# A Useful Formulas

## A.1 Finite Difference Operator Notation

$$u'(t_n) \approx [D_t u]^n = \frac{u^{n+\frac{1}{2}} - u^{n-\frac{1}{2}}}{\Delta t} \tag{A.1}$$

$$u'(t_n) \approx [D_{2t} u]^n = \frac{u^{n+1} - u^{n-1}}{2\Delta t} \tag{A.2}$$

$$u'(t_n) = [D_t^- u]^n = \frac{u^n - u^{n-1}}{\Delta t} \tag{A.3}$$

$$u'(t_n) \approx [D_t^+ u]^n = \frac{u^{n+1} - u^n}{\Delta t} \tag{A.4}$$

$$u'(t_{n+\theta}) = [\bar{D}_t u]^{n+\theta} = \frac{u^{n+1} - u^n}{\Delta t} \tag{A.5}$$

$$u'(t_n) \approx [D_t^{2-} u]^n = \frac{3u^n - 4u^{n-1} + u^{n-2}}{2\Delta t} \tag{A.6}$$

$$u''(t_n) \approx [D_t D_t u]^n = \frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2} \tag{A.7}$$

$$u\left(t_{n+\frac{1}{2}}\right) \approx [\overline{u}^t]^{n+\frac{1}{2}} = \frac{1}{2}(u^{n+1} + u^n) \tag{A.8}$$

$$u\left(t_{n+\frac{1}{2}}\right)^2 \approx [\overline{u^2}^{t,g}]^{n+\frac{1}{2}} = u^{n+1}u^n \tag{A.9}$$

$$u\left(t_{n+\frac{1}{2}}\right) \approx [\overline{u}^{t,h}]^{n+\frac{1}{2}} = \frac{2}{\frac{1}{u^{n+1}} + \frac{1}{u^n}} \tag{A.10}$$

$$u(t_{n+\theta}) \approx [\overline{u}^{t,\theta}]^{n+\theta} = \theta u^{n+1} + (1-\theta)u^n, \tag{A.11}$$

$$t_{n+\theta} = \theta t_{n+1} + (1-\theta)t_{n-1} \tag{A.12}$$

Some may wonder why $\theta$ is absent on the right-hand side of (A.5). The fraction is an approximation to the derivative at the point $t_{n+\theta} = \theta t_{n+1} + (1-\theta)t_n$.

© The Editor(s) (if applicable) and The Author(s) 2017

H.P. Langtangen, S. Linge, *Finite Difference Computing with PDEs*,
Texts in Computational Science and Engineering 16, DOI 10.1007/978-3-319-55456-3

## A.2 Truncation Errors of Finite Difference Approximations

$$u_e'(t_n) = [D_t u_e]^n + R^n = \frac{u_e^{n+\frac{1}{2}} - u_e^{n-\frac{1}{2}}}{\Delta t} + R^n,$$
$$R^n = -\frac{1}{24}u_e'''(t_n)\Delta t^2 + \mathcal{O}(\Delta t^4) \tag{A.13}$$
$$u_e'(t_n) = [D_{2t} u_e]^n + R^n = \frac{u_e^{n+1} - u_e^{n-1}}{2\Delta t} + R^n,$$
$$R^n = -\frac{1}{6}u_e'''(t_n)\Delta t^2 + \mathcal{O}(\Delta t^4) \tag{A.14}$$
$$u_e'(t_n) = [D_t^- u_e]^n + R^n = \frac{u_e^n - u_e^{n-1}}{\Delta t} + R^n,$$
$$R^n = -\frac{1}{2}u_e''(t_n)\Delta t + \mathcal{O}(\Delta t^2) \tag{A.15}$$
$$u_e'(t_n) = [D_t^+ u_e]^n + R^n = \frac{u_e^{n+1} - u_e^n}{\Delta t} + R^n,$$
$$R^n = \frac{1}{2}u_e''(t_n)\Delta t + \mathcal{O}(\Delta t^2) \tag{A.16}$$
$$u_e'(t_{n+\theta}) = [\bar{D}_t u_e]^{n+\theta} + R^{n+\theta} = \frac{u_e^{n+1} - u_e^n}{\Delta t} + R^{n+\theta},$$
$$R^{n+\theta} = -\frac{1}{2}(1-2\theta)u_e''(t_{n+\theta})\Delta t + \frac{1}{6}((1-\theta)^3 - \theta^3)u_e'''(t_{n+\theta})\Delta t^2 + \mathcal{O}(\Delta t^3) \tag{A.17}$$
$$u_e'(t_n) = [D_t^{2-} u_e]^n + R^n = \frac{3u_e^n - 4u_e^{n-1} + u_e^{n-2}}{2\Delta t} + R^n,$$
$$R^n = \frac{1}{3}u_e'''(t_n)\Delta t^2 + \mathcal{O}(\Delta t^3) \tag{A.18}$$
$$u_e''(t_n) = [D_t D_t u_e]^n + R^n = \frac{u_e^{n+1} - 2u_e^n + u_e^{n-1}}{\Delta t^2} + R^n,$$
$$R^n = -\frac{1}{12}u_e''''(t_n)\Delta t^2 + \mathcal{O}(\Delta t^4) \tag{A.19}$$
$$u_e(t_{n+\theta}) = [\overline{u_e}^{t,\theta}]^{n+\theta} + R^{n+\theta} = \theta u_e^{n+1} + (1-\theta)u_e^n + R^{n+\theta},$$
$$R^{n+\theta} = -\frac{1}{2}u_e''(t_{n+\theta})\Delta t^2\theta(1-\theta) + \mathcal{O}(\Delta t^3)\,. \tag{A.20}$$

## A.3 Finite Differences of Exponential Functions

**Complex exponentials** Let $u^n = \exp(i\omega n\Delta t) = e^{i\omega t_n}$.

$$[D_tD_tu]^n = u^n\frac{2}{\Delta t}(\cos\omega\Delta t - 1) = -\frac{4}{\Delta t}\sin^2\left(\frac{\omega\Delta t}{2}\right), \tag{A.21}$$

$$[D_t^+u]^n = u^n\frac{1}{\Delta t}(\exp(i\omega\Delta t) - 1), \tag{A.22}$$

$$[D_t^-u]^n = u^n\frac{1}{\Delta t}(1 - \exp(-i\omega\Delta t)), \tag{A.23}$$

$$[D_tu]^n = u^n\frac{2}{\Delta t}i\sin\left(\frac{\omega\Delta t}{2}\right), \tag{A.24}$$

$$[D_{2t}u]^n = u^n\frac{1}{\Delta t}i\sin(\omega\Delta t)\,. \tag{A.25}$$

**Real exponentials** Let $u^n = \exp(\omega n\Delta t) = e^{\omega t_n}$.

$$[D_tD_tu]^n = u^n\frac{2}{\Delta t}(\cos\omega\Delta t - 1) = -\frac{4}{\Delta t}\sin^2\left(\frac{\omega\Delta t}{2}\right), \tag{A.26}$$

$$[D_t^+u]^n = u^n\frac{1}{\Delta t}(\exp(i\omega\Delta t) - 1), \tag{A.27}$$

$$[D_t^-u]^n = u^n\frac{1}{\Delta t}(1 - \exp(-i\omega\Delta t)), \tag{A.28}$$

$$[D_tu]^n = u^n\frac{2}{\Delta t}i\sin\left(\frac{\omega\Delta t}{2}\right), \tag{A.29}$$

$$[D_{2t}u]^n = u^n\frac{1}{\Delta t}i\sin(\omega\Delta t)\,. \tag{A.30}$$

## A.4 Finite Differences of $t^n$

The following results are useful when checking if a polynomial term in a solution fulfills the discrete equation for the numerical method.

$$[D_t^+t]^n = 1, \tag{A.31}$$

$$[D_t^-t]^n = 1, \tag{A.32}$$

$$[D_tt]^n = 1, \tag{A.33}$$

$$[D_{2t}t]^n = 1, \tag{A.34}$$

$$[D_tD_tt]^n = 0\,. \tag{A.35}$$

The next formulas concern the action of difference operators on a $t^2$ term.

$$[D_t^+ t^2]^n = (2n + 1)\Delta t, \tag{A.36}$$
$$[D_t^- t^2]^n = (2n - 1)\Delta t, \tag{A.37}$$
$$[D_t t^2]^n = 2n\Delta t, \tag{A.38}$$
$$[D_{2t} t^2]^n = 2n\Delta t, \tag{A.39}$$
$$[D_t D_t t^2]^n = 2\,. \tag{A.40}$$

Finally, we present formulas for a $t^3$ term:

$$[D_t^+ t^3]^n = 3(n\Delta t)^2 + 3n\Delta t^2 + \Delta t^2, \tag{A.41}$$
$$[D_t^- t^3]^n = 3(n\Delta t)^2 - 3n\Delta t^2 + \Delta t^2, \tag{A.42}$$
$$[D_t t^3]^n = 3(n\Delta t)^2 + \frac{1}{4}\Delta t^2, \tag{A.43}$$
$$[D_{2t} t^3]^n = 3(n\Delta t)^2 + \Delta t^2, \tag{A.44}$$
$$[D_t D_t t^3]^n = 6n\Delta t\,. \tag{A.45}$$

### A.4.1 Software

Application of finite difference operators to polynomials and exponential functions, resulting in the formulas above, can easily be computed by some `sympy` code (from the file `lib.py`):

```
from sympy import *
t, dt, n, w = symbols('t dt n w', real=True)

# Finite difference operators

def D_t_forward(u):
    return (u(t + dt) - u(t))/dt

def D_t_backward(u):
    return (u(t) - u(t-dt))/dt

def D_t_centered(u):
    return (u(t + dt/2) - u(t-dt/2))/dt

def D_2t_centered(u):
    return (u(t + dt) - u(t-dt))/(2*dt)

def D_t_D_t(u):
    return (u(t + dt) - 2*u(t) + u(t-dt))/(dt**2)


op_list = [D_t_forward, D_t_backward,
           D_t_centered, D_2t_centered, D_t_D_t]
```

```
def ft1(t):
    return t

def ft2(t):
    return t**2

def ft3(t):
    return t**3

def f_expiwt(t):
    return exp(I*w*t)

def f_expwt(t):
    return exp(w*t)

func_list = [ft1, ft2, ft3, f_expiwt, f_expwt]
```

To see the results, one can now make a simple loop over the different types of functions and the various operators associated with them:

```
for func in func_list:
    for op in op_list:
        f = func
        e = op(f)
        e = simplify(expand(e))
        print e
        if func in [f_expiwt, f_expwt]:
            e = e/f(t)
        e = e.subs(t, n*dt)
        print expand(e)
        print factor(simplify(expand(e)))
```

# Truncation Error Analysis

# B

Truncation error analysis provides a widely applicable framework for analyzing the accuracy of finite difference schemes. This type of analysis can also be used for finite element and finite volume methods if the discrete equations are written in finite difference form. The result of the analysis is an asymptotic estimate of the error in the scheme on the form $Ch^r$, where $h$ is a discretization parameter ($\Delta t$, $\Delta x$, etc.), $r$ is a number, known as the convergence rate, and $C$ is a constant, typically dependent on the derivatives of the exact solution.

Knowing $r$ gives understanding of the accuracy of the scheme. But maybe even more important, a powerful verification method for computer codes is to check that the empirically observed convergence rates in experiments coincide with the theoretical value of $r$ found from truncation error analysis.

The analysis can be carried out by hand, by symbolic software, and also numerically. All three methods will be illustrated. From examining the symbolic expressions of the truncation error we can add correction terms to the differential equations in order to increase the numerical accuracy.

In general, the term truncation error refers to the discrepancy that arises from performing a finite number of steps to approximate a process with infinitely many steps. The term is used in a number of contexts, including truncation of infinite series, finite precision arithmetic, finite differences, and differential equations. We shall be concerned with computing truncation errors arising in finite difference formulas and in finite difference discretizations of differential equations.

## B.1 Overview of Truncation Error Analysis

### B.1.1 Abstract Problem Setting

Consider an abstract differential equation

$$\mathcal{L}(u) = 0,$$

where $\mathcal{L}(u)$ is some formula involving the unknown $u$ and its derivatives. One example is $\mathcal{L}(u) = u'(t) + a(t)u(t) - b(t)$, where $a$ and $b$ are constants or functions of time. We can discretize the differential equation and obtain a corresponding

discrete model, here written as

$$\mathcal{L}_\Delta(u) = 0\,.$$

The solution $u$ of this equation is the *numerical solution*. To distinguish the numerical solution from the exact solution of the differential equation problem, we denote the latter by $u_{\rm e}$ and write the differential equation and its discrete counterpart as

$$\mathcal{L}(u_{\rm e}) = 0,$$
$$\mathcal{L}_\Delta(u) = 0\,.$$

Initial and/or boundary conditions can usually be left out of the truncation error analysis and are omitted in the following.

The numerical solution $u$ is, in a finite difference method, computed at a collection of mesh points. The discrete equations represented by the abstract equation $\mathcal{L}_\Delta(u) = 0$ are usually algebraic equations involving $u$ at some neighboring mesh points.

### B.1.2 Error Measures

A key issue is how accurate the numerical solution is. The ultimate way of addressing this issue would be to compute the error $u_{\rm e} - u$ at the mesh points. This is usually extremely demanding. In very simplified problem settings we may, however, manage to derive formulas for the numerical solution $u$, and therefore closed form expressions for the error $u_{\rm e} - u$. Such special cases can provide considerable insight regarding accuracy and stability, but the results are established for special problems.

The error $u_{\rm e} - u$ can be computed empirically in special cases where we know $u_{\rm e}$. Such cases can be constructed by the method of manufactured solutions, where we choose some exact solution $u_{\rm e} = v$ and fit a source term $f$ in the governing differential equation $\mathcal{L}(u_{\rm e}) = f$ such that $u_{\rm e} = v$ is a solution (i.e., $f = \mathcal{L}(v)$). Assuming an error model of the form $Ch^r$, where $h$ is the discretization parameter, such as $\Delta t$ or $\Delta x$, one can estimate the convergence rate $r$. This is a widely applicable procedure, but the validity of the results is, strictly speaking, tied to the chosen test problems.

Another error measure arises by asking to what extent the exact solution $u_{\rm e}$ fits the discrete equations. Clearly, $u_{\rm e}$ is in general not a solution of $\mathcal{L}_\Delta(u) = 0$, but we can define the residual

$$R = \mathcal{L}_\Delta(u_{\rm e}),$$

and investigate how close $R$ is to zero. A small $R$ means intuitively that the discrete equations are close to the differential equation, and then we are tempted to think that $u^n$ must also be close to $u_{\rm e}(t_n)$.

The residual $R$ is known as the truncation error of the finite difference scheme $\mathcal{L}_\Delta(u) = 0$. It appears that the truncation error is relatively straightforward to compute by hand or symbolic software *without specializing the differential equation and the discrete model to a special case*. The resulting $R$ is found as a power

series in the discretization parameters. The leading-order terms in the series provide an asymptotic measure of the accuracy of the numerical solution method (as the discretization parameters tend to zero). An advantage of truncation error analysis, compared to empirical estimation of convergence rates, or detailed analysis of a special problem with a mathematical expression for the numerical solution, is that the truncation error analysis reveals the accuracy of the various building blocks in the numerical method and how each building block impacts the overall accuracy. The analysis can therefore be used to detect building blocks with lower accuracy than the others.

Knowing the truncation error or other error measures is important for verification of programs by empirically establishing convergence rates. The forthcoming text will provide many examples on how to compute truncation errors for finite difference discretizations of ODEs and PDEs.

## B.2 Truncation Errors in Finite Difference Formulas

The accuracy of a finite difference formula is a fundamental issue when discretizing differential equations. We shall first go through a particular example in detail and thereafter list the truncation error in the most common finite difference approximation formulas.

### B.2.1 Example: The Backward Difference for $u'(t)$

Consider a backward finite difference approximation of the first-order derivative $u'$:

$$[D_t^- u]^n = \frac{u^n - u^{n-1}}{\Delta t} \approx u'(t_n)\,. \tag{B.1}$$

Here, $u^n$ means the value of some function $u(t)$ at a point $t_n$, and $[D_t^- u]^n$ is the *discrete derivative* of $u(t)$ at $t = t_n$. The discrete derivative computed by a finite difference is, in general, not exactly equal to the derivative $u'(t_n)$. The error in the approximation is

$$R^n = [D_t^- u]^n - u'(t_n)\,. \tag{B.2}$$

The common way of calculating $R^n$ is to

1. expand $u(t)$ in a Taylor series around the point where the derivative is evaluated, here $t_n$,
2. insert this Taylor series in (B.2), and
3. collect terms that cancel and simplify the expression.

The result is an expression for $R^n$ in terms of a power series in $\Delta t$. The error $R^n$ is commonly referred to as the *truncation error* of the finite difference formula.

The Taylor series formula often found in calculus books takes the form

$$f(x+h) = \sum_{i=0}^{\infty} \frac{1}{i!} \frac{d^i f}{dx^i}(x) h^i\,.$$

In our application, we expand the Taylor series around the point where the finite difference formula approximates the derivative. The Taylor series of $u^n$ at $t_n$ is simply $u(t_n)$, while the Taylor series of $u^{n-1}$ at $t_n$ must employ the general formula,

$$u(t_{n-1}) = u(t - \Delta t) = \sum_{i=0}^{\infty} \frac{1}{i!}\frac{d^i u}{dt^i}(t_n)(-\Delta t)^i$$
$$= u(t_n) - u'(t_n)\Delta t + \frac{1}{2}u''(t_n)\Delta t^2 + \mathcal{O}(\Delta t^3),$$

where $\mathcal{O}(\Delta t^3)$ means a power-series in $\Delta t$ where the lowest power is $\Delta t^3$. We assume that $\Delta t$ is small such that $\Delta t^p \gg \Delta t^q$ if $p$ is smaller than $q$. The details of higher-order terms in $\Delta t$ are therefore not of much interest. Inserting the Taylor series above in the right-hand side of (B.2) gives rise to some algebra:

$$[D_t^- u]^n - u'(t_n) = \frac{u(t_n) - u(t_{n-1})}{\Delta t} - u'(t_n)$$
$$= \frac{u(t_n) - (u(t_n) - u'(t_n)\Delta t + \frac{1}{2}u''(t_n)\Delta t^2 + \mathcal{O}(\Delta t^3))}{\Delta t} - u'(t_n)$$
$$= -\frac{1}{2}u''(t_n)\Delta t + \mathcal{O}(\Delta t^2),$$

which is, according to (B.2), the truncation error:

$$R^n = -\frac{1}{2}u''(t_n)\Delta t + \mathcal{O}(\Delta t^2)\,. \qquad \text{(B.3)}$$

The dominating term for small $\Delta t$ is $-\frac{1}{2}u''(t_n)\Delta t$, which is proportional to $\Delta t$, and we say that the truncation error is of *first order* in $\Delta t$.

### B.2.2 Example: The Forward Difference for $u'(t)$

We can analyze the approximation error in the forward difference

$$u'(t_n) \approx [D_t^+ u]^n = \frac{u^{n+1} - u^n}{\Delta t},$$

by writing

$$R^n = [D_t^+ u]^n - u'(t_n),$$

and expanding $u^{n+1}$ in a Taylor series around $t_n$,

$$u(t_{n+1}) = u(t_n) + u'(t_n)\Delta t + \frac{1}{2}u''(t_n)\Delta t^2 + \mathcal{O}(\Delta t^3)\,.$$

The result becomes

$$R = \frac{1}{2}u''(t_n)\Delta t + \mathcal{O}(\Delta t^2),$$

showing that also the forward difference is of first order.

### B.2.3 Example: The Central Difference for $u'(t)$

For the central difference approximation,

$$u'(t_n) \approx [D_t u]^n, \quad [D_t u]^n = \frac{u^{n+\frac{1}{2}} - u^{n-\frac{1}{2}}}{\Delta t},$$

we write

$$R^n = [D_t u]^n - u'(t_n),$$

and expand $u(t_{n+\frac{1}{2}})$ and $u(t_{n-\frac{1}{2}})$ in Taylor series around the point $t_n$ where the derivative is evaluated. We have

$$\begin{aligned}
u\left(t_{n+\frac{1}{2}}\right) =& u(t_n) + u'(t_n)\frac{1}{2}\Delta t + \frac{1}{2}u''(t_n)\left(\frac{1}{2}\Delta t\right)^2 + \\
& \frac{1}{6}u'''(t_n)\left(\frac{1}{2}\Delta t\right)^3 + \frac{1}{24}u''''(t_n)\left(\frac{1}{2}\Delta t\right)^4 + \\
& \frac{1}{120}u''''(t_n)\left(\frac{1}{2}\Delta t\right)^5 + \mathcal{O}(\Delta t^6), \\
u\left(t_{n-\frac{1}{2}}\right) =& u(t_n) - u'(t_n)\frac{1}{2}\Delta t + \frac{1}{2}u''(t_n)\left(\frac{1}{2}\Delta t\right)^2 - \\
& \frac{1}{6}u'''(t_n)\left(\frac{1}{2}\Delta t\right)^3 + \frac{1}{24}u''''(t_n)\left(\frac{1}{2}\Delta t\right)^4 - \\
& \frac{1}{120}u'''''(t_n)\left(\frac{1}{2}\Delta t\right)^5 + \mathcal{O}(\Delta t^6).
\end{aligned}$$

Now,

$$u\left(t_{n+\frac{1}{2}}\right) - u\left(t_{n-\frac{1}{2}}\right) = u'(t_n)\Delta t + \frac{1}{24}u'''(t_n)\Delta t^3 + \frac{1}{960}u'''''(t_n)\Delta t^5 + \mathcal{O}(\Delta t^7).$$

By collecting terms in $[D_t u]^n - u'(t_n)$ we find the truncation error to be

$$R^n = \frac{1}{24}u'''(t_n)\Delta t^2 + \mathcal{O}(\Delta t^4), \tag{B.4}$$

with only even powers of $\Delta t$. Since $R \sim \Delta t^2$ we say the centered difference is of *second order* in $\Delta t$.

### B.2.4 Overview of Leading-Order Error Terms in Finite Difference Formulas

Here we list the leading-order terms of the truncation errors associated with several common finite difference formulas for the first and second derivatives.

$$[D_t u]^n = \frac{u^{n+\frac{1}{2}} - u^{n-\frac{1}{2}}}{\Delta t} = u'(t_n) + R^n, \tag{B.5}$$

$$R^n = \frac{1}{24}u'''(t_n)\Delta t^2 + \mathcal{O}(\Delta t^4) \tag{B.6}$$

$$[D_{2t} u]^n = \frac{u^{n+1} - u^{n-1}}{2\Delta t} = u'(t_n) + R^n, \tag{B.7}$$

$$R^n = \frac{1}{6}u'''(t_n)\Delta t^2 + \mathcal{O}(\Delta t^4) \tag{B.8}$$

$$[D_t^- u]^n = \frac{u^n - u^{n-1}}{\Delta t} = u'(t_n) + R^n, \tag{B.9}$$

$$R^n = -\frac{1}{2}u''(t_n)\Delta t + \mathcal{O}(\Delta t^2) \tag{B.10}$$

$$[D_t^+ u]^n = \frac{u^{n+1} - u^n}{\Delta t} = u'(t_n) + R^n, \tag{B.11}$$

$$R^n = \frac{1}{2}u''(t_n)\Delta t + \mathcal{O}(\Delta t^2) \tag{B.12}$$

$$[\bar{D}_t u]^{n+\theta} = \frac{u^{n+1} - u^n}{\Delta t} = u'(t_{n+\theta}) + R^{n+\theta}, \tag{B.13}$$

$$R^{n+\theta} = \frac{1}{2}(1-2\theta)u''(t_{n+\theta})\Delta t - \frac{1}{6}((1-\theta)^3 - \theta^3)u'''(t_{n+\theta})\Delta t^2 + \mathcal{O}(\Delta t^3) \tag{B.14}$$

$$[D_t^{2-} u]^n = \frac{3u^n - 4u^{n-1} + u^{n-2}}{2\Delta t} = u'(t_n) + R^n, \tag{B.15}$$

$$R^n = -\frac{1}{3}u'''(t_n)\Delta t^2 + \mathcal{O}(\Delta t^3) \tag{B.16}$$

$$[D_t D_t u]^n = \frac{u^{n+1} - 2u^n + u^{n-1}}{\Delta t^2} = u''(t_n) + R^n, \tag{B.17}$$

$$R^n = \frac{1}{12}u''''(t_n)\Delta t^2 + \mathcal{O}(\Delta t^4) \tag{B.18}$$

It will also be convenient to have the truncation errors for various means or averages. The weighted arithmetic mean leads to

$$[\overline{u}^{t,\theta}]^{n+\theta} = \theta u^{n+1} + (1-\theta)u^n = u(t_{n+\theta}) + R^{n+\theta}, \tag{B.19}$$

$$R^{n+\theta} = \frac{1}{2}u''(t_{n+\theta})\Delta t^2\theta(1-\theta) + \mathcal{O}(\Delta t^3). \tag{B.20}$$

The standard arithmetic mean follows from this formula when $\theta = \frac{1}{2}$. Expressed at point $t_n$ we get

$$[\overline{u}^t]^n = \frac{1}{2}\left(u^{n-\frac{1}{2}} + u^{n+\frac{1}{2}}\right) = u(t_n) + R^n, \tag{B.21}$$

$$R^n = \frac{1}{8}u''(t_n)\Delta t^2 + \frac{1}{384}u''''(t_n)\Delta t^4 + \mathcal{O}(\Delta t^6)\,. \tag{B.22}$$

The geometric mean also has an error $\mathcal{O}(\Delta t^2)$:

$$\left[\overline{u^2}^{t,g}\right]^n = u^{n-\frac{1}{2}}u^{n+\frac{1}{2}} = (u^n)^2 + R^n, \tag{B.23}$$

$$R^n = -\frac{1}{4}u'(t_n)^2\Delta t^2 + \frac{1}{4}u(t_n)u''(t_n)\Delta t^2 + \mathcal{O}(\Delta t^4)\,. \tag{B.24}$$

The harmonic mean is also second-order accurate:

$$\left[\overline{u}^{t,h}\right]^n = u^n = \frac{2}{\frac{1}{u^{n-\frac{1}{2}}} + \frac{1}{u^{n+\frac{1}{2}}}} + R^{n+\frac{1}{2}}, \tag{B.25}$$

$$R^n = -\frac{u'(t_n)^2}{4u(t_n)}\Delta t^2 + \frac{1}{8}u''(t_n)\Delta t^2\,. \tag{B.26}$$

### B.2.5 Software for Computing Truncation Errors

We can use `sympy` to aid calculations with Taylor series. The derivatives can be defined as symbols, say `D3f` for the 3rd derivative of some function $f$. A truncated Taylor series can then be written as `f + D1f*h + D2f*h**2/2`. The following class takes some symbol `f` for the function in question and makes a list of symbols for the derivatives. The `__call__` method computes the symbolic form of the series truncated at `num_terms` terms.

```
import sympy as sym

class TaylorSeries:
    """Class for symbolic Taylor series."""
    def __init__(self, f, num_terms=4):
        self.f = f
        self.N = num_terms
        # Introduce symbols for the derivatives
        self.df = [f]
        for i in range(1, self.N+1):
            self.df.append(sym.Symbol('D%d%s' % (i, f.name)))

    def __call__(self, h):
        """Return the truncated Taylor series at x+h."""
        terms = self.f
        for i in range(1, self.N+1):
            terms += sym.Rational(1, sym.factorial(i))*self.df[i]*h**i
        return terms
```

We may, for example, use this class to compute the truncation error of the Forward Euler finite difference formula:

```
>>> from truncation_errors import TaylorSeries
>>> from sympy import *
>>> u, dt = symbols('u dt')
>>> u_Taylor = TaylorSeries(u, 4)
>>> u_Taylor(dt)
D1u*dt + D2u*dt**2/2 + D3u*dt**3/6 + D4u*dt**4/24 + u
>>> FE = (u_Taylor(dt) - u)/dt
>>> FE
(D1u*dt + D2u*dt**2/2 + D3u*dt**3/6 + D4u*dt**4/24)/dt
>>> simplify(FE)
D1u + D2u*dt/2 + D3u*dt**2/6 + D4u*dt**3/24
```

The truncation error consists of the terms after the first one ($u'$).

