddot{x}_n$ by v_n and a_n, respectively. An additional numerical index will be used to indicate the ordinal position of an estimate (first, second, etc.). With this notation, the kth estimate of a quantity χ_n ($\chi = x$, $v = \dot{x}$, $a = \ddot{x}$) will be represented by the symbol $\chi_{k,n}$.

We can find the first estimate of $\ddot{x}_n$ by using Eq. (4.1):

$$a_{1,n} = f(x_n, \dot{x}_n, t_n) = f(x_n, v_n, t_n) \ .$$

At the same time, the first derivative is known at the beginning of the time step:

$$v_{1,n} = \dot{x}_n = v_n \ .$$

The next step on the route is to use Euler's method to find $\dot{x}(t)$ and $x(t)$ in the middle of the step:

$$x_{2,n} = x_{1,n} + v_{1,n}\frac{\Delta t}{2} \ ,$$

$$v_{2,n} = v_{1,n} + a_{1,n}\frac{\Delta t}{2} \ .$$

Furthermore, we can find an estimate of the second derivative at the midpoint of the step by using $v_{2,n}$, $x_{2,n}$ and Eq. (4.2):

$$a_{2,n} = f(x_{2,n}, v_{2,n}, t_n + \Delta t/2) \ .$$

The next step now is to use the new value for the second derivative at the midpoint in order to find a new estimate of $x(t)$ and $\dot{x}(t)$ at the midpoint of the step using Euler's method:

$$x_{3,n} = x_{1,n} + v_{2,n}\frac{\Delta t}{2} \ ,$$

$$v_{3,n} = v_{1,n} + a_{2,n}\frac{\Delta t}{2} \ .$$

With the new estimate of $x(t)$ and $\dot{x}(t)$ at the midpoint of the step, we can find a new estimate for the second derivative at the midpoint:

$$a_{3,n} = f(x_{3,n}, v_{3,n}, t_n + \Delta t/2) \ .$$

Using the new estimate of the second derivative in addition to the estimate of the first served in the middle range, we can now use Euler's method to estimate $x(t)$ and

$\dot{x}(t)$ at the end of step. This is done as follows:

$$x_{4,n} = x_{1,n} + v_{3,n}\Delta t \ ,$$

$$v_{4,n} = v_{1,n} + a_{3,n}\Delta t \ .$$

Finally, in the same way as before, we can estimate $\ddot{x}(t)$ at the end of the step using these new values:

$$a_{4,n} = f(x_{4,n}, v_{4,n}, t_n + \Delta t) \ .$$

We can now calculate a weighted average of the estimates, and then we get reasonable estimates of the average values of the first and second derivatives in the step:

$$\overline{a_n} = \tfrac{1}{6}\left(a_{1,n} + 2a_{2,n} + 2a_{3,n} + a_{4,n}\right) \ , \tag{4.4}$$

$$\overline{v_n} = \tfrac{1}{6}\left(v_{1,n} + 2v_{2,n} + 2v_{3,n} + v_{4,n}\right) \ . \tag{4.5}$$

Using these averages, which are quite good approximations to the mean values of the slopes over the entire step, we can use Euler's method of finding a good estimate of $x(t)$ and $\dot{x}(t)$ at the end of the step:

$$x_{n+1} = x_n + \overline{v_n}\Delta t \tag{4.6}$$

$$v_{n+1} = v_n + \overline{a_n}\Delta t \tag{4.7}$$

$$t_{n+1} = t_n + \Delta t \tag{4.8}$$

These are equivalent to the initial values for the next step.

In the Runge–Kutta method (see Fig. 4.4), we extract much more information from the differential equation than in Euler's method. This makes the Runge–Kutta method significantly more stable than Euler's method, Euler-Cromer method and Euler's midpoint method. The Runge–Kutta method does not make an excessive demand on the resources of a computer, but it is relatively simple to program. The Runge–Kutta method, in one or other variant, is therefore often the method we first turn to when we want to solve ordinary differential equations numerically.

Programming of the basic part of the Runge–Kutta method is done almost once and for all. It is usually only a small file that changes from one problem to another. The file specifies exactly the differential equations that will be used in exactly the calculations that will be performed. See example code later in the chapter.

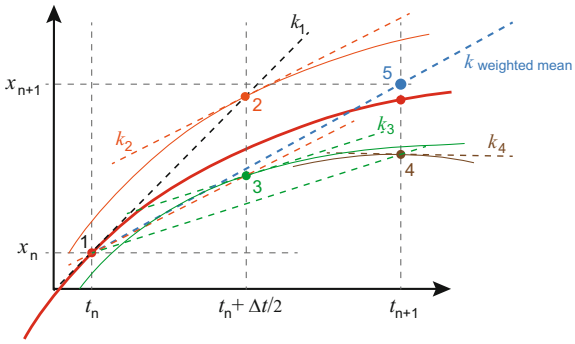

Fig. 4.4 Summary of the fourth-order Runge–Kutta method. See text

Some concentration is required to fully understand Fig. 4.4: In point 1 (x_n, t_n), the slope is k_1. We follow the tangent line at point 1 for half a step to point 2 (pink). This point is based on another solution of differential equation (thin pink line) than the one we seek. We calculate the slope k_2 at point 2 for this solution (pink dotted line). We then draw a line from point 1 again, but now with the gradient we found at point 2. Again we only go half the step length and find point 3 (green). There is yet another solution of the differential equation that goes through this point (thin green line). We calculate the slope k_3 at point 3 for this solution (dotted green line). We then draw a line through point 1 again, but now with the slope we just found. Now we go all the way up to point 4 (brown). Again there is a new solution of the differential equation that goes through this point. We calculate the slope k_4 of this solution at point 4.

The final step is to calculate the weighted mean of four different slopes and use this from the starting point 1 in the figure a full time span Δt to get the estimate (point 5) for the change of our function in the current time interval. The result is relatively close to the correct value (compare point 5 by a red dot in the figure).

4.6 Partial Differential Equations

Many physical problems are described by partial differential equations, perhaps the most well known are Maxwell's equations, Schrödinger equation and wave equation. The term "partial differential equation" means that the unknown function depends on two or more variables, and that derivatives with respect to these occur in the differential equation.

