Many of the fundamental physical laws that are essential for a description of fluid or solid systems in materials science are formulated most conveniently in terms of differential equations. Deriving and solving differential equations are thus common tasks when modeling material systems. Hence, this chapter first provides an overview of differential equations and then several numerical solution techniques – frequently used in materials science – are discussed.

4.1 Basics of Ordinary and Partial Differential Equations in Physics

There are physical quantities which - with appropriately chosen units - can be expressed by a single real-valued number. Very often however, one is looking for an unknown function, the derivation of which yields \( f(x) \). In physics, the quantity \( x \) will most often be the time parameter \( t \) and thus \( f(t) \) describes a process in time. In standard notation this problem can be written as

\[
\frac{dy}{dx} = f(x) .
\] (4.1)

Equations involving one or more scalar or tensorial dependent variables, unknown functions of these variables, as well as their corresponding derivatives, are called differential equations. For a modern and comprehensive treatment of differential equations see e.g. [268, 269]. Equations such as (4.1) with only one dependent and independent variable and their respective derivatives are called ordinary differential equations (ODE)s. The most general notation is the implicit form:

\[
f \left( y^{(n)}(x), y^{(n-1)}(x), ..., y'(x), y(x), x \right) = 0 .
\] (4.2)

The order of a differential equation (DE) is the highest order of any of the derivatives occurring in the equation. For example, the relation between the
motion of a point mass $m$ under the influence of an external force $\vec{F}$ is a second order differential equation. In one dimension, according to Newton’s third axiom

$$ma = m \frac{d^2 x(t)}{dt^2} = F(x, t) .$$  \hfill (4.3)

A linear DE is a DE which is only linear in its unknown function $y(x)$ and its derivatives $y', \ldots$. The most general form of such a DE is obtained when using the linear differential operator

$$L^{(n)}(x) = \sum_{k=0}^{n} f_k(x) \frac{d^k}{dx^k}$$ \hfill (4.4)

in the form:

$$L^{(n)}(x)y(x) = f(x) .$$ \hfill (4.5)

The great importance and abundance of linear differential equations in the natural sciences has some good reasons. Physically, linearity means that phenomena occurring in nature exhibit superposition, e.g. electromagnetic waves in empty space. The realization of the fact that nature itself, to a good deal, “behaves linear” is neither self-evident nor in any way “obvious”. The linearity of many natural phenomena in any case reduces the possible number of useful equations that might be used for theoretical model building in physics and thus leads to a powerful heuristic approach. However, there are also examples of fundamental equations which are highly non-linear, e.g. Einstein’s field equations which were treated in Sect. 3.7 where the gravitational field interacts with the masses which are themselves the sources of the gravitational field.

Linearity mathematically speaking, in essence means that for these equations the superposition principle applies, i.e. linear combinations of solutions are also solutions of the differential equation. These solutions even form a (linear) vector space (for the definition of a linear vector space see Chap. 3). This means that for obtaining all possible solutions to a linear problem, it is sufficient to obtain one particular solution that forms a basis (a “fundamental system”) of this vector space.

An explicit DE is one which can be resolved for the highest occurring derivative, i.e. for which $f^{(n)}$ can be written explicitly as a function of $y^{(n-1)}, \ldots, y$ and $x$.

We now return to the simple DE of the motion of a point mass in (4.3). If the external force $F(x, t)$ is given by the gravitational force $m'g$ where $g$ is the gravitational acceleration directed at the direction of fall, and $m'$ is the gravitational mass, we obtain a one-dimensional ODE of motion:

$$m\ddot{x}(t) = m'g .$$ \hfill (4.6)

All experimental evidence (see the discussion on p. 169) shows that the inertial mass $m$ equals the gravitational mass $m'$, thus:
\[ \ddot{x}(t) = g. \] (4.7)

Integrating this equation yields
\[ \int_{\dot{x}_0}^{\dot{x}} d\dot{x}'(t) = \int_{t_0}^{t} g dt', \] (4.8)
and
\[ \dot{x}(t) - \dot{x}_0 = g(t - t_0). \] (4.9)

A second integration of (4.9) finally yields the equation of motion. Resolved for \( x(t) \) the solution reads
\[ x(t) = \frac{g}{2}(t - t_0)^2 + \dot{x}_0(t - t_0) + x_0, \] (4.10)

where \( \dot{x}_0 \) is the initial velocity at time \( t = t_0 \), \( t_0 \) is the initial time and \( x_0 \) is the initial position of the point mass at \( t = t_0 \).

In this very simple example from classical point mechanics we see that a general solution of an ODE still contains some undetermined constants. Thus, the set of functions \( y(x) \) which solve the ODE builds a family of curves within a domain \( D \subset \mathbb{R} \), cf. Fig. 4.1.

Which one of these solutions is chosen is determined by initial and boundary conditions.

With a special solution these constants are already determined. A function \( y(x) \) which fulfills the initial and boundary value problem of a DE in an

![Fig. 4.1](image)

**Fig. 4.1.** A sample set of curves within \( D = \{a, b\} \) for the solution (4.10) of (4.6) with different initial conditions of the initial position \( x_0 \). The dotted lines are samples for curves with different initial velocity \( \dot{x}_0 \) and the same \( x_0 \).
interval \( I \subset \mathbb{R} \), and which is continuously differentiable, is called a solution of the DE. Hence, with ODEs the complete task of solving an initial-value problem as in the example of the point mass above needs three specifications:

**Initial Value Problem:**
- Differential equation, e.g. \( y'(x) = f(x, y) \),
- Initial condition, e.g. \( y(x_0) = y(t = 0) = y_0 \),
- Domain of the unknown function, i.e. the set of all points \( (x, y) \in D \subset \mathbb{R} \).

Can a differential equation always be solved? This question is answered by Peano’s theorem, see e.g. [270], which states that an initial value problem of ODEs such as in (4.3) has at least one solution on a planar domain \( D \) if \( F(x, y) \) is continuous on \( D \). The advantage of this theorem is that it has weaker premises than required by the Picard-Lindelöf theorem, cf. Box 4.1 which guarantees the existence and uniqueness of the solution to an initial value problem.

**Example 33.** Consider the initial value problem
\[
y' = x\sqrt{y}, \quad y(0) = 0, \quad D = \{ x \in \mathbb{R}, y \geq 0 \}. \tag{4.12}
\]
It is easy to verify that this equation has two solutions, namely \( y = 0 \) and \( y = x^4/16 \). Thus, not every initial value problem has a unique solution. If one simply relies on a computer code, e.g. in some simulation run, then this code will produce a solution, but one does not know which one.

The Picard-Lindelöf theorem (see Box 4.1) provides a sufficient condition for the existence and uniqueness of solutions which avoids the situation in the above example. This theorem states, that the initial value problem of ODEs has a unique solution if the function \( f(x, y) \) obeys a Lipschitz condition on a domain \( D \subset \mathbb{R}^2 \). A Lipschitz condition is a smoothness condition for functions which is stronger than regular continuity. Intuitively, a Lipschitz continuous function is limited in how fast it can change; a line joining any two points on the graph of this function will never have a slope steeper than a certain number called the Lipschitz constant of the function. For more details see e.g. [271].

Another aspect of the solution of ODEs is the question of stability of the solution with respect to small changes in the initial conditions. There are

---

**Box 4.1 Lipschitz condition**

Consider a function \( f(x, y) \) in a domain \( D \subset \mathbb{R}^2 \). If for all pairs \( (y, y_1), (y, y_2) \in D \) the inequality
\[
| f(x, y_1) - f(x, y_2) | \leq L \ | y_1 - y_2 | \tag{4.11}
\]
is valid for any value \( L > 0 \), then \( f(x, y) \) is said to satisfy a Lipschitz condition in \( y \). The number \( L \) is called Lipschitz constant. One form of the Picard-Lindelöf theorem states that the initial value problem has a unique solution if \( f(x, y) \) satisfies a Lipschitz condition in \( y \) on the interval \( D \).
systems of DE where small changes in the initial conditions lead to an exponentially growing deviation of different solutions*. These systems are “chaotic” in nature. In particular when using numerical solution methods the question of stability of solutions is of great relevance.

The same equations of motion as in (4.3) are obtained for a mass point \( m \) when using the canonical Hamilton equations:

\[
\dot{p}_i = -\frac{\partial H}{\partial q^i}, \quad \dot{q}^i = \frac{\partial H}{\partial p_i},
\]

(4.13)

where

\[
H = H(p_i, q^i, t) = E(p_i) + \Phi(q^i, t) \quad (i = 1, 2, 3)
\]

(4.14)

is the total energy of mass point \( m \) and \( p_i \) and \( q^i \) are the \( i \)-th components of the generalized momenta and coordinates in phase space. The Hamilton function of a simple point mass is \( H = \frac{1}{2}mq^2 + \Phi(q, t) \). Inserting this into (4.13) yields Newton’s law \( F_i = -\frac{\partial \Phi}{\partial q_i} \) for the \( i \)-th component of the force.