The module file `trunc/truncation_errors.py` contains another class `DiffOp` with symbolic expressions for most of the truncation errors listed in the previous section. For example:

```
>>> from truncation_errors import DiffOp
>>> from sympy import *
>>> u = Symbol('u')
>>> diffop = DiffOp(u, independent_variable='t')
>>> diffop['geometric_mean']
-D1u**2*dt**2/4 - D1u*D3u*dt**4/48 + D2u**2*dt**4/64 + ...
>>> diffop['Dtm']
D1u + D2u*dt/2 + D3u*dt**2/6 + D4u*dt**3/24
>>> >>> diffop.operator_names()
['geometric_mean', 'harmonic_mean', 'Dtm', 'D2t', 'DtDt',
 'weighted_arithmetic_mean', 'Dtp', 'Dt']
```

The indexing of `diffop` applies names that correspond to the operators: `Dtp` for $D_t^+$, `Dtm` for $D_t^-$, `Dt` for $D_t$, `D2t` for $D_{2t}$, `DtDt` for $D_t D_t$.

## B.3 Exponential Decay ODEs

We shall now compute the truncation error of a finite difference scheme for a differential equation. Our first problem involves the following linear ODE that models exponential decay,

$$u'(t) = -au(t)\,. \tag{B.27}$$

### B.3.1 Forward Euler Scheme

We begin with the Forward Euler scheme for discretizing (B.27):

$$[D_t^+ u = -au]^n\,. \tag{B.28}$$

The idea behind the truncation error computation is to insert the exact solution $u_e$ of the differential equation problem (B.27) in the discrete equations (B.28) and find the residual that arises because $u_e$ does not solve the discrete equations. Instead, $u_e$ solves the discrete equations with a residual $R^n$:

$$[D_t^+ u_e + a u_e = R]^n \,. \tag{B.29}$$

From (B.11)–(B.12) it follows that

$$[D_t^+ u_e]^n = u_e'(t_n) + \frac{1}{2}u_e''(t_n)\Delta t + \mathcal{O}(\Delta t^2),$$

which inserted in (B.29) results in

$$u_e'(t_n) + \frac{1}{2}u_e''(t_n)\Delta t + \mathcal{O}(\Delta t^2) + a u_e(t_n) = R^n \,.$$

Now, $u_e'(t_n) + a u_e^n = 0$ since $u_e$ solves the differential equation. The remaining terms constitute the residual:

$$R^n = \frac{1}{2}u_e''(t_n)\Delta t + \mathcal{O}(\Delta t^2)\,. \tag{B.30}$$

This is the truncation error $R^n$ of the Forward Euler scheme.

Because $R^n$ is proportional to $\Delta t$, we say that the Forward Euler scheme is of first order in $\Delta t$. However, the truncation error is just one error measure, and it is not equal to the true error $u_e^n - u^n$. For this simple model problem we can compute a range of different error measures for the Forward Euler scheme, including the true error $u_e^n - u^n$, and all of them have dominating terms proportional to $\Delta t$.

### B.3.2 Crank-Nicolson Scheme

For the Crank-Nicolson scheme,

$$[D_t u = -au]^{n+\frac{1}{2}}, \tag{B.31}$$

we compute the truncation error by inserting the exact solution of the ODE and adding a residual $R$,

$$[D_t u_e + a\overline{u_e}^t = R]^{n+\frac{1}{2}}\,. \tag{B.32}$$

The term $[D_t u_e]^{n+\frac{1}{2}}$ is easily computed from (B.5)–(B.6) by replacing $n$ with $n+\frac{1}{2}$ in the formula,

$$[D_t u_e]^{n+\frac{1}{2}} = u_e'\left(t_{n+\frac{1}{2}}\right) + \frac{1}{24}u_e'''\left(t_{n+\frac{1}{2}}\right)\Delta t^2 + \mathcal{O}(\Delta t^4)\,.$$

The arithmetic mean is related to $u(t_{n+\frac{1}{2}})$ by (B.21)–(B.22) so

$$[a\overline{u_e}^t]^{n+\frac{1}{2}} = u_e\left(t_{n+\frac{1}{2}}\right) + \frac{1}{8}u_e''(t_n)\Delta t^2 + \mathcal{O}(\Delta t^4)\,.$$

Inserting these expressions in (B.32) and observing that $u_e'(t_{n+\frac{1}{2}}) + au_e^{n+\frac{1}{2}} = 0$, because $u_e(t)$ solves the ODE $u'(t) = -au(t)$ at any point $t$, we find that

$$R^{n+\frac{1}{2}} = \left(\frac{1}{24}u_e'''\left(t_{n+\frac{1}{2}}\right) + \frac{1}{8}u_e''(t_n)\right)\Delta t^2 + \mathcal{O}(\Delta t^4). \tag{B.33}$$

Here, the truncation error is of second order because the leading term in $R$ is proportional to $\Delta t^2$.

At this point it is wise to redo some of the computations above to establish the truncation error of the Backward Euler scheme, see Exercise B.4.

### B.3.3 The $\theta$-Rule

We may also compute the truncation error of the $\theta$-rule,

$$[\bar{D}_t u = -a\overline{u}^{t,\theta}]^{n+\theta}.$$

Our computational task is to find $R^{n+\theta}$ in

$$[\bar{D}_t u_e + a\overline{u_e}^{t,\theta} = R]^{n+\theta}.$$

From (B.13)–(B.14) and (B.19)–(B.20) we get expressions for the terms with $u_e$. Using that $u_e'(t_{n+\theta}) + au_e(t_{n+\theta}) = 0$, we end up with

$$\begin{aligned} R^{n+\theta} &= \left(\frac{1}{2} - \theta\right)u_e''(t_{n+\theta})\Delta t + \frac{1}{2}\theta(1-\theta)u_e''(t_{n+\theta})\Delta t^2 \\ &\quad + \frac{1}{2}(\theta^2 - \theta + 3)u_e'''(t_{n+\theta})\Delta t^2 + \mathcal{O}(\Delta t^3). \end{aligned} \tag{B.34}$$

For $\theta = \frac{1}{2}$ the first-order term vanishes and the scheme is of second order, while for $\theta \neq \frac{1}{2}$ we only have a first-order scheme.

### B.3.4 Using Symbolic Software

The previously mentioned `truncation_error` module can be used to automate the Taylor series expansions and the process of collecting terms. Here is an example on possible use:

```
from truncation_error import DiffOp
from sympy import *

def decay():
    u, a = symbols('u a')
    diffop = DiffOp(u, independent_variable='t',
                    num_terms_Taylor_series=3)
    D1u = diffop.D(1)   # symbol for du/dt
    ODE = D1u + a*u     # define ODE
```

```
    # Define schemes
    FE = diffop['Dtp'] + a*u
    CN = diffop['Dt' ] + a*u
    BE = diffop['Dtm'] + a*u
    theta = diffop['barDt'] + a*diffop['weighted_arithmetic_mean']
    theta = sm.simplify(sm.expand(theta))
    # Residuals (truncation errors)
    R = {'FE': FE-ODE, 'BE': BE-ODE, 'CN': CN-ODE,
         'theta': theta-ODE}
    return R
```

The returned dictionary becomes

```
decay: {
 'BE': D2u*dt/2 + D3u*dt**2/6,
 'FE': -D2u*dt/2 + D3u*dt**2/6,
 'CN': D3u*dt**2/24,
 'theta': -D2u*a*dt**2*theta**2/2 + D2u*a*dt**2*theta/2 -
           D2u*dt*theta + D2u*dt/2 + D3u*a*dt**3*theta**3/3 -
           D3u*a*dt**3*theta**2/2 + D3u*a*dt**3*theta/6 +
           D3u*dt**2*theta**2/2 - D3u*dt**2*theta/2 + D3u*dt**2/6,
}
```

The results are in correspondence with our hand-derived expressions.

### B.3.5 Empirical Verification of the Truncation Error

The task of this section is to demonstrate how we can compute the truncation error $R$ numerically. For example, the truncation error of the Forward Euler scheme applied to the decay ODE $u' = -ua$ is

$$R^n = [D_t^+ u_e + a u_e]^n\,. \tag{B.35}$$

If we happen to know the exact solution $u_e(t)$, we can easily evaluate $R^n$ from the above formula.

To estimate how $R$ varies with the discretization parameter $\Delta t$, which has been our focus in the previous mathematical derivations, we first make the assumption that $R = C\Delta t^r$ for appropriate constants $C$ and $r$ and small enough $\Delta t$. The rate $r$ can be estimated from a series of experiments where $\Delta t$ is varied. Suppose we have $m$ experiments $(\Delta t_i, R_i)$, $i = 0,\ldots,m-1$. For two consecutive experiments $(\Delta t_{i-1}, R_{i-1})$ and $(\Delta t_i, R_i)$, a corresponding $r_{i-1}$ can be estimated by

$$r_{i-1} = \frac{\ln(R_{i-1}/R_i)}{\ln(\Delta t_{i-1}/\Delta t_i)}, \tag{B.36}$$

for $i = 1,\ldots,m-1$. Note that the truncation error $R_i$ varies through the mesh, so (B.36) is to be applied pointwise. A complicating issue is that $R_i$ and $R_{i-1}$ refer to different meshes. Pointwise comparisons of the truncation error at a certain point in all meshes therefore requires any computed $R$ to be restricted to the *coarsest mesh*

and that all finer meshes contain all the points in the coarsest mesh. Suppose we have $N_0$ intervals in the coarsest mesh. Inserting a superscript $n$ in (B.36), where $n$ counts mesh points in the coarsest mesh, $n = 0, \ldots, N_0$, leads to the formula

$$r_{i-1}^n = \frac{\ln(R_{i-1}^n/R_i^n)}{\ln(\Delta t_{i-1}/\Delta t_i)}\,. \tag{B.37}$$

Experiments are most conveniently defined by $N_0$ and a number of refinements $m$. Suppose each mesh has twice as many cells $N_i$ as the previous one:

$$N_i = 2^i N_0, \quad \Delta t_i = T N_i^{-1},$$

where $[0, T]$ is the total time interval for the computations. Suppose the computed $R_i$ values on the mesh with $N_i$ intervals are stored in an array `R[i]` (`R` being a list of arrays, one for each mesh). Restricting this $R_i$ function to the coarsest mesh means extracting every $N_i/N_0$ point and is done as follows:

```
stride = N[i]/N_0
R[i] = R[i][::stride]
```

The quantity `R[i][n]` now corresponds to $R_i^n$.

In addition to estimating $r$ for the pointwise values of $R = C\Delta t^r$, we may also consider an integrated quantity on mesh $i$,

$$R_{I,i} = \left(\Delta t_i \sum_{n=0}^{N_i} (R_i^n)^2\right)^{\frac{1}{2}} \approx \int_0^T R_i(t)dt\,. \tag{B.38}$$

The sequence $R_{I,i}$, $i = 0, \ldots, m-1$, is also expected to behave as $C\Delta t^r$, with the same $r$ as for the pointwise quantity $R$, as $\Delta t \to 0$.

The function below computes the $R_i$ and $R_{I,i}$ quantities, plots them and compares with the theoretically derived truncation error (`R_a`) if available.

```
import numpy as np
import scitools.std as plt

def estimate(truncation_error, T, N_0, m, makeplot=True):
    """
    Compute the truncation error in a problem with one independent
    variable, using m meshes, and estimate the convergence
    rate of the truncation error.

    The user-supplied function truncation_error(dt, N) computes
    the truncation error on a uniform mesh with N intervals of
    length dt::

      R, t, R_a = truncation_error(dt, N)

    where R holds the truncation error at points in the array t,
    and R_a are the corresponding theoretical truncation error
    values (None if not available).
```

```
    The truncation_error function is run on a series of meshes
    with 2**i*N_0 intervals, i=0,1,...,m-1.
    The values of R and R_a are restricted to the coarsest mesh.
    and based on these data, the convergence rate of R (pointwise)
    and time-integrated R can be estimated empirically.
    """
    N = [2**i*N_0 for i in range(m)]

    R_I = np.zeros(m) # time-integrated R values on various meshes
    R   = [None]*m    # time series of R restricted to coarsest mesh
    R_a = [None]*m    # time series of R_a restricted to coarsest mesh
    dt = np.zeros(m)
    legends_R = [];  legends_R_a = []  # all legends of curves

    for i in range(m):
        dt[i] = T/float(N[i])
        R[i], t, R_a[i] = truncation_error(dt[i], N[i])

        R_I[i] = np.sqrt(dt[i]*np.sum(R[i]**2))

        if i == 0:
            t_coarse = t           # the coarsest mesh

        stride = N[i]/N_0
        R[i] = R[i][::stride]      # restrict to coarsest mesh
        R_a[i] = R_a[i][::stride]

        if makeplot:
            plt.figure(1)
            plt.plot(t_coarse, R[i], log='y')
            legends_R.append('N=%d' % N[i])
            plt.hold('on')

            plt.figure(2)
            plt.plot(t_coarse, R_a[i] - R[i], log='y')
            plt.hold('on')
            legends_R_a.append('N=%d' % N[i])

    if makeplot:
        plt.figure(1)
        plt.xlabel('time')
        plt.ylabel('pointwise truncation error')
        plt.legend(legends_R)
        plt.savefig('R_series.png')
        plt.savefig('R_series.pdf')
        plt.figure(2)
        plt.xlabel('time')
        plt.ylabel('pointwise error in estimated truncation error')
        plt.legend(legends_R_a)
        plt.savefig('R_error.png')
        plt.savefig('R_error.pdf')
```

```
    # Convergence rates
    r_R_I = convergence_rates(dt, R_I)
    print 'R integrated in time; r:',
    print ' '.join(['%.1f' % r for r in r_R_I])
    R = np.array(R)  # two-dim. numpy array
    r_R = [convergence_rates(dt, R[:,n])[-1]
           for n in range(len(t_coarse))]
```

The first `makeplot` block demonstrates how to build up two figures in parallel, using `plt.figure(i)` to create and switch to figure number `i`. Figure numbers start at 1. A logarithmic scale is used on the $y$ axis since we expect that $R$ as a function of time (or mesh points) is exponential. The reason is that the theoretical estimate (B.30) contains $u_e''$, which for the present model goes like $e^{-at}$. Taking the logarithm makes a straight line.

The code follows closely the previously stated mathematical formulas, but the statements for computing the convergence rates might deserve an explanation. The generic help function `convergence_rate(h, E)` computes and returns $r_{i-1}$, $i = 1, \ldots, m-1$ from (B.37), given $\Delta t_i$ in `h` and $R_i^n$ in `E`:

```
def convergence_rates(h, E):
    from math import log
    r = [log(E[i]/E[i-1])/log(h[i]/h[i-1])
         for i in range(1, len(h))]
    return r
```

Calling `r_R_I = convergence_rates(dt, R_I)` computes the sequence of rates $r_0, r_1, \ldots, r_{m-2}$ for the model $R_I \sim \Delta t^r$, while the statements

```
R = np.array(R)  # two-dim. numpy array
r_R = [convergence_rates(dt, R[:,n])[-1]
       for n in range(len(t_coarse))]
```

compute the final rate $r_{m-2}$ for $R^n \sim \Delta t^r$ at each mesh point $t_n$ in the coarsest mesh. This latter computation deserves more explanation. Since `R[i][n]` holds the estimated truncation error $R_i^n$ on mesh $i$, at point $t_n$ in the coarsest mesh, `R[:,n]` picks out the sequence $R_i^n$ for $i = 0, \ldots, m-1$. The `convergence_rate` function computes the rates at $t_n$, and by indexing `[-1]` on the returned array from `convergence_rate`, we pick the rate $r_{m-2}$, which we believe is the best estimation since it is based on the two finest meshes.

The `estimate` function is available in a module `trunc_empir.py`. Let us apply this function to estimate the truncation error of the Forward Euler scheme. We need a function `decay_FE(dt, N)` that can compute (B.35) at the points in a mesh with time step `dt` and `N` intervals:

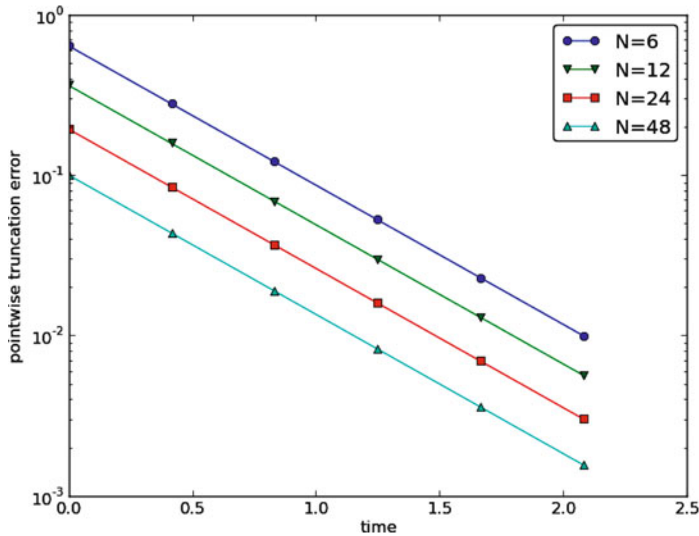


**Fig. B.1** Estimated truncation error at mesh points for different meshes

```
import numpy as np
import trunc_empir

def decay_FE(dt, N):
    dt = float(dt)
    t = np.linspace(0, N*dt, N+1)
    u_e = I*np.exp(-a*t)  # exact solution, I and a are global
    u = u_e  # naming convention when writing up the scheme
    R = np.zeros(N)

    for n in range(0, N):
        R[n] = (u[n+1] - u[n])/dt + a*u[n]

    # Theoretical expression for the trunction error
    R_a = 0.5*I*(-a)**2*np.exp(-a*t)*dt

    return R, t[:-1], R_a[:-1]

if __name__ == '__main__':
    I = 1; a = 2  # global variables needed in decay_FE
    trunc_empir.estimate(decay_FE, T=2.5, N_0=6, m=4, makeplot=True)
```

The estimated rates for the integrated truncation error $R_I$ become 1.1, 1.0, and 1.0 for this sequence of four meshes. All the rates for $R^n$, computed as `r_R`, are also very close to 1 at all mesh points. The agreement between the theoretical formula (B.30) and the computed quantity (ref(B.35)) is very good, as illustrated in Fig. B.1 and B.2. The program `trunc_decay_FE.py` was used to perform the simulations and it can easily be modified to test other schemes (see also Exercise B.5).

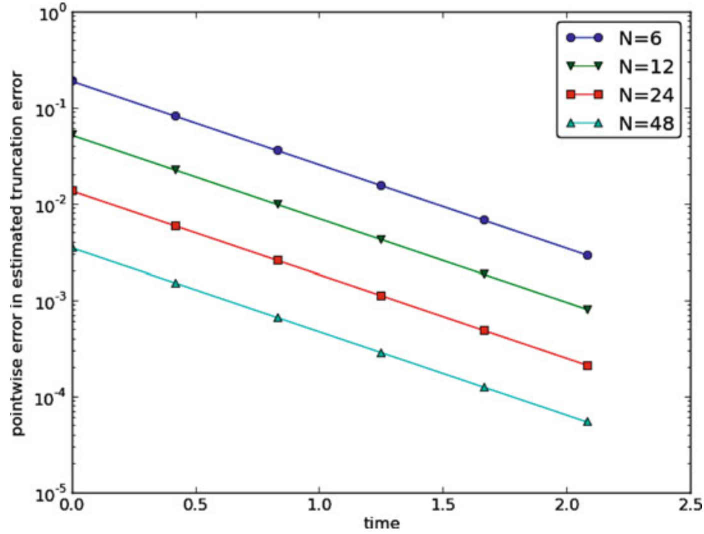


**Fig. B.2** Difference between theoretical and estimated truncation error at mesh points for different meshes

### B.3.6 Increasing the Accuracy by Adding Correction Terms

Now we ask the question: can we add terms in the differential equation that can help increase the order of the truncation error? To be precise, let us revisit the Forward Euler scheme for $u' = -au$, insert the exact solution $u_{\rm e}$, include a residual $R$, but also include new terms $C$:

$$[D_t^+ u_{\rm e} + au_{\rm e} = C + R]^n\,. \tag{B.39}$$

Inserting the Taylor expansions for $[D_t^+ u_{\rm e}]^n$ and keeping terms up to 3rd order in $\Delta t$ gives the equation

$$\frac{1}{2}u_{\rm e}''(t_n)\Delta t - \frac{1}{6}u_{\rm e}'''(t_n)\Delta t^2 + \frac{1}{24}u_{\rm e}''''(t_n)\Delta t^3 + \mathcal{O}(\Delta t^4) = C^n + R^n\,.$$

Can we find $C^n$ such that $R^n$ is $\mathcal{O}(\Delta t^2)$? Yes, by setting

$$C^n = \frac{1}{2}u_{\rm e}''(t_n)\Delta t,$$

we manage to cancel the first-order term and

$$R^n = \frac{1}{6}u_{\rm e}'''(t_n)\Delta t^2 + \mathcal{O}(\Delta t^3)\,.$$

The correction term $C^n$ introduces $\frac{1}{2}\Delta t u''$ in the discrete equation, and we have to get rid of the derivative $u''$. One idea is to approximate $u''$ by a second-order accurate finite difference formula, $u'' \approx (u^{n+1} - 2u^n + u^{n-1})/\Delta t^2$, but this introduces

an additional time level with $u^{n-1}$. Another approach is to rewrite $u''$ in terms of $u'$ or $u$ using the ODE:

$$u' = -au \quad \Rightarrow \quad u'' = -au' = -a(-au) = a^2u\,.$$

This means that we can simply set $C^n = \frac{1}{2}a^2\Delta t u^n$. We can then either solve the discrete equation

$$\left[D_t^+ u = -au + \frac{1}{2}a^2\Delta t u\right]^n, \tag{B.40}$$

or we can equivalently discretize the perturbed ODE

$$u' = -\hat{a}u, \quad \hat{a} = a\left(1 - \frac{1}{2}a\Delta t\right), \tag{B.41}$$

by a Forward Euler method. That is, we replace the original coefficient $a$ by the perturbed coefficient $\hat{a}$. Observe that $\hat{a} \to a$ as $\Delta t \to 0$.

The Forward Euler method applied to (B.41) results in

$$\left[D_t^+ u = -a\left(1 - \frac{1}{2}a\Delta t\right)u\right]^n.$$

We can control our computations and verify that the truncation error of the scheme above is indeed $\mathcal{O}(\Delta t^2)$.

Another way of revealing the fact that the perturbed ODE leads to a more accurate solution is to look at the amplification factor. Our scheme can be written as

$$u^{n+1} = Au^n, \quad A = 1 - \hat{a}\Delta t = 1 - p + \frac{1}{2}p^2, \quad p = a\Delta t,$$

The amplification factor $A$ as a function of $p = a\Delta t$ is seen to be the first three terms of the Taylor series for the exact amplification factor $e^{-p}$. The Forward Euler scheme for $u = -au$ gives only the first two terms $1 - p$ of the Taylor series for $e^{-p}$. That is, using $\hat{a}$ increases the order of the accuracy in the amplification factor.

Instead of replacing $u''$ by $a^2u$, we use the relation $u'' = -au'$ and add a term $-\frac{1}{2}a\Delta t u'$ in the ODE:

$$u' = -au - \frac{1}{2}a\Delta t u' \quad \Rightarrow \quad \left(1 + \frac{1}{2}a\Delta t\right)u' = -au\,.$$

Using a Forward Euler method results in

$$\left(1 + \frac{1}{2}a\Delta t\right)\frac{u^{n+1} - u^n}{\Delta t} = -au^n,$$

which after some algebra can be written as

$$u^{n+1} = \frac{1 - \frac{1}{2}a\Delta t}{1 + \frac{1}{2}a\Delta t}u^n\,.$$

This is the same formula as the one arising from a Crank-Nicolson scheme applied to $u' = -au$! It is now recommended to do Exercise B.6 and repeat the above steps to see what kind of correction term is needed in the Backward Euler scheme to make it second order.

The Crank-Nicolson scheme is a bit more challenging to analyze, but the ideas and techniques are the same. The discrete equation reads

$$[D_t u = -au]^{n+\frac{1}{2}},$$

and the truncation error is defined through

$$[D_t u_e + a\overline{u_e}^t = C + R]^{n+\frac{1}{2}},$$

where we have added a correction term. We need to Taylor expand both the discrete derivative and the arithmetic mean with aid of (B.5)–(B.6) and (B.21)–(B.22), respectively. The result is

$$\frac{1}{24}u_e'''\left(t_{n+\frac{1}{2}}\right)\Delta t^2 + \mathcal{O}(\Delta t^4) + \frac{a}{8}u_e''\left(t_{n+\frac{1}{2}}\right)\Delta t^2 + \mathcal{O}(\Delta t^4) = C^{n+\frac{1}{2}} + R^{n+\frac{1}{2}}.$$

The goal now is to make $C^{n+\frac{1}{2}}$ cancel the $\Delta t^2$ terms:

$$C^{n+\frac{1}{2}} = \frac{1}{24}u_e'''\left(t_{n+\frac{1}{2}}\right)\Delta t^2 + \frac{a}{8}u_e''(t_n)\Delta t^2.$$

Using $u' = -au$, we have that $u'' = a^2 u$, and we find that $u''' = -a^3 u$. We can therefore solve the perturbed ODE problem

$$u' = -\hat{a}u, \quad \hat{a} = a\left(1 - \frac{1}{12}a^2\Delta t^2\right),$$

by the Crank-Nicolson scheme and obtain a method that is of fourth order in $\Delta t$. Exercise B.7 encourages you to implement these correction terms and calculate empirical convergence rates to verify that higher-order accuracy is indeed obtained in real computations.

### B.3.7 Extension to Variable Coefficients

Let us address the decay ODE with variable coefficients,

$$u'(t) = -a(t)u(t) + b(t),$$

discretized by the Forward Euler scheme,

$$[D_t^+ u = -au + b]^n. \quad \text{(B.42)}$$

The truncation error $R$ is as always found by inserting the exact solution $u_e(t)$ in the discrete scheme:

$$[D_t^+ u_e + au_e - b = R]^n. \quad \text{(B.43)}$$

Using (B.11)–(B.12),

$$u_e'(t_n) - \frac{1}{2}u_e''(t_n)\Delta t + \mathcal{O}(\Delta t^2) + a(t_n)u_e(t_n) - b(t_n) = R^n \,.$$

Because of the ODE,

$$u_e'(t_n) + a(t_n)u_e(t_n) - b(t_n) = 0,$$

we are left with the result

$$R^n = -\frac{1}{2}u_e''(t_n)\Delta t + \mathcal{O}(\Delta t^2)\,. \tag{B.44}$$

We see that the variable coefficients do not pose any additional difficulties in this case. Exercise B.8 takes the analysis above one step further to the Crank-Nicolson scheme.

### B.3.8 Exact Solutions of the Finite Difference Equations

Having a mathematical expression for the numerical solution is very valuable in program verification, since we then know the exact numbers that the program should produce. Looking at the various formulas for the truncation errors in (B.5)–(B.6) and (B.25)–(B.26) in Sect. B.2.4, we see that all but two of the $R$ expressions contain a second or higher order derivative of $u_e$. The exceptions are the geometric and harmonic means where the truncation error involves $u_e'$ and even $u_e$ in case of the harmonic mean. So, apart from these two means, choosing $u_e$ to be a linear function of $t$, $u_e = ct + d$ for constants $c$ and $d$, will make the truncation error vanish since $u_e'' = 0$. Consequently, the truncation error of a finite difference scheme will be zero since the various approximations used will all be exact. This means that the linear solution is an exact solution of the discrete equations.

