Introduction to computing with finite difference methods

Hans Petter Langtangen[1,2]

[1]Center for Biomedical Computing, Simula Research Laboratory
[2]Department of Informatics, University of Oslo

Oct 5, 2013

Note: **PRELIMINARY VERSION**

Contents

List of Exercises, Problems, and Projects

Finite difference methods for partial differential equations (PDEs) employ a ange of concepts and tools that can be introduced and illustrated in the context f simple ordinary differential equation (ODE) examples. This is what we do in ie present document. By first working with ODEs, we keep the mathematical roblems to be solved as simple as possible (but no simpler), thereby allowing ill focus on understanding the key concepts and tools. The choice of topics i the forthcoming treatment of ODEs is therefore solely dominated by what arries over to numerical methods for PDEs.

Theory and practice are primarily illustrated by solving the very simple DE $u' = -au$, $u(0) = I$, where $a > 0$ is a constant, but we also address the eneralized problem $u' = -a(t)u + b(t)$ and the nonlinear problem $u' = f(u, t)$. he following topics are introduced:

- How to think when constructing finite difference methods, with special focus on the Forward Euler, Backward Euler, and Crank-Nicolson (midpoint) schemes
- How to formulate a computational algorithm and translate it into Python code
- How to make curve plots of the solutions
- How to compute numerical errors
- How to compute convergence rates
- How to verify an implementation and automate verification through nose tests in Python
- How to structure code in terms of functions, classes, and modules
- How to work with Python concepts such as arrays, lists, dictionaries, lambda functions, functions in functions (closures), doctests, unit tests, command-line interfaces, graphical user interfaces
- How to perform array computing and understand the difference from scalar computing
- How to conduct and automate large-scale numerical experiments
- How to generate scientific reports
- How to uncover numerical artifacts in the computed solution
- How to analyze the numerical schemes mathematically to understand why artifacts occur
- How to derive mathematical expressions for various measures of the error in numerical methods, frequently by using the `sympy` software for symbolic computation

- Introduce concepts such as finite difference operators, mesh (grid functions, stability, truncation error, consistency, and convergence
- Present additional methods for the general nonlinear ODE $u' =$ which is either a scalar ODE or a system of ODEs
- How to access professional packages for solving ODEs
- How the model equation $u' = -au$ arises in a wide range of pheno physics, biology, and finance

The exposition in a nutshell.
Everything we cover is put into a practical, hands-on context. All math ics is translated into working computing codes, and all the mathem theory of finite difference methods presented here is motivated fr strong need to understand strange behavior of programs. Two fundam questions saturate the text:

- How to we solve a differential equation problem and produce num
- How to we trust the answer?

1 Finite difference methods

Goal.
We explain the basic ideas of finite difference methods using a simple ord differential equation $u' = -au$ as primary example. Emphasis is put o reasoning when discretizing the problem and introduction of key con such as mesh, mesh function, finite difference approximations, averagi a mesh, deriation of algorithms, and discrete operator notation.

1.1 A basic model for exponential decay

Our model problem is perhaps the simplest ordinary differential equation

$$u'(t) = -au(t),$$

Here, $a > 0$ is a constant and $u'(t)$ means differentiation with respect t This type of equation arises in a number of widely different phenomen some quantity u undergoes exponential reduction. Examples include rad decay, population decay, investment decay, cooling of an object, pressu in the atmosphere, and retarded motion in fluids (for some of these m can be negative as well), see Section 11 for details and motivation. V chosen this particular ODE not only because its applications are relev even more because studying numerical solution methods for this simp

ives important insight that can be reused in much more complicated settings, ı particular when solving diffusion-type partial differential equations.

The analytical solution of the ODE is found by the method of separation of ariables, which results in

$$u(t) = Ce^{-at},$$

or any arbitrary constant C. To formulate a mathematical problem for which here is a unique solution, we need a condition to fix the value of C. This ondition is known as the *initial condition* and stated as $u(0) = I$. That is, we now the value I of u when the process starts at $t = 0$. The exact solution is hen $u(t) = Ie^{-at}$.

We seek the solution $u(t)$ of the ODE for $t \in (0, T]$. The point $t = 0$ is not ıcluded since we know u here and assume that the equation governs u for $t > 0$. he complete ODE problem then reads: find $u(t)$ such that

$$u' = -au, \ t \in (0, T], \quad u(0) = I\,. \tag{1}$$

his is known as a *continuous problem* because the parameter t varies continuously om 0 to T. For each t we have a corresponding $u(t)$. There are hence infinitely ıany values of t and $u(t)$. The purpose of a numerical method is to formulate corresponding *discrete* problem whose solution is characterized by a finite umber of values, which can be computed in a finite number of steps on a ɔmputer.

.2 The Forward Euler scheme

olving an ODE like (1) by a finite difference method consists of the following ɔur steps:

1. discretizing the domain,
2. fulfilling the equation at discrete time points,
3. replacing derivatives by finite differences,
4. formulating a recursive algorithm.

tep 1: Discretizing the domain. The time domain $[0, T]$ is represented y a finite number of $N_t + 1$ points

$$0 = t_0 < t_1 < t_2 < \cdots < t_{N_t-1} < t_{N_t} = T\,. \tag{2}$$

he collection of points $t_0, t_1, \ldots, t_{N_t}$ constitutes a *mesh* or *grid*. Often the ıesh points will be uniformly spaced in the domain $[0, T]$, which means that ıe spacing $t_{n+1} - t_n$ is the same for all n. This spacing is often denoted by Δt, ı this case $t_n = n\Delta t$.

We seek the solution u at the mesh points: $u(t_n)$, $n = 1, 2, \ldots, N_t$. Note that ^{0} is already known as I. A notational short-form for $u(t_n)$, which will be used

extensively, is u^n. More precisely, we let u^n be the *numerical approx* to the exact solution $u(t_n)$ at $t = t_n$. The numerical approximation is *function*, here defined only at the mesh points. When we need to distinguish between the numerical and the exact solution, we often subscript e on the exact solution, as in $u_e(t_n)$. Figure 1 shows the t_n points for $n = 0, 1, \ldots, N_t = 7$ as well as $u_e(t)$ as the dashed line. The g numerical method for ODEs is to compute the mesh function by solving set of *algebraic equations* derived from the original ODE problem.

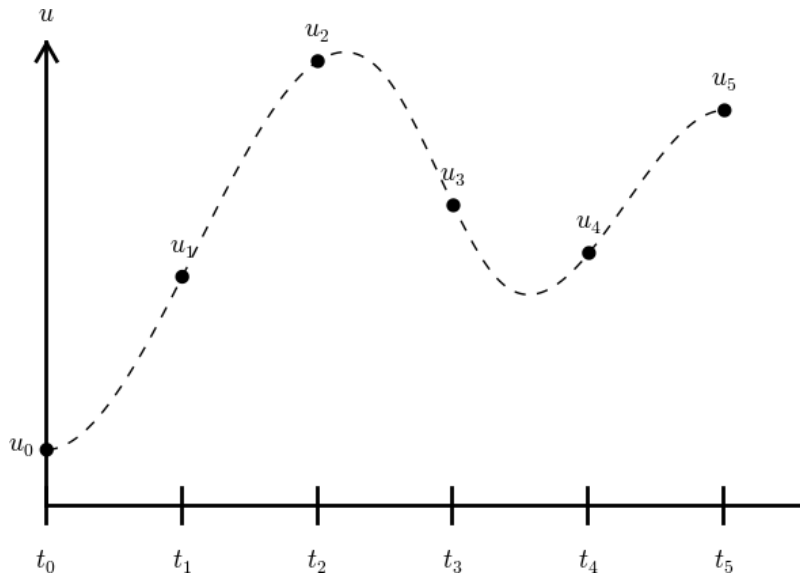
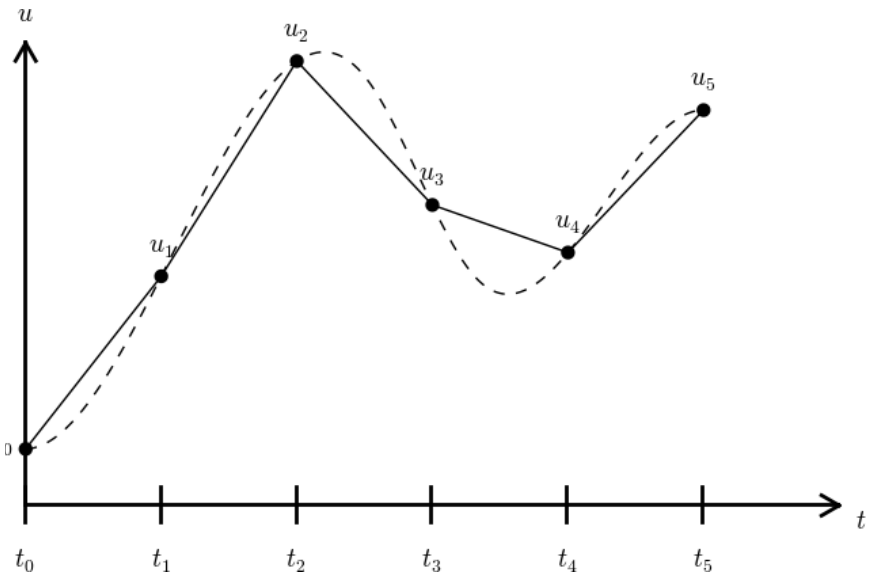