The physical world is not simply one-dimensional. Most problems in computational physics lead in their mathematical formulation to “partial differential equations” (PDE)s which involve derivatives with respect to several (space and time) variables. The Dirac equation of relativistic quantum mechanics is an example of a PDE of first order. The Schrödinger equation of non-relativistic quantum mechanics is an example of a second order PDE. While for the solution of PDEs of first order there is a number of different standard procedures, for the solution of second or higher order PDEs one has to adopt an individual approach. PDEs of up to second order can be written in a general form as:

\[
\sum_{i,j=1}^{k} \alpha_{ij} \partial_i \partial_j F(x_1, ..., x_k) = g(x_1, ..., x_k),
\]

(4.15)

where \( \alpha_{ij} = \alpha_{ji} \) are the elements of a symmetric matrix. With this matrix one can write down the quadratic form

\[
F(x_1, ..., x_k) = \sum_{i,j=1}^{k} \alpha_{ij} x_i x_j.
\]

(4.16)

If \( F \) is a constant then (4.16) describes higher order curves in \( k \) dimensions. Whether these curves assume an elliptic, hyperbolic or parabolic character, depends on the signs of the always existent Eigenvalues of the matrix \( \alpha_{ij} \). Depending on the relative signs of their respective derivatives, PDEs are associated with either one of the three categories of higher order curves. The most common partial differential equations in physics contain derivatives up to second order and can be grouped into and are named after one of the three categories:

*For example, in molecular dynamics simulations, particle trajectories, which were initially very close, diverge exponentially with time. This well-known deviation of trajectories in MD is called Lyapunov instability.
4.1.1 Elliptic Type

A prototype of an elliptic PDE in physics is the

\[
\text{Laplace equation} \quad \Delta \phi = 0 , \tag{4.17}
\]

which, in Cartesian coordinates \((x, y, z)\) and in three dimensions reads:

\[
\left( \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2} \right) \phi(x, y, z) = 0 \tag{4.18}
\]

In electrostatics, this equation describes the electric potential in a domain without charges. The distribution of charges is given by boundary conditions – the values of the potential at the boundary of the domain. In thermodynamics, the solution of (4.17) characterizes the temperature distribution at equilibrium in a domain without heat sinks or sources; thus, it is the stationary solution of the heat diffusion equation. If one introduces explicit charges, i.e. sources of the electric potential, into (4.17) one obtains the

\[
\text{Poisson equation} \quad \Delta \phi(\vec{x}) = \rho(\vec{x}) \tag{4.19}
\]

in SI-units with \(\rho(x)\) being the charge density. In the often used cgs-system there is an additional factor \(4\pi\) at the right side of (4.19)\(^1\). Point charges are denoted with delta functions.

**Elliptic PDEs describe pure boundary value problems**, i.e. the solution function \(\phi\) obeys certain conditions at the boundary of the considered domain \(D\). The solution function is typically independent of time, thus elliptic equations describe stationary situations. The specific type of boundary conditions is determined by the PDE and depends on the specific mathematical or physical problem. Two main types of boundary conditions are distinguished:

**Dirichlet boundary conditions,**

where the values of the function \(\phi\) are given at the boundary \(\partial D\):

\[
\phi(\partial D) = ... , \tag{4.20}
\]

and **von Neumann boundary conditions.**

\(^1\) For a thorough discussion of systems of units in classical electrodynamics, see e.g. [272].
Here, the values of the derivative of the function $\phi$ are given normal to the boundary curve or boundary area:

$$\nabla \phi \cdot \mathbf{n}|_{\partial D} = ... . \quad (4.21)$$

Also mixed type boundary conditions are possible. It can be shown that the solution of an elliptic PDE is uniquely determined by specification of either von Neumann or Dirichlet boundary values at each point of the total boundary of the domain.

**Remark 31.** A general formal solution of the Poisson equation (4.19) can be obtained by using the Green’s function $G(\vec{x} - \vec{x}')$, which obeys (4.19) for a unit point charge:

$$\Delta (G(\vec{x} - \vec{x}')) = \nabla^2 (G(\vec{x} - \vec{x}')) = -4\pi \delta(\vec{x} - \vec{x}'). \quad (4.22)$$

$G(\vec{x} - \vec{x}')$ is given by

$$G(\vec{x} - \vec{x}') = \frac{1}{|\vec{x} - \vec{x}'|} + F(\vec{x}, \vec{x}'), \quad (4.23)$$

where the function $F(\vec{x}, \vec{x}')$ has to obey the Poisson equation within the considered integration volume $V$. The first term in (4.23) is the potential of a unit point charge. Thus $F(\vec{x}, \vec{x}')$ is the potential of an external charge distribution, i.e. a charge source outside of $V$, cf. Fig. 4.2.

Using definition (4.23) and Green’s theorem, the formal solution of (4.19) can be written as [272]:

$$\Phi(\vec{x}) = \int_V \rho(\vec{x}') G(\vec{x}, \vec{x}') d^3 x' + \frac{1}{4\pi} \oint_A \left[ G(\vec{x} - \vec{x}') \frac{\partial \Phi}{\partial n'} - \Phi(\vec{x}') \frac{\partial G(\vec{x} - \vec{x}')}{\partial n'} \right] d\alpha'. \quad (4.24)$$

While the particular solution $\Phi(\vec{x}, \vec{x}') = \frac{1}{|\vec{x} - \vec{x}'|}$ satisfies the Poisson equation (4.22), none of the boundary conditions are satisfied, as the surface $A$ lies at infinity. However, by the definition of the Green’s function in (4.23) and its additional term $F(\vec{x}, \vec{x}')$ one has the freedom to eliminate one of the two surface integrals in (4.24) by an appropriate choice of $F$, either corresponding to von Neumann or to Dirichlet boundary conditions. In practice however, it is often very difficult, if not impossible, to determine $G(\vec{x} - \vec{x}')$ as it depends on the shape of the considered integration surface $A$.

### 4.1.2 Parabolic Type

Here, aside from the spacial derivatives in the Laplace operator there is an additional derivative of first order; in most typical physical applications this will be a derivative with respect to time $t$. 
Fig. 4.2. Physical meaning of the function $F(\vec{x}, \vec{x}')$ in (4.23). $F(\vec{x}, \vec{x}')$ solves the Laplace equation within the domain on the left and thus is the potential of a charge distribution outside the volume $V$, which is chosen in such a way that $F(\vec{x}, \vec{x}')$ along with the potential $\frac{1}{|\vec{x} - \vec{x}'|}$ of a unit point charge located at $\vec{x}'$ leads to a value of the Green’s function $G_D(\vec{x}, \vec{x}') = 0$ and $\frac{\partial}{\partial n} G_N(\vec{x}, \vec{x}') = -4\pi/A$ at the boundary ($\partial D$) for which $\vec{x} = \vec{x}'(\partial D)$. $A$ is the area of the considered volume, subscripts $D$ and $N$ stand for Dirichlet and von Neumann boundary conditions and the value $-4\pi/A$ for the latter case is due to the Gaussian law of electrodynamics. The function $F(\vec{x}, \vec{x}')$, i.e. the external charge distribution depends on parameter $\vec{x}'$, which points to the point charge distribution in $V$.

**Diffusion equation**

$$\Delta \phi - \frac{1}{\kappa} \frac{\partial \phi}{\partial t} = 0 \, ,$$

(4.25)

with $\Delta$ being the Laplace operator in $k$ dimensions and $\phi$ being dependent of $k$ spacial variables and of time $t$. The solution of this equation might be a temperature distribution that depends on time $t$. In the limit $t \to \infty$ a stationary state will be reached, thus in the limit of $t \to \infty$ the ordinary Laplace equation has to be fulfilled: $\lim_{t \to \infty} \frac{\partial \phi}{\partial t} = 0$. Most often the physical situation requires a statement on the initial value of $\phi$ at time $t = 0$ in the total domain $D$:

$$\phi(x, y, ..., t_0) = ... \, .$$

(4.26)

Additionally one may restrict the solution further by providing geometric restrictions to the domain $D$ for points in time with $t > t_0$. The values of the solution at some time $t_1 > t_0$, $\phi(t = t_1)$ are part of the solution and need not be provided. Considering the total domain of $k + 1$ dimensions, boundary conditions are needed only for an open subset of the surface, namely

$$\phi(D, t = t_0) \, ,$$

(4.27)
4.1 Basics of Ordinary and Partial Differential Equations in Physics

Fig. 4.3. Parabolic differential equations need as input an initial value, e.g. at time \( t = t_0 \). If additional boundary values are provided, the equation constitutes a mixed initial/boundary value problem

\[
\phi(\partial D, t_0 < t < t_1),
\]

but no specification of \( \phi(t_1) \) is needed in order to definitely determine the solution. If no boundary values are given the problem is an initial value problem, cf. Fig. 4.3.

4.1.3 Hyperbolic Type

A hyperbolic PDE contains – besides the Laplace operator – a second derivative with respect to time, which has the opposite sign with respect to the spacial derivatives. In one dimension, this equation reads:

\[
\text{Wave equation} \quad \Delta \phi - \frac{1}{c^2} \frac{\partial^2 \phi}{\partial t^2} = g(x, t).
\]

In physics this equation describes the motion of waves in spacetime. Parameter \( c \) is the wave propagation speed. If \( g = 0 \) the equation is called homogeneous, otherwise inhomogeneous. The function \( \phi(x, t) \) in one dimension might describe the elongation of a point of an oscillating linear chain of mass points. In three spacial dimensions \( \phi(\vec{x}) \) can describe sound waves (density fluctuations) or the components of the electromagnetic field.

A hyperbolic PDE describes a typical initial value problem. A solution requires the initial values of both \( \phi \) and its derivatives (Cauchy-problem) and - depending on the specific problem - boundary values at the boundary of the domain.
Plane Waves

Consider the homogeneous wave equation in one dimension:

$$\frac{\partial^2 \phi(x,t)}{\partial x^2} - \frac{1}{c^2} \frac{\partial^2 \phi(x,t)}{\partial t^2} = 0.$$  \hspace{1cm} (4.30)

One can prove easily by insertion that any two times differentiable function of the form $f(x - ct)$ or $f(x + ct)$ is a solution of (4.30), see Problem 4.1. The function $f(x - ct)$ describes a function which, without changing its shape, moves with constant velocity in positive $x$ direction and $f(x + ct)$ describes a similar function moving in negative $x$ direction. The solution $\phi$ depends on the phase $\varphi_\pm = x \pm ct$ and moves with velocity $c$ in positive ($\varphi_-$) or negative ($\varphi_+$) $x$-direction.