There are several methods for solving partial differential equations, but a key concept is finite differences. It is about replacing the differentials in the differential equation with final differences. Consider the simple differential equation

$$\frac{\partial y}{\partial x} = K \frac{\partial y}{\partial t} \,. \tag{4.9}$$

The simplest way to convert the derivatives in this equation into difference quotients is to use the definition of the derivative, as we have done before. The above equation will then become

$$\frac{y(x+\Delta x,t)-y(x,t)}{\Delta x}=K\frac{y(x,t+\Delta t)-y(x,t)}{\Delta t}\,.$$

This equation can be solved for $y(x,t+\Delta t)$, which gives

$$y(x,t+\Delta t)=y(x,t)+\frac{\Delta t}{K\,\Delta x}\big[y(x+\Delta x,t)-y(x,t)\big]\,.$$

Suppose that $y(x,t)$ is known at a time $t=t_0$ for the interesting interval in x. The right-hand side of the above equation gives $y(x,t_0+\Delta t)$, the value of the function at a later time $t+\Delta t$. However, note that we also need the value of the function at a different x from that appearing on the left-hand side. This means that we will encounter a problem when we come to calculating the value of the function at a point x near the outer limit of the region over which the calculation is to be performed. From the equation above, we see that we need to know what the function was at the next x coordinate at the last instant, and at the extreme x point, this is not feasible.

This means that, in order to find a unique solution to our problem, we must know the *boundary conditions*, that is, the state of the system at the boundary of the region of interest. These must be specified before the calculations can even begin.
Note: Initial and boundary conditions are two different things and must not be mixed together. Initial conditions specify the state of the system at the very beginning of the calculations and must also be used here. Boundary conditions specify the state of the system at the endpoints of the calculations at *all* time.

The finite differences introduced above are, however, rarely used, since they can be replaced by something that is better and not much more difficult to understand. Instead of using Euler's method in the above differentials, Euler's midpoint method, which significantly reduces the error in the calculations, is used. If we do this, the discretization of Eq. (4.9) leads to the following result:

$$\frac{y(x+\Delta x,t)-y(x-\Delta x,t)}{2\Delta x}=K\frac{y(x,t+\Delta t)-y(x,t-\Delta t)}{2\Delta t}\,.$$

It is not hard to understand that the result will now be better, for instead of calculating the average growth through the current point and the next point, the average growth is used through the previous and next point. In the same way as before, this equation can be solved with regard to $y(x,t+\Delta t)$, and the result will be:

$$y(x, t + \Delta t) = y(x, t - \Delta t) + \frac{\Delta t}{K \Delta x} [y(x + \Delta x, t) - y(x - \Delta x, t)] \ .$$

We see that we get the same problem with boundary conditions as above; in fact, an extra boundary condition is needed, even at the beginning of the x grid. Since this is a problem that concerns a spatial dimension, we need to set two boundary conditions to make the solution unique (there are two boundaries). To use the first one the update equation must therefore take into account the other boundary as well.

In the same way as we replaced first derivative with a finite difference quotient, the nth derivative can be approximated in the same way. An example is the second derivative that can be approximated with the following difference quotient:

$$f''(x) \approx \frac{f(x + \Delta x) - 2f(x) + f(x - \Delta x)}{\Delta x^2} \ . \tag{4.10}$$

Proof

$$f''(x) \approx \frac{f(x + \Delta x) - 2f(x) + f(x - \Delta x)}{(\Delta x)^2} \qquad \text{(start)}$$

$$= \frac{[f(x + \Delta x) - f(x)] - [f(x) - f(x - \Delta x)]}{(\Delta x)^2} \tag{4.11}$$

$$= \frac{1}{\Delta x} \left[\underbrace{\frac{f(x + \Delta x) - f(x)}{\Delta x}}_{\approx f'(x)} - \underbrace{\frac{f(x) - f(x - \Delta x)}{\Delta x}}_{\approx f'(x - \Delta x)} \right] \tag{4.12}$$

$$= \frac{f'(x) - f'(x - \Delta x)}{\Delta x} \ . \qquad \text{(end)}$$

This expression is nothing more than the definition of the derivative; thus, it is a proof of the validity of Eq. (4.10). The expressions make it clear why we must know the value of the function at three points (at least) in order to be able to calculate a second derivative.

As with the ordinary differential equations, we can move on and use methods that provide an even better result.

There are a number of methods available for different parts of physics. Interested refer to special courses/books in numerical calculations.

4.7 Example of Numerical Solution: Simple Pendulum

Let us take a concrete example, namely a pendulum that can swing with arbitrary large amplitudes (up to $\pm\pi$) without collapsing (i.e. the suspending rod is "rigid"). We expect all mass to be in a tiny ball (or bob) at the end of the rod.

Mechanics tell us that the force that pulls the pendulum along the path towards the equilibrium point is

$$F_\theta = -mg \sin\theta$$

where θ denotes the angular amplitude. If the length of the rod is L, the moment of this force around the pivot (suspension point) is:

$$\tau = -mgL \sin\theta \ .$$

The torque applied around the pivot can also be written as:

$$\tau = I\alpha = I\ddot{\theta} \ .$$

Here $\alpha = \ddot{\theta}$ is the angular acceleration and I the moment of inertia about the axis of rotation (which passes through the pivot and is perpendicular to the plane in which motion takes place). By using our simplifying assumptions for the pendulum, we have:

$$I = mL^2$$

which leads to the differential equation for the motion of the bob:

$$mL^2\ddot{\theta} = -mgL \sin\theta \ ,$$

$$\ddot{\theta} = -\frac{g}{L} \sin\theta \ .$$

In an elementary mechanics course, this equation is usually solved by assuming that the angle θ is so small that $\sin\theta \approx \theta$. The solution then turns out to be a simple harmonic motion with swing frequency (angular frequency) given by:

$$\omega = \sqrt{\frac{g}{L}} \ .$$

The approximation $\sin\theta \approx \theta$ was made to use analytical methods. This approach was not absolutely necessary in just this particular case, because we *can* solve the original differential equation analytically also for large angles by utilizing the series expansion of the sinus function. However, it is by far easier to use numerical methods.

The result of numerical calculations where we use fourth-order Runge–Kutta method is shown in Fig. 4.5. We see that the motion is near harmonic for small angular amplitudes, but very different from a sinusoid for a large swing amplitude.