Figure 1: Time mesh with discrete solution values.

Since finite difference methods produce solutions at the mesh points is an open question what the solution is between the mesh points. One methods for interpolation to compute the value of u between mesh poir simplest (and most widely used) interpolation method is to assume that linearly between the mesh points, see Figure 2. Given u^n and u^{n+1}, the u at some $t \in [t_n, t_{n+1}]$ is by linear interpolation

$$u(t) \approx u^n + \frac{u^{n+1} - u^n}{t_{n+1} - t_n}(t - t_n)\,.$$

Step 2: Fulfilling the equation at discrete time points. The supposed to hold for all $t \in (0, T]$, i.e., at an infinite number of points. relax that requirement and require that the ODE is fulfilled at a finit discrete points in time. The mesh points $t_1, t_2, \ldots, t_{N_t}$ are a natural c points. The original ODE is then reduced to the following N_t equation:



igure 2: Linear interpolation between the discrete solution values (dashed ırve is exact solution).

$$u'(t_n) = -au(t_n), \quad n = 1, \ldots, N_t\,. \tag{4}$$

tep 3: Replacing derivatives by finite differences. The next and most ssential step of the method is to replace the derivative u' by a finite differ- ıce approximation. Let us first try a one-sided difference approximation (see igure 3),

$$u'(t_n) \approx \frac{u^{n+1} - u^n}{t_{n+1} - t_n}\,. \tag{5}$$

ıserting this approximation in (4) results in

$$\frac{u^{n+1} - u^n}{t_{n+1} - t_n} = -au^n, \quad n = 0, 1, \ldots, N_t - 1\,. \tag{6}$$

his equation is the discrete counterpart to the original ODE problem (1), and ften referred to as *finite difference scheme* or more generally as the *discrete quations* of the problem. The fundamental feature of these equations is that ıey are *algebraic* and can hence be straightforwardly solved to produce the ıesh function, i.e., the values of u at the mesh points $(u^n,\, n = 1, 2, \ldots, N_t)$.

tep 4: Formulating a recursive algorithm. The final step is to identify ıe computational algorithm to be implemented in a program. The key obser- ıtion here is to realize that (6) can be used to compute u^{n+1} if u^n is known.

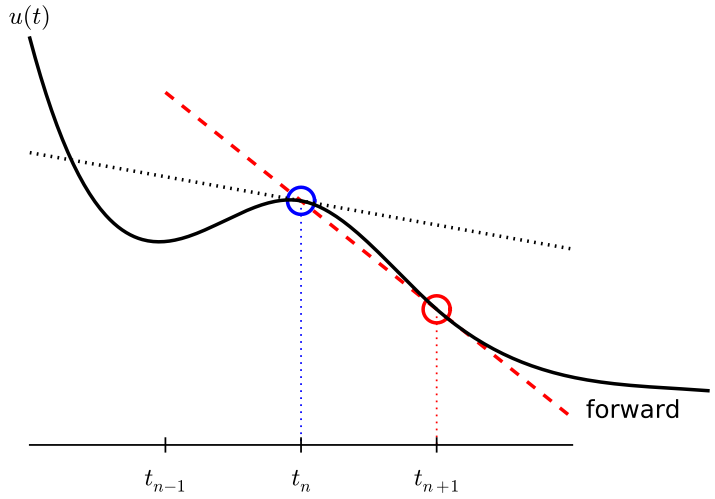


Figure 3: Illustration of a forward difference.

Starting with $n = 0$, u^0 is known since $u^0 = u(0) = I$, and (6) gives an e for u^1. Knowing u^1, u^2 can be found from (6). In general, u^n in (6) assumed known, and then we can easily solve for the unknown u^{n+1}:

$$u^{n+1} = u^n - a(t_{n+1} - t_n)u^n\,.$$

We shall refer to (7) as the Forward Euler (FE) scheme for our model p From a mathematical point of view, equations of the form (7) are kı *difference equations* since they express how differences in u, like $u^{n+1} - u^n$ with n. The finite difference method can be viewed as a method for tu differential equation into a difference equation.

Computation with (7) is straightforward:

$$\begin{aligned}
u_0 &= I,\\
u_1 &= u^0 - a(t_1 - t_0)u^0 = I(1 - a(t_1 - t_0)),\\
u_2 &= u^1 - a(t_2 - t_1)u^1 = I(1 - a(t_1 - t_0))(1 - a(t_2 - t_1)),\\
u^3 &= u^2 - a(t_3 - t_2)u^2 = I(1 - a(t_1 - t_0))(1 - a(t_2 - t_1))(1 - a(t_3 -
\end{aligned}$$

and so on until we reach u^{N_t}. Very often, $t_{n+1} - t_n$ is constant for ε we can introduce the common symbol Δt for the time step: $\Delta t = t_n$ $n = 0, 1, \ldots, N_t - 1$. Using a constant time step Δt in the above calc gives

$$
\begin{aligned}
u_0 &= I,\\
u_1 &= I(1-a\Delta t),\\
u_2 &= I(1-a\Delta t)^2,\\
u^3 &= I(1-a\Delta t)^3,\\
&\vdots\\
u^{N_t} &= I(1-a\Delta t)^{N_t}\,.
\end{aligned}
$$

his means that we have found a closed formula for u^n, and there is no need o let a computer generate the sequence $u^1, u^2, u^3, \ldots$. However, finding such formula for u^n is possible only for a few very simple problems, so in general nite difference equations must be solved on a computer.

As the next sections will show, the scheme (7) is just one out of many lternative finite difference (and other) methods for the model problem (1).

.3 The Backward Euler scheme

here are several choices of difference approximations in step 3 of the finite ifference method as presented in the previous section. Another alternative is

$$
u'(t_n) \approx \frac{u^n - u^{n-1}}{t_n - t_{n-1}}\,. \tag{8}
$$

ince this difference is based on going backward in time (t_{n-1}) for information, is known as the Backward Euler difference. Figure 4 explains the idea.

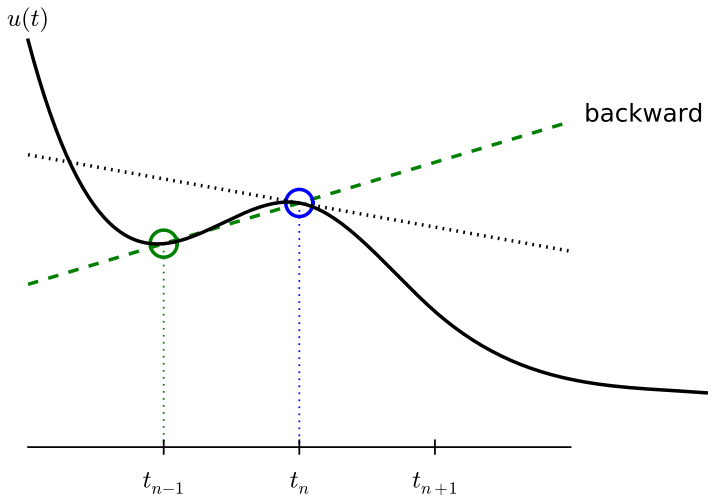


Figure 4: Illustration of a backward difference.