In a particular differential equation problem, the reasoning above can be used to determine if we expect a linear $u_e$ to fulfill the discrete equations. To actually prove that this is true, we can either compute the truncation error and see that it vanishes, or we can simply insert $u_e(t) = ct + d$ in the scheme and see that it fulfills the equations. The latter method is usually the simplest. It will often be necessary to add some source term to the ODE in order to allow a linear solution.

Many ODEs are discretized by centered differences. From Sect. B.2.4 we see that all the centered difference formulas have truncation errors involving $u_e'''$ or higher-order derivatives. A quadratic solution, e.g., $u_e(t) = t^2 + ct + d$, will then make the truncation errors vanish. This observation can be used to test if a quadratic solution will fulfill the discrete equations. Note that a quadratic solution will not obey the equations for a Crank-Nicolson scheme for $u' = -au + b$ because the approximation applies an arithmetic mean, which involves a truncation error with $u_e''$.

### B.3.9 Computing Truncation Errors in Nonlinear Problems

The general nonlinear ODE

$$u' = f(u,t), \tag{B.45}$$

can be solved by a Crank-Nicolson scheme

$$[D_t u = \overline{f}^t]^{n+\frac{1}{2}}. \tag{B.46}$$

The truncation error is as always defined as the residual arising when inserting the exact solution $u_e$ in the scheme:

$$[D_t u_e - \overline{f}^t = R]^{n+\frac{1}{2}}. \tag{B.47}$$

Using (B.21)–(B.22) for $\overline{f}^t$ results in

$$\begin{aligned}
[\overline{f}^t]^{n+\frac{1}{2}} &= \frac{1}{2}(f(u_e^n, t_n) + f(u_e^{n+1}, t_{n+1})) \\
&= f\left(u_e^{n+\frac{1}{2}}, t_{n+\frac{1}{2}}\right) + \frac{1}{8}u_e''\left(t_{n+\frac{1}{2}}\right)\Delta t^2 + \mathcal{O}(\Delta t^4).
\end{aligned}$$

With (B.5)–(B.6) the discrete equations (B.47) lead to

$$\begin{aligned}
&u_e'\left(t_{n+\frac{1}{2}}\right) + \frac{1}{24}u_e'''\left(t_{n+\frac{1}{2}}\right)\Delta t^2 - f\left(u_e^{n+\frac{1}{2}}, t_{n+\frac{1}{2}}\right) \\
&- \frac{1}{8}u_e''\left(t_{n+\frac{1}{2}}\right)\Delta t^2 + \mathcal{O}(\Delta t^4) = R^{n+\frac{1}{2}}.
\end{aligned}$$

Since $u_e'(t_{n+\frac{1}{2}}) - f(u_e^{n+\frac{1}{2}}, t_{n+\frac{1}{2}}) = 0$, the truncation error becomes

$$R^{n+\frac{1}{2}} = \left(\frac{1}{24}u_e'''\left(t_{n+\frac{1}{2}}\right) - \frac{1}{8}u_e''\left(t_{n+\frac{1}{2}}\right)\right)\Delta t^2.$$

The computational techniques worked well even for this nonlinear ODE.

## B.4 Vibration ODEs

### B.4.1 Linear Model Without Damping

The next example on computing the truncation error involves the following ODE for vibration problems:

$$u''(t) + \omega^2 u(t) = 0. \tag{B.48}$$

Here, $\omega$ is a given constant.

**The truncation error of a centered finite difference scheme** Using a standard, second-ordered, central difference for the second-order derivative in time, we have the scheme

$$[D_t D_t u + \omega^2 u = 0]^n . \tag{B.49}$$

Inserting the exact solution $u_e$ in this equation and adding a residual $R$ so that $u_e$ can fulfill the equation results in

$$[D_t D_t u_e + \omega^2 u_e = R]^n . \tag{B.50}$$

To calculate the truncation error $R^n$, we use (B.17)–(B.18), i.e.,

$$[D_t D_t u_e]^n = u_e''(t_n) + \frac{1}{12} u_e''''(t_n)\Delta t^2 + \mathcal{O}(\Delta t^4),$$

and the fact that $u_e''(t) + \omega^2 u_e(t) = 0$. The result is

$$R^n = \frac{1}{12} u_e''''(t_n)\Delta t^2 + \mathcal{O}(\Delta t^4) . \tag{B.51}$$

**The truncation error of approximating $u'(0)$** The initial conditions for (B.48) are $u(0) = I$ and $u'(0) = V$. The latter involves a finite difference approximation. The standard choice

$$[D_{2t} u = V]^0,$$

where $u^{-1}$ is eliminated with the aid of the discretized ODE for $n = 0$, involves a centered difference with an $\mathcal{O}(\Delta t^2)$ truncation error given by (B.7)–(B.8). The simpler choice

$$[D_t^+ u = V]^0,$$

is based on a forward difference with a truncation error $\mathcal{O}(\Delta t)$. A central question is if this initial error will impact the order of the scheme throughout the simulation. Exercise B.11 asks you to perform an experiment to investigate this question.

**Truncation error of the equation for the first step** We have shown that the truncation error of the difference used to approximate the initial condition $u'(0) = 0$ is $\mathcal{O}(\Delta t^2)$, but we can also investigate the difference equation used for the first step. In a truncation error setting, the right way to view this equation is not to use the initial condition $[D_{2t} u = V]^0$ to express $u^{-1} = u^1 - 2\Delta t V$ in order to eliminate $u^{-1}$ from the discretized differential equation, but the other way around: the fundamental equation is the discretized initial condition $[D_{2t} u = V]^0$ and we use the discretized ODE $[D_t D_t + \omega^2 u = 0]^0$ to eliminate $u^{-1}$ in the discretized initial condition. From $[D_t D_t + \omega^2 u = 0]^0$ we have

$$u^{-1} = 2u^0 - u^1 - \Delta t^2 \omega^2 u^0,$$

which inserted in $[D_{2t} u = V]^0$ gives

$$\frac{u^1 - u^0}{\Delta t} + \frac{1}{2}\omega^2 \Delta t u^0 = V . \tag{B.52}$$

The first term can be recognized as a forward difference such that the equation can be written in operator notation as

$$\left[D_t^+ u + \frac{1}{2}\omega^2 \Delta t u = V\right]^0 .$$

The truncation error is defined as

$$\left[D_t^+ u_e + \frac{1}{2}\omega^2 \Delta t u_e - V = R\right]^0 .$$

Using (B.11)–(B.12) with one more term in the Taylor series, we get that

$$u_e'(0) + \frac{1}{2}u_e''(0)\Delta t + \frac{1}{6}u_e'''(0)\Delta t^2 + \mathcal{O}(\Delta t^3) + \frac{1}{2}\omega^2 \Delta t u_e(0) - V = R^n .$$

Now, $u_e'(0) = V$ and $u_e''(0) = -\omega^2 u_e(0)$ so we get

$$R^n = \frac{1}{6}u_e'''(0)\Delta t^2 + \mathcal{O}(\Delta t^3) .$$

There is another way of analyzing the discrete initial condition, because eliminating $u^{-1}$ via the discretized ODE can be expressed as

$$[D_{2t} u + \Delta t(D_t D_t u - \omega^2 u) = V]^0 . \tag{B.53}$$

Writing out (B.53) shows that the equation is equivalent to (B.52). The truncation error is defined by

$$[D_{2t} u_e + \Delta t(D_t D_t u_e - \omega^2 u_e) = V + R]^0 .$$

Replacing the difference via (B.7)–(B.8) and (B.17)–(B.18), as well as using $u_e'(0) = V$ and $u_e''(0) = -\omega^2 u_e(0)$, gives

$$R^n = \frac{1}{6}u_e'''(0)\Delta t^2 + \mathcal{O}(\Delta t^3) .$$

**Computing correction terms** The idea of using correction terms to increase the order of $R^n$ can be applied as described in Sect. B.3.6. We look at

$$[D_t D_t u_e + \omega^2 u_e = C + R]^n ,$$

and observe that $C^n$ must be chosen to cancel the $\Delta t^2$ term in $R^n$. That is,

$$C^n = \frac{1}{12}u_e''''(t_n)\Delta t^2 .$$

To get rid of the 4th-order derivative we can use the differential equation: $u'' = -\omega^2 u$, which implies $u'''' = \omega^4 u$. Adding the correction term to the ODE results in

$$u'' + \omega^2\left(1 - \frac{1}{12}\omega^2 \Delta t^2\right) u = 0 . \tag{B.54}$$

Solving this equation by the standard scheme

$$\left[D_t D_t u + \omega^2 \left(1 - \frac{1}{12}\omega^2 \Delta t^2\right) u = 0\right]^n,$$

will result in a scheme with truncation error $\mathcal{O}(\Delta t^4)$.

We can use another set of arguments to justify that (B.54) leads to a higher-order method. Mathematical analysis of the scheme (B.49) reveals that the numerical frequency $\tilde{\omega}$ is (approximately as $\Delta t \to 0$)

$$\tilde{\omega} = \omega \left(1 + \frac{1}{24}\omega^2 \Delta t^2\right).$$

One can therefore attempt to replace $\omega$ in the ODE by a slightly smaller $\omega$ since the numerics will make it larger:

$$\left[u'' + \left(\omega \left(1 - \frac{1}{24}\omega^2 \Delta t^2\right)\right)^2 u = 0\right]^n.$$

Expanding the squared term and omitting the higher-order term $\Delta t^4$ gives exactly the ODE (B.54). Experiments show that $u^n$ is computed to 4th order in $\Delta t$. You can confirm this by running a little program in the `vib` directory:

```
from vib_undamped import convergence_rates, solver_adjust_w

r = convergence_rates(
    m=5, solver_function=solver_adjust_w, num_periods=8)
```

One will see that the rates `r` lie around 4.

### B.4.2 Model with Damping and Nonlinearity

The model (B.48) can be extended to include damping $\beta u'$, a nonlinear restoring (spring) force $s(u)$, and some known excitation force $F(t)$:

$$mu'' + \beta u' + s(u) = F(t). \tag{B.55}$$

The coefficient $m$ usually represents the mass of the system. This governing equation can be discretized by centered differences:

$$[mD_t D_t u + \beta D_{2t} u + s(u) = F]^n. \tag{B.56}$$

The exact solution $u_e$ fulfills the discrete equations with a residual term:

$$[mD_t D_t u_e + \beta D_{2t} u_e + s(u_e) = F + R]^n. \tag{B.57}$$

Using (B.17)–(B.18) and (B.7)–(B.8) we get

$$[mD_tD_tu_e + \beta D_{2t}u_e]^n = mu_e''(t_n) + \beta u_e'(t_n) + \left(\frac{m}{12}u_e''''(t_n) + \frac{\beta}{6}u_e'''(t_n)\right)\Delta t^2 + \mathcal{O}(\Delta t^4)\,.$$

Combining this with the previous equation, we can collect the terms

$$mu_e''(t_n) + \beta u_e'(t_n) + \omega^2 u_e(t_n) + s(u_e(t_n)) - F^n,$$

and set this sum to zero because $u_e$ solves the differential equation. We are left with the truncation error

$$R^n = \left(\frac{m}{12}u_e''''(t_n) + \frac{\beta}{6}u_e'''(t_n)\right)\Delta t^2 + \mathcal{O}(\Delta t^4), \quad \text{(B.58)}$$

so the scheme is of second order.

According to (B.58), we can add correction terms

$$C^n = \left(\frac{m}{12}u_e''''(t_n) + \frac{\beta}{6}u_e'''(t_n)\right)\Delta t^2,$$

to the right-hand side of the ODE to obtain a fourth-order scheme. However, expressing $u''''$ and $u'''$ in terms of lower-order derivatives is now harder because the differential equation is more complicated:

$$\begin{aligned}
u''' &= \frac{1}{m}(F' - \beta u'' - s'(u)u'),\\
u'''' &= \frac{1}{m}(F'' - \beta u''' - s''(u)(u')^2 - s'(u)u''),\\
&= \frac{1}{m}(F'' - \beta\frac{1}{m}(F' - \beta u'' - s'(u)u') - s''(u)(u')^2 - s'(u)u'')\,.
\end{aligned}$$

It is not impossible to discretize the resulting modified ODE, but it is up to debate whether correction terms are feasible and the way to go. Computing with a smaller $\Delta t$ is usually always possible in these problems to achieve the desired accuracy.

### B.4.3 Extension to Quadratic Damping

Instead of the linear damping term $\beta u'$ in (B.55) we now consider quadratic damping $\beta|u'|u'$:

$$mu'' + \beta|u'|u' + s(u) = F(t)\,. \quad \text{(B.59)}$$

A centered difference for $u'$ gives rise to a nonlinearity, which can be linearized using a geometric mean: $[|u'|u']^n \approx |[u']^{n-\frac{1}{2}}|[u']^{n+\frac{1}{2}}$. The resulting scheme becomes

$$[mD_tD_tu]^n + \beta|[D_tu]^{n-\frac{1}{2}}|[D_tu]^{n+\frac{1}{2}} + s(u^n) = F^n\,. \quad \text{(B.60)}$$

The truncation error is defined through

$$[mD_tD_tu_e]^n + \beta|[D_tu_e]^{n-\frac{1}{2}}|[D_tu_e]^{n+\frac{1}{2}} + s(u_e^n) - F^n = R^n\,. \tag{B.61}$$

We start with expressing the truncation error of the geometric mean. According to (B.23)–(B.24),

$$|[D_tu_e]^{n-\frac{1}{2}}|[D_tu_e]^{n+\frac{1}{2}} = [|D_tu_e|D_tu_e]^n - \frac{1}{4}u_e'(t_n)^2\Delta t^2 + \frac{1}{4}u_e(t_n)u_e''(t_n)\Delta t^2 + \mathcal{O}(\Delta t^4)\,.$$

Using (B.5)–(B.6) for the $D_tu_e$ factors results in

$$[|D_tu_e|D_tu_e]^n = \left|u_e' + \frac{1}{24}u_e'''(t_n)\Delta t^2 + \mathcal{O}(\Delta t^4)\right|\left(u_e' + \frac{1}{24}u_e'''(t_n)\Delta t^2 + \mathcal{O}(\Delta t^4)\right)\,.$$

We can remove the absolute value since it essentially gives a factor 1 or −1 only. Calculating the product, we have the leading-order terms

$$[D_tu_eD_tu_e]^n = (u_e'(t_n))^2 + \frac{1}{12}u_e(t_n)u_e'''(t_n)\Delta t^2 + \mathcal{O}(\Delta t^4)\,.$$

With

$$m[D_tD_tu_e]^n = mu_e''(t_n) + \frac{m}{12}u_e''''(t_n)\Delta t^2 + \mathcal{O}(\Delta t^4),$$

and using the differential equation on the form $mu'' + \beta(u')^2 + s(u) = F$, we end up with

$$R^n = \left(\frac{m}{12}u_e''''(t_n) + \frac{\beta}{12}u_e(t_n)u_e'''(t_n)\right)\Delta t^2 + \mathcal{O}(\Delta t^4)\,.$$

This result demonstrates that we have second-order accuracy also with quadratic damping. The key elements that lead to the second-order accuracy is that the difference approximations are $\mathcal{O}(\Delta t^2)$ *and* the geometric mean approximation is also $\mathcal{O}(\Delta t^2)$.

### B.4.4 The General Model Formulated as First-Order ODEs

The second-order model (B.59) can be formulated as a first-order system,

$$v' = \frac{1}{m}\left(F(t) - \beta|v|v - s(u)\right), \tag{B.62}$$

$$u' = v\,. \tag{B.63}$$

The system (B.63)–(B.63) can be solved either by a forward-backward scheme (the Euler-Cromer method) or a centered scheme on a staggered mesh.

**A centered scheme on a staggered mesh** We now introduce a staggered mesh where we seek $u$ at mesh points $t_n$ and $v$ at points $t_{n+\frac{1}{2}}$ in between the $u$ points. The staggered mesh makes it easy to formulate centered differences in the system (B.63)–(B.63):

$$[D_t u = v]^{n-\frac{1}{2}}, \tag{B.64}$$

$$\left[D_t v = \frac{1}{m}(F(t) - \beta|v|v - s(u))\right]^n. \tag{B.65}$$

The term $|v^n|v^n$ causes trouble since $v^n$ is not computed, only $v^{n-\frac{1}{2}}$ and $v^{n+\frac{1}{2}}$. Using geometric mean, we can express $|v^n|v^n$ in terms of known quantities: $|v^n|v^n \approx |v^{n-\frac{1}{2}}|v^{n+\frac{1}{2}}$. We then have

$$[D_t u]^{n-\frac{1}{2}} = v^{n-\frac{1}{2}}, \tag{B.66}$$

$$[D_t v]^n = \frac{1}{m}\left(F(t_n) - \beta\left|v^{n-\frac{1}{2}}\right|v^{n+\frac{1}{2}} - s(u^n)\right). \tag{B.67}$$

The truncation error in each equation fulfills

$$[D_t u_e]^{n-\frac{1}{2}} = v_e\left(t_{n-\frac{1}{2}}\right) + R_u^{n-\frac{1}{2}},$$

$$[D_t v_e]^n = \frac{1}{m}\left(F(t_n) - \beta\left|v_e\left(t_{n-\frac{1}{2}}\right)\right|v_e\left(t_{n+\frac{1}{2}}\right) - s(u^n)\right) + R_v^n.$$

The truncation error of the centered differences is given by (B.5)–(B.6), and the geometric mean approximation analysis can be taken from (B.23)–(B.24). These results lead to

$$u_e'\left(t_{n-\frac{1}{2}}\right) + \frac{1}{24}u_e'''\left(t_{n-\frac{1}{2}}\right)\Delta t^2 + \mathcal{O}(\Delta t^4) = v_e\left(t_{n-\frac{1}{2}}\right) + R_u^{n-\frac{1}{2}},$$

and

$$v_e'(t_n) = \frac{1}{m}(F(t_n) - \beta|v_e(t_n)|v_e(t_n) + \mathcal{O}(\Delta t^2) - s(u^n)) + R_v^n.$$

The ODEs fulfilled by $u_e$ and $v_e$ are evident in these equations, and we achieve second-order accuracy for the truncation error in both equations:

$$R_u^{n-\frac{1}{2}} = \mathcal{O}(\Delta t^2), \quad R_v^n = \mathcal{O}(\Delta t^2).$$

## B.5 Wave Equations

### B.5.1 Linear Wave Equation in 1D

The standard, linear wave equation in 1D for a function $u(x,t)$ reads

$$\frac{\partial^2 u}{\partial t^2} = c^2\frac{\partial^2 u}{\partial x^2} + f(x,t), \quad x \in (0,L),\ t \in (0,T], \tag{B.68}$$

where $c$ is the constant wave velocity of the physical medium in $[0, L]$. The equation can also be more compactly written as

$$u_{tt} = c^2 u_{xx} + f, \quad x \in (0, L),\ t \in (0, T]\,. \tag{B.69}$$

Centered, second-order finite differences are a natural choice for discretizing the derivatives, leading to

$$[D_t D_t u = c^2 D_x D_x u + f]^n_i\,. \tag{B.70}$$

Inserting the exact solution $u_{\rm e}(x, t)$ in (B.70) makes this function fulfill the equation if we add the term $R$:

$$[D_t D_t u_{\rm e} = c^2 D_x D_x u_{\rm e} + f + R]^n_i\,. \tag{B.71}$$

Our purpose is to calculate the truncation error $R$. From (B.17)–(B.18) we have that

$$[D_t D_t u_{\rm e}]^n_i = u_{{\rm e},tt}(x_i, t_n) + \frac{1}{12} u_{{\rm e},tttt}(x_i, t_n)\Delta t^2 + \mathcal{O}(\Delta t^4),$$

when we use a notation taking into account that $u_{\rm e}$ is a function of two variables and that derivatives must be partial derivatives. The notation $u_{{\rm e},tt}$ means $\partial^2 u_{\rm e}/\partial t^2$.

The same formula may also be applied to the $x$-derivative term:

$$[D_x D_x u_{\rm e}]^n_i = u_{{\rm e},xx}(x_i, t_n) + \frac{1}{12} u_{{\rm e},xxxx}(x_i, t_n)\Delta x^2 + \mathcal{O}(\Delta x^4)\,.$$

Equation (B.71) now becomes

$$\begin{aligned} u_{{\rm e},tt} + \frac{1}{12} u_{{\rm e},tttt}(x_i, t_n)\Delta t^2 &= c^2 u_{{\rm e},xx} + c^2 \frac{1}{12} u_{{\rm e},xxxx}(x_i, t_n)\Delta x^2 + f(x_i, t_n) \\ &\quad + \mathcal{O}(\Delta t^4, \Delta x^4) + R^n_i\,. \end{aligned}$$

Because $u_{\rm e}$ fulfills the partial differential equation (PDE) (B.69), the first, third, and fifth term cancel out, and we are left with

$$R^n_i = \frac{1}{12} u_{{\rm e},tttt}(x_i, t_n)\Delta t^2 - c^2 \frac{1}{12} u_{{\rm e},xxxx}(x_i, t_n)\Delta x^2 + \mathcal{O}(\Delta t^4, \Delta x^4), \tag{B.72}$$

showing that the scheme (B.70) is of second order in the time and space mesh spacing.

### B.5.2 Finding Correction Terms

Can we add correction terms to the PDE and increase the order of $R^n_i$ in (B.72)? The starting point is

$$[D_t D_t u_{\rm e} = c^2 D_x D_x u_{\rm e} + f + C + R]^n_i\,. \tag{B.73}$$

From the previous analysis we simply get (B.72) again, but now with $C$:

$$R_i^n + C_i^n = \frac{1}{12}u_{\mathrm{e},tttt}(x_i,t_n)\Delta t^2 - c^2\frac{1}{12}u_{\mathrm{e},xxxx}(x_i,t_n)\Delta x^2 + \mathcal{O}(\Delta t^4,\Delta x^4)\,. \tag{B.74}$$

The idea is to let $C_i^n$ cancel the $\Delta t^2$ and $\Delta x^2$ terms to make $R_i^n = \mathcal{O}(\Delta t^4,\Delta x^4)$:

$$C_i^n = \frac{1}{12}u_{\mathrm{e},tttt}(x_i,t_n)\Delta t^2 - c^2\frac{1}{12}u_{\mathrm{e},xxxx}(x_i,t_n)\Delta x^2\,.$$

Essentially, it means that we add a new term

$$C = \frac{1}{12}\left(u_{tttt}\Delta t^2 - c^2 u_{xxxx}\Delta x^2\right),$$

to the right-hand side of the PDE. We must either discretize these 4th-order derivatives directly or rewrite them in terms of lower-order derivatives with the aid of the PDE. The latter approach is more feasible. From the PDE we have the operator equality

$$\frac{\partial^2}{\partial t^2} = c^2\frac{\partial^2}{\partial x^2},$$

so

$$u_{tttt} = c^2 u_{xxtt},\quad u_{xxxx} = c^{-2}u_{ttxx}\,.$$

Assuming $u$ is smooth enough, so that $u_{xxtt} = u_{ttxx}$, these relations lead to

$$C = \frac{1}{12}((c^2\Delta t^2 - \Delta x^2)u_{xx})_{tt}\,.$$

A natural discretization is

$$C_i^n = \frac{1}{12}((c^2\Delta t^2 - \Delta x^2)[D_xD_xD_tD_tu]_i^n\,.$$

Writing out $[D_xD_xD_tD_tu]_i^n$ as $[D_xD_x(D_tD_tu)]_i^n$ gives

$$\frac{1}{\Delta t^2}\Bigg(\frac{u_{i+1}^{n+1} - 2u_{i+1}^n + u_{i+1}^{n-1}}{\Delta x^2} - 2\frac{u_i^{n+1} - 2u_i^n + u_i^{n-1}}{\Delta x^2} + \frac{u_{i-1}^{n+1} - 2u_{i-1}^n + u_{i-1}^{n-1}}{\Delta x^2}\Bigg).$$

Now the unknown values $u_{i+1}^{n+1}$, $u_i^{n+1}$, and $u_{i-1}^{n+1}$ are *coupled*, and we must solve a tridiagonal system to find them. This is in principle straightforward, but it results in an implicit finite difference scheme, while we had a convenient explicit scheme without the correction terms.

### B.5.3 Extension to Variable Coefficients

Now we address the variable coefficient version of the linear 1D wave equation,

$$\frac{\partial^2 u}{\partial t^2} = \frac{\partial}{\partial x}\left(\lambda(x)\frac{\partial u}{\partial x}\right),$$

or written more compactly as

$$u_{tt} = (\lambda u_x)_x\,. \tag{B.75}$$

The discrete counterpart to this equation, using arithmetic mean for $\lambda$ and centered differences, reads

$$\left[D_t D_t u = D_x \overline{\lambda}^x D_x u\right]_i^n\,. \tag{B.76}$$

The truncation error is the residual $R$ in the equation

$$\left[D_t D_t u_{\rm e} = D_x \overline{\lambda}^x D_x u_{\rm e} + R\right]_i^n\,. \tag{B.77}$$

The difficulty with (B.77) is how to compute the truncation error of the term $[D_x \overline{\lambda}^x D_x u_{\rm e}]_i^n$.

We start by writing out the outer operator:

$$\left[D_x \overline{\lambda}^x D_x u_{\rm e}\right]_i^n = \frac{1}{\Delta x}\left(\left[\overline{\lambda}^x D_x u_{\rm e}\right]_{i+\frac{1}{2}}^n - \left[\overline{\lambda}^x D_x u_{\rm e}\right]_{i-\frac{1}{2}}^n\right). \tag{B.78}$$

With the aid of (B.5)–(B.6) and (B.21)–(B.22) we have

$$\begin{aligned}
[D_x u_{\rm e}]_{i+\frac{1}{2}}^n &= u_{{\rm e},x}\left(x_{i+\frac{1}{2}}, t_n\right) + \frac{1}{24} u_{{\rm e},xxx}\left(x_{i+\frac{1}{2}}, t_n\right)\Delta x^2 + \mathcal{O}(\Delta x^4),\\
\left[\overline{\lambda}^x\right]_{i+\frac{1}{2}} &= \lambda\left(x_{i+\frac{1}{2}}\right) + \frac{1}{8}\lambda''\left(x_{i+\frac{1}{2}}\right)\Delta x^2 + \mathcal{O}(\Delta x^4),\\
\left[\overline{\lambda}^x D_x u_{\rm e}\right]_{i+\frac{1}{2}}^n &= \left(\lambda\left(x_{i+\frac{1}{2}}\right) + \frac{1}{8}\lambda''\left(x_{i+\frac{1}{2}}\right)\Delta x^2 + \mathcal{O}(\Delta x^4)\right)\\
&\quad\times\left(u_{{\rm e},x}\left(x_{i+\frac{1}{2}}, t_n\right) + \frac{1}{24} u_{{\rm e},xxx}\left(x_{i+\frac{1}{2}}, t_n\right)\Delta x^2 + \mathcal{O}(\Delta x^4)\right)\\
&= \lambda\left(x_{i+\frac{1}{2}}\right) u_{{\rm e},x}\left(x_{i+\frac{1}{2}}, t_n\right) + \lambda\left(x_{i+\frac{1}{2}}\right)\frac{1}{24} u_{{\rm e},xxx}\left(x_{i+\frac{1}{2}}, t_n\right)\Delta x^2\\
&\quad + u_{{\rm e},x}\left(x_{i+\frac{1}{2}}, t_n\right)\frac{1}{8}\lambda''\left(x_{i+\frac{1}{2}}\right)\Delta x^2 + \mathcal{O}(\Delta x^4)\\
&= [\lambda u_{{\rm e},x}]_{i+\frac{1}{2}}^n + G_{i+\frac{1}{2}}^n \Delta x^2 + \mathcal{O}(\Delta x^4),
\end{aligned}$$

where we have introduced the short form

$$G_{i+\frac{1}{2}}^n = \frac{1}{24} u_{{\rm e},xxx}\left(x_{i+\frac{1}{2}}, t_n\right)\lambda\left(x_{i+\frac{1}{2}}\right) + u_{{\rm e},x}\left(x_{i+\frac{1}{2}}, t_n\right)\frac{1}{8}\lambda''\left(x_{i+\frac{1}{2}}\right).$$

Similarly, we find that

$$\left[\overline{\lambda}^x D_x u_{\rm e}\right]_{i-\frac{1}{2}}^n = [\lambda u_{{\rm e},x}]_{i-\frac{1}{2}}^n + G_{i-\frac{1}{2}}^n \Delta x^2 + \mathcal{O}(\Delta x^4)\,.$$

Inserting these expressions in the outer operator (B.78) results in

$$\begin{aligned}
\left[D_x \overline{\lambda}^x D_x u_{\rm e}\right]_i^n &= \frac{1}{\Delta x}\left(\left[\overline{\lambda}^x D_x u_{\rm e}\right]_{i+\frac{1}{2}}^n - \left[\overline{\lambda}^x D_x u_{\rm e}\right]_{i-\frac{1}{2}}^n\right)\\
&= \frac{1}{\Delta x}\left([\lambda u_{{\rm e},x}]_{i+\frac{1}{2}}^n + G_{i+\frac{1}{2}}^n\Delta x^2 - [\lambda u_{{\rm e},x}]_{i-\frac{1}{2}}^n - G_{i-\frac{1}{2}}^n\Delta x^2 + \mathcal{O}(\Delta x^4)\right)\\
&= [D_x \lambda u_{{\rm e},x}]_i^n + [D_x G]_i^n \Delta x^2 + \mathcal{O}(\Delta x^4)\,.
\end{aligned}$$

The reason for $\mathcal{O}(\Delta x^4)$ in the remainder is that there are coefficients in front of this term, say $H\Delta x^4$, and the subtraction and division by $\Delta x$ results in $[D_x H]_i^n \Delta x^4$.