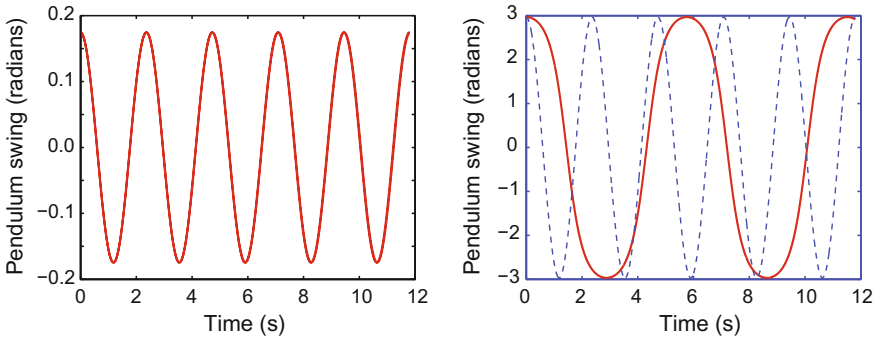

Fig. 4.5 A pendulum swings harmonically when the amplitude is small, but the swinging motion changes considerably when the swing angle increases. The swing period changes as well. See also the text

Moreover, the period has changed a lot. Note that in the right-hand part of the figure, we have chosen a motion where the pendulum *almost* reaches the "right-up" direction both "forward" and "return" (swing angle near $+\pi$ and $-\pi$).

If we wanted to include friction in the description of the pendulum motion, it would represent a more complex expression of the effective force than we had in our case. For nonlinear description of friction, there is no analytical solution.

Since the main structure of a numerical solution would be the same, irrespective of our description of the effective force acting on the system, the more complicated physical conditions can often be handled surprisingly easily with numerical solution methods (see Fig. 4.7 in one of the tasks in the problem section below).

This is an added bonus of numerical solutions: the force that works—and thereby the actual physics of the problem—becomes more central in our search for the solution! What force produces which result? Numbers are numbers, and there is no need to figure out different—occasionally intricate—analytical methods and tricks especially adapted for each functional representation of the force. The focus is where it should be: basically, the effective force, the governing differential equation, the pertinent initial condition(s), and the results that emerge from the analysis.

4.8 Test of Implementation

It is so easy to make a mistake, either in analytical calculations or in writing a computer program for obtaining numerical solutions. We have examples of many disasters in such contexts.

It is therefore very important to test the results of numerical solutions to detect as many errors as we can. It is often easier said than done! We often use numerical methods because we do not have any analytical methods to fall back on.

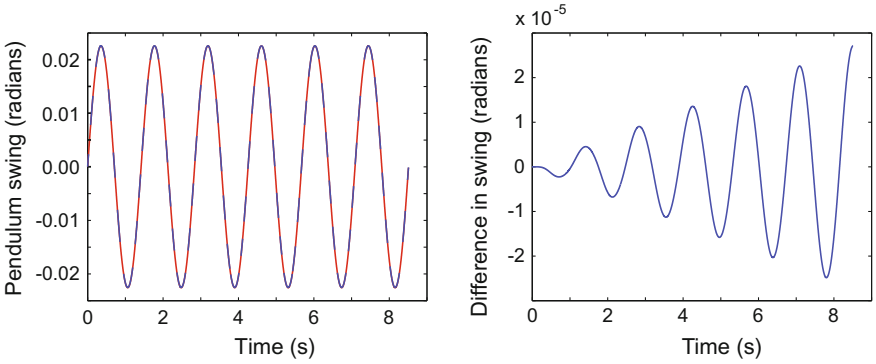

Fig. 4.6 Comparison between analytical and numerical solution of a shuttle movement. For explanations: See the text

In the case of the simple pendulum, there happens to be a trick up our sleeve. There is an analytical solution that is approximately correct for *small* amplitude. For *this* special case, we can test if the numerical solution becomes nearly the same as the analytical. If there is a serious disagreement between these two solutions, there must be an error somewhere.

That the numerical solution is close to its analytical counterpart in this special case, is unfortunately not a proof that the program is flawless! The implementation of the program beyond the special case may give incorrect results. Here it is necessary to consider the physical predictions: Do they seem reasonable or otherwise? It is often impossible to be absolutely sure that a computer program is completely correct. Within numerical analysis, there are special techniques that can be used in some cases. We cannot go into these. The main point is that we must be humble and alert to the possibility of errors and try to test the implementation of numerical methods every time we develop a computer program.

As an example, we will now try to check the program we used in the calculations that led to Fig. 4.5. We will use only the small amplitude case in our test.

In Fig. 4.6, the results of the numerical calculations (red curve) are shown on the left with an analytical solution (dashed blue curve) for the special case when the pendulum swing is small (maximum ± 0.023 rad). There is no perceptible difference between the two curves.

Plotting analytical and numerical solutions in the same figure are a common way to check that two solutions are in agreement with each other. However, this is a very rough test, because there is limited resolution in a graphical representation. In the right part of the figure, we have chosen a better test. Here, the *difference* between analytical and numeric results is plotted, and we see that there were certainly some differences, although we did not see this in the left part.

We can now see that the difference is increasing systematically. After six periods, the difference has increased to 2.7×10^{-5} rad. Is this an indication that our computer program is incorrect?

We know, however, that the analytical solution is *itself* only an approximation, and the smaller the swing angle, the smaller will be the error in the approximation. We can then reduce the amplitude and see what happens. Calculations show that if the amplitude is reduced to 1/10 of what we have in the figure, the maximum difference is reduced after six periods to 1/1000 of the earlier value. If we reduce the amplitude to 1/100 of the original, the maximum difference is reduced to 10^{-6} of the original difference. We see that numerical and analytical solutions are becoming more and more similar and in a way that we would expect. If we take a look at the series development for the sine function, it gives us a further clue that our results are what we would expect.

We can then feel reasonably sure that the program behaves as it should for small angular displacements, and that it seems to handle larger angles as it should, at least as long as they remain small.

There is also another test we often have to do in connection with numerical calculations. We chose to use 1000 steps within each period in the calculations whose results are plotted in Figs. 4.5 and 4.6. For calculations that span very many periods, we cannot use such small time steps. If we go down to, e.g., 100 calculations per period, the result will still be acceptable usually (depending on what requirements we impose), but if we go down to, say 10 steps per period, the result will almost certainly depend markedly on the choice of the step size. We often have to do a set of calculations to make sure that the "resolution" in the calculations is appropriate and manageable (neither too high nor too low).

4.9 Reproducibility Requirements

Today it is easy to change a program from one run to another. Ironically, this presents extra challenges that need to be taken seriously. When we make calculations to be used in a scientific article, a master's thesis, a project assignment, and almost in any context where our program is used, we must know the exact program and parameters that are used if the results are to have full value. In experimental physics, we know that it is important to enter in the laboratory journal all details of how the experiments have been performed. The purpose is that it should be possible to test the results we get. This is essential for reproducibility and for achieving so-called intersubjectivity (that the result should be independent of which person actually executes the experiment), which is extremely important in science and development.