Inserting (8) in (4) yields the Backward Euler (BE) scheme:

$$
\frac{u^n - u^{n-1}}{t_n - t_{n-1}} = -au^n\,.
$$

We assume, as explained under step 4 in Section 1.2, that we have co $u^0, u^1, \ldots, u^{n-1}$ such that (9) can be used to compute u^n. For direct si with the Forward Euler scheme (7) we replace n by $n+1$ in (9) and s the unknown value u^{n+1}:

$$
u^{n+1} = \frac{1}{1 + a(t_{n+1} - t_n)}u^n\,.
$$

1.4 The Crank-Nicolson scheme

The finite difference approximations used to derive the schemes (7) and both one-sided differences, known to be less accurate than central (or m differences. We shall now construct a central difference at $t_{n+1/2} = \frac{1}{2}(t_n$ or $t_{n+1/2} = (n+\frac{1}{2})\Delta t$ if the mesh spacing is uniform in time. The approx reads

$$
u'(t_{n+\frac{1}{2}}) \approx \frac{u^{n+1} - u^n}{t_{n+1} - t_n}\,.
$$

Note that the fraction on the right-hand side is the same as for the I Euler approximation (5) and the Backward Euler approximation (8) replaced by $n+1$). The accuracy of this fraction as an approximatio derivative of u depends on *where* we seek the derivative: in the cente interval $[t_n, t_{n+1}]$ or at the end points.

With the formula (11), where u' is evaluated at $t_{n+1/2}$, it is na demand the ODE to be fulfilled at the time points between the mesh p

$$
u'(t_{n+\frac{1}{2}}) = -au(t_{n+\frac{1}{2}}), \quad n = 0, \ldots, N_t - 1\,.
$$

Using (11) in (12) results in

$$
\frac{u^{n+1} - u^n}{t_{n+1} - t_n} = -au^{n+\frac{1}{2}},
$$

where $u^{n+\frac{1}{2}}$ is a short form for $u(t_{n+\frac{1}{2}})$. The problem is that we aim to c u^n for integer n, implying that $u^{n+\frac{1}{2}}$ is not a quantity computed by our It must therefore be expressed by the quantities that we actually prod the numerical solution at the mesh points. One possibility is to appr $u^{n+\frac{1}{2}}$ as an arithmetic mean of the u values at the neighboring mesh p

$$
u^{n+\frac{1}{2}} \approx \frac{1}{2}(u^n + u^{n+1})\,.
$$

Using (14) in (13) results in

$$\frac{u^{n+1}-u^n}{t_{n+1}-t_n} = -a\frac{1}{2}(u^n + u^{n+1})\,. \tag{15}$$

igure 5 sketches the geometric interpretation of such a centered difference.

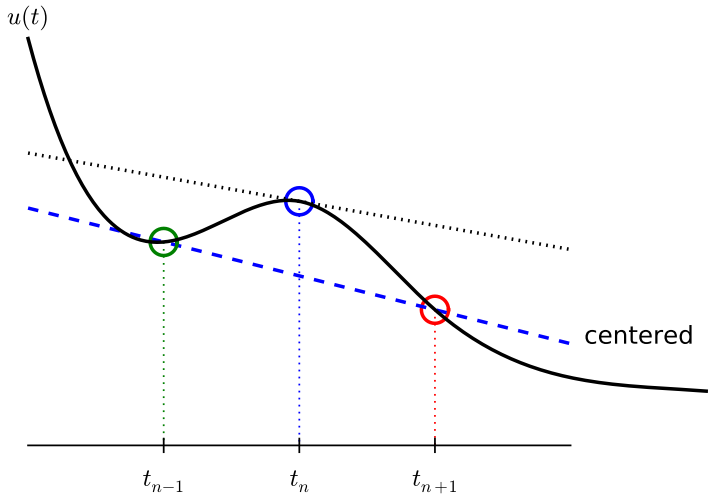


Figure 5: Illustration of a centered difference.

We assume that u^n is already computed so that u^{n+1} is the unknown, which e can solve for:

$$u^{n+1} = \frac{1-\frac{1}{2}a(t_{n+1}-t_n)}{1+\frac{1}{2}a(t_{n+1}-t_n)}u^n\,. \tag{16}$$

he finite difference scheme (16) is often called the Crank-Nicolson (CN) scheme r a midpoint or centered scheme.

.5 The unifying θ-rule

he Forward Euler, Backward Euler, and Crank-Nicolson schemes can be formu- ıted as one scheme with a varying parameter θ:

$$\frac{u^{n+1}-u^n}{t_{n+1}-t_n} = -a(\theta u^{n+1} + (1-\theta)u^n)\,. \tag{17}$$

Observe:

- $\theta = 0$ gives the Forward Euler scheme
- $\theta = 1$ gives the Backward Euler scheme, and
- $\theta = 1/2$ gives the Crank-Nicolson scheme.

- We may alternatively choose any other value of θ in $[0, 1]$.

As before, u^n is considered known and u^{n+1} unknown, so we solve for th

$$u^{n+1} = \frac{1-(1-\theta)a(t_{n+1}-t_n)}{1+\theta a(t_{n+1}-t_n)}\,.$$

This scheme is known as the θ-rule, or alternatively written as the "the

Derivation.
We start with replacing u' by the fraction

$$\frac{u^{n+1}-u^n}{t_{n+1}-t_n},$$

in the Forward Euler, Backward Euler, and Crank-Nicolson schemes. we observe that the difference between the methods concerns which this fraction approximates the derivative. Or in other words, at which we sample the ODE. So far this has been the end points or the midpo $[t_n, t_{n+1}]$. However, we may choose any point $\tilde{t} \in [t_n, t_{n+1}]$. The difficu that evaluating the right-hand side $-au$ at an arbitrary point faces the problem as in Section 1.4: the point value must be expressed by the di u quantities that we compute by the scheme, i.e., u^n and u^{n+1}. Foll the averaging idea from Section 1.4, the value of u at an arbitrary p can be calculated as a *weighted average*, which generalizes the arith mean $\frac{1}{2}u^n + \frac{1}{2}u^{n+1}$. If we express $\tilde{t}$ as a weighted average

$$t_{n+\theta} = \theta t_{n+1} + (1-\theta)t_n,$$

where $\theta \in [0, 1]$ is the weighting factor, we can write

$$u(\tilde{t}) = u(\theta t_{n+1} + (1-\theta)t_n) \approx \theta u^{n+1} + (1-\theta)u^n\,.$$

We can now let the ODE hold at the point $\tilde{t} \in [t_n, t_{n+1}]$, approxi u' by the fraction $(u^{n+1}-u^n)/(t_{n+1}-t_n)$, and approximate the right- side $-au$ by the weighted average (19). The result is (17).

1.6 Constant time step

All schemes up to now have been formulated for a general non-uniform time: $t_0, t_1, \ldots, t_{N_t}$. Non-uniform meshes are highly relevant since one many points in regions where u varies rapidly, and save points in region u is slowly varying. This is the key idea of *adaptive* methods where the of the mesh points are determined as the computations proceed.

However, a uniformly distributed set of mesh points is very comm sufficient for many applications. It therefore makes sense to present tl difference schemes for a uniform point distribution $t_n = n\Delta t$, where Δ constant spacing between the mesh points, also referred to as the *time st* resulting formulas look simpler and are perhaps more well known.

Summary of schemes for constant time step.

$$u^{n+1} = (1 - a\Delta t)u^n \quad \text{Forward Euler} \tag{20}$$

$$u^{n+1} = \frac{1}{1 + a\Delta t}u^n \quad \text{Backward Euler} \tag{21}$$

$$u^{n+1} = \frac{1 - \frac{1}{2}a\Delta t}{1 + \frac{1}{2}a\Delta t}u^n \quad \text{Crank-Nicolson} \tag{22}$$

$$u^{n+1} = \frac{1 - (1 - \theta)a\Delta t}{1 + \theta a\Delta t}u^n \quad \text{The } \theta - \text{rule} \tag{23}$$

Not surprisingly, we present these three alternative schemes because they ave different pros and cons, both for the simple ODE in question (which can asily be solved as accurately as desired), and for more advanced differential quation problems.