We can now use (B.5)–(B.6) to express the $D_x$ operator in $[D_x \lambda u_{e,x}]_i^n$ as a derivative and a truncation error:

$$[D_x \lambda u_{e,x}]_i^n = \frac{\partial}{\partial x}\lambda(x_i) u_{e,x}(x_i, t_n) + \frac{1}{24}(\lambda u_{e,x})_{xxx}(x_i, t_n)\Delta x^2 + \mathcal{O}(\Delta x^4)\,.$$

Expressions like $[D_x G]_i^n \Delta x^2$ can be treated in an identical way,

$$[D_x G]_i^n \Delta x^2 = G_x(x_i, t_n)\Delta x^2 + \frac{1}{24}G_{xxx}(x_i, t_n)\Delta x^4 + \mathcal{O}(\Delta x^4)\,.$$

There will be a number of terms with the $\Delta x^2$ factor. We lump these now into $\mathcal{O}(\Delta x^2)$. The result of the truncation error analysis of the spatial derivative is therefore summarized as

$$\left[D_x \overline{\lambda}^x D_x u_e\right]_i^n = \frac{\partial}{\partial x}\lambda(x_i) u_{e,x}(x_i, t_n) + \mathcal{O}(\Delta x^2)\,.$$

After having treated the $[D_t D_t u_e]_i^n$ term as well, we achieve

$$R_i^n = \mathcal{O}(\Delta x^2) + \frac{1}{12}u_{e,tttt}(x_i, t_n)\Delta t^2\,.$$

The main conclusion is that the scheme is of second-order in time and space also in this variable coefficient case. The key ingredients for second order are the centered differences and the arithmetic mean for $\lambda$: all those building blocks feature second-order accuracy.

### B.5.4 Linear Wave Equation in 2D/3D

The two-dimensional extension of (B.68) takes the form

$$\frac{\partial^2 u}{\partial t^2} = c^2\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right) + f(x, y, t), \quad (x, y) \in (0, L) \times (0, H),\ t \in (0, T], \tag{B.79}$$

where now $c(x, y)$ is the constant wave velocity of the physical medium $[0, L] \times [0, H]$. In compact notation, the PDE (B.79) can be written

$$u_{tt} = c^2(u_{xx} + u_{yy}) + f(x, y, t), \quad (x, y) \in (0, L) \times (0, H),\ t \in (0, T], \tag{B.80}$$

in 2D, while the 3D version reads

$$u_{tt} = c^2(u_{xx} + u_{yy} + u_{zz}) + f(x, y, z, t), \tag{B.81}$$

for $(x, y, z) \in (0, L) \times (0, H) \times (0, B)$ and $t \in (0, T]$.

Approximating the second-order derivatives by the standard formulas (B.17)–(B.18) yields the scheme

$$[D_t D_t u = c^2(D_x D_x u + D_y D_y u + D_z D_z u) + f]_{i,j,k}^n\,. \tag{B.82}$$

The truncation error is found from

$$[D_tD_tu_e = c^2(D_xD_xu_e + D_yD_yu_e + D_zD_zu_e) + f + R]^n_{i,j,k}\,. \tag{B.83}$$

The calculations from the 1D case can be repeated with the terms in the $y$ and $z$ directions. Collecting terms that fulfill the PDE, we end up with

$$R^n_{i,j,k} = \left[\frac{1}{12}u_{e,tttt}\Delta t^2 - c^2\frac{1}{12}\left(u_{e,xxxx}\Delta x^2 + u_{e,yyyy}\Delta x^2 + u_{e,zzzz}\Delta z^2\right)\right]^n_{i,j,k} + \mathcal{O}(\Delta t^4, \Delta x^4, \Delta y^4, \Delta z^4)\,. \tag{B.84}$$

## B.6 Diffusion Equations

### B.6.1 Linear Diffusion Equation in 1D

The standard, linear, 1D diffusion equation takes the form

$$\frac{\partial u}{\partial t} = \alpha\frac{\partial^2 u}{\partial x^2} + f(x,t), \quad x \in (0,L),\ t \in (0,T], \tag{B.85}$$

where $\alpha > 0$ is a constant diffusion coefficient. A more compact form of the diffusion equation is $u_t = \alpha u_{xx} + f$.

The spatial derivative in the diffusion equation, $\alpha u_{xx}$, is commonly discretized as $[D_xD_xu]^n_i$. The time-derivative, however, can be treated by a variety of methods.

**The Forward Euler scheme in time** Let us start with the simple Forward Euler scheme:

$$[D_t^+u = \alpha D_xD_xu + f]^n_i\,.$$

The truncation error arises as the residual $R$ when inserting the exact solution $u_e$ in the discrete equations:

$$[D_t^+u_e = \alpha D_xD_xu_e + f + R]^n_i\,.$$

Now, using (B.11)–(B.12) and (B.17)–(B.18), we can transform the difference operators to derivatives:

$$u_{e,t}(x_i,t_n) + \frac{1}{2}u_{e,tt}(t_n)\Delta t + \mathcal{O}(\Delta t^2) = \alpha u_{e,xx}(x_i,t_n) + \frac{\alpha}{12}u_{e,xxxx}(x_i,t_n)\Delta x^2 + \mathcal{O}(\Delta x^4) + f(x_i,t_n) + R^n_i\,.$$

The terms $u_{e,t}(x_i,t_n) - \alpha u_{e,xx}(x_i,t_n) - f(x_i,t_n)$ vanish because $u_e$ solves the PDE. The truncation error then becomes

$$R^n_i = \frac{1}{2}u_{e,tt}(t_n)\Delta t + \mathcal{O}(\Delta t^2) - \frac{\alpha}{12}u_{e,xxxx}(x_i,t_n)\Delta x^2 + \mathcal{O}(\Delta x^4)\,.$$

**The Crank-Nicolson scheme in time** The Crank-Nicolson method consists of using a centered difference for $u_t$ and an arithmetic average of the $u_{xx}$ term:

$$[D_t u]_i^{n+\frac{1}{2}} = \alpha\frac{1}{2}\left([D_xD_xu]_i^n + [D_xD_xu]_i^{n+1}\right) + f_i^{n+\frac{1}{2}}\,.$$

The equation for the truncation error is

$$[D_t u_{\mathrm{e}}]_i^{n+\frac{1}{2}} = \alpha\frac{1}{2}\left([D_xD_xu_{\mathrm{e}}]_i^n + [D_xD_xu_{\mathrm{e}}]_i^{n+1}\right) + f_i^{n+\frac{1}{2}} + R_i^{n+\frac{1}{2}}\,.$$

To find the truncation error, we start by expressing the arithmetic average in terms of values at time $t_{n+\frac{1}{2}}$. According to (B.21)–(B.22),

$$\frac{1}{2}\left([D_xD_xu_{\mathrm{e}}]_i^n + [D_xD_xu_{\mathrm{e}}]_i^{n+1}\right) = [D_xD_xu_{\mathrm{e}}]_i^{n+\frac{1}{2}} + \frac{1}{8}[D_xD_xu_{\mathrm{e},tt}]_i^{n+\frac{1}{2}}\Delta t^2 + \mathcal{O}(\Delta t^4)\,.$$

With (B.17)–(B.18) we can express the difference operator $D_xD_xu$ in terms of a derivative:

$$[D_xD_xu_{\mathrm{e}}]_i^{n+\frac{1}{2}} = u_{\mathrm{e},xx}\left(x_i, t_{n+\frac{1}{2}}\right) + \frac{1}{12}u_{\mathrm{e},xxxx}\left(x_i, t_{n+\frac{1}{2}}\right)\Delta x^2 + \mathcal{O}(\Delta x^4)\,.$$

The error term from the arithmetic mean is similarly expanded,

$$\frac{1}{8}[D_xD_xu_{\mathrm{e},tt}]_i^{n+\frac{1}{2}}\Delta t^2 = \frac{1}{8}u_{\mathrm{e},ttxx}\left(x_i, t_{n+\frac{1}{2}}\right)\Delta t^2 + \mathcal{O}(\Delta t^2\Delta x^2)\,.$$

The time derivative is analyzed using (B.5)–(B.6):

$$[D_t u]_i^{n+\frac{1}{2}} = u_{\mathrm{e},t}\left(x_i, t_{n+\frac{1}{2}}\right) + \frac{1}{24}u_{\mathrm{e},ttt}\left(x_i, t_{n+\frac{1}{2}}\right)\Delta t^2 + \mathcal{O}(\Delta t^4)\,.$$

Summing up all the contributions and notifying that

$$u_{\mathrm{e},t}\left(x_i, t_{n+\frac{1}{2}}\right) = \alpha u_{\mathrm{e},xx}\left(x_i, t_{n+\frac{1}{2}}\right) + f\left(x_i, t_{n+\frac{1}{2}}\right),$$

the truncation error is given by

$$R_i^{n+\frac{1}{2}} = \frac{1}{8}u_{\mathrm{e},xx}\left(x_i, t_{n+\frac{1}{2}}\right)\Delta t^2 + \frac{1}{12}u_{\mathrm{e},xxxx}\left(x_i, t_{n+\frac{1}{2}}\right)\Delta x^2 + \frac{1}{24}u_{\mathrm{e},ttt}\left(x_i, t_{n+\frac{1}{2}}\right)\Delta t^2 + \mathcal{O}(\Delta x^4) + \mathcal{O}(\Delta t^4) + \mathcal{O}(\Delta t^2\Delta x^2)\,.$$

### B.6.2 Nonlinear Diffusion Equation in 1D

We address the PDE

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial x}\left(\alpha(u)\frac{\partial u}{\partial x}\right) + f(u),$$

with two potentially nonlinear coefficients $q(u)$ and $\alpha(u)$. We use a Backward Euler scheme with arithmetic mean for $\alpha(u)$,

$$\left[D^-u = D_x\overline{\alpha(u)}^x D_x u + f(u)\right]_i^n .$$

Inserting $u_e$ defines the truncation error $R$:

$$\left[D^-u_e = D_x\overline{\alpha(u_e)}^x D_x u_e + f(u_e) + R\right]_i^n .$$

The most computationally challenging part is the variable coefficient with $\alpha(u)$, but we can use the same setup as in Sect. B.5.3 and arrive at a truncation error $\mathcal{O}(\Delta x^2)$ for the $x$-derivative term. The nonlinear term $[f(u_e)]_i^n = f(u_e(x_i, t_n))$ matches $x$ and $t$ derivatives of $u_e$ in the PDE. We end up with

$$R_i^n = -\frac{1}{2}\frac{\partial^2}{\partial t^2}u_e(x_i, t_n)\Delta t + \mathcal{O}(\Delta x^2) .$$

## B.7 Exercises

**Exercise B.1: Truncation error of a weighted mean**
Derive the truncation error of the weighted mean in (B.19)–(B.20).

*Hint* Expand $u_e^{n+1}$ and $u_e^n$ around $t_{n+\theta}$.
Filename: `trunc_weighted_mean`.

**Exercise B.2: Simulate the error of a weighted mean**
We consider the weighted mean

$$u_e(t_n) \approx \theta u_e^{n+1} + (1-\theta)u_e^n .$$

Choose some specific function for $u_e(t)$ and compute the error in this approximation for a sequence of decreasing $\Delta t = t_{n+1} - t_n$ and for $\theta = 0, 0.25, 0.5, 0.75, 1$. Assuming that the error equals $C\Delta t^r$, for some constants $C$ and $r$, compute $r$ for the two smallest $\Delta t$ values for each choice of $\theta$ and compare with the truncation error (B.19)–(B.20).
Filename: `trunc_theta_avg`.

**Exercise B.3: Verify a truncation error formula**
Set up a numerical experiment as explained in Sect. B.3.5 for verifying the formulas (B.15)–(B.16).
Filename: `trunc_backward_2level`.

**Problem B.4: Truncation error of the Backward Euler scheme**
Derive the truncation error of the Backward Euler scheme for the decay ODE $u' = -au$ with constant $a$. Extend the analysis to cover the variable-coefficient case $u' = -a(t)u + b(t)$.
Filename: `trunc_decay_BE`.

**Exercise B.5: Empirical estimation of truncation errors**
Use the ideas and tools from Sect. B.3.5 to estimate the rate of the truncation error of the Backward Euler and Crank-Nicolson schemes applied to the exponential decay model $u' = -au, u(0) = I$.

*Hint* In the Backward Euler scheme, the truncation error can be estimated at mesh points $n = 1, \ldots, N$, while the truncation error must be estimated at midpoints $t_{n+\frac{1}{2}}$, $n = 0, \ldots, N-1$ for the Crank-Nicolson scheme. The `truncation_error(dt, N)` function to be supplied to the `estimate` function needs to carefully implement these details and return the right `t` array such that `t[i]` is the time point corresponding to the quantities `R[i]` and `R_a[i]`.
Filename: `trunc_decay_BNCN`.

**Exercise B.6: Correction term for a Backward Euler scheme**
Consider the model $u' = -au$, $u(0) = I$. Use the ideas of Sect. B.3.6 to add a correction term to the ODE such that the Backward Euler scheme applied to the perturbed ODE problem is of second order in $\Delta t$. Find the amplification factor.
Filename: `trunc_decay_BE_corr`.

**Problem B.7: Verify the effect of correction terms**
Make a program that solves $u' = -au$, $u(0) = I$, by the $\theta$-rule and computes convergence rates. Adjust $a$ such that it incorporates correction terms. Run the program to verify that the error from the Forward and Backward Euler schemes with perturbed $a$ is $\mathcal{O}(\Delta t^2)$, while the error arising from the Crank-Nicolson scheme with perturbed $a$ is $\mathcal{O}(\Delta t^4)$.
Filename: `trunc_decay_corr_verify`.

**Problem B.8: Truncation error of the Crank-Nicolson scheme**
The variable-coefficient ODE $u' = -a(t)u+b(t)$ can be discretized in two different ways by the Crank-Nicolson scheme, depending on whether we use averages for $a$ and $b$ or compute them at the midpoint $t_{n+\frac{1}{2}}$:

$$[D_t u = -a\overline{u}^t + b]^{n+\frac{1}{2}}, \tag{B.86}$$

$$\left[D_t u = \overline{-au + b}^t\right]^{n+\frac{1}{2}}. \tag{B.87}$$

Compute the truncation error in both cases.
Filename: `trunc_decay_CN_vc`.

**Problem B.9: Truncation error of $u' = f(u, t)$**
Consider the general nonlinear first-order scalar ODE

$$u'(t) = f(u(t), t)\,.$$

Show that the truncation error in the Forward Euler scheme,

$$[D_t^+ u = f(u, t)]^n,$$

and in the Backward Euler scheme,

$$[D_t^- u = f(u,t)]^n,$$

both are of first order, regardless of what $f$ is.

Showing the order of the truncation error in the Crank-Nicolson scheme,

$$[D_t u = f(u,t)]^{n+\frac{1}{2}},$$

is somewhat more involved: Taylor expand $u_e^n$, $u_e^{n+1}$, $f(u_e^n, t_n)$, and $f(u_e^{n+1}, t_{n+1})$ around $t_{n+\frac{1}{2}}$, and use that

$$\frac{df}{dt} = \frac{\partial f}{\partial u}u' + \frac{\partial f}{\partial t}\,.$$

Check that the derived truncation error is consistent with previous results for the case $f(u,t) = -au$.
Filename: `trunc_nonlinear_ODE`.

**Exercise B.10: Truncation error of $[D_t D_t u]^n$**
Derive the truncation error of the finite difference approximation (B.17)–(B.18) to the second-order derivative.
Filename: `trunc_d2u`.

**Exercise B.11: Investigate the impact of approximating $u'(0)$**
Section B.4.1 describes two ways of discretizing the initial condition $u'(0) = V$ for a vibration model $u'' + \omega^2 u = 0$: a centered difference $[D_{2t} u = V]^0$ or a forward difference $[D_t^+ u = V]^0$. The program `vib_undamped.py` solves $u'' + \omega^2 u = 0$ with $[D_{2t} u = 0]^0$ and features a function `convergence_rates` for computing the order of the error in the numerical solution. Modify this program such that it applies the forward difference $[D_t^+ u = 0]^0$ and report how this simpler and more convenient approximation impacts the overall convergence rate of the scheme.
Filename: `trunc_vib_ic_fw`.

**Problem B.12: Investigate the accuracy of a simplified scheme**
Consider the ODE

$$mu'' + \beta|u'|u' + s(u) = F(t)\,.$$

The term $|u'|u'$ quickly gives rise to nonlinearities and complicates the scheme. Why not simply apply a backward difference to this term such that it only involves known values? That is, we propose to solve

$$[mD_t D_t u + \beta|D_t^- u|D_t^- u + s(u) = F]^n\,.$$

Drop the absolute value for simplicity and find the truncation error of the scheme. Perform numerical experiments with the scheme and compared with the one based on centered differences. Can you illustrate the accuracy loss visually in real computations, or is the asymptotic analysis here mainly of theoretical interest?
Filename: `trunc_vib_bw_damping`.

# Software Engineering; Wave Equation Model — C

## C.1 A 1D Wave Equation Simulator

### C.1.1 Mathematical Model

Let $u_t, u_{tt}, u_x, u_{xx}$ denote derivatives of $u$ with respect to the subscript, i.e., $u_{tt}$ is a second-order time derivative and $u_x$ is a first-order space derivative. The initial-boundary value problem implemented in the `wave1D_dn_vc.py` code is

$$u_{tt} = (q(x)u_x)_x + f(x,t), \quad x \in (0,L),\ t \in (0,T] \tag{C.1}$$

$$u(x,0) = I(x), \quad x \in [0,L] \tag{C.2}$$

$$u_t(x,0) = V(t), \quad x \in [0,L] \tag{C.3}$$

$$u(0,t) = U_0(t) \quad \text{or} \quad u_x(0,t) = 0, \quad t \in (0,T] \tag{C.4}$$

$$u(L,t) = U_L(t) \quad \text{or} \quad u_x(L,t) = 0, \quad t \in (0,T]\,. \tag{C.5}$$

We allow variable wave velocity $c^2(x) = q(x)$, and Dirichlet or homogeneous Neumann conditions at the boundaries.

### C.1.2 Numerical Discretization

The PDE is discretized by second-order finite differences in time and space, with arithmetic mean for the variable coefficient

$$[D_t D_t u = D_x \overline{q}^x D_x u + f]_i^n\,. \tag{C.6}$$

The Neumann boundary conditions are discretized by

$$[D_{2x} u]_i^n = 0,$$

at a boundary point $i$. The details of how the numerical scheme is worked out are described in Sect. 2.6 and 2.7.

### C.1.3 A Solver Function

The general initial-boundary value problem (C.1)–(C.5) solved by finite difference methods can be implemented as shown in the following `solver` function (taken from the file `wave1D_dn_vc.py`). This function builds on simpler versions described in Sect. 2.3, 2.4 2.6, and 2.7. There are several quite advanced constructs that will be commented upon later. The code is lengthy, but that is because we provide a lot of flexibility with respect to input arguments, boundary conditions, and optimization (scalar versus vectorized loops).

```
def solver(
    I, V, f, c, U_0, U_L, L, dt, C, T,
    user_action=None, version='scalar',
    stability_safety_factor=1.0):
    """Solve u_tt=(c^2*u_x)_x + f on (0,L)x(0,T]."""

    # --- Compute time and space mesh ---
    Nt = int(round(T/dt))
    t = np.linspace(0, Nt*dt, Nt+1)      # Mesh points in time

    # Find max(c) using a fake mesh and adapt dx to C and dt
    if isinstance(c, (float,int)):
        c_max = c
    elif callable(c):
        c_max = max([c(x_) for x_ in np.linspace(0, L, 101)])
    dx = dt*c_max/(stability_safety_factor*C)
    Nx = int(round(L/dx))
    x = np.linspace(0, L, Nx+1)          # Mesh points in space
    # Make sure dx and dt are compatible with x and t
    dx = x[1] - x[0]
    dt = t[1] - t[0]

    # Make c(x) available as array
    if isinstance(c, (float,int)):
        c = np.zeros(x.shape) + c
    elif callable(c):
        # Call c(x) and fill array c
        c_ = np.zeros(x.shape)
        for i in range(Nx+1):
            c_[i] = c(x[i])
        c = c_

    q = c**2
    C2 = (dt/dx)**2; dt2 = dt*dt    # Help variables in the scheme

    # --- Wrap user-given f, I, V, U_0, U_L if None or 0 ---
    if f is None or f == 0:
        f = (lambda x, t: 0) if version == 'scalar' else \
            lambda x, t: np.zeros(x.shape)
    if I is None or I == 0:
        I = (lambda x: 0) if version == 'scalar' else \
            lambda x: np.zeros(x.shape)
    if V is None or V == 0:
        V = (lambda x: 0) if version == 'scalar' else \
            lambda x: np.zeros(x.shape)
```

```
    if U_0 is not None:
        if isinstance(U_0, (float,int)) and U_0 == 0:
            U_0 = lambda t: 0
    if U_L is not None:
        if isinstance(U_L, (float,int)) and U_L == 0:
            U_L = lambda t: 0

    # --- Make hash of all input data ---
    import hashlib, inspect
    data = inspect.getsource(I) + '_' + inspect.getsource(V) + \
           '_' + inspect.getsource(f) + '_' + str(c) + '_' + \
           ('None' if U_0 is None else inspect.getsource(U_0)) + \
           ('None' if U_L is None else inspect.getsource(U_L)) + \
           '_' + str(L) + str(dt) + '_' + str(C) + '_' + str(T) + \
           '_' + str(stability_safety_factor)
    hashed_input = hashlib.sha1(data).hexdigest()
    if os.path.isfile('.' + hashed_input + '_archive.npz'):
        # Simulation is already run
        return -1, hashed_input


    # --- Allocate memomry for solutions ---
    u     = np.zeros(Nx+1)   # Solution array at new time level
    u_n   = np.zeros(Nx+1)   # Solution at 1 time level back
    u_nm1 = np.zeros(Nx+1)   # Solution at 2 time levels back

    import time;  t0 = time.clock()  # CPU time measurement

    # --- Valid indices for space and time mesh ---
    Ix = range(0, Nx+1)
    It = range(0, Nt+1)

    # --- Load initial condition into u_n ---
    for i in range(0,Nx+1):
        u_n[i] = I(x[i])

    if user_action is not None:
        user_action(u_n, x, t, 0)

    # --- Special formula for the first step ---
    for i in Ix[1:-1]:
        u[i] = u_n[i] + dt*V(x[i]) + \
        0.5*C2*(0.5*(q[i] + q[i+1])*(u_n[i+1] - u_n[i]) - \
                0.5*(q[i] + q[i-1])*(u_n[i] - u_n[i-1])) + \
        0.5*dt2*f(x[i], t[0])

    i = Ix[0]
    if U_0 is None:
        # Set boundary values (x=0: i-1 -> i+1 since u[i-1]=u[i+1]
        # when du/dn = 0, on x=L: i+1 -> i-1 since u[i+1]=u[i-1])
        ip1 = i+1
        im1 = ip1  # i-1 -> i+1
        u[i] = u_n[i] + dt*V(x[i]) + \
               0.5*C2*(0.5*(q[i] + q[ip1])*(u_n[ip1] - u_n[i])  - \
                       0.5*(q[i] + q[im1])*(u_n[i] - u_n[im1])) + \
        0.5*dt2*f(x[i], t[0])
    else:
        u[i] = U_0(dt)
```

```
    i = Ix[-1]
    if U_L is None:
        im1 = i-1
        ip1 = im1  # i+1 -> i-1
        u[i] = u_n[i] + dt*V(x[i]) + \
               0.5*C2*(0.5*(q[i] + q[ip1])*(u_n[ip1] - u_n[i])  - \
                       0.5*(q[i] + q[im1])*(u_n[i] - u_n[im1])) + \
        0.5*dt2*f(x[i], t[0])
    else:
        u[i] = U_L(dt)

    if user_action is not None:
        user_action(u, x, t, 1)

    # Update data structures for next step
    #u_nm1[:] = u_n;  u_n[:] = u  # safe, but slower
    u_nm1, u_n, u = u_n, u, u_nm1


    # --- Time loop ---
    for n in It[1:-1]:
        # Update all inner points
        if version == 'scalar':
            for i in Ix[1:-1]:
                u[i] = - u_nm1[i] + 2*u_n[i] + \
                    C2*(0.5*(q[i] + q[i+1])*(u_n[i+1] - u_n[i])  - \
                        0.5*(q[i] + q[i-1])*(u_n[i] - u_n[i-1])) + \
                dt2*f(x[i], t[n])

        elif version == 'vectorized':
            u[1:-1] = - u_nm1[1:-1] + 2*u_n[1:-1] + \
            C2*(0.5*(q[1:-1] + q[2:])*(u_n[2:] - u_n[1:-1]) -
                0.5*(q[1:-1] + q[:-2])*(u_n[1:-1] - u_n[:-2])) + \
            dt2*f(x[1:-1], t[n])
        else:
            raise ValueError('version=%s' % version)

        # Insert boundary conditions
        i = Ix[0]
        if U_0 is None:
            # Set boundary values
            # x=0: i-1 -> i+1 since u[i-1]=u[i+1] when du/dn=0
            # x=L: i+1 -> i-1 since u[i+1]=u[i-1] when du/dn=0
            ip1 = i+1
            im1 = ip1
            u[i] = - u_nm1[i] + 2*u_n[i] + \
                   C2*(0.5*(q[i] + q[ip1])*(u_n[ip1] - u_n[i])  - \
                       0.5*(q[i] + q[im1])*(u_n[i] - u_n[im1])) + \
            dt2*f(x[i], t[n])
        else:
            u[i] = U_0(t[n+1])

        i = Ix[-1]
        if U_L is None:
            im1 = i-1
            ip1 = im1
```

```
            u[i] = - u_nm1[i] + 2*u_n[i] + \
                   C2*(0.5*(q[i] + q[ip1])*(u_n[ip1] - u_n[i])  - \
                       0.5*(q[i] + q[im1])*(u_n[i] - u_n[im1])) + \
            dt2*f(x[i], t[n])
        else:
            u[i] = U_L(t[n+1])

        if user_action is not None:
            if user_action(u, x, t, n+1):
                break

        # Update data structures for next step
        u_nm1, u_n, u = u_n, u, u_nm1

    cpu_time = time.clock() - t0
    return cpu_time, hashed_input
```

## C.2 Saving Large Arrays in Files

Numerical simulations produce large arrays as results and the software needs to store these arrays on disk. Several methods are available in Python. We recommend to use tailored solutions for large arrays and not standard file storage tools such as `pickle` (`cPickle` for speed in Python version 2) and `shelve`, because the tailored solutions have been optimized for array data and are hence much faster than the standard tools.