In experimental work, one occasionally succumbs to the temptation of not jotting down all relevant details while the experiment is underway. Being interested primarily in the result, we think that when we have come a little further and got even better results, *then* we would write down all the details. Such practice often causes some frustration at a later date, because suddenly we discover that an important piece of information was never actually noted. At worst, the consequence of this lapse may be that we have to repeat the experiment, and hunt for the conditions under which the previous experiment, the results of which proved to be particularly interesting, was performed.

Modern use of numerical methods can in many ways be compared to experimental work in the laboratory. We test how different parameters in the calculation affect the results, and we use different numerical methods in a similar manner as we use different measuring instruments and protocols in experiments. This means that there are stringent requirements for documentation for those who use numerical methods as for the experimentalist.

In order to comply with this requirement, we should incorporate good habits in the programming. One way we can comply with reproducibility requirements is to do the following:

- In the program code, insert a "version number" for your application.
- In the result file you generate, the version number must be entered automatically.
- Every time you change the program in advance of a calculation that you would like to make, the version number must be updated.
- Each version of the program (actually used in practice) must be saved to disk so that it is always possible to rerun an application with a given version number.
- Parameters which are used and which vary from run to run within the same version of the program must be printed to a file along with the result of the run.

If we keep to these rules, we will always be able to return and reproduce the results obtained in the past. It is assumed here that the results are independent of the computer used for the calculations. If we suspect that a compiler or an underlying program or an operating system might malfunction, it may be appropriate to provide additional information about this along with the results (in a result file).

In the specimen programs given in this book, the lines needed for documentation of parameters and version number are, for the most part, *not* included in the code. The reason is that the program pieces provided here are intended primarily for showing how the calculations can be performed.

4.10 Some Hints on the Use of Numerical Methods

In our context, it is often necessary to create relatively small computer programs to get a specific type of calculation. There is usually no need to have the fancy interface to select parameters and fancy presentations of the results as it is for commercial programs. We need to do a specific task, and the program is usually not used by many, or very often. This is the starting point for the tips that follow.

Many of the issues we encounter in this book are related to the integration of differential equations that describe the processes we are interested in. The following hints are partly influenced by this preoccupation.

Planning

Before we get to the computer, we should have a clear notion of what we want to achieve. We must have already established the differential equation that describes the process of our interest and have pondered over the parameters that are to be included

in the calculations. Current parameter values and initial values need to be looked up or chosen by ourselves.

It may be useful to outline how we can partition the program into main components, each of which has its separate function. We also have to decide the order in which we will work through the various parts of the program and have thoughts of how we can test the different parts individually and together.

It is also natural to ask: Do we want to provide parameters while the program is running or is it sufficient to insert them into the program code before the program starts? How will we take care of the results? Should it be in the form of plots or animations or numbers are printed on screen, or should the final results be written to file(s) for later processing?

Writing of Code

There should be a one-to-one correspondence between the mathematical description of a problem (algorithm) and the code. It applies to variables, formulas, etc.

It is recommended to adhere to the programming language guidelines, such as "PEP 8—Style Guide for Python Code" or "MATLAB Style Guidelines 2.0".

Try to collect the code lines where parameters are given special values already as part of the code. This makes it easier to change parameters for later runs. Reset arrays or give arrays values.

Put together all expressions of fixed constants which will be used in that part of the program that is most frequently run, in order to avoid more calculation operations than necessary in a loop. For example, it is a good idea to create a parameter

```
coeff = 4.0*3.141926*epsilon0*epsilonR*mu0*muR
```

and use this coefficient in a loop that is recalled many times, instead of having to repeat all these multiplications each time the loop is run (the parameters in this example have been selected randomly).

A code should be broken up into logical functions. In Python, multiple functions can be added to one and the same file. In Matlab, various functions are often allocated to separate files (although it is actually possible to use a similar layout in Matlab as in Python).

Generalize when you are writing a program, unless it seems inadvisable. For example, when integrating an expression, a general integral of $f(x)$ is programmed and then a special f is chosen as its argument. This requires frequent use of functions. Do not overdo it though, because it obstructs a survey and the readability of the program.

Testing and Debugging

Make an effort to construct test problems for checking that the implementation is correct. Functions should be tested as they are written. Do not postpone testing until code writing is finished!

There are several types of errors that may occur. Some errors are detected by the compiler. Read the error message carefully to see how such errors can be corrected.

Other errors appear when running the program. For example, we can end up in an infinite loop and must terminate the program manually. It is not always easy to find out where in the program code such a fault is located. It is then useful to add dummy print-to-screen here and there in the code so we can locate that line in the code where the problem occurs.

While we are going through program development and testing, it is important to save the program several times along the way, and preferably change names sometimes, in order to avoid a potential catastrophe. Then we will not have to start all over again if you lose everything in a file.

Check that the program provides the correct result for a simplified version of the problem, where there is also an analytical solution. This is crucial!

Repeat the calculations using different resolutions (often given by Δt) to see how many points are needed to get a good match with the analytical answer or to verify that the result depends only to a small extent on moderate changes in resolution.

Forms of Presentation

Plot the results or present them in some other form. Save data to file if desired.

Simple plots are often sufficient, but we can rarely read precise details from a plot, at least not without having chosen a very special plot that displays just what we want to show. Sometimes, the choice of linear or logarithmic axes in a plot is crucial for whether we discover interesting relationships or not.

Make sure that the axes in the plot are labelled properly that symbol sizes and line thicknesses and other details in the presentation meet the expected requirements.

In reports, articles and theses, one is a requirement that numbers and text along the axes of the plots must be readable without the use of magnifying glass (!) *in the final size the characters have in a document.* This means that numbers and letters should have a size between 9 and 12 pt in *final size*, and indexes may be even a bit smaller).

When using Matlab, it is a good idea to save figures which do *not* fill the entire screen (use default display of figures on screen). Then the font size will be sufficiently large even if the figure is reduced to approximately the same format as used in this book. However, if the image size is reduced too much, the font size in the final document will become too small. You can choose, for example, line thickness and font size in plots generated by Matlab and Python. The following code piece indicates some of the possibilities that exist (the example is in Matlab, but there are similar solutions in Python):

```
...
axes('LineWidth',1,'FontSize',14,'FontName','Arial');
plot(t,z,'-r','LineWidth',1);
xlabel('Time (s)','FontSize',16,'FontName','Arial');
...
```

Learn good habits as early as possible—it will pay off in the long run!