Test the understanding.
At this point it can be good training to apply the explained finite difference discretization techniques to a slightly different equation. Exercise 1 is therefore highly recommended to check that the key concepts are understood.

.7 Compact operator notation for finite differences

inite difference formulas can be tedious to write and read, especially for differential equations with many terms and many derivatives. To save space and help the eader of the scheme to quickly see the nature of the difference approximations, e introduce a compact notation. A forward difference approximation is denoted y the D_t^+ operator:

$$[D_t^+ u]^n = \frac{u^{n+1} - u^n}{\Delta t} \approx \frac{d}{dt}u(t_n)\,. \tag{24}$$

he notation consists of an operator that approximates differentiation with espect to an independent variable, here t. The operator is built of the symbol D, ith the variable as subscript and a superscript denoting the type of difference. he superscript $^+$ indicates a forward difference. We place square brackets round the operator and the function it operates on and specify the mesh point, here the operator is acting, by a superscript.

The corresponding operator notation for a centered difference and a backward ifference reads

$$[D_t u]^n = \frac{u^{n+\frac{1}{2}} - u^{n-\frac{1}{2}}}{\Delta t} \approx \frac{d}{dt}u(t_n), \tag{25}$$

nd

$$[D_t^- u]^n = \frac{u^n - u^{n-1}}{\Delta t} \approx \frac{d}{dt}u(t_n)\,. \tag{26}$$

Note that the superscript $^-$ denotes the backward difference, while no sup implies a central difference.

An averaging operator is also convenient to have:

$$[\overline{u}^t]^n = \frac{1}{2}(u^{n-\frac{1}{2}} + u^{n+\frac{1}{2}}) \approx u(t_n)$$

The superscript t indicates that the average is taken along the time coo The common average $(u^n + u^{n+1})/2$ can now be expressed as $[\overline{u}^t]^{n+1/2}$. also spatial coordinates enter the problem, we need the explicit specific the coordinate after the bar.)

The Backward Euler finite difference approximation to $u' = -au$ written as follows utilizing the compact notation:

$$[D_t^- u]^n = -au^n\,.$$

In difference equations we often place the square brackets around th equation, to indicate at which mesh point the equation applies, since ea is supposed to be approximated at the same point:

$$[D_t^- u = -au]^n\,.$$

The Forward Euler scheme takes the form

$$[D_t^+ u = -au]^n,$$

while the Crank-Nicolson scheme is written as

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

Question.
Apply (25) and (27) and write out the expressions to see that (30) is i the Crank-Nicolson scheme.

The θ-rule can be specified by

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

if we define a new time difference and a *weighted averaging operator*:

$$[\bar{D}_t u]^{n+\theta} = \frac{u^{n+1} - u^n}{t^{n+1} - t^n},$$

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

where $\theta \in [0, 1]$. Note that for $\theta = 1/2$ we recover the standard centered di and the standard arithmetic mean. The idea in (31) is to sample the e at $t_{n+\theta}$, use a skew difference at that point $[\bar{D}_t u]^{n+\theta}$, and a skew mea An alternative notation is

$$[D_t u]^{n+1/2} = \theta[-au]^{n+1} + (1 - \theta)[-au]^n\,.$$

Looking at the various examples above and comparing them with the under-
ing differential equations, we see immediately which difference approximations
iat have been used and at which point they apply. Therefore, the compact
otation effectively communicates the reasoning behind turning a differential
quation into a difference equation.

Implementation

Goal.
We want make a computer program for solving

$$u'(t) = -au(t), \quad t \in (0,T], \quad u(0) = I,$$

by finite difference methods. The program should also display the numerical solution as a curve on the screen, preferably together with the exact solution. We shall also be concerned with program testing, user interfaces, and computing convergence rates.

All programs referred to in this section are found in the `src/decay`[1] directory
ve use the classical Unix term *directory* for what many others nowadays call
older).

Mathematical problem. We want to explore the Forward Euler scheme, the
ackward Euler, and the Crank-Nicolson schemes applied to our model problem.
rom an implementational point of view, it is advantageous to implement the
-rule

$$u^{n+1} = \frac{1-(1-\theta)a\Delta t}{1+\theta a\Delta t}u^n,$$

nce it can generate the three other schemes by various of choices of θ: $\theta = 0$ for
orward Euler, $\theta = 1$ for Backward Euler, and $\theta = 1/2$ for Crank-Nicolson. Given
, $u^0 = I$, T, and Δt, our task is to use the θ-rule to compute $u^1, u^2, \ldots, u^{N_t}$,
here $t_{N_t} = N_t\Delta t$, and N_t the closest integer to $T/\Delta t$.

Computer Language: Python. Any programming language can be used to
enerate the u^{n+1} values from the formula above. However, in this document
e shall mainly make use of Python of several reasons:

- Python has a very clean, readable syntax (often known as "executable pseudo-code").
- Python code is very similar to MATLAB code (and MATLAB has a particularly widespread use for scientific computing).
- Python is a full-fledged, very powerful programming language.
- Python is similar to, but much simpler to work with and results in more reliable code than C++.

[1] `http://tinyurl.com/jvzzcfn/decay`

- Python has a rich set of modules for scientific computing, and its po
in scientific computing is rapidly growing.
- Python was made for being combined with compiled languages (C
Fortran) to reuse existing numerical software and to reach high c
tional performance of new implementations.
- Python has extensive support for administrative task needed whe
large-scale computational investigations.
- Python has extensive support for graphics (visualization, user in
web applications).
- FEniCS, a very powerful tool for solving PDEs by the finite element
is most human-efficient to operate from Python.

Learning Python is easy. Many newcomers to the language will probab
enough from the forthcoming examples to perform their own computer
ments. The examples start with simple Python code and gradually mak
more powerful constructs as we proceed. As long as it is not inconven
the problem at hand, our Python code is made as close as possible to M
code for easy transition between the two languages.

Readers who feel the Python examples are too hard to follow will p
benefit from read a tutorial, e.g.,

- The Official Python Tutorial[2]
- Python Tutorial on tutorialspoint.com[3]
- Interactive Python tutorial site[4]
- A Beginner's Python Tutorial[5]

The author also has a book [**?**] that introduces scientific programmi
Python.

2.1 Making a solver function

We choose to have an array `u` for storing the u^n values, $n = 0, 1, \ldots, l$
algorithmic steps are

1. initialize u^0
2. for $t = t_n$, $n = 1, 2, \ldots, N_t$: compute u_n using the θ-rule formula

[2] `http://docs.python.org/2/tutorial/`
[3] `http://www.tutorialspoint.com/python/`
[4] `http://www.learnpython.org/`
[5] `http://en.wikibooks.org/wiki/A_Beginner's_Python_Tutorial`

unction for computing the numerical solution. The following Python unction takes the input data of the problem (I, a, T, Δt, θ) as arguments and eturns two arrays with the solution $u^0, \ldots, u^{N_t}$ and the mesh points $t_0, \ldots, t_{N_t}$, espectively:

```
rom numpy import *

def solver(I, a, T, dt, theta):
    """Solve u'=-a*u, u(0)=I, for t in (0,T] with steps of dt."""
    Nt = int(T/dt)            # no of time intervals
    T = Nt*dt                 # adjust T to fit time step dt
    u = zeros(Nt+1)           # array of u[n] values
    t = linspace(0, T, Nt+1)  # time mesh

    u[0] = I                  # assign initial condition
    for n in range(0, Nt):    # n=0,1,...,Nt-1
        u[n+1] = (1 - (1-theta)*a*dt)/(1 + theta*dt*a)*u[n]
    return u, t
```

The `numpy` library contains a lot of functions for array computing. Most f the function names are similar to what is found in the alternative scientific omputing language MATLAB. Here we make use of

- `zeros(Nt+1)` for creating an array of a size `Nt+1` and initializing the elements to zero
- `linspace(0, T, Nt+1)` for creating an array with `Nt+1` coordinates uniformly distributed between `0` and `T`

he `for` loop deserves a comment, especially for newcomers to Python. The onstruction `range(0, Nt, s)` generates all integers from `0` to `Nt` in steps of , *but not including* `Nt`. Omitting `s` means `s=1`. For example, `range(0, 6, 3)` ives `0` and `3`, while `range(0, Nt)` generates `0`, `1`, ..., `Nt-1`. Our loop implies ne following assignments to `u[n+1]`: `u[1]`, `u[2]`, ..., `u[Nt]`, which is what we ant since `u` has length `Nt+1`. The first index in Python arrays or lists is *always* and the last is then `len(u)-1`. The length of an array `u` is obtained by `len(u)` r `u.size`.