Remark 32. Also non-linear PDEs may have special solutions which propagate through space with constant velocity. Such solutions are called solitary. In contrast to the linear wave equation (4.30) which is solved by any function that moves with velocity $c$, solitary solutions always have a special form $\phi(u)$. In some special cases solitary solutions might even superimpose without interfering with each other. In this case one speaks of solitons.

The general solution of the wave equation in three dimensions due to the above said is a function

$$\phi(\vec{x},t) = f_+(\vec{x} - ct) + f_-(\vec{x} + ct).$$ \hspace{1cm} (4.31)

As any function $f(\vec{u}_\pm)$ solves the wave equation, so do the periodic functions

$$f_\pm(\vec{x},t) = C_\pm e^{i(\vec{k}\cdot\vec{x} - \omega t)},$$ \hspace{1cm} (4.32)

with a complex prefactor $C_\pm = \tilde{C}_\pm e^{i\alpha}$ and the phase $\alpha$ which is determined only up to modulo $2\pi$. This solution has the property of being spatially periodic with wave length $\lambda = 2\pi/k$ ($k = |\vec{k}|$) for a fixed value $t_0$ of time and it is temporally periodic with the period $T = 1/\nu = 2\pi/ck = \lambda/c$ for a fixed position $\vec{x}_0$. It is

$$\nu = \frac{1}{T} = \frac{c}{\lambda} = \frac{ck}{2\pi},$$ \hspace{1cm} (4.33)

and

$$\omega = 2\pi \nu = kc,$$ \hspace{1cm} (4.34)

where $\nu$ is the frequency, $k$ is the wave number and $\omega = |\vec{\omega}| = c \ |\vec{k}|$ is the angular frequency of the solution.

We consider in the following only the partial solution $f_+$. If the phase $\varphi_+$ is constant, then the surfaces of constant phase are given by the condition $\vec{k}\vec{x} = \text{const}$. This is the equation of a surface (wavefront) perpendicular to $\vec{k}$. For all points $\vec{x}$ for which the projection $\vec{k}\vec{x}$ onto the direction of $\vec{k}$ has the same value, the solution $f_+$ is a constant and is called a plane wave solution, cf. Fig. 4.4.
Spherical Waves

Another important solution of the wave equation (4.29) are spherical waves. Such a solution is of the form (see Problem 4.2)

$$\phi_{\pm} = \frac{A_{\pm}}{r} e^{i(kr \pm \omega t)}$$  (4.35)

with $r = |\vec{x}|$ and the following properties:

1. The phase $\phi_{\pm} = kr \pm \omega t$ only depends on the absolute value of the position vector $r = |\vec{x}|$. For a fixed time $t = t_0$ the points of equal phase, i.e. points for which $\phi$ has the same value, are the ones that have equal distance from the origin. These points lie on a sphere with radius $r$.
2. The amplitude of a spherical wave decreases with increasing distance from the origin as $1/r$.
3. The phase velocity $u_{\pm}$ of the spherical wave front is given by the velocity of all points with equal phase, i.e. $u_{\pm} = \frac{d\phi_{\pm}}{dt} = \frac{dr}{dt} \pm \omega = 0$. The solution (4.35) thus is an undulation with spherical wave fronts that propagate with phase velocity $u_{\pm}$, where $u_+$ is called outbound spherical wave and $u_-$ is called inbound spherical wave.

4.2 Numerical Solution of Differential Equations

An analytic closed-form solution of differential equations is only possible in very rare cases. Sometimes certain symmetry conditions allow for a simplification of a problem, e.g. by reducing the number of dimensions. A good
overview of many analytically solvable ODEs can be found in Kamke [273] or Boas [274]. For PDEs, Courant and Hilbert [275] and Morse and Feshbach are classics [276]. Farlow [277] discusses many analytic approaches to PDEs, including the separation of variables. The general solutions of PDEs have to take into account boundary values. Very often, it is possible to formulate an equivalent integral equation which implicitly contains the boundary conditions. The reformulation of PDEs as equivalent integral equations is discussed in Courant and Hilbert [275] and Porter and Stirling [278]. However, for most applications of practical interest with complex geometries involved, analytic solutions of PDEs are very rarely possible, and one has to make use of numerical methods. For a good overview of numerical solution methods of ODEs and PDEs see e.g. Morton and Mayers [279], Stoer and Bulirsch [280], Press et al. [281], Burden and Faires [282], Gould and Tobochnik [283] or Cohen [284]. For a compact and succinct presentation of numerical methods, see e.g. Abramovitz and Segun [285] or Bronstein and Semendjajew [286]. As an example for a program package devoted to the numerical and analytical treatment of ODEs the reader is refered to Wolfram [287]. A corresponding detailed discussion of ODEs can be found in Abell and Braselton [288] and PDEs are treated in Vvedensky [289].

In the next section, we describe a number of different numerical techniques which are commonly used in simulation codes to obtain approximate solutions to boundary and initial value problems. Such problems are abundant in materials science, e.g. when solving the classical Newtonian equations of motion (discussed in detail in Sect. 6.4.1), or some other basic physical equations.

4.2.1 Mesh-based and Mesh-free Methods

When numerically solving differential equations on a computer one has to handle two principal problems: First, the evaluation of derivatives which are actually defined by a limiting process, and second, the mapping of a continuous variable time onto a discrete scheme with intervals of length \( h = \Delta t \). Hence, all numerical techniques for solving differential equations are based on the discretization of the time and space variables and the replacement of differential operators by difference operators.

One possible classification of methods could be done by distinguishing methods that are either based on discrete particles, so-called mesh-free methods – typically used in basic research in physics, chemistry, biology or genomics – and methods with a mesh-based discretization of the domain for which a solution is sought. The latter methods, based e.g. on finite elements (FE) have been used almost exclusively in engineering applications for more than sixty years since their first introduction which is generally attributed to A. Hrennikov [290] and R. Courant [291]. Finite element methods (FEM) – discussed in Sec. 4.2.4 – are designed to solve numerically both, complex boundary-value, and initial-value problems. The common feature of all FEM is the reformulation of the investigated differential equation as a variational
problem. Variational problems have solutions in a special class of function spaces, the so-called Sobolev spaces, cf. the discussion in Chap. 3. For a numerical treatment, a minimization of the variational problem is done in finite-dimensional subspaces, the so-called finite element spaces. Characteristic for the numerical treatment is a discrete division of the spacial domain $\Omega$ into a number of sub-domains, the so-called finite elements. Within these elements one considers polynomial trial functions as an approximation of the “true” solution.

Finite difference methods and even more finite element methods – in particular within typical engineering applications – are generally associated with the treatment of (spacial) large-scale problems. While finite element methods are in fact prevailing in virtually all engineering applications on the meso- and macroscale in industry and science (and are often in fact the only modeling tools available to engineers) it is very important to understand that there is no intrinsic length or time scale associated with FEM, or generally, with any numerical mesh-based method as such. The same is true for particle methods such as molecular dynamics (MD), smooth particle hydrodynamics (SPH) or discrete element methods (DEM). Thus, in principle, one could also use finite elements for the modeling of atomic systems in thermodynamics or one could as well use a particle based approach – solving Newtonian equations of motion – for any large-scale structural problem. It is the underlying physics that determines a length (and associated time-scale) of a problem, whereas the different numerical schemes for the solution of basic equations are not intrinsically coupled to any specific scale. This is the reason why we will discuss the FEM method as such separately in this chapter in Sect. 4.2.4 and not in the corresponding chapters devoted to meso- and macroscales in Part II of this book. In those chapters we will focus on the physical foundations of continuum theory and the phenomenological assumptions (so-called constitutive equations) on macroscopic material behavior that are used along with finite element approaches.

One of the main limitations of a finite element approximation is that it only works on a prearranged topological environment – a mesh – which is a rather artificial constraint to ensure compatibility of the finite element interpolation. A mesh itself is not compatible, and in fact is often in conflict with the real physical compatibility condition that a continuum posesses. As soon as conflicts between the mesh and a physical compatibility occur, a FEM simulation is stopped due to numerical instabilities, and remeshing becomes inevitable. This again not only is a time consuming process but also leads to a degradation of computational accuracy, finally tainting the numerical results. A too large distortion of a mesh usually leads to numerical instabilities which is often the case e.g. in shock wave applications which are discussed in Chap. 8. Since the beginning 90s of the last century there has been an increasing interest in the development of meshfree interpolant schemes that can relieve the burden of successive remeshing and mesh generation, which have led to serious technical obstacles in some FE procedures such as adaptive refinement.
and simulations of solid objects moving in a fluid field. The investigation of meshfree interpolants by e.g. Needleman [292], Liberski et al. [293], Nayroles et al. [294], Belytschko et al. [295, 296], Lui et al. [297] and others have led to the adaptation of meshfree methods – which were originally developed in other research areas – such as Smooth Particle Hydrodynamics (SPH) or discrete element methods (DEM) (see Chap. 7) to solid mechanics problems. During the past decade, meshfree approaches have attracted much attention due to their applications to computational failure mechanics, crack growth, multiscale computations and even nano-technology. As a growing field within engineering applications, meshfree methods have shown promising potential to become a major numerical methodology in engineering computations.