### C.2.1 Using `savez` to Store Arrays in Files

**Storing individual arrays** The `numpy.storez` function can store a set of arrays to a named file in a zip archive. An associated function `numpy.load` can be used to read the file later. Basically, we call `numpy.storez(filename, **kwargs)`, where `kwargs` is a dictionary containing array names as keys and the corresponding array objects as values. Very often, the solution at a time point is given a natural name where the name of the variable and the time level counter are combined, e.g., `u11` or `v39`. Suppose `n` is the time level counter and we have two solution arrays, `u` and `v`, that we want to save to a zip archive. The appropriate code is

```
import numpy as np
u_name = 'u%04d' % n   # array name
v_name = 'v%04d' % n   # array name
kwargs = {u_name: u, v_name: v}   # keyword args for savez
fname = '.mydata%04d.dat' % n
np.savez(fname, **kwargs)
if n == 0:            # store x once
    np.savez('.mydata_x.dat', x=x)
```

Since the name of the array must be given as a keyword argument to `savez`, and the name must be constructed as shown, it becomes a little tricky to do the call, but with

a dictionary `kwargs` and `**kwargs`, which sends each key-value pair as individual keyword arguments, the task gets accomplished.

**Merging zip archives** Each separate call to `np.savez` creates a new file (zip archive) with extension `.npz`. It is very convenient to collect all results in one archive instead. This can be done by merging all the individual `.npz` files into a single zip archive:

```
def merge_zip_archives(individual_archives, archive_name):
    """
    Merge individual zip archives made with numpy.savez into
    one archive with name archive_name.
    The individual archives can be given as a list of names
    or as a Unix wild chard filename expression for glob.glob.
    The result of this function is that all the individual
    archives are deleted and the new single archive made.
    """
    import zipfile
    archive = zipfile.ZipFile(
        archive_name, 'w', zipfile.ZIP_DEFLATED,
        allowZip64=True)
    if isinstance(individual_archives, (list,tuple)):
        filenames = individual_archives
    elif isinstance(individual_archives, str):
        filenames = glob.glob(individual_archives)

    # Open each archive and write to the common archive
    for filename in filenames:
        f = zipfile.ZipFile(filename,  'r',
                            zipfile.ZIP_DEFLATED)
        for name in f.namelist():
            data = f.open(name, 'r')
            # Save under name without .npy
            archive.writestr(name[:-4], data.read())
        f.close()
        os.remove(filename)
    archive.close()
```

Here we remark that `savez` automatically adds the `.npz` extension to the names of the arrays we store. We do not want this extension in the final archive.

**Reading arrays from zip archives** Archives created by `savez` or the merged archive we describe above with name of the form `myarchive.npz`, can be conveniently read by the `numpy.load` function:

```
import numpy as np
array_names = np.load(‘myarchive.npz‘)
for array_name in array_names:
    # array_names[array_name] is the array itself
    # e.g. plot(array_names['t'], array_names[array_name])
```

### C.2.2 Using joblib to Store Arrays in Files

The Python package joblib has nice functionality for efficient storage of arrays on disk. The following class applies this functionality so that one can save an array, or in fact any Python data structure (e.g., a dictionary of arrays), to disk under a certain name. Later, we can retrieve the object by use of its name. The name of the directory under which the arrays are stored by joblib can be given by the user.

```
class Storage(object):
    """
    Store large data structures (e.g. numpy arrays) efficiently
    using joblib.

    Use:

    >>> from Storage import Storage
    >>> storage = Storage(cachedir='tmp_u01', verbose=1)
    >>> import numpy as np
    >>> a = np.linspace(0, 1, 100000) # large array
    >>> b = np.linspace(0, 1, 100000) # large array
    >>> storage.save('a', a)
    >>> storage.save('b', b)
    >>> # later
    >>> a = storage.retrieve('a')
    >>> b = storage.retrieve('b')
    """
    def __init__(self, cachedir='tmp', verbose=1):
        """
        Parameters
        ----------
        cachedir: str
             Name of directory where objects are stored in files.
        verbose: bool, int
             Let joblib and this class speak when storing files
             to disk.
        """
        import joblib
        self.memory = joblib.Memory(cachedir=cachedir,
                                    verbose=verbose)
        self.verbose = verbose
        self.retrieve = self.memory.cache(
            self.retrieve, ignore=['data'])
        self.save = self.retrieve

    def retrieve(self, name, data=None):
        if self.verbose > 0:
            print 'joblib save of', name
        return data
```

The retrive and save functions, which do the work, seem quite magic. The idea is that joblib looks at the name parameter and saves the return value data to disk if the name parameter has not been used in a previous call. Otherwise, if name is already registered, joblib fetches the data object from file and returns it (this is an example of a memoize function, see Section 2.1.4 in [11] for a brief explanation]).

### C.2.3 Using a Hash to Create a File or Directory Name

Array storage techniques like those outlined in Sect. C.2.2 and C.2.1 demand the user to assign a name for the file(s) or directory where the solution is to be stored. Ideally, this name should reflect parameters in the problem such that one can recognize an already run simulation. One technique is to make a hash string out of the input data. A hash string is a 40-character long hexadecimal string that uniquely reflects another potentially much longer string. (You may be used to hash strings from the Git version control system: every committed version of the files in Git is recognized by a hash string.)

Suppose you have some input data in the form of functions, `numpy` arrays, and other objects. To turn these input data into a string, we may grab the source code of the functions, use a very efficient hash method for potentially large arrays, and simply convert all other objects via `str` to a string representation. The final string, merging all input data, is then converted to an SHA1 hash string such that we represent the input with a 40-character long string.

```
def myfunction(func1, func2, array1, array2, obj1, obj2):
    # Convert arguments to hash
    import inspect, joblib, hashlib
    data = (inspect.getsource(func1),
            inspect.getsource(func2),
        joblib.hash(array1),
        joblib.hash(array2),
        str(obj1),
        str(obj2))
    hash_input = hashlib.sha1(data).hexdigest()
```

It is wise to use `joblib.hash` and not try to do a `str(array1)`, since that string can be *very* long, and `joblib.hash` is more efficient than `hashlib` when turning these data into a hash.

**Remark: turning function objects into their source code is unreliable!**

The idea of turning a function object into a string via its source code may look smart, but is not a completely reliable solution. Suppose we have some function

```
x0 = 0.1
f = lambda x: 0 if x <= x0 else 1
```

The source code will be `f = lambda x: 0 if x <= x0 else 1`, so if the calling code changes the value of `x0` (which `f` remembers - it is a closure), the source remains unchanged, the hash is the same, and the change in input data is unnoticed. Consequently, the technique above must be used with care. The user can always just remove the stored files in disk and thereby force a recomputation (provided the software applies a hash to test if a zip archive or `joblib` subdirectory exists, and if so, avoids recomputation).

## C.3 Software for the 1D Wave Equation

We use `numpy.storez` to store the solution at each time level on disk. Such actions must be taken care of outside the `solver` function, more precisely in the `user_action` function that is called at every time level.

We have, in the `wave1D_dn_vc.py` code, implemented the `user_action` callback function as a class `PlotAndStoreSolution` with a `__call__(self, x, t, t, n)` method for the `user_action` function. Basically, `__call__` stores and plots the solution. The storage makes use of the `numpy.savez` function for saving a set of arrays to a zip archive. Here, in this callback function, we want to save one array, `u`. Since there will be many such arrays, we introduce the array names `'u%04d' % n` and closely related filenames. The usage of `numpy.savez` in `__call__` goes like this:

```
from numpy import savez
name = 'u%04d' % n   # array name
kwargs = {name: u}   # keyword args for savez
fname = '.' + self.filename + '_' + name + '.dat'
self.t.append(t[n])  # store corresponding time value
savez(fname, **kwargs)
if n == 0:           # store x once
    savez('.' + self.filename + '_x.dat', x=x)
```

For example, if `n` is 10 and `self.filename` is `tmp`, the above call to `savez` becomes `savez('.tmp_u0010.dat', u0010=u)`. The actual filename becomes `.tmp_u0010.dat.npz`. The actual array name becomes `u0010.npy`.

Each `savez` call results in a file, so after the simulation we have one file per time level. Each file produced by `savez` is a zip archive. It makes sense to merge all the files into one. This is done in the `close_file` method in the `PlotAndStoreSolution` class. The code goes as follows.

```
class PlotAndStoreSolution:
    ...
    def close_file(self, hashed_input):
        """
        Merge all files from savez calls into one archive.
        hashed_input is a string reflecting input data
        for this simulation (made by solver).
        """
        if self.filename is not None:
            # Save all the time points where solutions are saved
            savez('.' + self.filename + '_t.dat',
                  t=array(self.t, dtype=float))
            # Merge all savez files to one zip archive
            archive_name = '.' + hashed_input + '_archive.npz'
            filenames = glob.glob('.' + self.filename + '*.dat.npz')
            merge_zip_archives(filenames, archive_name)
```

We use various `ZipFile` functionality to extract the content of the individual files (each with name `filename`) and write it to the merged archive (`archive`). There is only one array in each individual file (`filename`) so strictly speaking, there is

no need for the loop `for name in f.namelist()` (as `f.namelist()` returns a list of length 1). However, in other applications where we compute more arrays at each time level, `savez` will store all these and then there is need for iterating over `f.namelist()`.

Instead of merging the archives written by `savez` we could make an alternative implementation that writes all our arrays into one archive. This is the subject of Exercise C.2.

### C.3.1 Making Hash Strings from Input Data

The `hashed_input` argument, used to name the resulting archive file with all solutions, is supposed to be a hash reflecting all import parameters in the problem such that this simulation has a unique name. The `hashed_input` string is made in the `solver` function, using the `hashlib` and `inspect` modules, based on the arguments to `solver`:

```
# Make hash of all input data
import hashlib, inspect
data = inspect.getsource(I) + '_' + inspect.getsource(V) + \
       '_' + inspect.getsource(f) + '_' + str(c) + '_' + \
       ('None' if U_0 is None else inspect.getsource(U_0)) + \
       ('None' if U_L is None else inspect.getsource(U_L)) + \
       '_' + str(L) + str(dt) + '_' + str(C) + '_' + str(T) + \
       '_' + str(stability_safety_factor)
hashed_input = hashlib.sha1(data).hexdigest()
```

To get the source code of a function `f` as a string, we use `inspect.getsource(f)`. All input, functions as well as variables, is then merged to a string `data`, and then `hashlib.sha1` makes a unique, much shorter (40 characters long), fixed-length string out of `data` that we can use in the archive filename.

**Remark**

Note that the construction of the `data` string is not fool proof: if, e.g., `I` is a formula with parameters and the parameters change, the source code is still the same and `data` and hence the hash remains unaltered. The implementation must therefore be used with care!

### C.3.2 Avoiding Rerunning Previously Run Cases

If the archive file whose name is based on `hashed_input` already exists, the simulation with the current set of parameters has been done before and one can avoid redoing the work. The `solver` function returns the CPU time and `hashed_input`, and a negative CPU time means that no simulation was run. In that case we should not call the `close_file` method above (otherwise we overwrite the archive with just the `self.t` array). The typical usage goes like

```
action = PlotAndStoreSolution(...)
dt = (L/Nx)/C  # choose the stability limit with given Nx
cpu, hashed_input = solver(
    I=lambda x: ...,
    V=0, f=0, c=1, U_0=lambda t: 0, U_L=None, L=1,
    dt=dt, C=C, T=T,
    user_action=action, version='vectorized',
    stability_safety_factor=1)
action.make_movie_file()
if cpu > 0:  # did we generate new data?
    action.close_file(hashed_input)
```

### C.3.3 Verification

**Vanishing approximation error** Exact solutions of the numerical equations are always attractive for verification purposes since the software should reproduce such solutions to machine precision. With Dirichlet boundary conditions we can construct a function that is linear in $t$ and quadratic in $x$ that is also an exact solution of the scheme, while with Neumann conditions we are left with testing just a constant solution (see comments in Sect. 2.6.5).

**Convergence rates** A more general method for verification is to check the convergence rates. We must introduce one discretization parameter $h$ and assume an error model $E = Ch^r$, where $C$ and $r$ are constants to be determine (i.e., $r$ is the rate that we are interested in). Given two experiments with different resolutions $h_i$ and $h_i-1$, we can estimate $r$ by

$$r = \frac{\ln(E_i/E_{i-1})}{\ln(h_i/h_{i-1})},$$

where $E_i$ is the error corresponding to $h_i$ and $E_{i-1}$ corresponds to $h_{i-1}$. Section 2.2.2 explains the details of this type of verification and how we introduce the single discretization parameter $h = \Delta t = \hat{c}\Delta t$, for some constant $\hat{c}$. To compute the error, we had to rely on a global variable in the user action function. Below is an implementation where we have a more elegant solution in terms of a class: the `error` variable is not a class attribute and there is no need for a global error (which is always considered an advantage).

```
def convergence_rates(
    u_exact,
    I, V, f, c, U_0, U_L, L,
    dt0, num_meshes,
    C, T, version='scalar',
    stability_safety_factor=1.0):
    """
    Half the time step and estimate convergence rates for
    for num_meshes simulations.
    """
    class ComputeError:
        def __init__(self, norm_type):
            self.error = 0
```

```
    def __call__(self, u, x, t, n):
        """Store norm of the error in self.E."""
        error = np.abs(u - u_exact(x, t[n])).max()
        self.error = max(self.error, error)

E = []
h = []  # dt, solver adjusts dx such that C=dt*c/dx
dt = dt0
for i in range(num_meshes):
    error_calculator = ComputeError('Linf')
    solver(I, V, f, c, U_0, U_L, L, dt, C, T,
           user_action=error_calculator,
           version='scalar',
           stability_safety_factor=1.0)
    E.append(error_calculator.error)
    h.append(dt)
    dt /= 2  # halve the time step for next simulation
print 'E:', E
print 'h:', h
r = [np.log(E[i]/E[i-1])/np.log(h[i]/h[i-1])
     for i in range(1,num_meshes)]
return r
```

The returned sequence `r` should converge to 2 since the error analysis in Sect. 2.10 predicts various error measures to behave like $\mathcal{O}(\Delta t^2) + \mathcal{O}(\Delta x^2)$. We can easily run the case with standing waves and the analytical solution $u(x,t) = \cos(\frac{2\pi}{L}t)\sin(\frac{2\pi}{L}x)$. The call will be very similar to the one provided in the `test_convrate_sincos` function in Sect. 2.3.4, see the file `wave1D_dn_vc.py` for details.

## C.4 Programming the Solver with Classes

Many who know about class programming prefer to organize their software in terms of classes. This gives a richer application programming interface (API) since a function solver must have all its input data in terms of arguments, while a class-based solver naturally has a mix of method arguments and user-supplied methods. (Well, to be more precise, our solvers have demanded `user_action` to be a function provided by the user, so it is possible to mix variables and functions in the input also with a solver function.)

We will next illustrate how some of the functionality in `wave1D_dn_vc.py` may be implemented by using classes. Focusing on class implementation aspects, we restrict the example case to a simpler wave with constant wave speed $c$. Applying the method of manufactured solutions, we test whether the class based implementation is able to compute the known exact solution within machine precision.

We will create a class `Problem` to hold the physical parameters of the problem and a class `Solver` to hold the numerical solution parameters besides the solver function itself. As the number of parameters increases, so does the amount of repetitive code. We therefore take the opportunity to illustrate how this may be counteracted by introducing a super class `Parameters` that allows code to be parameterized. In addition, it is convenient to collect the arrays that describe the mesh

in a special Mesh class and make a class Function for a mesh function (mesh point values and its mesh). All the following code is found in wave1D_oo.py.

### C.4.1 Class Parameters

The classes Problem and Solver both inherit class Parameters, which handles reading of parameters from the command line and has methods for setting and getting parameter values. Since processing dictionaries is easier than processing a collection of individual attributes, the class Parameters requires each class Problem and Solver to represent their parameters by dictionaries, one compulsory and two optional ones. The compulsory dictionary, self.prm, contains all parameters, while a second and optional dictionary, self.type, holds the associated object types, and a third and optional dictionary, self.help, stores help strings. The Parameters class may be implemented as follows:

```
class Parameters(object):
    def __init__(self):
        """
        Subclasses must initialize self.prm with
        parameters and default values, self.type with
        the corresponding types, and self.help with
        the corresponding descriptions of parameters.
        self.type and self.help are optional, but
        self.prms must be complete and contain all parameters.
        """
        pass

    def ok(self):
        """Check if attr. prm, type, and help are defined."""
        if hasattr(self, 'prm') and \
           isinstance(self.prm, dict) and \
           hasattr(self, 'type') and \
           isinstance(self.type, dict) and \
           hasattr(self, 'help') and \
           isinstance(self.help, dict):
            return True
        else:
            raise ValueError(
                'The constructor in class %s does not '\
                'initialize the\ndictionaries '\
                'self.prm, self.type, self.help!' %
                self.__class__.__name__)

    def _illegal_parameter(self, name):
        """Raise exception about illegal parameter name."""
        raise ValueError(
            'parameter "%s" is not registered.\nLegal '\
            'parameters are\n%s' %
            (name, ' '.join(list(self.prm.keys()))))
```

```
    def set(self, **parameters):
        """Set one or more parameters."""
        for name in parameters:
            if name in self.prm:
                self.prm[name] = parameters[name]
            else:
                self._illegal_parameter(name)

    def get(self, name):
        """Get one or more parameter values."""
        if isinstance(name, (list,tuple)):   # get many?
            for n in name:
                if n not in self.prm:
                    self._illegal_parameter(name)
            return [self.prm[n] for n in name]
        else:
            if name not in self.prm:
                self._illegal_parameter(name)
            return self.prm[name]

    def __getitem__(self, name):
        """Allow obj[name] indexing to look up a parameter."""
        return self.get(name)

    def __setitem__(self, name, value):
        """
        Allow obj[name] = value syntax to assign a parameter's value.
        """
        return self.set(name=value)


    def define_command_line_options(self, parser=None):
        self.ok()
        if parser is None:
            import argparse
            parser = argparse.ArgumentParser()

        for name in self.prm:
            tp = self.type[name] if name in self.type else str
            help = self.help[name] if name in self.help else None
            parser.add_argument(
                '--' + name, default=self.get(name), metavar=name,
                type=tp, help=help)

        return parser

    def init_from_command_line(self, args):
        for name in self.prm:
            self.prm[name] = getattr(args, name)
```

### C.4.2 Class Problem

Inheriting the `Parameters` class, our class `Problem` is defined as:

```
class Problem(Parameters):
    """
    Physical parameters for the wave equation
    u_tt = (c**2*u_x)_x + f(x,t) with t in [0,T] and
    x in (0,L). The problem definition is implied by
    the method of manufactured solution, choosing
    u(x,t)=x(L-x)(1+t/2) as our solution. This solution
    should be exactly reproduced when c is const.
    """

    def __init__(self):
        self.prm  = dict(L=2.5, c=1.5, T=18)
        self.type = dict(L=float, c=float, T=float)
        self.help = dict(L='1D domain',
                         c='coefficient (wave velocity) in PDE',
                         T='end time of simulation')
    def u_exact(self, x, t):
        L = self['L']
        return x*(L-x)*(1+0.5*t)
    def I(self, x):
        return self.u_exact(x, 0)
    def V(self, x):
        return 0.5*self.u_exact(x, 0)
    def f(self, x, t):
        c = self['c']
        return 2*(1+0.5*t)*c**2
    def U_0(self, t):
        return self.u_exact(0, t)
    U_L = None
```

### C.4.3 Class Mesh

The `Mesh` class can be made valid for a space-time mesh in any number of space dimensions. To make the class versatile, the constructor accepts either a tuple/list of number of cells in each spatial dimension or a tuple/list of cell spacings. In addition, we need the size of the hypercube mesh as a tuple/list of 2-tuples with lower and upper limits of the mesh coordinates in each direction. For 1D meshes it is more natural to just write the number of cells or the cell size and not wrap it in a list. We also need the time interval from `t0` to `T`. Giving no spatial discretization information implies a time mesh only, and vice versa. The `Mesh` class with documentation and a doc test should now be self-explanatory:

```
import numpy as np

class Mesh(object):
    """
    Holds data structures for a uniform mesh on a hypercube in
    space, plus a uniform mesh in time.

    ======== ==================================================
    Argument                Explanation
    ======== ==================================================
    L        List of 2-lists of min and max coordinates
             in each spatial direction.
    T        Final time in time mesh.
    Nt       Number of cells in time mesh.
    dt       Time step. Either Nt or dt must be given.
    N        List of number of cells in the spatial directions.
    d        List of cell sizes in the spatial directions.
             Either N or d must be given.
    ======== ==================================================

    Users can access all the parameters mentioned above, plus
    ``x[i]`` and ``t`` for the coordinates in direction ``i``
    and the time coordinates, respectively.

    Examples:

    >>> from UniformFDMesh import Mesh
    >>>
    >>> # Simple space mesh
    >>> m = Mesh(L=[0,1], N=4)
    >>> print m.dump()
    space: [0,1] N=4 d=0.25
    >>>
    >>> # Simple time mesh
    >>> m = Mesh(T=4, dt=0.5)
    >>> print m.dump()
    time: [0,4] Nt=8 dt=0.5
    >>>
    >>> # 2D space mesh
    >>> m = Mesh(L=[[0,1], [-1,1]], d=[0.5, 1])
    >>> print m.dump()
    space: [0,1]x[-1,1] N=2x2 d=0.5,1
    >>>
    >>> # 2D space mesh and time mesh
    >>> m = Mesh(L=[[0,1], [-1,1]], d=[0.5, 1], Nt=10, T=3)
    >>> print m.dump()
    space: [0,1]x[-1,1] N=2x2 d=0.5,1 time: [0,3] Nt=10 dt=0.3

    """
    def __init__(self,
                 L=None, T=None, t0=0,
                 N=None, d=None,
                 Nt=None, dt=None):
        if N is None and d is None:
            # No spatial mesh
            if Nt is None and dt is None:
                raise ValueError(
                'Mesh constructor: either Nt or dt must be given')
            if T is None:
                raise ValueError(
                'Mesh constructor: T must be given')
```

```
        if Nt is None and dt is None:
            if N is None and d is None:
                raise ValueError(
                'Mesh constructor: either N or d must be given')
            if L is None:
                raise ValueError(
                'Mesh constructor: L must be given')

        # Allow 1D interface without nested lists with one element
        if L is not None and isinstance(L[0], (float,int)):
            # Only an interval was given
            L = [L]
        if N is not None and isinstance(N, (float,int)):
            N = [N]
        if d is not None and isinstance(d, (float,int)):
            d = [d]

        # Set all attributes to None
        self.x = None
        self.t = None
        self.Nt = None
        self.dt = None
        self.N = None
        self.d = None
        self.t0 = t0

        if N is None and d is not None and L is not None:
            self.L = L
            if len(d) != len(L):
                raise ValueError(
                    'd has different size (no of space dim.) from '
                    'L: %d vs %d', len(d), len(L))
            self.d = d
            self.N = [int(round(float(self.L[i][1] -
                                      self.L[i][0])/d[i]))
                      for i in range(len(d))]
        if d is None and N is not None and L is not None:
            self.L = L
            if len(N) != len(L):
                raise ValueError(
                    'N has different size (no of space dim.) from '
                    'L: %d vs %d', len(N), len(L))
            self.N = N
            self.d = [float(self.L[i][1] - self.L[i][0])/N[i]
                      for i in range(len(N))]

        if Nt is None and dt is not None and T is not None:
            self.T = T
            self.dt = dt
            self.Nt = int(round(T/dt))
        if dt is None and Nt is not None and T is not None:
            self.T = T
            self.Nt = Nt
            self.dt = T/float(Nt)

        if self.N is not None:
            self.x = [np.linspace(
                      self.L[i][0], self.L[i][1], self.N[i]+1)
                      for i in range(len(self.L))]
        if Nt is not None:
            self.t = np.linspace(self.t0, self.T, self.Nt+1)
```

```
    def get_num_space_dim(self):
        return len(self.d) if self.d is not None else 0

    def has_space(self):
        return self.d is not None

    def has_time(self):
        return self.dt is not None

    def dump(self):
        s = ''
        if self.has_space():
            s += 'space: ' + \
                 'x'.join(['[%g,%g]' % (self.L[i][0], self.L[i][1])
                           for i in range(len(self.L))]) + ' N='
            s += 'x'.join([str(Ni) for Ni in self.N]) + ' d='
            s += ','.join([str(di) for di in self.d])
        if self.has_space() and self.has_time():
            s += ' '
        if self.has_time():
            s += 'time: ' + '[%g,%g]' % (self.t0, self.T) + \
                 ' Nt=%g' % self.Nt + ' dt=%g' % self.dt
        return s
```

**We rely on attribute access – not get/set functions!**

Java programmers, in particular, are used to get/set functions in classes to access internal data. In Python, we usually apply direct access of the attribute, such as `m.N[i]` if `m` is a `Mesh` object. A widely used convention is to do this as long as access to an attribute does not require additional code. In that case, one applies a property construction. The original interface remains the same after a property is introduced (in contrast to Java), so user will not notice a change to properties.

The only argument against direct attribute access in class `Mesh` is that the attributes are read-only so we could avoid offering a set function. Instead, we rely on the user that she does not assign new values to the attributes.