To compute with the `solver` function, we need to *call* it. Here is a sample all:

```
, t = solver(I=1, a=2, T=8, dt=0.8, theta=1)
```

nteger division. The shown implementation of the `solver` may face problems nd wrong results if `T`, `a`, `dt`, and `theta` are given as integers, see Exercises 4 nd 5. The problem is related to *integer division* in Python (as well as in Fortran, , C++, and many other computer languages): `1/2` becomes `0`, while `1.0/2`, `/2.0`, or `1.0/2.0` all become `0.5`. It is enough that at least the nominator r the denominator is a real number (i.e., a `float` object) to ensure correct athematical division. Inserting a conversion `dt = float(dt)` guarantees that `t` is `float` and avoids problems in Exercise 5.

Another problem with computing $N_t = T/\Delta t$ is that we should roun the nearest integer. With `Nt = int(T/dt)` the `int` operation picks the integer smaller than `T/dt`. Correct mathematical rounding as known fron is obtained by

```
Nt = int(round(T/dt))
```

The complete version of our improved, safer `solver` function then becc

```
from numpy import *

def solver(I, a, T, dt, theta):
    """Solve u'=-a*u, u(0)=I, for t in (0,T] with steps of dt."""
    dt = float(dt)            # avoid integer division
    Nt = int(round(T/dt))     # no of time intervals
    T = Nt*dt                 # adjust T to fit time step dt
    u = zeros(Nt+1)           # array of u[n] values
    t = linspace(0, T, Nt+1)  # time mesh

    u[0] = I                  # assign initial condition
    for n in range(0, Nt):    # n=0,1,...,Nt-1
        u[n+1] = (1 - (1-theta)*a*dt)/(1 + theta*dt*a)*u[n]
    return u, t
```

Doc strings. Right below the header line in the `solver` function th Python string enclosed in triple double quotes `"""`. The purpose of thi object is to document what the function does and what the arguments this case the necessary documentation do not span more than one line, b triple double quoted strings the text may span several lines:

```
def solver(I, a, T, dt, theta):
    """
    Solve

        u'(t) = -a*u(t),

    with initial condition u(0)=I, for t in the time interval
    (0,T]. The time interval is divided into time steps of
    length dt.

    theta=1 corresponds to the Backward Euler scheme, theta=0
    to the Forward Euler scheme, and theta=0.5 to the Crank-
    Nicolson method.
    """
    ...
```

Such documentation strings appearing right after the header of a f are called *doc strings*. There are tools that can automatically produc formatted documentation by extracting the definition of functions a contents of doc strings.

It is strongly recommended to equip any function whose purpos obvious with a doc string. Nevertheless, the forthcoming text deviates fi rule if the function is explained in the text.

ormatting of numbers. Having computed the discrete solution `u`, it is atural to look at the numbers:

```
# Write out a table of t and u values:
for i in range(len(t)):
    print t[i], u[i]
```

his compact `print` statement gives unfortunately quite ugly output because the and `u` values are not aligned in nicely formatted columns. To fix this problem, e recommend to use the *printf format*, supported most programming languages nherited from C. Another choice is Python's recent *format string syntax*.

Writing `t[i]` and `u[i]` in two nicely formatted columns is done like this with ne printf format:

```
print 't=%6.3f u=%g' % (t[i], u[i])
```

he percentage signs signify "slots" in the text where the variables listed at the nd of the statement are inserted. For each "slot" one must specify a format for ow the variable is going to appear in the string: `s` for pure text, `d` for an integer, for a real number written as compactly as possible, `9.3E` for scientific notation ith three decimals in a field of width 9 characters (e.g., `-1.351E-2`), or `.2f` for tandard decimal notation with two decimals formatted with minimum width. he printf syntax provides a quick way of formatting tabular output of numbers ith full control of the layout.

The alternative *format string syntax* looks like

```
print 't={t:6.3f} u={u:g}'.format(t=t[i], u=u[i])
```

s seen, this format allows logical names in the "slots" where `t[i]` and `u[i]` are be inserted. The "slots" are surrounded by curly braces, and the logical name followed by a colon and then the printf-like specification of how to format real umbers, integers, or strings.

unning the program. The function and main program shown above must e placed in a file, say with name `decay_v1.py`[6] (`v1` stands for "version 1" - we nall make numerous different versions of this program). Make sure you write the ode with a suitable text editor (Gedit, Emacs, Vim, Notepad++, or similar). he program is run by executing the file this way:

```
erminal> python decay_v1.py
```

he text `Terminal>` just indicates a prompt in a Unix/Linux or DOS terminal indow. After this prompt, which will look different in your terminal window, epending on the terminal application and how it is set up, commands like

[6] `http://tinyurl.com/jvzzcfn/decay/decay_v1.py`

`python decay_v1.py` can be issued. These commands are interpreted operating system.

We strongly recommend to run Python programs within the IPyth First start IPython by typing `ipython` in the terminal window. Ins IPython shell, our program `decay_v1.py` is run by the command `run dec`

```
Terminal> ipython

In [1]: run decay_v1.py
t= 0.000 u=1
t= 0.800 u=0.384615
t= 1.600 u=0.147929
t= 2.400 u=0.0568958
t= 3.200 u=0.021883
t= 4.000 u=0.00841653
t= 4.800 u=0.00323713
t= 5.600 u=0.00124505
t= 6.400 u=0.000478865
t= 7.200 u=0.000184179
t= 8.000 u=7.0838e-05

In [2]:
```

The advantage of running programs in IPython are many: previous co are easily recalled with the up arrow, `%pdb` turns on debugging so that v can be examined if the program aborts due to an exception, output of co are stored in variables, programs and statements can be profiled, any o system command can be executed, modules can be loaded automatic other customizations can be performed when starting IPython – to m few of the most useful features.

Although running programs in IPython is strongly recommende execution examples in the forthcoming text use the standard Python sh prompt `>>>` and run programs through a typesetting like

```
Terminal> python programname
```

The reason is that such typesetting makes the text more compact in the direction than showing sessions with IPython syntax.

2.2 Verifying the implementation

It is easy to make mistakes while deriving and implementing numeri rithms, so we should never believe in the printed u values before they h thoroughly verified. The most obvious idea is to compare the computed with the exact solution, when that exists, but there will always be a disc between these two solutions because of the numerical approximatio challenging question is whether we have the mathematically correct disc

r if we have another, maybe small, discrepancy due to both an approximation
:ror and an error in the implementation.

The purpose of *verifying* a program is to bring evidence for the property
iat there are no errors in the implementation. To avoid mixing unavoidable
pproximation errors and undesired implementation errors, we should try to
iake tests where we have some exact computation of the discrete solution or at
:ast parts of it. Examples will show how this can be done.

Running a few algorithmic steps by hand. The simplest approach to
roduce a correct reference for the discrete solution u of finite difference equations
to compute a few steps of the algorithm by hand. Then we can compare the
and calculations with numbers produced by the program.