Generally speaking, it seems obvious that a good simulation program requires a good algorithm to integrate the underlying basic equations, e.g. Newton’s equation of motion. At first sight, one might think, that speed of an algorithm is important, but this is usually not the case, as the time spent with integrating the equations of motion is small compared to the time needed to search for interacting particles, cf. Sect. 6.4.1. Although it seems easy to recognize a bad algorithm, it is not directly clear what criteria a good algorithm should satisfy. The main demands on an integrator are generally.

- Energy conservation
- Accuracy for large timesteps
- Numerical efficiency
- Time reversibility
- Low memory requirements

Energy conservation is an important issue when solving physical equations of motion, described e.g. using Hamiltonian dynamics. Sophisticated higher order integration schemes often have an excellent energy conservation for short times while they show an undesired (and sometimes catastrophic) long term energy drift [47]. Accuracy of an algorithm for large time steps is important, because the larger the time step one can use, the fewer calculations of the interactions have to be done per unit of simulation time. Higher order schemes achieve this by storing information on the higher-order derivatives of the particle coordinates. It is important to understand that too high an accuracy of the integration algorithm is not necessary in MD simulations, as with MD, one is primarily interested in thermodynamic averages, i.e. statistical properties of observables and not in an exact prediction of particle trajectories\(^2\). The trajectories, no matter how close at the beginning of a simulation, will diverge exponentially from their “true” trajectory which is caused by the numerical rounding errors. This well-known instability of particle trajectories is

\(^2\) In this respect, MD simulations differ fundamentally from other numerical schemes for, say, predicting the trajectory of satellites through space. Here, of course, one is interested in the exact ballistic curve of the considered object and cannot afford to just make statistical predictions on their whereabouts.
called Lyapunov instability. However, there is considerable numerical evidence (but no formal proof), that the results of MD simulations are representative of a “true” trajectory in phase space, albeit one cannot tell a priori which one [298, 299]. Thus, the confidence in molecular dynamics simulations is based largely on numerical evidence.

True Hamiltonian dynamics also leaves the size of a volume Ω in phase space, spanned by the coordinates \( \vec{r}_i \) and momenta \( \vec{p}_i \), invariant\(^3\). Many non-symplectic, i.e. non-time-reversible integration schemes do not reproduce this area-preserving property. Area-preserving, time-reversible algorithms do not change the phase space volume. Last but not least, the limited precision of representing real numbers on a computer by bits of memory, cf. Fig. 4.5 results in unavoidable rounding errors, which is why no implementation of a symplectic algorithm is truly time-reversible.

The basic equations of motion are given in Cartesian coordinates by the canonical equations

\[
\dot{\vec{r}}_i = \frac{\partial H}{\partial \vec{p}_i}, \quad \dot{\vec{p}}_i = -\frac{\partial H}{\partial \vec{r}_i},
\]

where the partial derivatives \( \partial \vec{r}_i \) and \( \partial \vec{p}_i \) are a symbolic (physicists’) notation for the derivative with respect to each of the \( i = N \) components of the generalized coordinates \( \vec{r}_i \) and momenta \( \vec{p}_i \). \( H = K + \Phi \) is the Hamiltonian, the sum of kinetic and potential energy for an ergodic system. Starting point for a discretization and subsequent MD simulation of a \( N \)-body system are the Newtonian equations of motion. The second axiom states these equations as

\(^3\) This is Liouville’s theorem, Theorem 7 on p. 280.

![Fig. 4.5. Single precision representation of a real number on a computer. (a) On 32-bit systems, for a single precision variable 32 bits of memory are allocated. Thus, there are 8 significant figures and a maximum exponent of +127. (b) Double precision increases accuracy by allocating 64 bits of memory. In this case there is a maximum of 16 significant figures and a maximum exponent of ±324](image-url)
\[ m_i \ddot{r}_i = -\frac{\partial \Phi}{\partial r_i} = \vec{f}_i, \quad (4.37) \]

where \( f_i \) are the forces, \( \Phi \) is the total potential and \( m_i \) the particles’ masses.

Several schemes for an integration of this system of coupled differential equations exist \([51, 300]\). Generally, one likes to make the time step during the integration as large as possible in order to quickly sample the phase space of the system. This is the reason why a low order integration scheme is more advantageous to this purpose, i.e. an integration scheme that does not involve calculating and storing higher order derivations of the positions or velocities. Such an integrator allows for a higher time step without jeopardizing energy conservation. If the system has reached equilibrium, measurements of observables are done. Due to the ergodic and chaotic nature of classical trajectories, it is useless to follow the detailed trajectory of single atoms or molecules for a longer time, as initially nearby trajectories of particles diverge from each other exponentially. Instead, the propagation of particle positions serves primarily for a sufficient sampling of the system’s phase space.

Standard Hamiltonian dynamics, cf. e.g. \([301]\), describes the flow of a representative \( N \)-particle systems through phase space with the classical statistical distribution function \( \rho(\vec{r}^N, \vec{p}^N, t) \), which obeys the Liouville equation

\[ \frac{\partial \rho}{\partial t} = \sum_i \left[ \dot{\vec{r}}_i \frac{\partial}{\partial \vec{r}_i} + \dot{\vec{p}}_i \frac{\partial}{\partial \vec{p}_i} \right] =: -i \mathcal{L}. \quad (4.38) \]

\( i \mathcal{L} \) is the Liouville operator which can be split in two parts \([302]\), the momentum part \( i \mathcal{L}_p \) and the coordinate part \( i \mathcal{L}_r \) such that

\[ i \mathcal{L} = i \mathcal{L}_p + i \mathcal{L}_r \quad (4.39) \]

\[ = \sum_i \dot{\vec{p}}_i \frac{\partial}{\partial \vec{p}_i} + \sum_i \dot{\vec{r}}_i \frac{\partial}{\partial \vec{r}_i} \]

\[ = \sum_i f_i \frac{\partial}{\partial \vec{p}_i} + \sum_i m_i^{-1} \vec{p}_i \frac{\partial}{\partial \vec{p}_i}. \]

The time evolution of a dynamic variable \( A = A(\vec{r}^N, \vec{p}^N, t) \) in Hamiltonian dynamics can be written as:

\[ \dot{A} = \sum_{i=1}^N \dot{r}_i \frac{\partial A}{\partial r_i} + \dot{p}_i \frac{\partial A}{\partial p_i} = i \mathcal{L} A \quad (4.40) \]

\[ \Leftrightarrow A(t) = \exp(i \mathcal{L} t) A(0). \quad (4.41) \]

The exponential operator \( \exp(i \mathcal{L} t) \) is called a propagator, due to the following identity:

\[ \exp(i \mathcal{L} t) = \lim_{k \to \infty} [\exp(i \mathcal{L}_p t/2) \exp(i \mathcal{L}_r t) \exp(i \mathcal{L}_p t/2)]^k. \quad (4.42) \]
4.2 Numerical Solution of Differential Equations

Using a small discrete timestep, 4.42 can be rewritten as an approximation to \( \exp(iL\Delta t) \) using the Trotter identity:

\[
\exp(iL\Delta t) = \exp(iL_p + iL_r) \Delta t = \exp(iL_p \Delta t/2) \exp(iL_r \Delta t) \exp(iL_p \Delta t/2).
\]

Thus, the Liouville operator propagates the equation of motion in several steps which ignore, in turn, the kinetic and the potential part of the Hamiltonian. Applying first the operator \( \exp(iL_p \Delta t/2) \) to the observable \( A \) one obtains:

\[
\exp(iL_p \Delta t/2) A [\vec{r}_i(t), \vec{p}_i(t)] = A [\vec{r}_i(t), \vec{p}_i(t) + \vec{p}_i(t + \Delta t/2)].
\]

The next part of the operator applied to \( A \) yields

\[
\exp(iL_r \Delta t) A [\vec{r}_i(t), \vec{p}_i(t) + \vec{p}_i(t + \Delta t/2)] = A [\vec{r}_i(t) + \vec{r}_i(t + \Delta t), \vec{p}_i(t) + \vec{p}_i(t + \Delta t/2)].
\]

Finally one obtains with the third step

\[
\exp(iL_p \Delta t/2) A [\vec{r}_i(t) + \vec{r}_i(t + \Delta t), \vec{p}_i(t) + \vec{p}_i(t + \Delta t/2) + \vec{p}_i(t + \Delta t/2)].
\]

From the previous steps one has obtained one of the most commonly used algorithms to integrate the classical Hamiltonian equations of a system, namely the **Verlet velocity algorithm**. Written in standard notation (dropping the vector symbol and index \( i \)), this algorithm consists of four steps, cf. Algorithm 4.1.

The Verlet algorithm exists in various, essentially equivalent versions, based on a Taylor expansion of the position vector, including the originally proposed method [35, 303] (see also Problem 4.3), the above velocity version and a leapfrog form [46], see (4.61) on p. 202. For almost all applications using molecular dynamics simulations, the Verlet algorithm is usually the best and

**Algorithm 4.1** The velocity Verlet algorithm

1. \( p(t + \frac{1}{2} \Delta t) = p(t + \frac{1}{2} \Delta t) f(t) \)  
2. \( r(t + \Delta t) = r(t) + \Delta t p(t + \frac{1}{2} \Delta t) \)  
3. \( f(t + \Delta t) = f \left( r(t) + \Delta t), p(t + \frac{1}{2} \Delta t) \right) \)  
4. \( p(t + \Delta t) = p(t + \frac{1}{2} \Delta t) + \frac{1}{2} \Delta t f(t + \Delta t) \)
the algorithm of choice. Occasionally however, one might want to use a larger
time step without loss of accuracy. As mentioned before, higher order schemes
or often not reversible or area conserving and require more memory. To eval-
uate the different merits of algorithms, the reader is referred to a review by
Berendsen and van Gunsteren [304].