Reproducibility

When we believe that the program as whole works as it should, we can finally embark upon the calculations for the particular project we are occupied with. Reproducibility requirements must be adhered to when the program now receives a solemn version number, and the program code must be saved and not changed without a new version number.

Files that document later runs must be preserved in a manner similar to a laboratory record.

4.11 Summary and Program Codes

Summary of the Chapter

Let us try to summarize the key points in our chapter:

- A second-order differential equation can be considered equivalent to two coupled first-order differential equations.
- In a single differential equation, we replace the derivative $\mathrm{d}f/\mathrm{d}t$ with the differential quotient $\Delta f/\Delta t$. Starting from this approximate equation and initial conditions, we can successively calculate all subsequent values of $f(t)$. This method is called Euler's method. The method often gives large errors, especially when we are dealing with oscillations!
- There are better methods for estimating the average slope of the function during the step Δt than just using, as we in Euler's method, the derivative at the beginning of the interval. One of the most practical and robust methods is called fourth-order Runge–Kutta method. In this method, a weighted average of four different calculated increments in the interval Δt is used as the starting point for the calculations. The method often provides good consistency with analytical solutions where these exist, also for oscillatory phenomena. However, we must be aware that this method is not exempt from error, and for some systems it will not work properly.
- For second-order ordinary differential equations, such as the equation for oscillation, we can find the solution if we know the differential equation and the initial conditions. For the second-order partial differential equations, for example, a wave equation, we must *in addition* know the so-called boundary conditions not only at the start but also throughout the calculations. This makes it often far more difficult to solve partial differential equations than ordinary other order diffusions.
- It is valuable to compare numerical calculations and analytical calculations (where these exist) to detect errors in our programming. However, even if the conformity is good in such special cases, there is no guarantee that the numerical solutions will be correct also for other parameter values (where analytical solutions are not available).

- The program code is divided into an appropriate number of separate functions that have their own task. In this way, the logical structure of the program will clarify. Some features can be made so general that they can be reused in many different contexts. For example, we can create one general Runge–Kutta function that calls for a more specialized function that contains the appropriate differential equation (where only the last small function will vary from problem to problem).
- Since we can easily change programs and parameters, it is a big challenge to keep track of how the computer program looked and what parameters we used when we made calculations and arrived at results we would use. Some systematic form of documentation is imperative, where program, input parameters and results can be linked to each other in a clear way.

Pseudocode for Runge–Kutta Method *

```
The input to this function is x[n-1], v[n-1] and t[n-1] and
returns x[n] and v[n].

1. Use the input parameters in order to find the
   acceleration, a1, in the start of the interval.
   The speed in the start of the interval, v1, is given as
   an input parameter.
   x1 = x[n-1]
   v1 = v[n-1]
   a1 = ...

2. Use this acceleration and speed to find an estimate for
   the speed (v2) and position in the middle of the interval.
   x2 = ...
   v2 = ...

3. Use the new position and speed to find an estimate for
   the acceleration, a2, in the middle of the interval.
   a2 = ...

4. Use this new acceleration and speed (a2 and v2) to find
   a new estimate for position and speed (v3) in the middle
   of the interval.
   x3 = ...
   v3 = ...

5. Use the new position, speed and time in the middle of
   the interval to find a new estimate for the acceleration,
   a3, in the middle of the interval.
   a3 = ...
```

```
6. Use the last estimate for the acceleration and speed in
   the middle of the interval to find a new estimate for the
   position and speed (v4) in the END of the interval.
   x4 = ...
   v4 = ...

7. Use the last estimate for position and speed to find an
   estimate for the acceleration in the END of the interval, a4.
   a4 = ...

8. A mean value for speed and acceleration in the interval
   is calculated by a weighted, normalized sum:
   vMiddle = 1.0/6.0 * (v1 + 2*v2 + 2*v3 + v4)
   aMmiddle = 1.0/6.0 * (a1 + 2*a2 + 2*a3 + a4)

9. Finally, use these weighted mean values for speed and
   acceleration in the interval to calculate the position
   and speed in the end of the interval.
   The function return this position and speed.
   x[n]  = ...
   v[n] = ...
   return x[n], v[n]
```

Matlab Code for Runge–Kutta Method

Important

The code of most of the example programs in this book is available (both for Matlab and Python) at a "Supplementary material" web page. At the same web page, files required for solving some of the problems are available as well as a list of reported errors, etc. The address for the "Supplementary material" web page is http://www.physics.uio.no/pow.

```
function [xp,vp,tp] = rk4x(xn,vn,tn,delta_t,param)

% Runge-Kutta integrator (4th order)
%************************************************************
% This version of a 4th order Runge-Kutta function for Matlab
% is written by AIV. Versjon 09282017.
% This function can be used for the case where we have two
% coupled difference equations
%    dv/dt = ffa(x,v,t,param)
%    dx/dt = v  NOTE: This part is taken care of automatically
%               in this fuction.
% Input parameters: x,v,t can be position, speed and time,
% respectively. delta_t is the step length in time.
% param is a structure in Matlab (in Python it is called a
```

```
% class). It contains various parameters that is used to
% describe the actual second order differential equation.
% It MUST contain the name of the function that contains
% the differential equation. The class "param" the user has
% to define.

% Input argumentents (n: "now")
%   [xn,vn,tn,delta_t,param] = values for x, v and t "now".
% Output argumentets (p : "n plus 1")
%   [xp,vp,tp] = new values for x, v and t after one step in
%   delta_t.
%************************************************************

ffa = eval(['@' param.fn]);   % Picks up the name of the
% Matlab-code for the second derivative. Given as a text
% string in a structure param.

half_delta_t = 0.5*delta_t;
t_p_half = tn + half_delta_t;

x1 = xn;
v1 = vn;
a1 = ffa(x1,v1,tn,param);

x2 = x1 + v1*half_delta_t;
v2 = v1 + a1*half_delta_t;
a2 = ffa(x2,v2,t_p_half,param);

x3 = x1 + v2*half_delta_t;
v3 = v1 + a2*half_delta_t;
a3 = ffa(x3,v3,t_p_half,param);

tp = tn + delta_t;
x4 = x1 + v3*delta_t;
v4 = v1 + a3*delta_t;
a4 = ffa(x4,v4,tp,param);

% Returns (estimated) (x,v,t) in the end of the interval.
delta_t6 = delta_t/6.0;
xp = xn + delta_t6*(v1 + 2.0*(v2+v3) + v4);
vp = vn + delta_t6*(a1 + 2.0*(a2+a3) + a4);
tp = tn + delta_t;
return;
```