A straightforward approach is to use a calculator and compute u^1, u^2, and
^{3}. With $I = 0.1$, $\theta = 0.8$, and $\Delta t = 0.8$ we get

$$A \equiv \frac{1-(1-\theta)a\Delta t}{1+\theta a\Delta t} = 0.298245614035$$

$$\begin{aligned} u^1 &= AI = 0.0298245614035, \\ u^2 &= Au^1 = 0.00889504462912, \\ u^3 &= Au^2 = 0.00265290804728 \end{aligned}$$

Comparison of these manual calculations with the result of the `solver`
unction is carried out in the function

```
lef verify_three_steps():
    """Compare three steps with known manual computations."""
    theta = 0.8; a = 2; I = 0.1; dt = 0.8
    u_by_hand = array([I,
                       0.0298245614035,
                       0.00889504462912,
                       0.00265290804728])

    Nt = 3  # number of time steps
    u, t = solver(I=I, a=a, T=Nt*dt, dt=dt, theta=theta)

    tol = 1E-15  # tolerance for comparing floats
    difference = abs(u - u_by_hand).max()
    success = difference <= tol
    return success
```

The main program, where we call the `solver` function and print `u`, is now
ut in a separate function `main`:

```
lef main():
    u, t = solver(I=1, a=2, T=8, dt=0.8, theta=1)
    # Write out a table of t and u values:
    for i in range(len(t)):
        print 't=%6.3f u=%g' % (t[i], u[i])
        # or print 't={t:6.3f} u={u:g}'.format(t=t[i], u=u[i])
```


The main program in the file may now first run the verification test a
go on with the real simulation (`main()`) only if the test is passed:

```
if verify_three_steps():
    main()
else:
    print 'Bug in the implementation!'
```

Since the verification test is always done, future errors introduced acci
in the program have a good chance of being detected.

> **Caution: choice of parameter values.**
> For the choice of values of parameters in verification tests one shoulc
> away from integers, especially 0 and 1, as these can simplify formula
> much for test purposes. For example, with $\theta = 1$ the nominator i
> formula for u^n will be the same for all a and Δt values. One should the
> choose more "arbitrary" values, say $\theta = 0.8$ and $I = 0.1$.

It is essential that verification tests can be automatically run at *any* ti
this purpose, there are test frameworks and corresponding programmi
that allow us to request running through a suite of test cases (see Secti
but in this very early stage of program development we just implement
the verification in our own code so that every detail is visible and unde

The complete program including the `verify_three_steps*` func
found in the file `decay_verf1.py`[7] (`verf1` is a short name for "veri
version 1").

Comparison with an exact discrete solution. Sometimes it is po
find a closed-form *exact discrete solution* that fulfills the discrete finite di
equations. The implementation can then be verified against the exact
solution. This is usually the best technique for verification.

Define

$$A = \frac{1-(1-\theta)a\Delta t}{1+\theta a\Delta t}\,.$$

Manual computations with the θ-rule results in

$$\begin{aligned} u^0 &= I, \\ u^1 &= Au^0 = AI, \\ u^2 &= Au^1 = A^2 I, \\ &\vdots \\ u^n &= A^n u^{n-1} = A^n I\,. \end{aligned}$$

We have then established the exact discrete solution as

$$u^n = IA^n\,.$$

[7] `http://tinyurl.com/jvzzcfn/decay/decay_verf1.py`

Caution.
One should be conscious about the different meanings of the notation on the left- and right-hand side of (34): on the left, n in u^n is a superscript reflecting a counter of mesh points (t_n), while on the right, n is the power in the exponentiation A^n.

Comparison of the exact discrete solution and the computed solution is done ı the following function:

```
lef verify_exact_discrete_solution():

    def exact_discrete_solution(n, I, a, theta, dt):
        A = (1 - (1-theta)*a*dt)/(1 + theta*dt*a)
        return I*A**n

    theta = 0.8; a = 2; I = 0.1; dt = 0.8
    Nt = int(8/dt)  # no of steps
    u, t = solver(I=I, a=a, T=Nt*dt, dt=dt, theta=theta)
    u_de = array([exact_discrete_solution(n, I, a, theta, dt)
                  for n in range(Nt+1)])
    difference = abs(u_de - u).max()  # max deviation
    tol = 1E-15  # tolerance for comparing floats
    success = difference <= tol
    return success
```

he complete program is found in the file `decay_verf2.py`[8] (`verf2` is a short ame for "verification, version 2").

Local functions.
One can define a function inside another function, here called a *local function* (also known as *closure*) inside a *parent function*. A local function is invisible outside the parent function. A convenient property is that any local function has access to all variables defined in the parent function, also if we send the local function to some other function as argument (!). In the present example, it means that the local function `exact_discrete_solution` does not need its five arguments as the values can alternatively be accessed through the local variables defined in the parent function `verify_exact_discrete_solution`. We can send such an `exact_discrete_solution` without arguments to any other function and `exact_discrete_solution` will still have access to `n`, `I`, `a`, and so forth defined in its parent function.

.3 Computing the numerical error as a mesh function

ow that we have evidence for a correct implementation, we are in a position to ɔmpare the computed u^n values in the `u` array with the exact u values at the ıesh points, in order to study the error in the numerical solution.

Let us first make a function for the analytical solution $u_{\mathrm{e}}(t) = Ie^{-at}$ of the ıodel problem:

[8] `http://tinyurl.com/jvzzcfn/decay/decay_verf2.py`


```
def exact_solution(t, I, a):
    return I*exp(-a*t)
```

A natural way to compare the exact and discrete solutions is to c their difference as a mesh function:

$$e^n = u_{\mathrm{e}}(t_n) - u^n, \quad n = 0, 1, \ldots, N_t\,.$$

We may view $u^n_{\mathrm{e}} = u_{\mathrm{e}}(t_n)$ as the representation of $u_{\mathrm{e}}(t)$ as a mesh f rather than a continuous function defined for all $t \in [0, T]$ (u^n_{e} is often c *representative* of u_{e} on the mesh). Then, $e^n = u^n_{\mathrm{e}} - u^n$ is clearly the di of two mesh functions. This interpretation of e^n is natural when progr

The error mesh function e^n can be computed by

```
u, t = solver(I, a, T, dt, theta)  # Numerical sol.
u_e = exact_solution(t, I, a)      # Representative of exact sol.
e = u_e - u
```

Note that the mesh functions `u` and `u_e` are represented by arrays and as with the points in the array `t`.

Array arithmetics.
The last statements

```
u_e = exact_solution(t, I, a)
e = u_e - u
```

are primary examples of array arithmetics: `t` is an array of mesh p that we pass to `exact_solution`. This function evaluates `-a*t`, whic scalar times an array, meaning that the scalar is multiplied with each element. The result is an array, let us call it `tmp1`. Then `exp(tmp1)` r applying the exponential function to each element in `tmp`, resulting an say `tmp2`. Finally, `I*tmp2` is computed (scalar times array) and `u_e` to this array returned from `exact_solution`. The expression `u_e -` the difference between two arrays, resulting in a new array referred `e`.

2.4 Computing the norm of the numerical error

Instead of working with the error e^n on the entire mesh, we often w number expressing the size of the error. This is obtained by taking the the error function.

Let us first define norms of a function $f(t)$ defined for all $t \in [0, T]$ common norms are

$$||f||_{L^2} = \left(\int_0^T f(t)^2 dt \right)^{1/2}, \tag{36}$$

$$||f||_{L^1} = \int_0^T |f(t)| dt, \tag{37}$$

$$||f||_{L^\infty} = \max_{t\in[0,T]} |f(t)|\,. \tag{38}$$

he L^2 norm (36) ("L-two norm") has nice mathematical properties and is the ıost popular norm. It is a generalization of the well-known Eucledian norm f vectors to functions. The L^∞ is also called the max norm or the supremum orm. In fact, there is a whole family of norms,

$$||f||_{L^p} = \left(\int_0^T f(t)^p dt \right)^{1/p}, \tag{39}$$

ith p real. In particular, $p = 1$ corresponds to the L^1 norm above while $p = \infty$ the L^∞ norm.