4.2.2 Finite Difference Methods

The numerical solution of a differential equation involves the *discretization*
of the independent variables of the equation (usually time and space coordi-
nates). In numerical analysis the term *discretization* refers to passing from
a continuous problem with an infinite number of degrees of freedom to one
considered at only a *finite* number of points which may or may not be located
on a regular discrete lattice. Thus, the discretization reduces the differential
equation to a system of algebraic equations. These equations determine the
values of the solution at a finite number of points within the domain $\Omega$ where
a solution is sought. One of the simplest techniques to discretize a differen-
tial equation is based on a discrete approximation of the differential operator
in the equation. Thus, this method substitutes the temporal and spacial differe-
tial operators of an equation by their finite parts of the infinite Taylor
Series of the respective differential operators. Such a technique is called *finite
difference method*. The variety of finite difference methods currently in use
for many different problems in engineering and science [279, 285, 284, 47, 55]
is evidence that there is no single algorithm superior under all different con-
tions encountered in practice. Thus, experience is needed to do the right
choices for certain problems.

*Example 34* (Finite difference discretization in one dimension). In the one-
dimensional case, the domain $\Omega$ on which a differential equation is to be
solved is an interval $\Omega := [0, a] \subset \mathbb{R}$. The interval is subdivided into $i_{max}$
subintervals of equal size $\delta x = a/i_{max}$. Thus, one obtains a one-dimensional
grid as depicted in Fig. 4.6.

The same scheme applies for a discretization of the time variable in the
case $\delta x = \Delta t$ Recalling the derivative of a function $u : \mathbb{R} \to \mathbb{R}$

$$
\frac{du}{dx} = \lim_{\delta x \to 0} \frac{u(x + \delta x) - u(x)}{\delta x},
$$

(4.51)

Fig. 4.6. Equispaced one-dimensional grid with $i_{max} = 6$
one can approximate the differential operator at grid point $x_i$ by its difference counterpart

$$\left[ \frac{du}{dx} \right]_i^f = \frac{u(x_{i+1}) - u(x_i)}{\delta x} \text{ (forward difference)}, \quad (4.52)$$

where $x_{i+1} = x_i + \delta x$ is the right neighboring grid point of $x_i$. Instead of this forward difference one could also use the backward difference

$$\left[ \frac{du}{dx} \right]_i^b = \frac{u(x_i) - u(x_{i+1})}{\delta x} \text{ (backward difference)}, \quad (4.53)$$

or the central difference

$$\left[ \frac{du}{dx} \right]_i^c = \frac{u(x_{i+1}) - u(x_{i-1})}{\delta x} \text{ (central difference)}. \quad (4.54)$$

From (4.52) to (4.54) – which are illustrated in Fig. 4.7 – it is immediately clear that a refinement of the grid, i.e. reducing the stepsize $\delta x$, results in a better approximation of the differential quotient.

The discretization error or truncation error\(^4\), i.e. the difference between the “true” solution and the approximate finite difference solution for the forward and backward difference is of the order $O(\delta x)$. Thus, in order to halve the error one has to halve the stepsize $\delta x$. The error of the central difference is of the order $O(\delta x^2)$. In this case, halving the stepsize $\delta x$ reduces the error by a factor of four.

\[
\text{Fig. 4.7. The finite difference scheme as an approximation of a derivative}
\]

\(^4\) This term reflects the fact that a finite difference operator is simply a finite part of a Taylor expansion of the differential operator.
Algebraic Equations

A finite difference approximation scheme provides an algebraic equation at each node; this equation contains the variable value at this node as well as values at neighboring nodes and may also include some non-linear terms. The general form of this equation is:

\[ A_P \Phi_P + \sum_l A_l \phi_l = Q_P , \quad (4.55) \]

where \( P \) denotes the node at which the partial differential equation is approximated and index \( l \) runs over the neighbor nodes involved in finite difference approximations. The coefficients \( A_l \) involve geometrical quantities, fluid properties and, in the case of non-linearity, the values themselves. \( Q_P \) contains all those terms which do not contain unknown variable values. The numbers of equations and unknowns must be equal, i.e. there has to be one equation for each grid node, which results in a large set of linear algebraic equations, that is sparse – each equation contains only a few unknowns. The algebraic system can be summarized as follows:

\[ A \Phi = Q , \quad (4.56) \]

where \( A \in \mathbb{R}^{(n \times n)} \) is the square sparse coefficient matrix, \( \Phi \) is a vector containing the variable values at the grid nodes, and \( Q \) is the vector, containing the terms on the right hand side of (4.55). The structure of matrix \( A \) is dependent upon the ordering of variables in the vector \( \Phi \). For structured grids, a lexicographic ordering leads to a poly-diagonal structure. The variables can be most efficiently stored in one-dimensional arrays. The usual conversion between the different node locations on the grid, cf. Fig. 4.8, and the storage location in computer memory, see Table 4.1.

For unstructured grids, the coefficient matrix remains sparse, but it has no longer the diagonal structure.

4.2.3 Finite Volume Method

The finite volume method (FV) is very common in computational fluid dynamics simulations when one wants to approximate the fluxes propagating through surfaces of volumes. Fluid dynamics is a field theory based on the general conservation equation of for a quantity \( \phi \) in their integral formulation, cf. Chap. 8. Here, the solution domain is subdivided into a grid of a finite number of contiguous control volumes (CVs), and the conservation equations are applied to each CV. At the centroid of each CV lies a computational node at which the variable values are to be calculated. Interpolation is used to express variable values at the CV surface in terms of the nodal (CV-center) values. Surface and volume integrals are simply approximated employing suitable quadrature formulae. Hence, one obtains as a result an
algebraic equation for each CV, in which the nodal values of neighbor nodes appear. An advantage of this method is that it can accommodate any type of grid, so this grid-based method can be applied to any type of geometry. The grid itself defines only the control volume boundaries and need not be related to a coordinate system. The method is conservative by construction, so long as surface integrals which represent the convective and diffusive fluxes, are the same for the CVs sharing the boundary. The FV approach is probably one the simplest approaches possible and very straightforward to program. This method is especially popular in computational engineering applications fundamentally based on meshing the considered solution domain with a finite number of small control volumes, cf. Fig. 4.9.

The approximation of surface integrals is straightforward. The CV surface in 2d (3D) can be subdivided into four (six) plane faces, with respect to the

<table>
<thead>
<tr>
<th>Grid location</th>
<th>Compass Notation</th>
<th>Memory location</th>
</tr>
</thead>
<tbody>
<tr>
<td>i,j,k</td>
<td>P</td>
<td>(l = (k-1)N_jN_i + (i-1)N_j + j)</td>
</tr>
<tr>
<td>i,j+1,k</td>
<td>N</td>
<td>(l + 1)</td>
</tr>
<tr>
<td>i-1,j,k</td>
<td>W</td>
<td>(l - N_j)</td>
</tr>
<tr>
<td>i,j-1,k</td>
<td>S</td>
<td>(l - 1)</td>
</tr>
<tr>
<td>i+1,j,k</td>
<td>E</td>
<td>(l + N_j)</td>
</tr>
<tr>
<td>i,j,k-1</td>
<td>B</td>
<td>(l - N_iN_j)</td>
</tr>
<tr>
<td>i,j,k+1</td>
<td>T</td>
<td>(l + N_iN_j)</td>
</tr>
</tbody>
</table>
Fig. 4.9. Two different types of Cartesian finite volume grids. (a) The grid nodes are centered in control volumes (CVs). (b) CV faces are centered between nodes pertaining to the volume enclosed by the surfaces. The net flux through the surfaces is the sum of integrals over the four (six) CV faces:

$$\int_{\partial \Omega} f \, dS = \sum_k \int_{\partial \Omega_k} f \, dS,$$

where $f$ is the component of the convective or diffusive vector in the direction normal to CV face. For the validity of conservation equations it is important that CV’s do not overlap; each CV is unique to two CV’s which lie on either side of it. To calculate the surface integral in (4.57) exactly, one had to know...
the integral kernel $f$ everywhere on the surface $S$. However, since only the nodal values are calculated, one has to introduce an approximation:

- The integral is approximated in terms of the variable values at one or more locations on the cell face,
- The cell face values are approximated in terms of the nodal (CV center) values.

The simplest access to approximate the calculation of the integral (4.57) is the midpoint rule which approximates the integral as a product of the integrand at the cell face center and the cell face area:

$$ F = \int_{\partial \Omega} f \, dS = \bar{f} S \approx f S . $$

This approximation is of second order. The values of $f$ have to be obtained by interpolation. In order to preserve the second order accuracy of the midpoint rule approximation of the surface integral, e.g. by the use of higher order schemes. A different simple method is the trapezoid rule:

$$ F = \int_{\partial \Omega} f \, dS \approx \frac{S}{2} (f_a + f_b) , $$

where $f_a$ and $f_b$ are the values of the integrand in opposite directions with respect to the control volume. If the variation of $f$ is assumed to have some particular simple shape (e.g. an interpolation polynomial), the integration is easy. The accuracy of the approximation then depends on the order of shape functions.

### Euler Methods

The most simple integration scheme is probably obtained by a truncated Taylor expansion of the particle coordinates $r$:

$$ r_{t+\Delta t} = r_t + v_t \Delta t + \frac{a_t}{2m} \delta t^2 + \cdots . $$

When the expansion is truncated as in (4.60) one obtains the Euler algorithm. This algorithm is known to suffer from a huge energy drift and it is neither symplectic\(^5\), nor area preserving. Hence, this algorithm should not be used for solving Newton’s equation of motion. In fact, it is mostly used in engineering contexts, as an implicit integration scheme.