The Function that Contains the Differential Equation

```
function dvdt = forced(y,v,t,param)

%*********************************************************
% This function is calculating the accelleration of a
% mass-spring oscillator that is influenced by an external
% periodic force that last only for a limited time interval.
% The trivial first order diff.eq. dx/dt = v is taken care
% of automatically in rk4x. The function "forced" is used
% by a RK4 function, but the necessary parameters are
% defined by the main program (given separately).
% Written by AIV. Versjon 09282017.

% Input parameters:
%   y = position
%   v = speed
%   t = time
% Output parameters:
%   dvdt = Left side of an equation in a difference equation
%   for v.

%*********************************************************

% The external periodic force last from the start of
% calculation until the time is param.end. See the main
% program for explanations of the other param items.

if (t < param.end)
    dvdt = - param.A*v - param.B*y + param.C*cos(param.D*t);
else
    dvdt = - param.A*v - param.B*y;
end;
return;
```

Example:
Matlab Program that Uses the Runge–Kutta Method

A program for calculating forced mechanical oscillations (spring pendulum) is given below. It shows how Runge–Kutta method is used in practice if we program the Runge–Kutta routine itself.

```
function forcedOscillations17

% An example program to study how forced oscillations which
% start with a mass-spring oscillator with no motions. The
% external force is removed after a while. The program calls
% the functions rk4r.m which is also using the function
% forced.m.
```

```
global param;

% Constants etc (see theory in previous chapters) in SI units
omega = 100;
Q = 25;
m = 1.0e-2;
k = m*omega*omega;
b = m*omega/Q;
F = 40;
time = 6.0; % Force only present halv of this time, see later
% Parameters used in the calculations (rk4.m, tvungen.m)
param.A = b/m;
param.B = omega*omega;
param.C = F/m;
param.D = omega*1.0;  % If this value is 1.0, the angular
                      % frequency of the force equals the
                      % angular frequency for the system.
param.end = time/2.0;
param.fn = 'forced'; % Name of Matlab file for 2. derivative

% Choose number steps and step size in the calculations
N = 2e4;                    % Number calculation points
delta_t = time/N;           % Time step in the calculations

% Allocate arrays, set initial conditions
y = zeros(1,N);
v = zeros(1,N);
t = zeros(1,N);
y(1) = 0.0;
v(1) = 0.0;
t(1) = 0.0;

% The loop where the calculations actually are done
for j = 1:N-1
    [y(j+1), v(j+1), t(j+1)]=rk4x(y(j),v(j),t(j),delta_t,param);
end;

% Plot the results
plot(t,y,'-b');
maxy = max(y);
xlabel('Time (rel units)');
ylabel('Position of the mass (rel. units)');
axis([-0.2   time   -maxy*1.2   maxy*1.2]); % want some
    % open space arround the calculated results

% We should also have compared our results with the analytical
% solution of the differential equation in order to verify
% that our program works fine. Not implementet in this
% version of the program.
```

Using Matlab's Built-in Runge–Kutta Function *

Finally, here is a specimen program for calculating damped oscillations, if we use Matlab's built-in solver of ordinary equations (ode) using the fourth-order Runge–Kutta method. First, we enter the main program we called *dampedOscill.m* (the name is insignificant here) and then follows a small application snap *ourDiffEq.m* that the main application calls. Matlab's equation solver requires a small additional function that specifies the current differential equation as such and that is the one given in vaarDiffLign.m.

```
function dampedOscill
% Program for simulation of damped oscillations.
% Written by FN. Version 09282017
% Solves two copuled differential equations
% dz/dt = v
% dv/dt = - coef1 v - coef2 z

clear all;

% Defines the physical properties for the oscillator
% (in SI units).
b = 3.0;  % Friction coefficient
m = 7.0;  % Mass
k = 73.0; % Spring constant
% Reminder:
%    Overcritical damping : b > 2 sqrt(k m)
%    Critical damping :     b = 2 sqrt(k m)
%    Undercritical damping: b < 2 sqrt(k m)

coef1 = b/m;
coef2 = k/m;

% Initialconditions (in SI-units)
z0  = 0.40; % Position rel. equilibrium point
v0  = 2.50; % Velocity

% Time we want to follow the system [start, end]
TIME = [0,20];

% Initial values
INITIAL=[z0,v0];

% We let Matlab perform a full 4th order Runge-Kutta
% integration of the differential equation. Our chosen
% differential equation is specified by the function
% ourDiffEq.

% T is time, F is the solutions [z v], corresponding to the
```

```
% running variable t (time) and f is the running variable
% [z(t) v(t)] that Matlab use through the calculations.
% Matlab chooses itself the step lengths in order to give
% proper accuracy. Thus, the calculated points are not
% equidistant in time!

[T F] = ode45(@(t,f) ourDiffEq(t,f,coef1,coef2),TIME, INITIAL);

% Plot the results, we choose to only  plot position vs time.
plot(T,F(:,1));

% length(T) % Option: Write to sceen how many points Matlab
%             actually used in the calculation. Can be useful
%             when we compare with our calculations with our
%             own Runge-Kutta function.

% We should also compare our results with the analytical
% solution of the differential equation in order to verify
% that our program works fine. Not implementet so far...
```

Our Own Differential Equation

Here comes the small function that gives the actual differential equation (in the form of two coupled difference equations):

```
function df = ourDiffEq(~,f,coef1,coef2)

% This function evaluate the functions f, where f(1) = z and
% f(2) = v. As the first variable in our input parameters we
% have written ~ since time does not enter explicitely in our
% expressions.

df = zeros(2,1);

%The important part: The first differential equation: dz/dt = v
df(1) = f(2);

% The second differential equation: dv/dt = -coef1 v - coef2 z
df(2) = -coef1*f(2)-coef2*f(1);
```

4.11.1 Suggestions for Further Reading

The following sources may be useful for those who want to go a little deeper into this material:

- Hans Petter Langtangen: A Primer on Scientific Programming with Python. 5th Ed. Springer, 2016.
- http://en.wikipedia.org/wiki/Semi-implicit_Euler_method (accessed 01.10.2017)
  - `http://en.wikipedia.org/wiki/Numerical_partial_differential_equations`

4.12 Learning Objectives

After working through this chapter, you should be able to:

- Know that a second-order differential equation can be considered equivalent to two coupled first-order differential equations.
- Solve a second-order differential equation numerically using the fourth-order Runge–Kutta method.
- Explain why numerical methods can handle, more frequently than analytical methods, complex physical situations, such as nonlinear friction.
- Point to some factors that could cause numerical calculations to fail.
- Explain in detail why the fourth-order Runge–Kutta method usually works better than Euler's method.
- Make a reasonably good test that a computer program that uses numerical solution methods works as it should.
- Put into practice your practical experience in using numerical methods to integrate an ordinary differential equation or a partial differential equation.
- Know and have some practical experience working out a computer program with several functions that interact with each other and could explain the purpose of such a partitioning of code.
- Know and have some experience with troubleshooting and know some principles that should be used to avoid postponing comprehensive troubleshooting until most of the code is written.
- Know how we can proceed to consolidate documentation of programs and parameters associated with the calculated values.
- Know why it is a good idea to save a computer program under a new name just as it is, while one is going through modifications to the program.