### C.4.4 Class Function

A class `Function` is handy to hold a mesh and corresponding values for a scalar or vector function over the mesh. Since we may have a time or space mesh, or a combined time and space mesh, with one or more components in the function, some if tests are needed for allocating the right array sizes. To help the user, an `indices` attribute with the name of the indices in the final array `u` for the function values is made. The examples in the doc string should explain the functionality.

```
class Function(object):
    """
    A scalar or vector function over a mesh (of class Mesh).

    ========== ===================================================
    Argument                       Explanation
    ========== ===================================================
    mesh       Class Mesh object: spatial and/or temporal mesh.
    num_comp   Number of components in function (1 for scalar).
    space_only True if the function is defined on the space mesh
               only (to save space). False if function has values
               in space and time.
    ========== ===================================================

    The indexing of ``u``, which holds the mesh point values of the
    function, depends on whether we have a space and/or time mesh.

    Examples:

    >>> from UniformFDMesh import Mesh, Function
    >>>
    >>> # Simple space mesh
    >>> m = Mesh(L=[0,1], N=4)
    >>> print m.dump()
    space: [0,1] N=4 d=0.25
    >>> f = Function(m)
    >>> f.indices
    ['x0']
    >>> f.u.shape
    (5,)
    >>> f.u[4]  # space point 4
    0.0
    >>>
    >>> # Simple time mesh for two components
    >>> m = Mesh(T=4, dt=0.5)
    >>> print m.dump()
    time: [0,4] Nt=8 dt=0.5
    >>> f = Function(m, num_comp=2)
    >>> f.indices
    ['time', 'component']
    >>> f.u.shape
    (9, 2)
    >>> f.u[3,1]  # time point 3, comp=1 (2nd comp.)
    0.0
    >>>
    >>> # 2D space mesh
    >>> m = Mesh(L=[[0,1], [-1,1]], d=[0.5, 1])
    >>> print m.dump()
    space: [0,1]x[-1,1] N=2x2 d=0.5,1
    >>> f = Function(m)
    >>> f.indices
    ['x0', 'x1']
    >>> f.u.shape
    (3, 3)
    >>> f.u[1,2]  # space point (1,2)
    0.0
```

```
    >>>
    >>> # 2D space mesh and time mesh
    >>> m = Mesh(L=[[0,1],[-1,1]], d=[0.5,1], Nt=10, T=3)
    >>> print m.dump()
    space: [0,1]x[-1,1] N=2x2 d=0.5,1 time: [0,3] Nt=10 dt=0.3
    >>> f = Function(m, num_comp=2, space_only=False)
    >>> f.indices
    ['time', 'x0', 'x1', 'component']
    >>> f.u.shape
    (11, 3, 3, 2)
    >>> f.u[2,1,2,0]  # time step 2, space point (1,2), comp=0
    0.0
    >>> # Function with space data only
    >>> f = Function(m, num_comp=1, space_only=True)
    >>> f.indices
    ['x0', 'x1']
    >>> f.u.shape
    (3, 3)
    >>> f.u[1,2]  # space point (1,2)
    0.0
    """

    def __init__(self, mesh, num_comp=1, space_only=True):
        self.mesh = mesh
        self.num_comp = num_comp
        self.indices = []

        # Create array(s) to store mesh point values
        if (self.mesh.has_space() and not self.mesh.has_time()) or \
           (self.mesh.has_space() and self.mesh.has_time() and \
            space_only):
            # Space mesh only
            if num_comp == 1:
                self.u = np.zeros(
                    [self.mesh.N[i] + 1
                     for i in range(len(self.mesh.N))])
                self.indices = [
                    'x'+str(i) for i in range(len(self.mesh.N))]
            else:
                self.u = np.zeros(
                    [self.mesh.N[i] + 1
                     for i in range(len(self.mesh.N))] +
                    [num_comp])
                self.indices = [
                    'x'+str(i)
                    for i in range(len(self.mesh.N))] +\
                    ['component']
        if not self.mesh.has_space() and self.mesh.has_time():
            # Time mesh only
            if num_comp == 1:
                self.u = np.zeros(self.mesh.Nt+1)
                self.indices = ['time']
            else:
                # Need num_comp entries per time step
                self.u = np.zeros((self.mesh.Nt+1, num_comp))
                self.indices = ['time', 'component']
```

```
        if self.mesh.has_space() and self.mesh.has_time() \
            and not space_only:
            # Space-time mesh
            size = [self.mesh.Nt+1] + \
                   [self.mesh.N[i]+1
                    for i in range(len(self.mesh.N))]
            if num_comp > 1:
                self.indices = ['time'] + \
                               ['x'+str(i)
                                for i in range(len(self.mesh.N))] +\
                               ['component']
                size += [num_comp]
            else:
                self.indices = ['time'] + ['x'+str(i)
                                for i in range(len(self.mesh.N))]
            self.u = np.zeros(size)
```

### C.4.5 Class Solver

With the Mesh and Function classes in place, we can rewrite the solver function, but we make it a method in class Solver:

```
class Solver(Parameters):
    """
    Numerical parameters for solving the wave equation
    u_tt = (c**2*u_x)_x + f(x,t) with t in [0,T] and
    x in (0,L). The problem definition is implied by
    the method of manufactured solution, choosing
    u(x,t)=x(L-x)(1+t/2) as our solution. This solution
    should be exactly reproduced, provided c is const.
    We simulate in [0, L/2] and apply a symmetry condition
    at the end x=L/2.
    """

    def __init__(self, problem):
        self.problem = problem
        self.prm  = dict(C = 0.75, Nx=3, stability_safety_factor=1.0)
        self.type = dict(C=float, Nx=int, stability_safety_factor=float)
        self.help = dict(C='Courant number',
                         Nx='No of spatial mesh points',
                         stability_safety_factor='stability factor')

        from UniformFDMesh import Mesh, Function
        # introduce some local help variables to ease reading
        L_end = self.problem['L']
        dx = (L_end/2)/float(self['Nx'])
        t_interval = self.problem['T']
        dt = dx*self['stability_safety_factor']*self['C']/ \
                                float(self.problem['c'])
        self.m = Mesh(L=[0,L_end/2],
                      d=[dx],
                      Nt = int(round(t_interval/float(dt))),
                      T=t_interval)
        # The mesh function f will, after solving, contain
        # the solution for the whole domain and all time steps.
        self.f = Function(self.m, num_comp=1, space_only=False)
```

```
def solve(self, user_action=None, version='scalar'):
    # ...use local variables to ease reading
    L, c, T = self.problem['L c T'.split()]
    L = L/2     # compute with half the domain only (symmetry)
    C, Nx, stability_safety_factor = self[
                         'C Nx stability_safety_factor'.split()]
    dx = self.m.d[0]
    I = self.problem.I
    V = self.problem.V
    f = self.problem.f
    U_0 = self.problem.U_0
    U_L = self.problem.U_L
    Nt = self.m.Nt
    t = np.linspace(0, T, Nt+1)      # Mesh points in time
    x = np.linspace(0, L, Nx+1)      # Mesh points in space

    # Make sure dx and dt are compatible with x and t
    dx = x[1] - x[0]
    dt = t[1] - t[0]


    # Treat c(x) as array
    if isinstance(c, (float,int)):
        c = np.zeros(x.shape) + c
    elif callable(c):
        # Call c(x) and fill array c
        c_ = np.zeros(x.shape)
        for i in range(Nx+1):
            c_[i] = c(x[i])
        c = c_

    q = c**2
    C2 = (dt/dx)**2; dt2 = dt*dt    # Help variables in the scheme

    # Wrap user-given f, I, V, U_0, U_L if None or 0
    if f is None or f == 0:
        f = (lambda x, t: 0) if version == 'scalar' else \
            lambda x, t: np.zeros(x.shape)
    if I is None or I == 0:
        I = (lambda x: 0) if version == 'scalar' else \
            lambda x: np.zeros(x.shape)
    if V is None or V == 0:
        V = (lambda x: 0) if version == 'scalar' else \
            lambda x: np.zeros(x.shape)
    if U_0 is not None:
        if isinstance(U_0, (float,int)) and U_0 == 0:
            U_0 = lambda t: 0
    if U_L is not None:
        if isinstance(U_L, (float,int)) and U_L == 0:
            U_L = lambda t: 0

    # Make hash of all input data
    import hashlib, inspect
    data = inspect.getsource(I) + '_' + inspect.getsource(V) + \
           '_' + inspect.getsource(f) + '_' + str(c) + '_' + \
           ('None' if U_0 is None else inspect.getsource(U_0)) + \
           ('None' if U_L is None else inspect.getsource(U_L)) + \
           '_' + str(L) + str(dt) + '_' + str(C) + '_' + str(T) + \
           '_' + str(stability_safety_factor)
```

```
        hashed_input = hashlib.sha1(data).hexdigest()
        if os.path.isfile('.' + hashed_input + '_archive.npz'):
            # Simulation is already run
            return -1, hashed_input

        # use local variables to make code closer to mathematical
        # notation in computational scheme
        u_1 = self.f.u[0,:]
        u   = self.f.u[1,:]

        import time;  t0 = time.clock()  # CPU time measurement

        Ix = range(0, Nx+1)
        It = range(0, Nt+1)

        # Load initial condition into u_1
        for i in range(0,Nx+1):
            u_1[i] = I(x[i])

        if user_action is not None:
            user_action(u_1, x, t, 0)


        # Special formula for the first step
        for i in Ix[1:-1]:
            u[i] = u_1[i] + dt*V(x[i]) + \
            0.5*C2*(0.5*(q[i] + q[i+1])*(u_1[i+1] - u_1[i]) - \
                    0.5*(q[i] + q[i-1])*(u_1[i] - u_1[i-1])) + \
            0.5*dt2*f(x[i], t[0])

        i = Ix[0]
        if U_0 is None:
            # Set boundary values (x=0: i-1 -> i+1 since u[i-1]=u[i+1]
            # when du/dn = 0, on x=L: i+1 -> i-1 since u[i+1]=u[i-1])
            ip1 = i+1
            im1 = ip1  # i-1 -> i+1
            u[i] = u_1[i] + dt*V(x[i]) + \
                   0.5*C2*(0.5*(q[i] + q[ip1])*(u_1[ip1] - u_1[i])  - \
                           0.5*(q[i] + q[im1])*(u_1[i] - u_1[im1])) + \
            0.5*dt2*f(x[i], t[0])
        else:
            u[i] = U_0(dt)

        i = Ix[-1]
        if U_L is None:
            im1 = i-1
            ip1 = im1  # i+1 -> i-1
            u[i] = u_1[i] + dt*V(x[i]) + \
                   0.5*C2*(0.5*(q[i] + q[ip1])*(u_1[ip1] - u_1[i])  - \
                           0.5*(q[i] + q[im1])*(u_1[i] - u_1[im1])) + \
            0.5*dt2*f(x[i], t[0])
        else:
            u[i] = U_L(dt)

        if user_action is not None:
            user_action(u, x, t, 1)
```

```
        for n in It[1:-1]:
            # u corresponds to u^{n+1} in the mathematical scheme
            u_2 = self.f.u[n-1,:]
            u_1 = self.f.u[n,:]
            u   = self.f.u[n+1,:]

            # Update all inner points
            if version == 'scalar':
                for i in Ix[1:-1]:
                    u[i] = - u_2[i] + 2*u_1[i] + \
                        C2*(0.5*(q[i] + q[i+1])*(u_1[i+1] - u_1[i])  - \
                            0.5*(q[i] + q[i-1])*(u_1[i] - u_1[i-1])) + \
                    dt2*f(x[i], t[n])

            elif version == 'vectorized':
                u[1:-1] = - u_2[1:-1] + 2*u_1[1:-1] + \
                C2*(0.5*(q[1:-1] + q[2:])*(u_1[2:] - u_1[1:-1]) -
                    0.5*(q[1:-1] + q[:-2])*(u_1[1:-1] - u_1[:-2])) + \
                dt2*f(x[1:-1], t[n])
            else:
                raise ValueError('version=%s' % version)


            # Insert boundary conditions
            i = Ix[0]
            if U_0 is None:
                # Set boundary values
                # x=0: i-1 -> i+1 since u[i-1]=u[i+1] when du/dn=0
                # x=L: i+1 -> i-1 since u[i+1]=u[i-1] when du/dn=0
                ip1 = i+1
                im1 = ip1
                u[i] = - u_2[i] + 2*u_1[i] + \
                       C2*(0.5*(q[i] + q[ip1])*(u_1[ip1] - u_1[i])  - \
                           0.5*(q[i] + q[im1])*(u_1[i] - u_1[im1])) + \
                dt2*f(x[i], t[n])
            else:
                u[i] = U_0(t[n+1])

            i = Ix[-1]
            if U_L is None:
                im1 = i-1
                ip1 = im1
                u[i] = - u_2[i] + 2*u_1[i] + \
                       C2*(0.5*(q[i] + q[ip1])*(u_1[ip1] - u_1[i])  - \
                           0.5*(q[i] + q[im1])*(u_1[i] - u_1[im1])) + \
                dt2*f(x[i], t[n])
            else:
                u[i] = U_L(t[n+1])

            if user_action is not None:
                if user_action(u, x, t, n+1):
                    break

        cpu_time = time.clock() - t0
        return cpu_time, hashed_input
```

```
    def assert_no_error(self):
        """Run through mesh and check error"""
        Nx = self['Nx']
        Nt = self.m.Nt
        L, T = self.problem['L T'.split()]
        L = L/2     # only half the domain used (symmetry)
        x = np.linspace(0, L, Nx+1)     # Mesh points in space
        t = np.linspace(0, T, Nt+1)     # Mesh points in time

        for n in range(len(t)):
            u_e = self.problem.u_exact(x, t[n])
            diff = np.abs(self.f.u[n,:] - u_e).max()
            print 'diff:', diff
            tol = 1E-13
            assert diff < tol
```

Observe that the solutions from all time steps are stored in the mesh function, which allows error assessment (in `assert_no_error`) to take place after all solutions have been found. Of course, in 2D or 3D, such a strategy may place too high demands on available computer memory, in which case intermediate results could be stored on file.

Running `wave1D_oo.py` gives a printout showing that the class-based implementation performs as expected, i.e. that the known exact solution is reproduced (within machine precision).

## C.5 Migrating Loops to Cython

We now consider the `wave2D_u0.py` code for solving the 2D linear wave equation with constant wave velocity and homogeneous Dirichlet boundary conditions $u = 0$. We shall in the present chapter extend this code with computational modules written in other languages than Python. This extended version is called `wave2D_u0_adv.py`.

The `wave2D_u0.py` file contains a `solver` function, which calls an `advance_*` function to advance the numerical scheme one level forward in time. The function `advance_scalar` applies standard Python loops to implement the scheme, while `advance_vectorized` performs corresponding vectorized arithmetics with array slices. The statements of this solver are explained in Sect. 2.12, in particular Sect. 2.12.1 and 2.12.2.

Although vectorization can bring down the CPU time dramatically compared with scalar code, there is still some factor 5-10 to win in these types of applications by implementing the finite difference scheme in compiled code, typically in Fortran, C, or C++. This can quite easily be done by adding a little extra code to our program. Cython is an extension of Python that offers the easiest way to nail our Python loops in the scalar code down to machine code and achieve the efficiency of C.

Cython can be viewed as an extended Python language where variables are declared with types and where functions are marked to be implemented in C. Migrating Python code to Cython is done by copying the desired code segments to

functions (or classes) and placing them in one or more separate files with extension `.pyx`.

### C.5.1 Declaring Variables and Annotating the Code

Our starting point is the plain `advance_scalar` function for a scalar implementation of the updating algorithm for new values $u_{i,j}^{n+1}$:

```
def advance_scalar(u, u_n, u_nm1, f, x, y, t, n, Cx2, Cy2, dt2,
                   V=None, step1=False):
    Ix = range(0, u.shape[0]);  Iy = range(0, u.shape[1])
    if step1:
        dt = sqrt(dt2)  # save
        Cx2 = 0.5*Cx2;  Cy2 = 0.5*Cy2; dt2 = 0.5*dt2  # redefine
        D1 = 1;  D2 = 0
    else:
        D1 = 2;  D2 = 1
    for i in Ix[1:-1]:
        for j in Iy[1:-1]:
            u_xx = u_n[i-1,j] - 2*u_n[i,j] + u_n[i+1,j]
            u_yy = u_n[i,j-1] - 2*u_n[i,j] + u_n[i,j+1]
            u[i,j] = D1*u_n[i,j] - D2*u_nm1[i,j] + \
                     Cx2*u_xx + Cy2*u_yy + dt2*f(x[i], y[j], t[n])
            if step1:
                u[i,j] += dt*V(x[i], y[j])
    # Boundary condition u=0
    j = Iy[0]
    for i in Ix: u[i,j] = 0
    j = Iy[-1]
    for i in Ix: u[i,j] = 0
    i = Ix[0]
    for j in Iy: u[i,j] = 0
    i = Ix[-1]
    for j in Iy: u[i,j] = 0
    return u
```

We simply take a copy of this function and put it in a file `wave2D_u0_loop_cy.pyx`. The relevant Cython implementation arises from declaring variables with types and adding some important annotations to speed up array computing in Cython. Let us first list the complete code in the `.pyx` file:

```
import numpy as np
cimport numpy as np
cimport cython
ctypedef np.float64_t DT    # data type

@cython.boundscheck(False)  # turn off array bounds check
@cython.wraparound(False)   # turn off negative indices (u[-1,-1])
cpdef advance(
    np.ndarray[DT, ndim=2, mode='c'] u,
    np.ndarray[DT, ndim=2, mode='c'] u_n,
    np.ndarray[DT, ndim=2, mode='c'] u_nm1,
    np.ndarray[DT, ndim=2, mode='c'] f,
    double Cx2, double Cy2, double dt2):
```

```
    cdef:
        int Ix_start = 0
        int Iy_start = 0
        int Ix_end = u.shape[0]-1
        int Iy_end = u.shape[1]-1
        int i, j
        double u_xx, u_yy

    for i in range(Ix_start+1, Ix_end):
        for j in range(Iy_start+1, Iy_end):
            u_xx = u_n[i-1,j] - 2*u_n[i,j] + u_n[i+1,j]
            u_yy = u_n[i,j-1] - 2*u_n[i,j] + u_n[i,j+1]
            u[i,j] = 2*u_n[i,j] - u_nm1[i,j] + \
                     Cx2*u_xx + Cy2*u_yy + dt2*f[i,j]
    # Boundary condition u=0
    j = Iy_start
    for i in range(Ix_start, Ix_end+1): u[i,j] = 0
    j = Iy_end
    for i in range(Ix_start, Ix_end+1): u[i,j] = 0
    i = Ix_start
    for j in range(Iy_start, Iy_end+1): u[i,j] = 0
    i = Ix_end
    for j in range(Iy_start, Iy_end+1): u[i,j] = 0
    return u
```

This example may act as a recipe on how to transform array-intensive code with loops into Cython.

1. Variables are declared with types: for example, `double v` in the argument list instead of just `v`, and `cdef double v` for a variable `v` in the body of the function. A Python `float` object is declared as `double` for translation to C by Cython, while an `int` object is declared by `int`.
2. Arrays need a comprehensive type declaration involving
   - the type `np.ndarray`,
   - the data type of the elements, here 64-bit floats, abbreviated as `DT` through `ctypedef np.float64_t DT` (instead of `DT` we could use the full name of the data type: `np.float64_t`, which is a Cython-defined type),
   - the dimensions of the array, here `ndim=2` and `ndim=1`,
   - specification of contiguous memory for the array (`mode='c'`).
3. Functions declared with `cpdef` are translated to C but are also accessible from Python.
4. In addition to the standard `numpy` import we also need a special Cython import of `numpy`: `cimport numpy as np`, to appear *after* the standard import.
5. By default, array indices are checked to be within their legal limits. To speed up the code one should turn off this feature for a specific function by placing `@cython.boundscheck(False)` above the function header.
6. Also by default, array indices can be negative (counting from the end), but this feature has a performance penalty and is therefore here turned off by writing `@cython.wraparound(False)` right above the function header.
7. The use of index sets `Ix` and `Iy` in the scalar code cannot be successfully translated to C. One reason is that constructions like `Ix[1:-1]` involve negative

indices, and these are now turned off. Another reason is that Cython loops must take the form `for i in xrange` or `for i in range` for being translated into efficient C loops. We have therefore introduced `Ix_start` as `Ix[0]` and `Ix_end` as `Ix[-1]` to hold the start and end of the values of index $i$. Similar variables are introduced for the $j$ index. A loop `for i in Ix` is with these new variables written as `for i in range(Ix_start, Ix_end+1)`.

**Array declaration syntax in Cython**

We have used the syntax `np.ndarray[DT, ndim=2, mode='c']` to declare numpy arrays in Cython. There is a simpler, alternative syntax, employing typed memory views[1], where the declaration looks like `double [:,:]`. However, the full support for this functionality is not yet ready, and in this text we use the full array declaration syntax.

### C.5.2 Visual Inspection of the C Translation

Cython can visually explain how successfully it translated a code from Python to C. The command

```
Terminal> cython -a wave2D_u0_loop_cy.pyx
```

produces an HTML file `wave2D_u0_loop_cy.html`, which can be loaded into a web browser to illustrate which lines of the code that have been translated to C. Figure C.1 shows the illustrated code. Yellow lines indicate the lines that Cython

Raw output: wave2D_u0_loop_cy.c

```
1: import numpy as np
2: cimport numpy as np
3: cimport cython
4: ctypedef np.float64_t DT    # data type
5:
6: @cython.boundscheck(False)  # turn off array bounds check
7: @cython.wraparound(False)   # turn off negative indices (u[-1,-1])
8: cpdef advance(
9:     np.ndarray[DT, ndim=2, mode='c'] u,
10:    np.ndarray[DT, ndim=2, mode='c'] u_1,
11:    np.ndarray[DT, ndim=2, mode='c'] u_2,
12:    np.ndarray[DT, ndim=2, mode='c'] f,
13:    double Cx2, double Cy2, double dt2):
14:
15:    cdef int Ix_start = 0
16:    cdef int Iy_start = 0
17:    cdef int Ix_end = u.shape[0]-1
18:    cdef int Iy_end = u.shape[1]-1
19:    cdef int i, j
20:    cdef double u_xx, u_yy
21:
22:    for i in range(Ix_start+1, Ix_end):
23:        for j in range(Iy_start+1, Iy_end):
24:            u_xx = u_1[i-1,j] - 2*u_1[i,j] + u_1[i+1,j]
25:            u_yy = u_1[i,j-1] - 2*u_1[i,j] + u_1[i,j+1]
26:            u[i,j] = 2*u_1[i,j] - u_2[i,j] + \
27:                     Cx2*u_xx + Cy2*u_yy + dt2*f[i,j]
28:    # Boundary condition u=0
29:    j = Iy_start
30:    for i in range(Ix_start, Ix_end+1): u[i,j] = 0
31:    j = Iy_end
32:    for i in range(Ix_start, Ix_end+1): u[i,j] = 0
33:    i = Ix_start
34:    for j in range(Iy_start, Iy_end+1): u[i,j] = 0
35:    i = Iy_end
36:    for j in range(Iy_start, Iy_end+1): u[i,j] = 0
37:    return u
```

**Fig. C.1** Visual illustration of Cython's ability to translate Python to C

[1] http://docs.cython.org/src/userguide/memoryviews.html

did not manage to translate to efficient C code and that remain in Python. For the present code we see that Cython is able to translate all the loops with array computing to C, which is our primary goal.

You can also inspect the generated C code directly, as it appears in the file `wave2D_u0_loop_cy.c`. Nevertheless, understanding this C code requires some familiarity with writing Python extension modules in C by hand. Deep down in the file we can see in detail how the compute-intensive statements have been translated into some complex C code that is quite different from what a human would write (at least if a direct correspondence to the mathematical notation was intended).

### C.5.3 Building the Extension Module

Cython code must be translated to C, compiled, and linked to form what is known in the Python world as a *C extension module*. This is usually done by making a `setup.py` script, which is the standard way of building and installing Python software. For an extension module arising from Cython code, the following `setup.py` script is all we need to build and install the module:

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

cymodule = 'wave2D_u0_loop_cy'
setup(
  name=cymodule
  ext_modules=[Extension(cymodule, [cymodule + '.pyx'],)],
  cmdclass={'build_ext': build_ext},
)
```

We run the script by

```
Terminal> python setup.py build_ext --inplace
```

The `-inplace` option makes the extension module available in the current directory as the file `wave2D_u0_loop_cy.so`. This file acts as a normal Python module that can be imported and inspected:

```
>>> import wave2D_u0_loop_cy
>>> dir(wave2D_u0_loop_cy)
['__builtins__', '__doc__', '__file__', '__name__',
 '__package__', '__test__', 'advance', 'np']
```

The important output from the `dir` function is our Cython function `advance` (the module also features the imported `numpy` module under the name `np` as well as many standard Python objects with double underscores in their names).

The setup.py file makes use of the distutils package in Python and Cython's extension of this package. These tools know how Python was built on the computer and will use compatible compiler(s) and options when building other code in Cython, C, or C++. Quite some experience with building large program systems is needed to do the build process manually, so using a setup.py script is strongly recommended.

**Simplified build of a Cython module**

When there is no need to link the C code with special libraries, Cython offers a shortcut for generating and importing the extension module:

```
import pyximport; pyximport.install()
```

This makes the setup.py script redundant. However, in the wave2D_u0_adv.py code we do not use pyximport and require an explicit build process of this and many other modules.

### C.5.4 Calling the Cython Function from Python

The wave2D_u0_loop_cy module contains our advance function, which we now may call from the Python program for the wave equation:

```
import wave2D_u0_loop_cy
advance = wave2D_u0_loop_cy.advance
...
for n in It[1:-1]:                  # time loop
    f_a[:,:] = f(xv, yv, t[n])     # precompute, size as u
    u = advance(u, u_n, u_nm1, f_a, x, y, t, Cx2, Cy2, dt2)
```

**Efficiency** For a mesh consisting of $120\times120$ cells, the scalar Python code requires 1370 CPU time units, the vectorized version requires 5.5, while the Cython version requires only 1! For a smaller mesh with $60 \times 60$ cells Cython is about 1000 times faster than the scalar Python code, and the vectorized version is about 6 times slower than the Cython version.

## C.6 Migrating Loops to Fortran

Instead of relying on Cython's (excellent) ability to translate Python to C, we can invoke a compiled language directly and write the loops ourselves. Let us start with Fortran 77, because this is a language with more convenient array handling than C (or plain C++), because we can use the same multi-dimensional indices in the Fortran code as in the numpy arrays in the Python code, while in C these arrays are one-dimensional and require us to reduce multi-dimensional indices to a single index.

### C.6.1 The Fortran Subroutine

We write a Fortran subroutine `advance` in a file `wave2D_u0_loop_f77.f` for implementing the updating formula (2.117) and setting the solution to zero at the boundaries:

```
      subroutine advance(u, u_n, u_nm1, f, Cx2, Cy2, dt2, Nx, Ny)
      integer Nx, Ny
      real*8 u(0:Nx,0:Ny), u_n(0:Nx,0:Ny), u_nm1(0:Nx,0:Ny)
      real*8 f(0:Nx,0:Ny), Cx2, Cy2, dt2
      integer i, j
      real*8 u_xx, u_yy
Cf2py intent(in, out) u

C     Scheme at interior points
      do j = 1, Ny-1
         do i = 1, Nx-1
            u_xx = u_n(i-1,j) - 2*u_n(i,j) + u_n(i+1,j)
            u_yy = u_n(i,j-1) - 2*u_n(i,j) + u_n(i,j+1)
            u(i,j) = 2*u_n(i,j) - u_nm1(i,j) + Cx2*u_xx + Cy2*u_yy +
     &               dt2*f(i,j)
         end do
      end do

C     Boundary conditions
      j = 0
      do i = 0, Nx
         u(i,j) = 0
      end do
      j = Ny
      do i = 0, Nx
         u(i,j) = 0
      end do
      i = 0
      do j = 0, Ny
         u(i,j) = 0
      end do
      i = Nx
      do j = 0, Ny
         u(i,j) = 0
      end do
      return
      end
```

This code is plain Fortran 77, except for the special `Cf2py` comment line, which here specifies that `u` is both an input argument *and* an object to be returned from the `advance` routine. Or more precisely, Fortran is not able return an array from a function, but we need a *wrapper code* in C for the Fortran subroutine to enable calling it from Python, and from this wrapper code one can return `u` to the calling Python code.

**Tip: Return all computed objects to the calling code**

It is not strictly necessary to return `u` to the calling Python code since the `advance` function will modify the elements of `u`, but the convention in Python

is to get all output from a function as returned values. That is, the right way of calling the above Fortran subroutine from Python is

```
u = advance(u, u_n, u_nm1, f, Cx2, Cy2, dt2)
```

The less encouraged style, which works and resembles the way the Fortran subroutine is called from Fortran, reads

```
advance(u, u_n, u_nm1, f, Cx2, Cy2, dt2)
```

### C.6.2 Building the Fortran Module with f2py

The nice feature of writing loops in Fortran is that, without much effort, the tool `f2py` can produce a C extension module such that we can call the Fortran version of `advance` from Python. The necessary commands to run are

Terminal