### Leap-Frog Method

The leap-frog method is equivalent to the Verlet algorithm. With this algorithm, the velocities are calculated at half-integer time steps and the velocities are used to compute the new positions as follows:

\(^5\) time reversible
\[ v_{t-\Delta t/2} = \frac{r_t - r_{t-\Delta t}}{\delta t}, \]  
\[ v_{t+\Delta t/2} = \frac{r_{t+\Delta t} - r_t}{\delta t}, \]

Resolving (4.61)b for the position \( r_{t+\Delta t} \) one obtains:

\[ r_{t+\Delta t} = r_t + \Delta t v_{t+\Delta t/2}. \]

From (4.47)d one gets the following equation for the updated velocities:

\[ r_{t+\Delta t/2} = v_{t-\Delta t/2} + \Delta t \frac{f_t}{m}. \]

The leapfrog algorithm is derived from the Verlet algorithm; thus, the obtained particle trajectories in a simulation are the same. A disadvantage of the leapfrog scheme is, that the positions and velocities are not calculated at the same time, and thus, one cannot calculate directly the total energy using this scheme.

### 4.2.4 Finite Element Methods

In the numerical treatment of elliptic and parabolic DEs, the finite element method is very often used; in particular in engineering applications this has long been the discretization method that has been used to solve stress analysis, heat transfer and other types of engineering problems. The FE method is very well suited to be used within the framework of a variational formulation of differential equations and their discretized algebraic counterparts of the problem under investigation. This is the reason why FEM – in contrast to finite difference or finite volume methods – is more flexible also for the treatment of difficult problems with complex geometries and boundary conditions. The development of the FE method has been restricted to engineering applications for a long time, until in the 70s the method was standardized as a theory by mathematicians. The discretization with the FE method may lead to very large systems of equations for which the effort is of the order \( O(n^2) \). In the last two decades, some new procedures such as multigrid-methods or conjugate gradient methods (which are also used in particle-based codes, e.g. in quantum chemistry) have been developed, which allow for the implementation of very efficient solvers. Here, the basics of the theory of finite elements are shortly discussed and in Chap. 7 typical applications are presented. The finite element method is based on solving a system of equations that describe some parameter (e.g. displacement) over the domain of a continuous physical system, (such as a part’s surface). The true power of FEM lies in its ability to solve problems that do not fit any standard formula. For example, prior to the use of FEM, stress analysis problems were usually matched to a handbooks formula, which was derived for a standard shape part. As the name FEM implies, this method involves the partitioning, or discretization of
a structure into a finite number of elements. Elements are connected to one another at their corner points. These corner points are called nodes or nodal points. Each element is a simple geometric shape, such as triangle or quadrilateral. Being a standard shape facilitates the development of the governing equations that relate the displacement and stress behavior within the element. For the completion of a finite element model, nodal points, elements, loads, supports and element related data must be defined. An FEM code formulates the equilibrium equations corresponding to each degree of freedom at each nodal point. The forces on each nodal point are calculated once the element is subject to loading and begins to deform. Hence, a finite element model, in essence, acts like a large system of springs which deflect until all forces balance. The FE method is well established in the engineering community and there is a multitude of commercial program packages to choose from. The algorithm shown in Fig. 4.10 summarizes the crucial steps of a FEM simulation. While conventional analytic variational methods aim to find a solution by using a single polynomial expression which is valid throughout the whole domain considered, FEM tries to find approximate solutions in each subdomain (each element). The subdomains are connected at joints which are called nodes. The (mostly linear or simple polynomial) interpolation functions used for an approximation, describe the course of the state variable in an element.

![Fig. 4.10. Basic flow scheme of an FEM analysis](image-url)
in terms of its node values. Most finite element methods use these functions to map both, the state variable and the topology of the element, which, by definition, is then referred to as an **isoparametric element**. The trial functions of isoparametric elements are thus also referred to as “shape” or “form functions”. The FE method represents a continuum type approach, that is, it does not incorporate the genuine dynamics of single lattice defects such as dislocations, but rather uses averaging constitutive laws for representing the materials reaction to load. For example, as a two-dimensional isoparametric element with four nodes, the following set of linear shape functions can be used:

\[
\begin{align*}
K_1(\xi, \kappa) &= \frac{1}{2}(1 + \xi)(1 + \kappa), \\
K_2(\xi, \kappa) &= \frac{1}{2}(1 + \xi)(1 - \kappa), \\
K_3(\xi, \kappa) &= \frac{1}{2}(1 - \xi)(1 + \kappa), \\
K_4(\xi, \kappa) &= \frac{1}{2}(1 - \xi)(1 - \kappa),
\end{align*}
\]

(4.64)

where the variables $\xi$ and $\kappa$ lie in the range $[-1 \leq \xi]$ and $[\kappa \leq +1]$. The approximated values of the considered field variable in one element may be calculated as:

\[
\vec{f}(\xi, \kappa) = \sum_{i=1}^{n=4} K_i(\xi, \kappa) \vec{f}_i,
\]

(4.65)

where $n$ is the number of nodes and $\vec{f}_i$ are the values of the field variable at the nodes. The application of shape functions to state variables allows one to map each element into a master generic element with a fixed length by using an isoparametric transformation. By taking the derivatives $K_{i,j} = \partial K / \partial x_j$ along $x_j$ of the $n$ ansatz functions in (4.65), one can finally calculate the corresponding stiffness matrix. In Chap. 7 several applications of the FE method are shown. One important difference between FEM and other methods for solving PDEs, such as FDM, is, that FEM approximates the solution to a differential equation, whereas FDM approximates the differential equation itself. Sophisticated implementations of FEM use **adaptive finite elements**, i.e. they assess the quality of the mesh during the solution aiming at achieving an approximate solution within some bounds from the exact solution of the continuum problem. The most common mesh-adaptivity methods are:

- moving nodes (r-adaptivity),
- refining and elements (h-adaptivity),
- changing order of base functions (p-adaptivity),
- combinations of the above (hp-adaptivity).
4.3 Elements of Software Design

Although leading-edge software-development practice has advanced rapidly in recent years, common practice hasn’t. Many programs written in software industry are still very buggy, late and over budget, and many fail to satisfy the needs of their users. On the other hand, many researchers who are active in computer simulation haven’t had any formal education in computer science, software design or software engineering, let alone have any professional experience in leading a team of software experts in an industrial large-scale environment, but instead they develop research codes in a “heuristic” process. This explains the abundance of passed on ancient Fortran 77 or Fortran 90 codes for e.g. solving Schrödinger’s wave function, performing Monte Carlo simulation runs or for fluid and structural dynamics applications in Engineering; It explains also to a certain degree the common practice in academia that consecutive generations of research students often have to write and invent software packages for similar purposes over and over again. This is an unfortunate situation as researchers in both the software industry and in academic settings have long since discovered effective practices that help to eliminate many programming problems and that makes codes or bits of codes reusable. These practices, when applied to scientific code development, may help considerably not only to reduce the number of bugs in codings and to be more efficient but also to make the research codes more readable and understandable to other scientists and thus allow for concentrating on the scientific parts of computational science and spending less time on implementation details. These practices or different stages in the so-called “software-development life cycle” include:

- Problem definition
- Requirements analysis
- Implementation planning
- High-level design or architecture
- Detailed design
- Implementation
- Integration
- Unit testing
- System testing

Probably more than 80% of the time in software development are spend with implementation. Implementation focuses on coding and debugging but also includes some detailed design and some unit testing. It comes after architectural design and before system testing and maintenance within the software-development life cycle. Many researchers – due to a lack in formal training – write their codes in a somewhat “intuitive” style, with not much architectural design, commenting of code, or planning ahead, just going along as certain needs arise. This is probably fine as long as one does not have to meet any specific goals or deadlines in scientific projects, and the code is only
used by one person, but often this practice results in “spaghetti-programs” that cannot really be handled by anybody else than the author. A good piece of code should be largely self-explanatory and “readable” just like an ordinary book.

As long as software projects are small (maybe only containing a few thousand lines of code and involving only one or a handful of people working with the code), an extended architectural design would be overdone. However, a large project in software industry will most likely involve many people with many different responsibilities. Writing a parallelized molecular dynamics simulation code can easily extend to more than 30000 lines of code which – in an industrial setting – would be a medium sized software project. Probably more than 95% of a research code is only data management, input/output and error handling and only a very small piece of the code does actually contain “science”, i.e. a physical model of the system under investigation. Large software projects (e.g. writing operating systems) may contain millions of lines of code.

The author of this book has written probably more than 500,000 lines of coding in mostly academic but also in industrial research environments and in software industry during the last 10 years. Scientific research codes usually have a very simple architecture and thus, as a rule of thumb, object oriented methods are way overdone for the (usually) very simple data structures involved. The advantage of object-oriented concepts is the re-usability of pieces of code which is made easy and supported by e.g. container classes and many other implicit language elements; also, combining some piece of code with a GUI is very easy in this case as most object-oriented languages provide appropriate GUI container classes. While the specific choice of programming language has nothing to do with the algorithms that are to be implemented to help solving a scientific problem, it is quite clear, that any software written today by a professional programmer would not be written e.g. using Fortran but rather in some object oriented language such as Java or C++, or even in C which is probably the best alternative to the overdone constructs which are possible in object oriented languages. Java has the advantage of being interpreted during runtime, so that one could change simulation parameters on-the-fly and directly watch the consequences in a Java Applet, instead of going through the process of re-compiling the complete code and visualizing the results using additional programs. However, with scientific programs, usually speed is the most important limiting factor which favors compiling a code into machine language. Most installations of super computers only have C, C++ and Fortran compilers.

A good place to start learning how to manage large software projects is Gilb [305] which focuses on the points a software manager should (be able to) do during the development stage. DeMarco [306] or Maguire [307] are good alternatives.