Numerical computations involving mesh functions need corresponding norms. iven a set of function values, f^n, and some associated mesh points, t_n, a umerical integration rule can be used to calculate the L^2 and L^1 norms defined bove. Imagining that the mesh function is extended to vary linearly between ıe mesh points, the Trapezoidal rule is in fact an exact integration rule. A ossible modification of the L^2 norm for a mesh function f^n on a uniform mesh ith spacing Δt is therefore the well-known Trapezoidal integration formula

$$||f^n|| = \left(\Delta t \left(\frac{1}{2}(f^0)^2 + \frac{1}{2}(f^{N_t})^2 + \sum_{n=1}^{N_t-1} (f^n)^2 \right) \right)^{1/2}$$

common approximation of this expression, motivated by the convenience of aving a simpler formula, is

$$||f^n||_{\ell^2} = \left(\Delta t \sum_{n=0}^{N_t} (f^n)^2 \right)^{1/2}.$$

his is called the discrete L^2 norm and denoted by ℓ^2. The error in $||f||^2_{\ell^2}$ ompared with the Trapezoidal integration formula is $\Delta t((f^0)^2 + (f^{N_t})^2)/2$, hich means perturbed weights at the end points of the mesh function, and the ror goes to zero as $\Delta t \to 0$. As long as we are consistent and stick to one kind f integration rule for the norm of a mesh function, the details and accuracy of ıis rule is not of concern.

The three discrete norms for a mesh function f^n, corresponding to the L^2, ^{1}, and L^∞ norms of $f(t)$ defined above, are defined by

$$||f^n||_{\ell^2} \left(\Delta t \sum_{n=0}^{N_t} (f^n)^2 \right)^{1/2},$$

$$||f^n||_{\ell^1} \Delta t \sum_{n=0}^{N_t} |f^n|$$

$$||f^n||_{\ell^\infty} \max_{0\le n\le N_t} |f^n|\,.$$

Note that the L^2, L^1, ℓ^2, and ℓ^1 norms depend on the length of the of interest (think of $f = 1$, then the norms are proportional to $\sqrt{T}$ oı some applications it is convenient to think of a mesh function as just a v function values and neglect the information of the mesh points. Then replace Δt by T/N_t and drop T. Moreover, it is convenient to divide total length of the vector, $N_t + 1$, instead of N_t. This reasoning gives ris *vector norms* for a vector $f = (f_0, \ldots, f_N)$:

$$||f||_2 = \left(\frac{1}{N+1} \sum_{n=0}^{N} (f_n)^2 \right)^{1/2},$$

$$||f||_1 = \frac{1}{N+1} \sum_{n=0}^{N} |f_n|$$

$$||f||_{\ell^\infty} = \max_{0\le n\le N} |f_n|\,.$$

Here we have used the common vector component notation with subscri and N as length. We will mostly work with mesh functions and use the ℓ^2 norm (40) or the max norm ℓ^∞ (42), but the corresponding vecto (43)-(45) are also much used in numerical computations, so it is impo know the different norms and the relations between them.

A single number that expresses the size of the numerical error will b as $||e^n||_{\ell^2}$ and called E:

$$E = \sqrt{\Delta t \sum_{n=0}^{N_t} (e^n)^2}$$

The corresponding Python code, using array arithmetics, reads

```
E = sqrt(dt*sum(e**2))
```

The `sum` function comes from `numpy` and computes the sum of the ele an array. Also the `sqrt` function is from `numpy` and computes the squar each element in the array argument.

calar computing. Instead of doing array computing `sqrt(dt*sum(e**2))` e can compute with one element at a time:

```
n = len(u)     # length of u array (alt: u.size)
u_e = zeros(m)
t = 0
for i in range(m):
    u_e[i] = exact_solution(t, a, I)
    t = t + dt
e = zeros(m)
for i in range(m):
    e[i] = u_e[i] - u[i]
s = 0  # summation variable
for i in range(m):
    s = s + e[i]**2
error = sqrt(dt*s)
```

uch element-wise computing, often called *scalar* computing, takes more code, less readable, and runs much slower than what we can achieve with array omputing.

.5 Plotting solutions

aving the `t` and `u` arrays, the approximate solution `u` is visualized by the ıtuitive command `plot(t, u)`:

```
from matplotlib.pyplot import *
plot(t, u)
show()
```

lotting multiple curves. It will be illustrative to also plot $u_{\mathrm{e}}(t)$ for comarison. Doing a `plot(t, u_e)` is not exactly what we want: the `plot` function raws straight lines between the discrete points `(t[n], u_e[n])` while $u_{\mathrm{e}}(t)$ aries as an exponential function between the mesh points. The technique for owing the "exact" variation of $u_{\mathrm{e}}(t)$ between the mesh points is to introduce a ery fine mesh for $u_{\mathrm{e}}(t)$:

```
t_e = linspace(0, T, 1001)      # fine mesh
u_e = exact_solution(t_e, I, a)
plot(t_e, u_e, 'b-')            # blue line for u_e
plot(t,   u,   'r--o')          # red dashes w/circles
```

With more than one curve in the plot we need to associate each curve ith a legend. We also want appropriate names on the axis, a title, and a file ontaining the plot as an image for inclusion in reports. The Matplotlib package `matplotlib.pyplot`) contains functions for this purpose. The names of the unctions are similar to the plotting functions known from MATLAB. A complete lot session then becomes

```
from matplotlib.pyplot import *

figure()                        # create new plot
```



```
t_e = linspace(0, T, 1001)        # fine mesh for u_e
u_e = exact_solution(t_e, I, a)
plot(t,   u,   'r--o')            # red dashes w/circles
plot(t_e, u_e, 'b-')              # blue line for exact sol.
legend(['numerical', 'exact'])
xlabel('t')
ylabel('u')
title('theta=%g, dt=%g' % (theta, dt))
savefig('%s_%g.png' % (theta, dt))
show()
```

Note that `savefig` here creates a PNG file whose name reflects the val and Δt so that we can easily distinguish files from different runs with θ

A bit more sophisticated and easy-to-read filename can be gener mapping the θ value to acronyms for the three common schemes: FE (I Euler, $\theta = 0$), BE (Backward Euler, $\theta = 1$), CN (Crank-Nicolson, $\theta =$ Python dictionary is ideal for such a mapping from numbers to strings:

```
theta2name = {0: 'FE', 1: 'BE', 0.5: 'CN'}
savefig('%s_%g.png' % (theta2name[theta], dt))
```

Experiments with computing and plotting. Let us wrap up the tation of the error measure and all the plotting statements in a function e This function can be called for various θ and Δt values to see how tl varies with the method and the mesh resolution:

```
def explore(I, a, T, dt, theta=0.5, makeplot=True):
    """
    Run a case with the solver, compute error measure,
    and plot the numerical and exact solutions (if makeplot=True)
    """
    u, t = solver(I, a, T, dt, theta)    # Numerical solution
    u_e = exact_solution(t, I, a)
    e = u_e - u
    E = sqrt(dt*sum(e**2))
    if makeplot:
        figure()                         # create new plot
        t_e = linspace(0, T, 1001)       # fine mesh for u_e
        u_e = exact_solution(t_e, I, a)
        plot(t,   u,   'r--o')           # red dashes w/circles
        plot(t_e, u_e, 'b-')             # blue line for exact so
        legend(['numerical', 'exact'])
        xlabel('t')
        ylabel('u')
        title('theta=%g, dt=%g' % (theta, dt))
        theta2name = {0: 'FE', 1: 'BE', 0.5: 'CN'}
        savefig('%s_%g.png' % (theta2name[theta], dt))
        savefig('%s_%g.pdf' % (theta2name[theta], dt))
        savefig('%s_%g.eps' % (theta2name[theta], dt))
        show()
    return E
```

The `figure()` call is key here: without it, a new `plot` command w the new pair of curves in the same plot window, while we want the c pairs to appear in separate windows and files. Calling `figure()` ensure

The `explore` function stores the plot in three different image file formats: NG, PDF, and EPS (Encapsulated PostScript). The PNG format is aimed at eing included in HTML files, the PDF format in pdfLaTeX documents, and the PS format in LaTeX documents. Frequently used viewers for these image files n Unix systems are `gv` (comes with Ghostscript) for the PDF and EPS formats nd `display` (from the ImageMagick) suite for PNG files:

```
erminal> gv BE_0.5.pdf
erminal> gv BE_0.5.eps
erminal> display BE_0.5.png
```

The complete code containing the functions above resides in the file `decay_lot_mpl.py`[9]. Running this program results in

```
erminal> python decay_plot_mpl.py
.0   0.40:    2.105E-01
.0   0.04:    1.449E-02
.5   0.40:    3.362E-02
.5   0.04:    1.887E-04
.0   0.40:    1.030E-01
.0   0.04:    1.382E-02
```

Ve observe that reducing Δt by a factor of 10 increases the accuracy for all ıree methods (θ values). We also see that the combination of $\theta = 0.5$ and a nall time step $\Delta t = 0.04$ gives a much more accurate solution, and that $\theta = 0$ nd $\theta = 1$ with $\Delta t = 0.4$ result in the least accurate solutions.