4.13 Exercises

Suggested concepts for student active learning activities: Discretizing, algorithm, numerical method, Euler's method, Runge–Kutta's method, accuracy, coupled differential equations, partial differential equation, documentation for programming activities.

Comprehension/discussion questions

1. Why does the fourth-order Runge–Kutta method usually work better than Euler's method?
2. Figure 4.7 shows the result of calculations of a pendulum motion for the case that there is some friction present. The figure shows position (angle) as a function of time (left part) and angular velocity as a function of position (angle) in the right part (also called a phase plane plot). The two upper figures result from an initial condition where the pendulum at time $t = 0$ hangs straight down, but at the same time has a small angular velocity. The lower figures result from an initial condition which is the same as for the upper part, but that the initial angular velocity is a good deal greater than in the first case.
   Explain what the figures say about the motion (try to bring as many interesting details as possible). How would the figure look if we increased the initial angular velocity even more than the one we have in the lower part of the figure?
3. Try to outline the working steps involved in analytical calculations of an oblique projectile throw with or without friction (or planetary motion around the sun). What do we spend most of the time on, and what do we concentrate on when we inspect the calculation afterwards? Attempt to outline the work plan for a numerical calculation and how we examine the result of such a calculation. What are the pros and cons of each method? Also try to incorporate physical understanding of the mechanisms of motion.

Problems

Remember: A "Supplementary material" web page for this book is available at http://www.physics.uio.no/pow.

4 The purpose of this composite task is to create your own program to solve different order differential equations using the fourth-order Runge–Kutta method (RK4) and to modify the program to cope with new challenges. Feel free to get extra help to get started! Specific assignments are as follows:
(a) Write a computer program in Matlab or Python that uses RK4 to calculate the damped harmonic motion of a spring pendulum. The program should consist of at least three different parts/functions following a similar scheme outlined in Sect. 4.7. You should not use Matlab's built-in Runge–Kutta function. The program should be tested for the case: $m = 100\,\text{g}$, $k = 10\,\text{N/m}$, and the friction

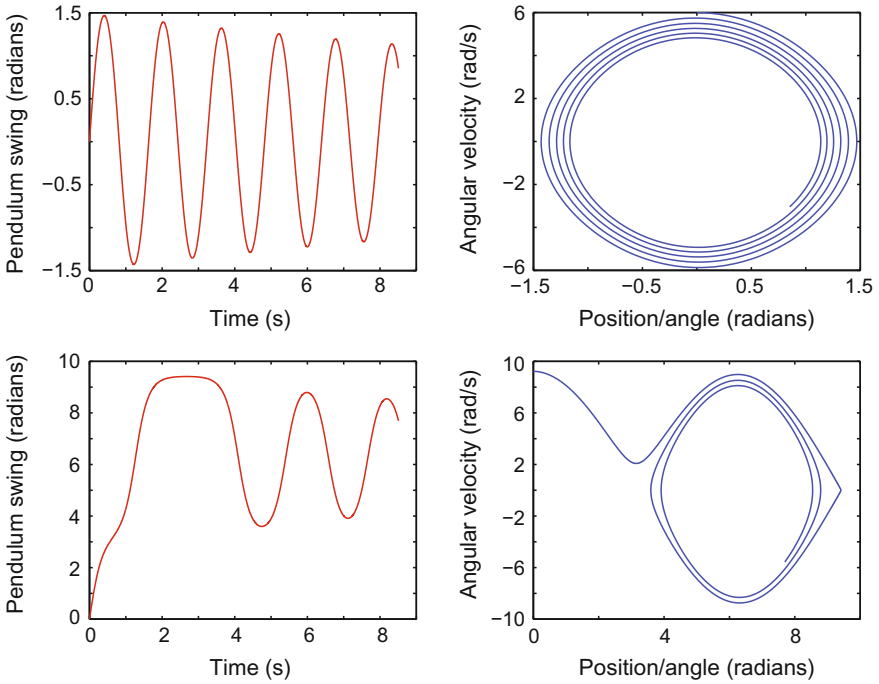

Fig. 4.7 Motion of a simple pendulum. Position vs time is shown to the left and phase space presentation of the motion to the right. See the text for a detailed description

is assumed to be linear with the coefficient of friction $b = 0.10\,\text{kg/s}$. Initial terms are $z(0) = 10\,\text{cm}$ and $[dz/dt]_{t=0} = 0\,\text{m/s}$. Conduct a test of which time steps are acceptable and check if there is agreement between numerical calculations and analytical solution. Put correct numbers, text and units along the axes of the plots. Add a copy of your code.

(b) Modify the program a little and change some parameters so that you can create a figure similar to Fig. 2.5 that shows the time course of the oscillation when we have subcritical, critical and supercritical damping. Explain how you chose the parameters. [We assume that the tests you did in (a) with respect to time resolution and comparison with analytical solutions do not need to be repeated here.]

(c) Modify the program so that it can also handle forced vibration (may last for the entire calculation period). Use $m = 100\,\text{g}$, $k = 10\,\text{N/m}$, $b = 0.040\,\text{kg/s}$ and $F = 0.10\,\text{N}$ in Eq. (3.1). Try to get a plot that corresponds to the initial part of each of the time courses we find in Fig. 3.7.

(d) Use this last version of the program to check that the "frequency response" of the system (à la Fig. 3.8) comes out to be correct, and that you can actually read the approximate Q value of the system from a plot made by you.