Maybe the first book on software architecture and design was Yourdon [308] who originally developed many key ideas of a structured design approach
to software development. A more modern book that describes object oriented
design stages is Booch [309]. He was a pioneer in the use of Ada and in object-
oriented methods, and this experience shows in this book.

Testing software is an important part in software development. Myers [310]
is a classic devoted to the topic, Hetzel [311] being a good alternative.

It is sometimes astounding how less knowledge and experience researchers
in academia or industry who write simulation codes or who are responsible
for managing scientific software projects actually have to show in project
planning, computer science, algorithms or data structures. We have discussed
some basics of the latter three topics in Sec. 2.6 of Chap. 2. One may consider
these things as “unscientific”, however when it comes to scientific computing,
the scientific results which are published in specialized physics or engineering
journals usually rely heavily and often exclusively on the results of numerical
simulations. Some good books on the subject not mentioned in Chap. 2 are
Sedgewick [312], Reingold/Hansen [313], Maguire [314], McConnell [315] and
of course the classics by Donald Knuth [316, 317, 318].

In the next few sections we will touch on some of the most important issues
in software development. We start with the first level of software development
which is software design.

4.3.1 Software Design

The phrase “software design” means the planning, conception or invention of
a scheme for turning a specification for a computer program into an opera-
tional program. Design is the activity that links requirements specifications
to coding and debugging. It is a heuristic process rather than a determinis-
tic one, requiring creativity and insight. A large part of the design process is
appropriate only for a specific project at hand.

On large, formal projects, design is often distinct from other activities such
as requirements analysis and coding. It might even be performed by several
different people. A really large project is often split into several stages of
design, beginning with the software architecture, a high-level module design
and detailed implementation design. On small, informal projects, most of the
design is done while the program is written. It might just be writing a routine
in program design language before writing it in some programming-language
or it might just involve sketching a few diagrams for a few routines before
writing them. In any case, small projects usually benefit from careful design
just as larger projects do.

Structured Design

A structured program design is made up of the following parts:

- Strategies for developing designs.
- Criteria for evaluating designs.
- Problem analysis as a guide to the solution to the problem.
- System organization into routines that have a well-defined, narrow interface and whose implementation details are hidden from other routines.
- Graphical tools for expressing design, including structure charts and program design language.

In Fig. 4.11 the main levels (or hierarchies) of code design are displayed which have proved useful in practice. On the first level, a system is split up into subsystems. The main point at this level is to identify all major subsystems and to divide the program into major components with respective interfaces. In a large project, some subsystems might be too big to be directly identified as modules on the first design level.

On a second level, the subsystems are further divided into modules, e.g. input/output functionality, exception handling, etc. As indicated in Fig 4.11 different design methods might be used within different parts of the system.

On a third design level each module is subdivided into the services offered by the module. This includes identifying routines for different tasks and at the same time specifying respective functions. This level of decomposition and design is needed on any project that takes more time than a few hours. It does not have to be done in a formal process, but it at least needs to be done mentally.

A fourth design level specifies the detailed functionality of the individual routines. In an industrial environment this internal routine design is usually left to the individual programmer working on an individual routine. This level

Fig. 4.11. Different software design levels
of design is always done in projects of any size, consciously or unconsciously, by every programmer.

In this short discussion we have only introduced a top down approach which consists in stepwise refinement. Sometimes this approach is so abstract that it hard to get started at all. In this case, also a bottom-up approach can be tried which identifies useful routines first and this usually results in a robust design.

Object-oriented Design

Object-oriented design is characterized by the identification of real-world and abstract objects which are then represented by programming-language objects. It is based on the assumption that the more closely a program models the real-world problem, the better the program will be. In many cases, the data definitions in programs are more stable than functionality, therefore a design based on the data model, as object oriented design is, is a more stable approach. This approach involves identifying objects, the operations on these objects and classes and then building a system from those objects, classes and operations. Object-oriented design uses key ideas of modern programming such as:

- Abstraction
- Encapsulation (information hiding)
- Modularity
- Hierarchy and Inheritance
- Objects and Classes

Object-oriented vs. Structured Design

With a top-down structured design as depicted in Fig. 4.11 one basically breaks a program into routines. This approach has an emphasis on functionality and does not stress data. A simple example of a system based on functionality would be a program that reads in data in batch mode. This is a predictable process on the data which performs in predictable order. An object-oriented design on the other hand is mainly a way of designing modules as collections of data and operations on the data. This method is great for decomposing a system from the highest level. It is applicable to any system which acts as objects in the real world do. Examples of such systems are highly interactive programs using windows, dialog boxes, and other constructs. Most of the work being done with object-oriented techniques is focused on the successful implementation of systems from at least 100,000 to multi-million lines of code. Empirically, structured techniques have often failed on such large projects and object-oriented design was developed to specifically allow for error-free programming of very large codes [319]. Figure 4.12 contrasts the two discussed design approaches.
Fig. 4.12. Contrasting the object-oriented approach (a) and the structured design approach (b). The circles represent operations or data structures. In a) objects are identified until no more useful partition into objects can be done. In b) first the top level of code design is identified. Then, consecutively, lower level design structures are identified until the whole system has been partitioned.

The main difference between structured and object-oriented design is that the latter works well at a higher level of abstraction. If data is likely to change, then the object-oriented design is appropriate because it isolates the data likely to change into individual objects (modules). However, if functionality is likely to change, the object-oriented approach is disadvantageous because the functionality is spread throughout many objects. A structured design approach on the other hand will always be a useful method of choice for all but the largest software projects. In scientific computing, large programs that exceed 50,000 lines of code are extremely rare. In addition, the number of different basic data types is usually extremely small, e.g. properties of particles, atoms, finite elements or fields, compared to, e.g. the thousands of different objects involved in an object-oriented design for controlling a modern aircraft. Thus, in science, in almost all cases a structured approach will be sufficient and there will be no real considerable advantage when using an object-oriented design, simply because the complexity of a scientific code usually is not very large. Examples for this will be provided in later sections.

Next we discuss the core task of programming, the building of a routine.

4.3.2 Writing a Routine

There are many low-level details that go into building a routine which are not done in any particular order, but usually there are four major activities, cf. Fig. 4.13. It is good practice to start a routine by writing down the specification demands in a program design language (PDL) [320] which is roughly the same as ordinary language. An example of a bad description of a routine is
Algorithm 4.2 A code example of badly written PDL

```
increment resource number by 1 allocate a dig struct using malloc if malloc() returns NULL then return 1 invoke OSrsrc_init to initialize a resource for the operating system *hRsrcPtr = resource number return 0
```
Algorithm 4.3 A code example of improved PDL

Keep track of current number of resources in use if another resource is available
Allocate a dialog box structure
If a dialog box structure could be allocated
    Note that one more resource is in use
    Initialize the resource
    Store the resource number at the location provided by the caller
Endif
Endif
Return TRUE if a new resource was created; else return FALSE

not only saves space but also modification are much easier done, as one only needs to change code at one location.

- **Limiting effects of changes.** Areas in a code which are likely to change, such as hardware dependencies, input/output handling, complex data structures, etc. should be such designed that the effects of a change are limited to the scope of a single routine or at least not more than a few routines.

- **Hiding data structures.** Routines that hide implementation details of data structures provide a valuable level of abstraction which reduces a program’s complexity. They centralize data structure operations in one place and make it easy to change the structure without changing most of the program.

- **Hiding pointer operations.** Pointer operations tend to be hard to read error prone. By isolating them in routines one can concentrate on the intend of the operation rather than the mechanics of pointer manipulation.

- **Avoiding global data structures.** Beginners in programming love global data structures and global variables as they make programming very easy. One never has to pass any variables to routines, one does not have to worry about pointer arithmetic and one can easily access all variables from all scopes. However, anybody who has ever tried to understand a program that makes extensive use of global variables will always try to avoid them. They make a code completely unreadable and completely destroy any architectural design, i.e. it is impossible to understand the purpose of routines as there is no interface – everything can be globally accessed. Worse, such a code is usually impossible to extend or optimize as there is no control as to where some variable might be changed or used in the code. Thus, in any good piece of code there are as less global variables as possible.

- **Saying Farwell to “goto” statements.** The general argument against *gotos* is that code without *gotos* is higher-quality code. The original controversy on this issue was sparked by Dijkstra [321] in 1968. He observed that the quality of code was inversely proportional to the number of *gotos* the programmers used. This *goto* controversy erupted when Fortran was
the most popular language, particularly in science. Fortran originally had no presentable loop structures, and in the absence of good advice on programming structured loops (using for, do, while constructs), programmers wrote a lot of spaghetti-code. Such goto-laden code was undoubtedly correlated with the production of low-quality programs; however its careful use to make up for a gap in a structured language’s capabilities might have been acceptable in the 1970s. Today, no high-quality code should contain any gotos. Some of the main arguments against it are:

- Code containing gotos is hard to format using indentation which should actually represent logical structure.
- Use of gotos defeats compiler optimizations. Many optimizations depend on a program’s flow of control within a few statements. Unconditional gotos make the flow difficult to analyze and reduce the ability of the compiler to optimize code.
- Use of gotos violates structured programming principles.

**Promoting code reuse.** Programs written in Fortran 77 or Fortran 90 are often written in one monolithic huge file. Writing one huge file with probably 50000 lines and several subroutines written in the same file makes a code virtually unreadable. This practice should be strictly avoided and instead a program should be given a modular code structure where certain routines that do similar things are collected in one file. Modular routines can be reused in other programs more easily than the same code embedded in one larger routine. A modular structure also makes a program more readable and understandable.