```
Terminal> f2py -m wave2D_u0_loop_f77 -h wave2D_u0_loop_f77.pyf \
          --overwrite-signature wave2D_u0_loop_f77.f
Terminal> f2py -c wave2D_u0_loop_f77.pyf --build-dir build_f77 \
          -DF2PY_REPORT_ON_ARRAY_COPY=1 wave2D_u0_loop_f77.f
```

The first command asks `f2py` to interpret the Fortran code and make a Fortran 90 specification of the extension module in the file `wave2D_u0_loop_f77.pyf`. The second command makes `f2py` generate all necessary wrapper code, compile our Fortran file and the wrapper code, and finally build the module. The build process takes place in the specified subdirectory `build_f77` so that files can be inspected if something goes wrong. The option `-DF2PY_REPORT_ON_ARRAY_COPY=1` makes `f2py` write a message for every array that is copied in the communication between Fortran and Python, which is very useful for avoiding unnecessary array copying (see below). The name of the module file is `wave2D_u0_loop_f77.so`, and this file can be imported and inspected as any other Python module:

```
>>> import wave2D_u0_loop_f77
>>> dir(wave2D_u0_loop_f77)
['__doc__', '__file__', '__name__', '__package__',
 '__version__', 'advance']
>>> print wave2D_u0_loop_f77.__doc__
This module 'wave2D_u0_loop_f77' is auto-generated with f2py....
Functions:
  u = advance(u,u_n,u_nm1,f,cx2,cy2,dt2,
      nx=(shape(u,0)-1),ny=(shape(u,1)-1))
```

**Examine the doc strings!**

Printing the doc strings of the module and its functions is extremely important after having created a module with `f2py`. The reason is that `f2py` makes Python interfaces to the Fortran functions that are different from how the functions are

declared in the Fortran code (!). The rationale for this behavior is that f2py creates *Pythonic* interfaces such that Fortran routines can be called in the same way as one calls Python functions. Output data from Python functions is always returned to the calling code, but this is technically impossible in Fortran. Also, arrays in Python are passed to Python functions without their dimensions because that information is packed with the array data in the array objects. This is not possible in Fortran, however. Therefore, f2py removes array dimensions from the argument list, and f2py makes it possible to return objects back to Python.

Let us follow the advice of examining the doc strings and take a close look at the documentation f2py has generated for our Fortran advance subroutine:

```
>>> print wave2D_u0_loop_f77.advance.__doc__
This module 'wave2D_u0_loop_f77' is auto-generated with f2py
Functions:
  u = advance(u,u_n,u_nm1,f,cx2,cy2,dt2,
              nx=(shape(u,0)-1),ny=(shape(u,1)-1))
.
advance - Function signature:
  u = advance(u,u_n,u_nm1,f,cx2,cy2,dt2,[nx,ny])
Required arguments:
  u : input rank-2 array('d') with bounds (nx + 1,ny + 1)
  u_n : input rank-2 array('d') with bounds (nx + 1,ny + 1)
  u_nm1 : input rank-2 array('d') with bounds (nx + 1,ny + 1)
  f : input rank-2 array('d') with bounds (nx + 1,ny + 1)
  cx2 : input float
  cy2 : input float
  dt2 : input float
Optional arguments:
  nx := (shape(u,0)-1) input int
  ny := (shape(u,1)-1) input int
Return objects:
  u : rank-2 array('d') with bounds (nx + 1,ny + 1)
```

Here we see that the nx and ny parameters declared in Fortran are optional arguments that can be omitted when calling advance from Python.

We strongly recommend to print out the documentation of *every* Fortran function to be called from Python and make sure the call syntax is exactly as listed in the documentation.

### C.6.3 How to Avoid Array Copying

Multi-dimensional arrays are stored as a stream of numbers in memory. For a two-dimensional array consisting of rows and columns there are two ways of creating such a stream: *row-major ordering*, which means that rows are stored consecutively in memory, or *column-major ordering*, which means that the columns are stored one after each other. All programming languages inherited from C, including Python, apply the row-major ordering, but Fortran uses column-major storage. Thinking of a two-dimensional array in Python or C as a matrix, it means that Fortran works with the transposed matrix.

Fortunately, f2py creates extra code so that accessing u(i,j) in the Fortran subroutine corresponds to the element u[i,j] in the underlying numpy array (without the extra code, u(i,j) in Fortran would access u[j,i] in the numpy array). Technically, f2py takes a copy of our numpy array and reorders the data before sending the array to Fortran. Such copying can be costly. For 2D wave simulations on a $60 \times 60$ grid the overhead of copying is a factor of 5, which means that almost the whole performance gain of Fortran over vectorized numpy code is lost!

To avoid having f2py to copy arrays with C storage to the corresponding Fortran storage, we declare the arrays with Fortran storage:

```
order = 'Fortran' if version == 'f77' else 'C'
u   = zeros((Nx+1,Ny+1), order=order)   # solution array
u_n = zeros((Nx+1,Ny+1), order=order)   # solution at t-dt
u_nm1 = zeros((Nx+1,Ny+1), order=order)   # solution at t-2*dt
```

In the compile and build step of using f2py, it is recommended to add an extra option for making f2py report on array copying:

Terminal

```
Terminal> f2py -c wave2D_u0_loop_f77.pyf --build-dir build_f77 \
          -DF2PY_REPORT_ON_ARRAY_COPY=1 wave2D_u0_loop_f77.f
```

It can sometimes be a challenge to track down which array that causes a copying. There are two principal reasons for copying array data: either the array does not have Fortran storage or the element types do not match those declared in the Fortran code. The latter cause is usually effectively eliminated by using real*8 data in the Fortran code and float64 (the default float type in numpy) in the arrays on the Python side. The former reason is more common, and to check whether an array before a Fortran call has the right storage one can print the result of isfortran(a), which is True if the array a has Fortran storage.

Let us look at an example where we face problems with array storage. A typical problem in the wave2D_u0.py code is to set

```
f_a = f(xv, yv, t[n])
```

before the call to the Fortran advance routine. This computation creates a new array with C storage. An undesired copy of f_a will be produced when sending f_a to a Fortran routine. There are two remedies, either direct insertion of data in an array with Fortran storage,

```
f_a = zeros((Nx+1, Ny+1), order='Fortran')
...
f_a[:,:] = f(xv, yv, t[n])
```

or remaking the f(xv, yv, t[n]) array,

```
f_a = asarray(f(xv, yv, t[n]), order='Fortran')
```

The former remedy is most efficient if the asarray operation is to be performed a large number of times.

**Efficiency** The efficiency of this Fortran code is very similar to the Cython code. There is usually nothing more to gain, from a computational efficiency point of view, by implementing the *complete* Python program in Fortran or C. That will just be a lot more code for all administering work that is needed in scientific software, especially if we extend our sample program wave2D_u0.py to handle a real scientific problem. Then only a small portion will consist of loops with intensive array calculations. These can be migrated to Cython or Fortran as explained, while the rest of the programming can be more conveniently done in Python.

## C.7 Migrating Loops to C via Cython

The computationally intensive loops can alternatively be implemented in C code. Just as Fortran calls for care regarding the storage of two-dimensional arrays, working with two-dimensional arrays in C is a bit tricky. The reason is that numpy arrays are viewed as one-dimensional arrays when transferred to C, while C programmers will think of u, u_n, and u_nm1 as two dimensional arrays and index them like u[i][j]. The C code must declare u as double* u and translate an index pair [i][j] to a corresponding single index when u is viewed as one-dimensional. This translation requires knowledge of how the numbers in u are stored in memory.

### C.7.1 Translating Index Pairs to Single Indices

Two-dimensional numpy arrays with the default C storage are stored row by row. In general, multi-dimensional arrays with C storage are stored such that the last index has the fastest variation, then the next last index, and so on, ending up with the slowest variation in the first index. For a two-dimensional u declared as zeros((Nx+1,Ny+1)) in Python, the individual elements are stored in the following order:

```
u[0,0], u[0,1], u[0,2], ..., u[0,Ny], u[1,0], u[1,1], ...,
u[1,Ny], u[2,0], ..., u[Nx,0], u[Nx,1], ..., u[Nx, Ny]
```

Viewing u as one-dimensional, the index pair $(i,j)$ translates to $i(N_y+1)+j$. So, where a C programmer would naturally write an index u[i][j], the indexing must read u[i*(Ny+1) + j]. This is tedious to write, so it can be handy to define a C macro,

```
#define idx(i,j) (i)*(Ny+1) + j
```

so that we can write u[idx(i,j)], which reads much better and is easier to debug.

**Be careful with macro definitions**

Macros just perform simple text substitutions: `idx(hello,world)` is expanded to `(hello)*(Ny+1) + world`. The parentheses in `(i)` are essential – using the natural mathematical formula `i*(Ny+1) + j` in the macro definition, `idx(i-1,j)` would expand to `i-1*(Ny+1) + j`, which is the wrong formula. Macros are handy, but require careful use. In C++, inline functions are safer and replace the need for macros.

### C.7.2 The Complete C Code

The C version of our function `advance` can be coded as follows.

```
#define idx(i,j) (i)*(Ny+1) + j

void advance(double* u, double* u_n, double* u_nm1, double* f,
             double Cx2, double Cy2, double dt2, int Nx, int Ny)
{
  int i, j;
  double u_xx, u_yy;
  /* Scheme at interior points */
  for (i=1; i<=Nx-1; i++) {
    for (j=1; j<=Ny-1; j++) {
      u_xx = u_n[idx(i-1,j)] - 2*u_n[idx(i,j)] + u_n[idx(i+1,j)];
      u_yy = u_n[idx(i,j-1)] - 2*u_n[idx(i,j)] + u_n[idx(i,j+1)];
      u[idx(i,j)] = 2*u_n[idx(i,j)] - u_nm1[idx(i,j)] +
    Cx2*u_xx + Cy2*u_yy + dt2*f[idx(i,j)];
    }
  }
  /* Boundary conditions */
  j = 0;  for (i=0; i<=Nx; i++) u[idx(i,j)] = 0;
  j = Ny; for (i=0; i<=Nx; i++) u[idx(i,j)] = 0;
  i = 0;  for (j=0; j<=Ny; j++) u[idx(i,j)] = 0;
  i = Nx; for (j=0; j<=Ny; j++) u[idx(i,j)] = 0;
}
```

### C.7.3 The Cython Interface File

All the code above appears in the file `wave2D_u0_loop_c.c`. We need to compile this file together with C wrapper code such that `advance` can be called from Python. Cython can be used to generate appropriate wrapper code. The relevant Cython code for interfacing C is placed in a file with extension `.pyx`. This file, called `wave2D_u0_loop_c_cy.pyx`[2], looks like

```
import numpy as np
cimport numpy as np
cimport cython
```

[2] http://tinyurl.com/nu656p2/softeng2/wave2D_u0_loop_c_cy.pyx

```
cdef extern from "wave2D_u0_loop_c.h":
    void advance(double* u, double* u_n, double* u_nm1, double* f,
                 double Cx2, double Cy2, double dt2,
                 int Nx, int Ny)

@cython.boundscheck(False)
@cython.wraparound(False)
def advance_cwrap(
    np.ndarray[double, ndim=2, mode='c'] u,
    np.ndarray[double, ndim=2, mode='c'] u_n,
    np.ndarray[double, ndim=2, mode='c'] u_nm1,
    np.ndarray[double, ndim=2, mode='c'] f,
    double Cx2, double Cy2, double dt2):
    advance(&u[0,0], &u_n[0,0], &u_nm1[0,0], &f[0,0],
            Cx2, Cy2, dt2,
            u.shape[0]-1, u.shape[1]-1)
    return u
```

We first declare the C functions to be interfaced. These must also appear in a C header file, `wave2D_u0_loop_c.h`,

```
extern void advance(double* u, double* u_n, double* u_nm1, double* f,
            double Cx2, double Cy2, double dt2,
            int Nx, int Ny);
```

The next step is to write a Cython function with Python objects as arguments. The name `advance` is already used for the C function so the function to be called from Python is named `advance_cwrap`. The contents of this function is simply a call to the `advance` version in C. To this end, the right information from the Python objects must be passed on as arguments to `advance`. Arrays are sent with their C pointers to the first element, obtained in Cython as `&u[0,0]` (the `&` takes the address of a C variable). The `Nx` and `Ny` arguments in `advance` are easily obtained from the shape of the `numpy` array `u`. Finally, `u` must be returned such that we can set `u = advance(...)` in Python.

### C.7.4 Building the Extension Module

It remains to build the extension module. An appropriate `setup.py` file is

```
from distutils.core import setup
from distutils.extension import Extension
from Cython.Distutils import build_ext

sources = ['wave2D_u0_loop_c.c', 'wave2D_u0_loop_c_cy.pyx']
module = 'wave2D_u0_loop_c_cy'
setup(
  name=module,
  ext_modules=[Extension(module, sources,
                         libraries=[], # C libs to link with
                         )],
  cmdclass={'build_ext': build_ext},
)
```

All we need to specify is the `.c` file(s) and the `.pyx` interface file. Cython is automatically run to generate the necessary wrapper code. Files are then compiled and linked to an extension module residing in the file `wave2D_u0_loop_c_cy.so`. Here is a session with running `setup.py` and examining the resulting module in Python

Terminal

```
Terminal> python setup.py build_ext --inplace
Terminal> python
>>> import wave2D_u0_loop_c_cy as m
>>> dir(m)
['__builtins__', '__doc__', '__file__', '__name__', '__package__',
 '__test__', 'advance_cwrap', 'np']
```

The call to the C version of `advance` can go like this in Python:

```
import wave2D_u0_loop_c_cy
advance = wave2D_u0_loop_c_cy.advance_cwrap
...
f_a[:,:] = f(xv, yv, t[n])
u = advance(u, u_n, u_nm1, f_a, Cx2, Cy2, dt2)
```

**Efficiency** In this example, the C and Fortran code runs at the same speed, and there are no significant differences in the efficiency of the wrapper code. The overhead implied by the wrapper code is negligible as long as there is little numerical work in the `advance` function, or in other words, that we work with small meshes.

## C.8 Migrating Loops to C via f2py

An alternative to using Cython for interfacing C code is to apply `f2py`. The C code is the same, just the details of specifying how it is to be called from Python differ. The `f2py` tool requires the call specification to be a Fortran 90 module defined in a `.pyf` file. This file was automatically generated when we interfaced a Fortran subroutine. With a C function we need to write this module ourselves, or we can use a trick and let `f2py` generate it for us. The trick consists in writing the signature of the C function with Fortran syntax and place it in a Fortran file, here `wave2D_u0_loop_c_f2py_signature.f`:

```
      subroutine advance(u, u_n, u_nm1, f, Cx2, Cy2, dt2, Nx, Ny)
Cf2py intent(c) advance
      integer Nx, Ny, N
      real*8 u(0:Nx,0:Ny), u_n(0:Nx,0:Ny), u_nm1(0:Nx,0:Ny)
      real*8 f(0:Nx, 0:Ny), Cx2, Cy2, dt2
Cf2py intent(in, out) u
Cf2py intent(c) u, u_n, u_nm1, f, Cx2, Cy2, dt2, Nx, Ny
      return
      end
```

Note that we need a special f2py instruction, through a Cf2py comment line, to specify that all the function arguments are C variables. We also need to tell that the function is actually in C: intent(c) advance.

Since f2py is just concerned with the function signature and not the complete contents of the function body, it can easily generate the Fortran 90 module specification based solely on the signature above:

Terminal

```
Terminal> f2py -m wave2D_u0_loop_c_f2py \
          -h wave2D_u0_loop_c_f2py.pyf --overwrite-signature \
          wave2D_u0_loop_c_f2py_signature.f
```

The compile and build step is as for the Fortran code, except that we list C files instead of Fortran files:

Terminal

```
Terminal> f2py -c wave2D_u0_loop_c_f2py.pyf \
          --build-dir tmp_build_c \
          -DF2PY_REPORT_ON_ARRAY_COPY=1 wave2D_u0_loop_c.c
```

As when interfacing Fortran code with f2py, we need to print out the doc string to see the exact call syntax from the Python side. This doc string is identical for the C and Fortran versions of advance.

### C.8.1 Migrating Loops to C++ via f2py

C++ is a much more versatile language than C or Fortran and has over the last two decades become very popular for numerical computing. Many will therefore prefer to migrate compute-intensive Python code to C++. This is, in principle, easy: just write the desired C++ code and use some tool for interfacing it from Python. A tool like SWIG[3] can interpret the C++ code and generate interfaces for a wide range of languages, including Python, Perl, Ruby, and Java. However, SWIG is a comprehensive tool with a correspondingly steep learning curve. Alternative tools, such as Boost Python[4], SIP[5], and Shiboken[6] are similarly comprehensive. Simpler tools include PyBindGen[7].

A technically much easier way of interfacing C++ code is to drop the possibility to use C++ classes directly from Python, but instead make a C interface to the C++ code. The C interface can be handled by f2py as shown in the example with pure C code. Such a solution means that classes in Python and C++ cannot be mixed and that only primitive data types like numbers, strings, and arrays can be transferred between Python and C++. Actually, this is often a very good solution because it

[3] http://swig.org/
[4] http://www.boost.org/doc/libs/1_51_0/libs/python/doc/index.html
[5] http://riverbankcomputing.co.uk/software/sip/intro
[6] http://qt-project.org/wiki/Category:LanguageBindings::PySide::Shiboken
[7] http://code.google.com/p/pybindgen/

forces the C++ code to work on array data, which usually gives faster code than if fancy data structures with classes are used. The arrays coming from Python, and looking like plain C/C++ arrays, can be efficiently wrapped in more user-friendly C++ array classes in the C++ code, if desired.

## C.9 Exercises

**Exercise C.1: Explore computational efficiency of numpy.sum versus built-in sum**
Using the task of computing the sum of the first `n` integers, we want to compare the efficiency of `numpy.sum` versus Python's built-in function `sum`. Use IPython's `%timeit` functionality to time these two functions applied to three different arguments: `range(n)`, `xrange(n)`, and `arange(n)`.
Filename: `sumn`.

**Exercise C.2: Make an improved `numpy.savez` function**
The `numpy.savez` function can save multiple arrays to a zip archive. Unfortunately, if we want to use `savez` in time-dependent problems and call it multiple times (once per time level), each call leads to a separate zip archive. It is more convenient to have all arrays in one archive, which can be read by `numpy.load`. Section C.2 provides a recipe for merging all the individual zip archives into one archive. An alternative is to write a new `savez` function that allows multiple calls and storage into the same archive prior to a final `close` method to close the archive and make it ready for reading. Implement such an improved `savez` function as a class `Savez`.

The class should pass the following unit test:

```
def test_Savez():
    import tempfile, os
    tmp = 'tmp_testarchive'
    database = Savez(tmp)
    for i in range(4):
        array = np.linspace(0, 5+i, 3)
        kwargs = {'myarray_%02d' % i: array}
        database.savez(**kwargs)
    database.close()

    database = np.load(tmp+'.npz')