5. Write your own program to calculate the time development of a damped oscillator using the fourth-order Runge–Kutta method. Test that it works by comparing the results for analytical solution and numerical solution for a case in which they should be identical. How large is the error in the numerical solution for the position (relative to maximum amplitude)? If you choose the time step Δt, we ask you to test at least two to three different options for Δt to see how much this choice means for accuracy.
6. Carry out calculations of forced oscillations for a variety of different applied frequencies and check that the quality factor expression in Chap. 2 corresponds to the frequency curve and the alternative calculation of Q based on the half-value and centre frequency.
7. Study how fast the amplitude grows by forced oscillations when the applied frequency is slightly different from the resonant frequency. Compare with the time course at the resonance frequency. Initial conditions: the system starts at rest from the equilibrium point.
8. Find out how the calculations in the previous tasks have to be modified if, for example, wanted to incorporate an additional term $-cv^2 \times (\vec{\boldsymbol{v}}/v)$ for the friction. Feel free to comment on why numerical methods have a certain advantage over analytical mathematical methods alone.
9. This task is to check if the superposition principles apply to a swinging spring pendulum with damping, first in the case that the friction can be described only with a $-bv$, that the friction must be described by $-bv - sv^2$, or rather: $-bv - s|v|v$ to take account of the direction (see Chap. 2 where this detail is mentioned). In practice, the task involves making calculations for one swing mode, then for another, and then checking if the sum of solutions is equal to the solution of the sum of states.
   The physical properties of the spring pendulum are characterized by $b = 2.0$, $s = 4.0$, $m = 8.0$ and $k = 73.0$, all in SI units. Make calculations first with the initial conditions $z_0 = 0.40$ and $v_0 = 2.50$, and then the initial conditions $z_0 = 0.40$ and $v_0 = -2.50$. Add the two solutions. Compare this sum with the solution of differential equation when the initial conditions are equal to the sum of the initial conditions we used in the first two runs. Remember to check the superposition principle both for runs where $-s|v|v$ is present and where it is absent. Can you draw a preliminary conclusion and put forward a hypothesis about the validity of the superposition principle based on the results you have achieved?

   Note: In case you use Matlab's built-in solver, the times will not match the two runs. You must then take into account the time series corresponding to one run and use interpolation when the addition of the result for the second run is to be performed. Below is an example of how such an addition can be made. Ask for help if you do not understand the code well enough to use it or something similar in your own program.

```
% Addition of two functions Z1(t) and Z2(t'), where t is
% elements in T1 and t' in T2. The two series have the same
% start value (and end value), but is different elsewhere.
% n1 = length(T1) and n2 = length(T2). The function only
```

```
% works for n2>=n1. Modify the code if that is not the case.

% Use T1 as basis for for the summation
Z12(1)=Z1(1)+Z2(1);
for i = 2:n1
    % Find index to the last point in T2 less than T1(i)
    j = 1;
    kL = -1;
    while kL<0
        if (T2(j)<T1(i)) j=j+1;
        else;
            kL=j-1;
        end;
    end;
    % The first point in T2 is then larger or equal the
    % T1(i) index:
    kH = kL+1;
    % Summation of the two solutions (linear interpolation)
    Z12(i) = Z1(i)+Z2(kL) + (Z2(kH)-Z2(kL))...
    *(T1(i)-T2(kL))/(T2(kH)-T2(kL));
end;
```

4.13.1 An Exciting Motion (Chaotic)

11. Let us look at a nonharmonic "swing" that is beyond analytical mathematics. We consider a ball that is bouncing vertically up and down influenced by gravity, and we assume, for the sake of simplicity, that there is no loss. The special aspect here is that the floor oscillates vertically and has much greater mass than the bouncing ball so that the motion of the floor is not affected by the ball.
The velocity of the floor is described as $u(t) = A\cos(\omega t) = A\cos(\phi(t))$. The ball has a speed of v_i down just before it hits the floor, but according to mechanics, the speed $v_{i+1} = v_i + 2u(t)$ will rise soon after the ball has hit the floor. We assume that the ball bounces so high in relation to the amplitude of the floor that we can make the approximation that the time the ball uses from leaving the floor until it hits the floor again is independent of the position of the floor and depends only on the speed the ball had when it last left the floor. This time is $\Delta t_i = 2v_i/g$ where g is the acceleration due to gravity. Note that Δt varies from bounce to bounce.
With these approximations, the phase difference between the floor oscillation and the oscillations of the ball until their next encounter is:

$$\Delta\phi_i = \Delta t_i\ \omega = \frac{2\omega}{g} v_i \equiv \gamma_i \tag{4.13}$$

where γ is a "normalized velocity" that depends on the constants g and ω and varies as v_i. The term "velocity" is a little misleading, but since g and ω are both constant in our context, γ varies linearly with the velocity of the floor at the instant the ball hits it. When $\gamma_i = 2\pi$, the bounce will equal exactly one period in the oscillation of the floor.
We can then set up the following algorithm to calculate a new bounce based on the knowledge of the previous bounce in the following way:

$$\phi(n+1) = [\text{modulo } 2\pi]\,(\phi(n) + \gamma(n)) \tag{4.14}$$

where [modulo 2π] means that we take the modulo of what we calculate (to ensure that ϕ is in the range of $[0, 2\pi >$). And further:

$$\gamma(n+1) = \gamma(n) + \alpha\cos(\phi(n+1)) \tag{4.15}$$

where $\alpha \propto A$.
In this description, we operate with "normalized velocity" $\gamma(n)$, which is proportional to the initial velocity of each bounce, and with $\phi(n)$, which is the phase of the floor motion just as nth bounce begins. The quantity α is proportional to the amplitude of the floor, and for simplicity we will choose an amplitude corresponding to $\alpha = 1.0$.
We will plot the results in a form of phase plot, but not quite. We let the phase of the oscillation $\phi(n)$ lie along the x-axis and "normalized velocity" $\gamma(n)$ along the y-axis.
Create a plot showing *points* $(\phi(n), \gamma(n))$ for N number of bounces. During the test you can, for example, take $N = 2 \times 10^3$, but when the program works without errors, you may want to expand this to e.g. $N = 2 \times 10^6$ if the calculation time is still acceptable.
Remember to allocate space to the "phi" and "gamma" array before you enter the loop using the algorithm in Eqs. (4.14) and (4.15).
Note: Do not connect the points with lines! Plotting of the points can be done in Matlab, for example, as follows:

```
plot(phi,gamma,'r','MarkerSize',2);
```

Try the following initial conditions for (phi, gamma): (0.0, 1.0), ($\pi/2$, 0.0), (1.4, 1.71), (1.4, 1.75). Also try other initial values to create a picture of various movements that may occur. Try to describe in words different forms of motion.