Often it is advisable to hide details of code in a routine which is then called from the main program. This often makes code less complex and automatically documents it. The following piece of code illustrates this:

```plaintext
if ( Node <> NULL )
    while ( Node.Next <> NULL ) do
        Node = Node.Next
        LeafName = Node.Name
    else
        LeafName = ''''
These six lines of code could easily be put in a routine as follows:

LeafName = GetLeafName ( Node )
```

This new routine is short and all that it needs for documentation is a good name.

**How long should a routine be?**

The theoretical best maximum length is often described as one or two screen pages of program listing, about 60 to 120 lines. In Frenkel and Smit [55] the
authors say that it is generally assumed that in every 200 lines of code there is one error. This is however not supported by many studies. A study by Basili [322] found that routine size was inversely correlated with errors; as the size of routines increased (more than 200 lines of code), the number of errors per line of code decreased. Card et al. [323] found in a study that small routines of 32 lines of code or fewer were not correlated with lower cost or fault rate. The evidence suggested that larger routines of 65 lines of code or more were cheaper to develop per line of code. Another study (Lind and Vairavan [324]) found that code needed to be changed least when routines averaged 100 to 150 lines of code. Jones et al. [325] found that routines beyond 500 lines were most error prone and that the error rate tended to be proportional to the size of the routine. A university study of graduate computer science students [326] found that their comprehension of a program that was extremely modularized into routines of about 10 lines of code was no better than their comprehension of a program that had no routines at all. Moreover, when the program was broken into routines of moderate length of about 25 lines, students scored 65% better on a test of comprehension.

4.3.3 Code-Tuning Strategies

In the early days of programming computer resources were severely limited, and efficiency was the prime concern. As computers become more powerful in the 1970s, programmers realized how much the focus on performance had hurt readability and maintainability, and code tuning received less attention. With the return of performance limitations with the microcomputer revolution of the 1980s, performance issues were considered again. Today, at the high-end are super computer systems with at least hundreds and even thousands of processors which allow for high-performance parallel computing, usually involving a fast network that links the CPUs. As for scientific computing, high-performance and in particular speed are of paramount importance. Therefore usually a lot of time is spend for code tuning in high-end parallelized scientific programs. Generally speaking, code efficiency may have to do with one or several of the following aspects:

- Hardware
- Operating-system interactions
- Code compilation
- Program design
- Module and routine design

One theory for faster and more efficient code is that one should optimize as one writes each routine. This, however is not true. It is generally much better to write a code completely first and then start optimizing when the whole code and all routines run correctly. Before a program is really working completely it is almost impossible to identify any bottlenecks. Thus, the above strategy would lead to many micro-optimizations but in the end the whole code might
exhibit inferior performance. Focusing too much on optimization during initial
development tends to detract from achieving other program objectives, such
as correctness, modularity, information hiding and readability. Optimizations
done after after a system is complete can identify each problem area and its
relative importance so that optimization time is allocated effectively.

The Pareto Principle

This principle, also known as the 80/20 rule, states that you can get 80% of
the result with 20% of the effort. This principle applies to many other areas
than programming, but it does definitely apply to program optimization.

Knuth [327] found in an empirical study that less than 4% of a program
usually accounts for more than 50% of its run time. Thus, since small parts
of a program usually take a disproportionate share of run time, one should
always measure the code to find the hot spots and then put one’s resources
into optimizing the few percent that are used most. An example of this can
be seen in Fig. 4.14 which shows the profiling output of one of the authors
molecular dynamics programs. The total run-time was 84 seconds and a total
of 51 different functions are called in the program. One can see that less than
4% of the program accounts for more than 50% of total run time. In this case,
a Langevin-thermostat was simulated which uses random numbers. There are
no general-purpose rules to optimization as the rules change each time one
change languages, hardware, compilers, or even compiler versions. This means
that techniques which improve performance in one environment can degrade
in others. Often optimizations are counter-intuitive. A simple example for this
is the piece of pseudo-code given in Algorithm 4.4.

<table>
<thead>
<tr>
<th>% cumulative</th>
<th>self time</th>
<th>self seconds</th>
<th>self calls</th>
<th>ms/call</th>
<th>ms/call name</th>
</tr>
</thead>
<tbody>
<tr>
<td>34.14</td>
<td>28.15</td>
<td>28.15</td>
<td>10001</td>
<td>2.81</td>
<td>3.65</td>
</tr>
<tr>
<td>18.47</td>
<td>43.38</td>
<td>15.23</td>
<td>229175856</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>11.54</td>
<td>53.14</td>
<td>9.76</td>
<td>957</td>
<td>10.20</td>
<td>10.20</td>
</tr>
<tr>
<td>10.59</td>
<td>61.71</td>
<td>8.57</td>
<td>180000000</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>8.59</td>
<td>68.80</td>
<td>7.08</td>
<td>10000</td>
<td>0.71</td>
<td>8.23</td>
</tr>
<tr>
<td>6.93</td>
<td>74.51</td>
<td>5.71</td>
<td>39762734</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>3.29</td>
<td>77.22</td>
<td>2.71</td>
<td>10001</td>
<td>0.27</td>
<td>0.27</td>
</tr>
<tr>
<td>3.24</td>
<td>79.89</td>
<td>2.67</td>
<td>24002400</td>
<td>0.00</td>
<td>0.00</td>
</tr>
<tr>
<td>1.46</td>
<td>81.09</td>
<td>1.20</td>
<td>10000</td>
<td>0.12</td>
<td>0.12</td>
</tr>
<tr>
<td>1.27</td>
<td>82.14</td>
<td>1.05</td>
<td>957</td>
<td>1.10</td>
<td>1.10</td>
</tr>
<tr>
<td>0.22</td>
<td>82.32</td>
<td>0.18</td>
<td>957</td>
<td>0.19</td>
<td>0.19</td>
</tr>
</tbody>
</table>

Fig. 4.14. Output of a profiler gprof on a UNIX system for one of the author’s
molecular dynamics simulation codes
Algorithm 4.4 Example of a straightforward piece of code for the summation of matrix elements

```
CalcSum = 0; for ( Row = 0; Row < RowCount; Row++ )
{
    for ( Column = 0; Column < ColumnCount; Column++ )
    {
        CalcSum += Matrix[ Row ][ Column ];
    }
}
```

This code sums up matrix elements. Generally speaking, it is always advisable to only use one-dimensional arrays, as the computer internally translates these arrays into one-dimensional ones anyway. But this is actually not the point here. In an attempt to optimize this matrix element addition which – for a $10 \times 10$ matrix consists of 100 multiplications and additions plus the loop overhead for incrementing the loop counter, one might try out pointer notation. Converting Algorithm 4.4 into pointer notation with only 100 relatively cheap pointer increment operations, one obtains the piece of code displayed in Algorithm 4.5.

When testing this program’s performance in C compiled with a standard C-compiler on a UNIX system, it might come as a surprise that there is no gain in speed at all compared to the un-optimized version. The reason is that the compiler’s optimizer already optimizes the first code version well enough. Thus, all that was gained here, would be turning a readable code into a harder to read code with no measurable gain in speed. Probably the best way to prepare for code tuning at the initial coding stage is to write clean code that is easy to understand and modify. As a summary of the short discussion of code tuning, we give some key points of an optimization strategy:

1. Use a good modular program design that is easy to understand and modify.
2. If performance is poor and the need arises, measure the systems performance to find hot spots.
3. Evaluate whether the performance comes from poor design, inadequate data structures or algorithms. If this is the case got back to step 1.

Algorithm 4.5 Example of an optimization strategy for the summation of matrix elements using pointers

```
CalcSum = 0; ElementPointer = Matrix; LastElementPointer = Matrix[ RowCount -1 ][ ColumnCount -1 ] + 1; while ( ElementPointer < LastElementPointer )
{
    Sum += *ElementPointer++;
}
```
4. Tune the bottleneck identified in step 3 and then measure each improvement; if it does not improve, take it out.
5. Repeat steps 2 to 5.

4.3.4 Suggested Reading

The discussion of structured code design has been necessarily short and superficial here; in particular a graphical representation of relations is very powerful. An excellent book to start with is Yourdon [308] which is written with obvious care. Another excellent book which is succinct, to the point and less technical is Myers [310]. The original article on structured design by Stevens et. al. [328] is a classic although many books today make a better presentation of the material. The fine volume by Meyer [329] and Coad [330] discusses virtually all aspects of object-oriented design, with the latter volume being a probably easier introduction to the subject. The classic article by Parnas [331] describes the gap between how programs are often really designed and how one wished they were designed; the main point of this book is that no one ever goes through a rational, orderly design process but that aiming for it makes for better designs in the end. The very readable book by Jackson [332] explains the full data-view design methodology. A discussion of creative thought process in programming can be found in Adams [333]. A very readable book by Simon [334] discusses the “science of design”. A discussion of quality considerations during all phases of software development is provided in Glass [335]. Some good books on performance engineering are Smith [336] which includes many examples and case studies or Bentley [337] which is an expert treatment of code tuning and optimization. Weinberg [338] is probably one of the most readable books on software development. This classic also contains many entertaining anecdotes about the human side of programming.

Problems

4.1. Plane waves as solution of the wave equation
Show that any function of the form \( f(x + ct) \) or \( f(x - ct) \) is a solution of the wave equation (4.30).

4.2. Spherical waves as solution of the wave equation
Show that a spherically symmetric solution of the wave function (4.30) is of the form \( \Phi = A \pm e^{i(kx \pm \omega t)} \).

4.3. Verlet Algorithm
Derive the Verlet algorithm by Taylor expansion of the coordinates \( r \) and show that it is of order \( O(\Delta t^2) \).