    expected = {
        'myarray_00': np.array([ 0. ,  2.5,  5. ]),
        'myarray_01': np.array([ 0.,  3.,  6.])
        'myarray_02': np.array([ 0. ,  3.5,  7. ]),
        'myarray_03': np.array([ 0.,  4.,  8.]),
        }
    for name in database:
        computed = database[name]
        diff = np.abs(expected[name] - computed).max()
        assert diff < 1E-13
    database.close
    os.remove(tmp+'.npz')
```

*Hint* Study the source code[8] for function `savez` (or more precisely, function `_savez`).
Filename: `Savez`.

**Exercise C.3: Visualize the impact of the Courant number**
Use the `pulse` function in the `wave1D_dn_vc.py` to simulate a pulse through two media with different wave velocities. The aim is to visualize the impact of the Courant number $C$ on the quality of the solution. Set `slowness_factor=4` and `Nx=100`.

Simulate for $C = 1, 0.9, 0.75$ and make an animation comparing the three curves (use the `animate_archives.py` program to combine the curves and make animations on the screen and video files). Perform the investigations for different types of initial profiles: a Gaussian pulse, a "cosine hat" pulse, half a "cosine hat" pulse, and a plug pulse.
Filename: `pulse1D_Courant`.

**Exercise C.4: Visualize the impact of the resolution**
We solve the same set of problems as in Exercise C.3, except that we now fix $C = 1$ and instead study the impact of $\Delta t$ and $\Delta x$ by varying the `Nx` parameter: 20, 40, 160. Make animations comparing three such curves.
Filename: `pulse1D_Nx`.

[8] https://github.com/numpy/numpy/blob/master/numpy/lib/npyio.py

# References


1. O. Axelsson. *Iterative Solution Methods*. Cambridge University Press, 1996.
2. R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. SIAM, second edition, 1994. http://www.netlib.org/linalg/html_templates/Templates.html.
3. D. Duran. *Numerical Methods for Fluid Dynamics - With Applications to Geophysics*. Springer, second edition, 2010.
4. C. A. J. Fletcher. *Computational Techniques for Fluid Dynamics, Vol. 1: Fundamental and General Techniques*. Springer, second edition, 2013.
5. C. Greif and U. M. Ascher. *A First Course in Numerical Methods*. Computational Science and Engineering. SIAM, 2011.
6. E. Hairer, S. P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I. Nonstiff Problems*. Springer, 1993.
7. M. Hjorth-Jensen. *Computational Physics*. Institute of Physics Publishing, 2016. https://github.com/CompPhysics/ComputationalPhysics1/raw/gh-pages/doc/L%ectures/lectures2015.pdf.
8. C. T. Kelley. *Iterative Methods for Linear and Nonlinear Equations*. SIAM, 1995.
9. H. P. Langtangen. *Finite Difference Computing with Exponential Decay Models*. Lecture Notes in Computational Science and Engineering. Springer, 2016. http://hplgit.github.io/decay-book/doc/web/.
10. H. P. Langtangen. *A Primer on Scientific Programming with Python*. Texts in Computational Science and Engineering. Springer, fifth edition, 2016.
11. H. P. Langtangen and G. K. Pedersen. *Scaling of Differential Equations*. Simula Springer Brief Series. Springer, 2016. http://hplgit.github.io/scaling-book/doc/web/.
12. L. Lapidus and G. F. Pinder. *Numerical Solution of Partial Differential Equations in Science and Engineering*. Wiley, 1982.
13. R. LeVeque. *Finite Difference Methods for Ordinary and Partial Differential Equations: Steady-State and Time-Dependent Problems*. SIAM, 2007.
14. I. P. Omelyan, I. M. Mryglod, and R. Folk. Optimized forest-ruth- and suzuki-like algorithms for integration of motion in many-body systems. *Computer Physics Communication*, 146(2):188–202, 2002.
15. R. Rannacher. Finite element solution of diffusion problems with irregular data. *Numerische Mathematik*, 43:309–327, 1984.
16. Y. Saad. *Iterative Methods for Sparse Linear Systems*. SIAM, second edition, 2003. http://www-users.cs.umn.edu/~saad/IterMethBook_2ndEd.pdf.
17. J. Strikwerda. *Numerical Solution of Partial Differential Equations in Science and Engineering*. SIAM, second edition, 2007.
18. L. N. Trefethen. *Trefethen's index cards - Forty years of notes about People, Words and Mathematics*. World Scientific, 2011.

# Index


1st-order ODE, 29
2nd-order ODE, 29
3D visualization, 177

**A**
accuracy, 234
Adams-Bashforth, 392
ADI methods, 387
`advec1D.py`, 328
alternating mesh, 46
amplification factor, 234
angular frequency, 1
animation, 13
animation speed, 110
`argparse` (Python module), 62
`ArgumentParser` (Python class), 62
arithmetic mean, 137, 359
array computing, 115
array slices, 115
array slices (2D), 174
array updating, 121
`as_ordered_terms`, 161
assert, 7
averaging
  arithmetic, 137
  geometric, 58, 137
  harmonic, 137

**B**
Bernoulli variable, 288
Bokeh, 15
boundary condition
  open (radiation), 149
boundary conditions
  Dirichlet, 126
  Neumann, 126
  periodic, 151
boundary layer, 344

**C**
C extension module, 479
C/Python array storage, 483
cable equation, 316
`__call__`, 110
callback function, 104, 263
centered difference, 2
central difference approximation, 208
CFL condition, 329
Cholesky factorization, 282
`class serial layers`, 248
`clock`, 210
closure, 110
coefficients
  variable, 140
column-major ordering, 483
conjugate gradient method, 285
constrained motion, 79
continuation
  method, 386
  parameter, 386
continuation method, 406
correction terms, 430
cosine hat, 144
cosine pulse
  half-truncated, 325
coupled system, 221, 367
Courant number, 160
`cumsum`, 290
cylindrical coordinates, 251, 314
Cython, 475
`cython -a` (Python-C translation in HTML), 478

**D**
Darcy's law, 312
decay ODE, 422
declaration of variables in Cython, 477
dense coefficient matrix, 260
`diags`, 223, 246
`diff`, 212
difference equations, 95

differential-algebraic equation, 79
diffu1D_u0.py, 210, 223
diffu1D_vc.py, 246
diffu2D_u0.py, 266
diffusion
  artificial, 331
diffusion coefficient, 207
  non-constant, 245
  piecewise constant, 247
diffusion equation
  1D, 207
  1D, boundary condition, 208
  1D, Crank-Nicolson scheme, 224
  1D, dense matrix, 223
  1D, discrete equations, 208
  1D, explicit scheme, 208
  1D, Forward Euler scheme, 208
  1D, Fourier number, 208
  1D, Implementation, 246
  1D, implementation (FE), 210
  1D, implementation (sparse), 223
  1D, implicit schemes, 218
  1D, initial boundary value problem, 208
  1D, initial condition, 208
  1D, mesh Fourier number, 208
  1D, numerical experiments, 215
  1D, sparse matrix, 223
  1D, theta rule, 226
  1D, tridiagonal matrix, 223
  1D, verification (BE), 223
  1D, verification (CN), 226
  1D, verification (FE), 212
  2D, 254
  2D, banded matrix, 258
  2D, implementation, 260
  2D, implementation (sparse), 266
  2D, numbering of mesh points, 255
  2D, sparse matrix, 257
  2D, verification (conv. rates), 265
  2D, verification (exact num. sol.), 264
  axi-symmetric diffusion, 251
  diffusion coefficient, 207
  implementation, 248
  numerical Fourier number, 234
  source term, 208
  spherically-symmetric diffusion, 252
  stationary solution, 207, 247
  truncation error, 234
diffusion limit of random walk, 293
dimensional splitting, 387
dimensionless number, 208
Dirac delta function, 231
Dirichlet conditions, 126
discontinuous initial condition, 227
discontinuous medium, 144
discontinuous plug, 223
discrete derivative, 417
discrete Fourier transform, 157
discretization of domain, 2
discretization parameter, 100, 144
dispersion relation, 337
  analytical, 160
  numerical, 160
distutils, 479
DOF (degree of freedom), 68
domain, 208
dynamic viscosity, 312

**E**
efficiency measurements, 119
energy estimates (diffusion), 316
energy principle, 36
equation of state, 309
error
  global, 25
error function (erf), 229
  complementary, 229
error mesh function, 24
error norm, 7, 25, 38
Euler-Cromer scheme, 40, 439
expectation, 288
explicit discretization methods, 208
extract_leading_order, 161

**F**
factor, 161
fast Fourier transform (FFT), 157
FD operator notation, 4
Fick’s law, 308
finite difference scheme, 94, 95
finite differences
  backward, 417
  centered, 2, 419
  forward, 418
fixed-point iteration, 357
Flash (video format), 13
Fokker-Planck equation, 304
forced vibrations, 57
Fortran 77, 480
Fortran 90, 482
Fortran array storage, 483
Fortran subroutine, 481
forward difference approximation, 208
forward-backward scheme, 40
Fourier series, 157
Fourier transform, 157
Fourier’s law, 309
fractional step methods, 387
free body diagram
  animated, 74
  dynamic, 74
frequency (of oscillations), 1
friction, 315
functools, 119

**G**
Gaussian elimination, 223
Gaussian function, 144
Gaussian pulse, 325
Gauss-Seidel method, 277
geometric mean, 58, 137, 359
ghost
  cells, 132
  points, 132
  values, 132
Gnuplot, 177
`Gnuplot.py`, 177

**H**
harmonic average, 137
hash, 457
heat capacity, 309
heat conduction
  coefficient of, 311
heat equation, 207
homogeneous Dirichlet conditions, 126
homogeneous Neumann conditions, 126
HTML5 video tag, 13
Hz (unit), 1

**I**
ImageMagic, 15
incompressible fluid, 311
index set notation, 128, 173
initial condition
  triangular, 328
interior spatial points, 221
internal energy, 309
interpolation, 137
interrupt a program by Ctrl+c, 300
iterative methods, 270, 353

**J**
Jacobi iterative method, 270
Jacobian, 369
`joblib`, 457

**K**
kinetic energy, 36

**L**
lambda function (Python), 118
`lambdify`, 161
Laplace equation, 207, 227
leading order term, 161
Leapfrog method, 3
Leapfrog scheme, 237
`limit`, 356
`linalg`, 223, 246, 260, 262
linear system, 224, 270, 369
linearization, 357
  explicit time integration, 355
  fixed-point iteration, 357
  Picard iteration, 357
  successive substitutions, 357
`load`, 455
logistic growth, 388
`logistic.py`, 362
LU factorization, 282

**M**
making movies, 13
manufactured solution, 100
mass balance, 311
material derivative, 350
matrix
  equation, 221
  form, 221
  half-bandwidth, 282
  positive definite, 285
`Mayavi`, 179
mechanical energy, 36
mechanical vibrations, 1
memoize function, 457
mesh
  finite differences, 2, 94
  parameters, 155
  uniform, 94
mesh function, 2, 94, 208
mesh points, 208
`mlab`, 179
MP4 (video format), 13

**N**
Navier-Stokes equations, 313
Neumann conditions, 126
neuronal fibers, 316
`newaxis`, 174, 263
Newton's 2nd law, 36
noise
  removing, 231
  sawtooth-like, 229
nonlinear restoring force, 57
nonlinear spring, 57
norm, 25
`nose`, 6, 106
`Numba`, 114
Nyquist frequency, 157

**O**
`ODE_Picard_tricks.py`, 365
Odespy, 32, 392
Ogg (video format), 13
open boundary condition, 149
operator splitting, 387
oscillations, 1

**P**
padding zeros, 110

parallelism, 114
PDE
  algebraic version, 94
pendulum
  elastic, 79
  physical, 74
  simple, 71
period (of oscillations), 1
periodic boundary conditions, 151
phase plane plot, 32
Picard iteration, 357
plotslopes.py, 9
Plotter class (SciTools), 300
plug, 144
Poisson equation, 227
potential energy, 36
preconditioning, 285, 318
pulse propagation, 144
Pysketcher, 74
pytest, 6, 106

**Q**
quadratic convergence, 360
quadratic solution, 106, 118

**R**
radiation condition, 149
radioactive rock, 309
random, 290
random walk, 287
ready-made software, 221
red-black numbering, 278
relaxation, 271
relaxation (nonlinear equations), 361
relaxation parameter, 361
remove0, 161
reshape, 174
resonance, 88
Richardson iteration, 318
round-off error, 81
row-major ordering, 483

**S**
sampling (a PDE), 95
savez, 455
sawtooth-like noise, 227
scalar code, 115
scaling, 81
scaling equations, 113
SciTools, 11
scitools movie command, 14
scitools.avplotter, 300
seed (random numbers), 291
semi-explicit Euler, 40
semi-implicit Euler, 40
series, 161
setup.py, 479
signal processing, 231
simplify, 212
single Picard iteration technique, 358
skipping frames, 110
slice, 115
slope marker (in convergence plots), 9
smooth Gaussian function, 223
smoothing, 231
solver_BE, 223
solver_dense, 260
solver_FE, 212
solver_FECS, 325
solver_FE_simple, 210
source term, 99
sparse, 223, 246
sparse Gaussian elimination, 269
special method, 110
spectral radius, 283
spherical coordinates, 252
split_diffu_react.py, 392
split_logistic.py, 388
split-step methods, 387
splitting ODEs, 387
spring constant, 36
spsolve, 223, 246, 269
stability, 234, 329
stability criterion, 26, 160
staggered Euler-Cromer scheme, 46
staggered mesh, 46, 439
stationary fluid flow, 313
stationary solution, 207
steady state, 227
stencil
  1D wave equation, 94
  Neumann boundary, 126
step function, 229
stiffness, 36
stochastic difference equation, 303
stochastic ODE, 304
stochastic variable, 288
Stoermer's method, 3
Stoermer-Verlet algorithm, 45
stopping criteria (nonlinear problems), 358, 371
storez, 455
Strang splitting, 388
stream function, 313
stress, 350
subs, 212
successive over-relaxation (SOR), 277
successive substitutions, 357
SuperLU, 269
switching references, 121
symmetric successive over-relaxation (SSOR), 285
symplectic scheme, 41
sympy, 22, 356, 412

system of algebraic equations, 367

**T**
Taylor series, 161, 235, 356
test function, 6, 106, 212
time, 210
time step
  spatially varying, 138
todense, 223
transport phenomena, 344
truncation error
  Backward Euler scheme, 417
  correction terms, 430
  Crank-Nicolson scheme, 419
  Forward Euler scheme, 418
  general, 415
  table of formulas, 420
trunc_decay_FE.py, 425

**U**
uniform, 290
unit testing, 6, 106
upwind difference, 331

**V**
vectorization, 6, 114, 115, 290
vectorized
  code, 114
  computing, 114
  loops, 114
verification, 292, 433
  convergence rates, 7, 44, 99, 100, 107, 214, 265, 392, 415, 461
  hand calculations, 6
  polynomial solution, 7, 106
Verlet integration, 3
vib_empirical_analysis.py, 19
vib_EulerCromer.py, 43
vib_plot_freq.py, 22
vib.py, 59
vibration ODE, 1
vib_undamped_EulerCromer.py, 43
vib_undamped_odespy.py, 32
vib_undamped.py, 4
vib_undamped_staggered.py, 49
video formats, 13
viscous boundary layer, 345
viscous effects, 312
visualization of 2D scalar fields, 177

**W**
wave
  complex component, 155
  damping, 140
  reflected, 135
  transmitted, 135
  variable velocity, 135
  velocity, 93
wave equation
  1D, 93
  1D, analytical properties, 155
  1D, discrete, 96
  1D, exact numerical solution, 158
  1D, finite difference method, 94
  1D, implementation, 104
  1D, stability, 160
  2D, implementation, 171
wave1D_dn.py, 128
wave1D_dn_vc.py, 141, 451
wave1D_n0_ghost.py, 132
wave1D_n0.py, 127
wave1D_oo.py, 462
wave1D_u0.py, 105
wave1D_u0v.py, 117
wave2D_u0_adv.py, 475
wave2D_u0_loop_c.c, 486
wave2D_u0_loop_c_f2py_signature.f, 488
wave2D_u0_loop_c.h, 486
wave2D_u0_loop_cy.pyx, 476
wave2D_u0_loop_f77.f, 481
wave2D_u0.py, 172, 475
waves
  on a string, 93
WebM (video format), 13
where, 290
Wiener process, 304
wrapper code, 481

**Z**
zeros, 6
zip archive, 455

## *Editorial Policy*

§1. Textbooks on topics in the field of computational science and engineering will be considered. They should be written for courses in CSE education. Both graduate and undergraduate textbooks will be published in TCSE. Multidisciplinary topics and multidisciplinary teams of authors are especially welcome.

§2. Format: Only works in English will be considered. For evaluation purposes, manuscripts may be submitted in print or electronic form, in the latter case, preferably as pdf- or zipped ps-files. Authors are requested to use the LaTeX style files available from Springer at: https://www.springer.com/gp/authors-editors/book-authors-editors/manuscript-preparation/5636 (Click on ⟶ Templates ⟶ LaTeX ⟶ monographs)
Electronic material can be included if appropriate. Please contact the publisher.

§3. Those considering a book which might be suitable for the series are strongly advised to contact the publisher or the series editors at an early stage.

## *General Remarks*

Careful preparation of manuscripts will help keep production time short and ensure a satisfactory appearance of the finished book.

The following terms and conditions hold:

Regarding free copies and royalties, the standard terms for Springer mathematics textbooks hold. Please write to martin.peters@springer.com for details.

Authors are entitled to purchase further copies of their book and other Springer books for their personal use, at a discount of 40% directly from Springer-Verlag.

## *Series Editors*

Timothy J. Barth
NASA Ames Research Center
NAS Division
Moffett Field, CA 94035, USA
barth@nas.nasa.gov

Michael Griebel
Institut für Numerische Simulation
der Universität Bonn
Wegelerstr. 6
53115 Bonn, Germany
griebel@ins.uni-bonn.de

David E. Keyes
Mathematical and Computer Sciences
and Engineering
King Abdullah University of Science
and Technology
P.O. Box 55455
Jeddah 21534, Saudi Arabia
david.keyes@kaust.edu.sa

and

Department of Applied Physics
and Applied Mathematics
Columbia University
500 W. 120 th Street
New York, NY 10027, USA
kd2112@columbia.edu

Risto M. Nieminen
Department of Applied Physics
Aalto University School of Science
and Technology
00076 Aalto, Finland
risto.nieminen@aalto.fi

Dirk Roose
Department of Computer Science
Katholieke Universiteit Leuven
Celestijnenlaan 200A
3001 Leuven-Heverlee, Belgium
dirk.roose@cs.kuleuven.be

Tamar Schlick
Department of Chemistry
and Courant Institute
of Mathematical Sciences
New York University
251 Mercer Street
New York, NY 10012, USA
schlick@nyu.edu

Editor for Computational Science
and Engineering at Springer:
Martin Peters
Springer-Verlag
Mathematics Editorial IV
Tiergartenstrasse 17
69121 Heidelberg, Germany
martin.peters@springer.com

# Texts in Computational Science and Engineering

1. H. P. Langtangen, *Computational Partial Differential Equations.* Numerical Methods and Diffpack Programming. 2nd Edition
2. A. Quarteroni, F. Saleri, P. Gervasio, *Scientific Computing with MATLAB and Octave.* 4th Edition
3. H. P. Langtangen, *Python Scripting for Computational Science*. 3rd Edition
4. H. Gardner, G. Manduchi, *Design Patterns for e-Science*.
5. M. Griebel, S. Knapek, G. Zumbusch, *Numerical Simulation in Molecular Dynamics*.
6. H. P. Langtangen, *A Primer on Scientific Programming with Python.* 5th Edition
7. A. Tveito, H. P. Langtangen, B. F. Nielsen, X. Cai, *Elements of Scientific Computing.*
8. B. Gustafsson, *Fundamentals of Scientific Computing.*
9. M. Bader, *Space-Filling Curves.*
10. M. Larson, F. Bengzon, *The Finite Element Method: Theory, Implementation and Applications.*
11. W. Gander, M. Gander, F. Kwok, *Scientific Computing: An Introduction using Maple and MATLAB.*
12. P. Deuflhard, S. Röblitz, *A Guide to Numerical Modelling in Systems Biology*.
13. M. H. Holmes, *Introduction to Scientific Computing and Data Analysis*.
14. S. Linge, H. P. Langtangen, *Programming for Computations* – A Gentle Introduction to Numerical Simulations with MATLAB/Octave.
15. S. Linge, H. P. Langtangen, *Programming for Computations* – A Gentle Introduction to Numerical Simulations with Python.
16. H. P. Langtangen, S. Linge, *Finite Difference Computing with PDEs* – A Modern Software Approach.

*For further information on these books please have a look at our mathematics catalogue at the following URL:* www.springer.com/series/5151

# Monographs in Computational Science and Engineering

1. J. Sundnes, G.T. Lines, X. Cai, B.F. Nielsen, K.-A. Mardal, A. Tveito, *Computing the Electrical Activity in the Heart.*

*For further information on this book, please have a look at our mathematics catalogue at the following URL:* www.springer.com/series/7417

# Lecture Notes in Computational Science and Engineering

1. D. Funaro, *Spectral Elements for Transport-Dominated Equations.*
2. H.P. Langtangen, *Computational Partial Differential Equations.* Numerical Methods and Diffpack Programming.
3. W. Hackbusch, G. Wittum (eds.), *Multigrid Methods V.*
4. P. Deuflhard, J. Hermans, B. Leimkuhler, A.E. Mark, S. Reich, R.D. Skeel (eds.), *Computational Molecular Dynamics: Challenges, Methods, Ideas.*
5. D. Kröner, M. Ohlberger, C. Rohde (eds.), *An Introduction to Recent Developments in Theory and Numerics for Conservation Laws.*
6. S. Turek, *Efficient Solvers for Incompressible Flow Problems.* An Algorithmic and Computational Approach.
7. R. von Schwerin, ***M**ulti **B**ody **S**ystem **SIM**ulation.* Numerical Methods, Algorithms, and Software.
8. H.-J. Bungartz, F. Durst, C. Zenger (eds.), *High Performance Scientific and Engineering Computing.*
9. T.J. Barth, H. Deconinck (eds.), *High-Order Methods for Computational Physics.*
10. H.P. Langtangen, A.M. Bruaset, E. Quak (eds.), *Advances in Software Tools for Scientific Computing*.
11. B. Cockburn, G.E. Karniadakis, C.-W. Shu (eds.), *Discontinuous Galerkin Methods.* Theory, Computation and Applications.
12. U. van Rienen, *Numerical Methods in Computational Electrodynamics.* Linear Systems in Practical Applications.
13. B. Engquist, L. Johnsson, M. Hammill, F. Short (eds.), *Simulation and Visualization on the Grid.*
14. E. Dick, K. Riemslagh, J. Vierendeels (eds.), *Multigrid Methods VI.*
15. A. Frommer, T. Lippert, B. Medeke, K. Schilling (eds.), *Numerical Challenges in Lattice Quantum Chromodynamics.*
16. J. Lang, *Adaptive Multilevel Solution of Nonlinear Parabolic PDE Systems.* Theory, Algorithm, and Applications.
17. B.I. Wohlmuth, *Discretization Methods and Iterative Solvers Based on Domain Decomposition.*
18. U. van Rienen, M. Günther, D. Hecht (eds.), *Scientific Computing in Electrical Engineering.*
19. I. Babuška, P.G. Ciarlet, T. Miyoshi (eds.), *Mathematical Modeling and Numerical Simulation in Continuum Mechanics.*
20. T.J. Barth, T. Chan, R. Haimes (eds.), *Multiscale and Multiresolution Methods.* Theory and Applications.
21. M. Breuer, F. Durst, C. Zenger (eds.), *High Performance Scientific and Engineering Computing.*
22. K. Urban, *Wavelets in Numerical Simulation.* Problem Adapted Construction and Applications.
23. L.F. Pavarino, A. Toselli (eds.), *Recent Developments in Domain Decomposition Methods.*
24. T. Schlick, H.H. Gan (eds.), *Computational Methods for Macromolecules: Challenges and Applications.*

25. T.J. Barth, H. Deconinck (eds.), *Error Estimation and Adaptive Discretization Methods in Computational Fluid Dynamics*.
26. M. Griebel, M.A. Schweitzer (eds.), *Meshfree Methods for Partial Differential Equations*.
27. S. Müller, *Adaptive Multiscale Schemes for Conservation Laws*.
28. C. Carstensen, S. Funken, W. Hackbusch, R.H.W. Hoppe, P. Monk (eds.), *Computational Electromagnetics*.
29. M.A. Schweitzer, *A Parallel Multilevel Partition of Unity Method for Elliptic Partial Differential Equations*.
30. T. Biegler, O. Ghattas, M. Heinkenschloss, B. van Bloemen Waanders (eds.), *Large-Scale PDE-Constrained Optimization*.
31. M. Ainsworth, P. Davies, D. Duncan, P. Martin, B. Rynne (eds.), *Topics in Computational Wave Propagation*. Direct and Inverse Problems.
32. H. Emmerich, B. Nestler, M. Schreckenberg (eds.), *Interface and Transport Dynamics.* Computational Modelling.
33. H.P. Langtangen, A. Tveito (eds.), *Advanced Topics in Computational Partial Differential Equations.* Numerical Methods and Diffpack Programming.
34. V. John, *Large Eddy Simulation of Turbulent Incompressible Flows.* Analytical and Numerical Results for a Class of LES Models.
35. E. Bänsch (ed.), *Challenges in Scientific Computing – CISC 2002.*
36. B.N. Khoromskij, G. Wittum, *Numerical Solution of Elliptic Differential Equations by Reduction to the Interface.*
37. A. Iske, *Multiresolution Methods in Scattered Data Modelling.*
38. S.-I. Niculescu, K. Gu (eds.), *Advances in Time-Delay Systems.*
39. S. Attinger, P. Koumoutsakos (eds.), *Multiscale Modelling and Simulation.*
40. R. Kornhuber, R. Hoppe, J. Périaux, O. Pironneau, O. Wildlund, J. Xu (eds.), *Domain Decomposition Methods in Science and Engineering.*
41. T. Plewa, T. Linde, V.G. Weirs (eds.), *Adaptive Mesh Refinement – Theory and Applications.*
42. A. Schmidt, K.G. Siebert, *Design of Adaptive Finite Element Software.* The Finite Element Toolbox ALBERTA.
43. M. Griebel, M.A. Schweitzer (eds.), *Meshfree Methods for Partial Differential Equations II.*
44. B. Engquist, P. Lötstedt, O. Runborg (eds.), *Multiscale Methods in Science and Engineering.*
45. P. Benner, V. Mehrmann, D.C. Sorensen (eds.), *Dimension Reduction of Large-Scale Systems.*
46. D. Kressner, *Numerical Methods for General and Structured Eigenvalue Problems.*
47. A. Boriçi, A. Frommer, B. Joó, A. Kennedy, B. Pendleton (eds.), *QCD and Numerical Analysis III*.
48. F. Graziani (ed.), *Computational Methods in Transport.*
49. B. Leimkuhler, C. Chipot, R. Elber, A. Laaksonen, A. Mark, T. Schlick, C. Schütte, R. Skeel (eds.), *New Algorithms for Macromolecular Simulation.*
50. M. Bücker, G. Corliss, P. Hovland, U. Naumann, B. Norris (eds.), *Automatic Differentiation: Applications, Theory, and Implementations.*
51. A.M. Bruaset, A. Tveito (eds.), *Numerical Solution of Partial Differential Equations on Parallel Computers.*
52. K.H. Hoffmann, A. Meyer (eds.), *Parallel Algorithms and Cluster Computing.*

53. H.-J. Bungartz, M. Schäfer (eds.), *Fluid-Structure Interaction.*

54. J. Behrens, *Adaptive Atmospheric Modeling.*

55. O. Widlund, D. Keyes (eds.), *Domain Decomposition Methods in Science and Engineering XVI.*

56. S. Kassinos, C. Langer, G. Iaccarino, P. Moin (eds.), *Complex Effects in Large Eddy Simulations.*

57. M. Griebel, M.A Schweitzer (eds.), *Meshfree Methods for Partial Differential Equations III.*

58. A.N. Gorban, B. Kégl, D.C. Wunsch, A. Zinovyev (eds.), *Principal Manifolds for Data Visualization and Dimension Reduction.*

59. H. Ammari (ed.), *Modeling and Computations in Electromagnetics: A Volume Dedicated to Jean-Claude Nédélec.*

60. U. Langer, M. Discacciati, D. Keyes, O. Widlund, W. Zulehner (eds.), *Domain Decomposition Methods in Science and Engineering XVII.*

61. T. Mathew, *Domain Decomposition Methods for the Numerical Solution of Partial Differential Equations.*

62. F. Graziani (ed.), *Computational Methods in Transport: Verification and Validation.*

63. M. Bebendorf, *Hierarchical Matrices.* A Means to Efficiently Solve Elliptic Boundary Value Problems.

64. C.H. Bischof, H.M. Bücker, P. Hovland, U. Naumann, J. Utke (eds.), *Advances in Automatic Differentiation.*

65. M. Griebel, M.A. Schweitzer (eds.), *Meshfree Methods for Partial Differential Equations IV*.

66. B. Engquist, P. Lötstedt, O. Runborg (eds.), *Multiscale Modeling and Simulation in Science.*

67. I.H. Tuncer, Ü. Gülcat, D.R. Emerson, K. Matsuno (eds.), *Parallel Computational Fluid Dynamics 2007.*

68. S. Yip, T. Diaz de la Rubia (eds.), *Scientific Modeling and Simulations*.

69. A. Hegarty, N. Kopteva, E. O'Riordan, M. Stynes (eds.), *BAIL* 2008 – *Boundary and Interior Layers.*

70. M. Bercovier, M.J. Gander, R. Kornhuber, O. Widlund (eds.), *Domain Decomposition Methods in Science and Engineering XVIII.*

71. B. Koren, C. Vuik (eds.), *Advanced Computational Methods in Science and Engineering.*

72. M. Peters (ed.), *Computational Fluid Dynamics for Sport Simulation.*

73. H.-J. Bungartz, M. Mehl, M. Schäfer (eds.), *Fluid Structure Interaction II – Modelling, Simulation, Optimization.*

74. D. Tromeur-Dervout, G. Brenner, D.R. Emerson, J. Erhel (eds.), *Parallel Computational Fluid Dynamics 2008.*

75. A.N. Gorban, D. Roose (eds.), *Coping with Complexity: Model Reduction and Data Analysis.*

76. J.S. Hesthaven, E.M. Rønquist (eds.), *Spectral and High Order Methods for Partial Differential Equations.*

77. M. Holtz, *Sparse Grid Quadrature in High Dimensions with Applications in Finance and Insurance*.

78. Y. Huang, R. Kornhuber, O.Widlund, J. Xu (eds.), *Domain Decomposition Methods in Science and Engineering XIX*.

79. M. Griebel, M.A. Schweitzer (eds.), *Meshfree Methods for Partial Differential Equations V*.

80. P.H. Lauritzen, C. Jablonowski, M.A. Taylor, R.D. Nair (eds.), *Numerical Techniques for Global Atmospheric Models*.

81. C. Clavero, J.L. Gracia, F.J. Lisbona (eds.), *BAIL 2010 – Boundary and Interior Layers, Computational and Asymptotic Methods*.

82. B. Engquist, O. Runborg, Y.R. Tsai (eds.), *Numerical Analysis and Multiscale Computations*.

83. I.G. Graham, T.Y. Hou, O. Lakkis, R. Scheichl (eds.), *Numerical Analysis of Multiscale Problems*.

84. A. Logg, K.-A. Mardal, G. Wells (eds.), *Automated Solution of Differential Equations by the Finite Element Method*.

85. J. Blowey, M. Jensen (eds.), *Frontiers in Numerical Analysis – Durham 2010*.

86. O. Kolditz, U.-J. Gorke, H. Shao, W. Wang (eds.), *Thermo-Hydro-Mechanical-Chemical Processes in Fractured Porous Media – Benchmarks and Examples*.

87. S. Forth, P. Hovland, E. Phipps, J. Utke, A. Walther (eds.), *Recent Advances in Algorithmic Differentiation*.

88. J. Garcke, M. Griebel (eds.), *Sparse Grids and Applications*.

89. M. Griebel, M.A. Schweitzer (eds.), *Meshfree Methods for Partial Differential Equations VI*.

90. C. Pechstein, *Finite and Boundary Element Tearing and Interconnecting Solvers for Multiscale Problems*.

91. R. Bank, M. Holst, O. Widlund, J. Xu (eds.), *Domain Decomposition Methods in Science and Engineering XX*.

92. H. Bijl, D. Lucor, S. Mishra, C. Schwab (eds.), *Uncertainty Quantification in Computational Fluid Dynamics*.

93. M. Bader, H.-J. Bungartz, T. Weinzierl (eds.), *Advanced Computing*.

94. M. Ehrhardt, T. Koprucki (eds.), *Advanced Mathematical Models and Numerical Techniques for Multi-Band Effective Mass Approximations*.

95. M. Azaïez, H. El Fekih, J.S. Hesthaven (eds.), *Spectral and High Order Methods for Partial Differential Equations ICOSAHOM 2012*.

96. F. Graziani, M.P. Desjarlais, R. Redmer, S.B. Trickey (eds.), *Frontiers and Challenges in Warm Dense Matter*.

97. J. Garcke, D. Pflüger (eds.), *Sparse Grids and Applications – Munich 2012*.

98. J. Erhel, M. Gander, L. Halpern, G. Pichot, T. Sassi, O. Widlund (eds.), *Domain Decomposition Methods in Science and Engineering XXI*.

99. R. Abgrall, H. Beaugendre, P.M. Congedo, C. Dobrzynski, V. Perrier, M. Ricchiuto (eds.), *High Order Nonlinear Numerical Methods for Evolutionary PDEs – HONOM 2013*.

100. M. Griebel, M.A. Schweitzer (eds.), *Meshfree Methods for Partial Differential Equations VII*.

101. R. Hoppe (ed.), *Optimization with PDE Constraints – OPTPDE 2014*.

102. S. Dahlke, W. Dahmen, M. Griebel, W. Hackbusch, K. Ritter, R. Schneider, C. Schwab, H. Yserentant (eds.), *Extraction of Quantifiable Information from Complex Systems*.

103. A. Abdulle, S. Deparis, D. Kressner, F. Nobile, M. Picasso (eds.), *Numerical Mathematics and Advanced Applications – ENUMATH 2013*.

104. T. Dickopf, M.J. Gander, L. Halpern, R. Krause, L.F. Pavarino (eds.), *Domain Decomposition Methods in Science and Engineering XXII*.

105. M. Mehl, M. Bischoff, M. Schäfer (eds.), *Recent Trends in Computational Engineering – CE2014*. Optimization, Uncertainty, Parallel Algorithms, Coupled and Complex Problems.

106. R.M. Kirby, M. Berzins, J.S. Hesthaven (eds.), *Spectral and High Order Methods for Partial Differential Equations – ICOSAHOM'14*.

107. B. Jüttler, B. Simeon (eds.), *Isogeometric Analysis and Applications 2014*.
108. P. Knobloch (ed.), *Boundary and Interior Layers, Computational and Asymptotic Methods – BAIL 2014*.
109. J. Garcke, D. Pflüger (eds.), *Sparse Grids and Applications – Stuttgart 2014*.
110. H.P. Langtangen, *Finite Difference Computing with Exponential Decay Models*.
111. A. Tveito, G.T. Lines, *Computing Characterizations of Drugs for Ion Channels and Receptors Using Markov Models*
112. B. Karazösen, M. Manguoğlu, M. Tezer-Sezgin, S. Göktepe, Ö. Uğur (eds.), *Numerical Mathematics and Advanced Applications – ENUMATH 2015*.
113. H.-J. Bungartz, P. Neumann, W.E. Nagel (eds.), *Software for Exascale Computing – SPPEXA 2013-2015*.
114. G.R. Barrenechea, F. Brezzi, A. Cangiani, E.H. Georgoulis (eds.), *Building Bridges: Connections and Challenges in Modern Approaches to Numerical Partial Differential Equations*.
115. M. Griebel, M.A. Schweitzer (eds.), *Meshfree Methods for Partial Differential Equations VIII*.
116. C.-O. Lee, X.-C. Cai, D. E. Keyes, H.H. Kim, A. Klawonn, E.-J. Park, O.B. Widlund (eds.), *Domain Decomposition Methods in Science and Engineering XXIII*.

*For further information on these books please have a look at our mathematics catalogue at the following URL:* www.springer.com/series/3527