Figure 6 demonstrates that the numerical solution for $\Delta t = 0.4$ clearly lies elow the exact curve, but that the accuracy improves considerably by reducing ıe time step by a factor of 10.

Combining plot files. Mounting two PNG files, as done in the figure, is easily one by the `montage`[10] program from the ImageMagick suite:

```
erminal> montage -background white -geometry 100% -tile 2x1 \
         FE_0.4.png FE_0.04.png FE1.png
erminal> convert -trim FE1.png FE1.png
```

he `-geometry` argument is used to specify the size of the image, and here we reserve the individual sizes of the images. The `-tile HxV` option specifies `H` nages in the horizontal direction and `V` images in the vertical direction. A series f image files to be combined are then listed, with the name of the resulting ombined image, here `FE1.png` at the end. The `convert -trim` command

[9] `http://tinyurl.com/jvzzcfn/decay/decay_plot_mpl.py`
[10] `http://www.imagemagick.org/script/montage.php`

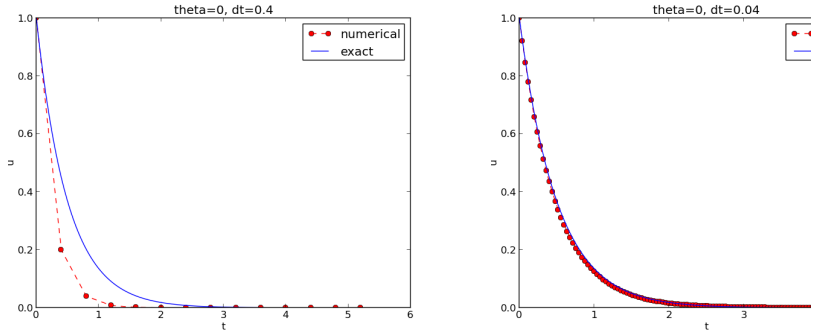


Figure 6: The Forward Euler scheme for two values of the time st

removes surrounding white areas in the figure (an operation usually kı *cropping* in image manipulation programs).

For LaTeX reports it is not recommended to use `montage` and PNG file result has too low resolution. Instead, plots should be made in the PDF and combined using the `pdftk`, `pdfnup`, and `pdfcrop` tools (on Linux/

```
Terminal> pdftk FE_0.4.png FE_0.04.png output tmp.pdf
Terminal> pdfnup --nup 2x1 tmp.pdf      # output in tmp-nup.pdf
Terminal> pdfcrop tmp-nup.pdf FE1.png   # output in FE1.png
```

Here, `pdftk` combines images into a multi-page PDF file, `pdfnup` comb images in individual pages to a table of images (pages), and `pdfcrop` white margins in the resulting combined image file.

The behavior of the two other schemes is shown in Figures 7 and 8. Nicolson is obviously the most accurate scheme from this visual point

Plotting with SciTools. The SciTools package[11] provides a unified interface, called Easyviz, to many different plotting packages, includi plotlib, Gnuplot, Grace, MATLAB, VTK, OpenDX, and VisIt. The s very similar to that of Matplotlib and MATLAB. In fact, the plotting co shown above look the same in SciTool's Easyviz interface, apart from th statement, which reads

```
from scitools.std import *
```

This statement performs a `from numpy import *` as well as an impor most common pieces of the Easyviz (`scitools.easyviz`) package, alo some additional numerical functionality.

[11] `http://code.google.com/p/scitools`

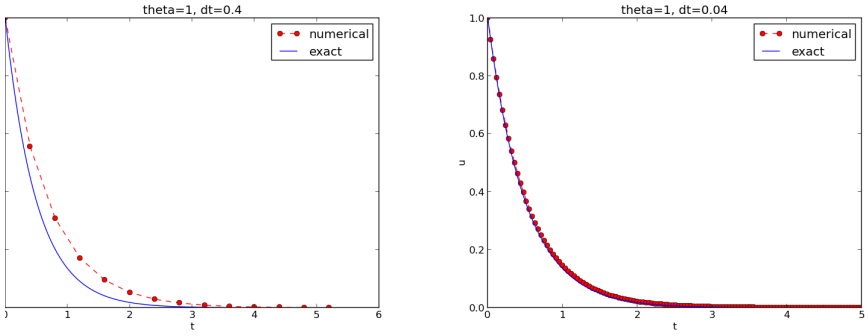


Figure 7: The Backward Euler scheme for two values of the time step.

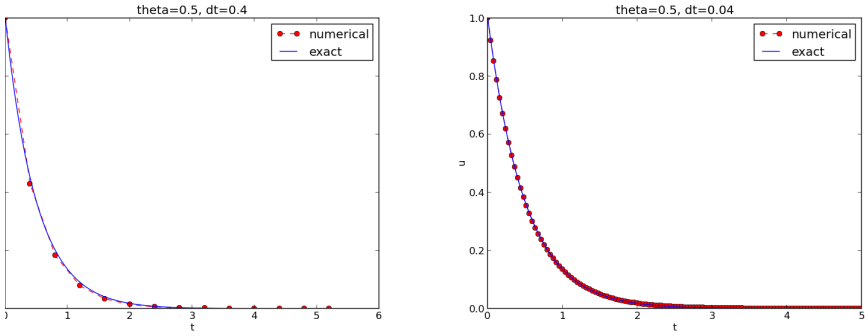


Figure 8: The Crank-Nicolson scheme for two values of the time step.

With Easyviz one can merge several plotting commands into a single one sing keyword arguments:

```
plot(t,   u,   'r--o',          # red dashes w/circles
     t_e, u_e, 'b-',            # blue line for exact sol.
     legend=['numerical', 'exact'],
     xlabel='t',
     ylabel='u',
     title='theta=%g, dt=%g' % (theta, dt),
     savefig='%s_%g.png' % (theta2name[theta], dt),
     show=True)
```

he `decay_plot_st.py`[12] file contains such a demo.

By default, Easyviz employs Matplotlib for plotting, but Gnuplot[13] and race[14] are viable alternatives:

[12] `http://tinyurl.com/jvzzcfn/decay/decay_plot_st.py`
[13] `http://www.gnuplot.info/`
[14] `http://plasma-gate.weizmann.ac.il/Grace/`


```
Terminal> python decay_plot_st.py --SCITOOLS_easyviz_backend gnupl
Terminal> python decay_plot_st.py --SCITOOLS_easyviz_backend grace
```

The backend used for creating plots (and numerous other options) permanently set in SciTool's configuration file.

All the Gnuplot windows are launched without any need to kill on the next one pops up (as is the case with Matplotlib) and one can press 'q' anywhere in a plot window to kill it. Another advantage of Gnuplo automatic choice of sensible and distinguishable line types in black-an PDF and PostScript files.

Regarding functionality for annotating plots with title, labels on t legends, etc., we refer to the documentation of Matplotlib and SciTools f detailed information on the syntax. The hope is that the programming explained so far suffices for understanding the code and learning more combination of the forthcoming examples and other resources such as bo web pages.

Test the understanding.
Exercise 2 asks you to implement a solver for a problem that is sli different from the one above. You may use the `solver` and `exp` functions explained above as a starting point. Apply the new solv Exercise 3.

2.6 Creating command-line interfaces

It is good programming practice to let programs read input from the use than require the user to edit the source code when trying out new values parameters. Reading input from the command line is a simple and flexi of interacting with the user. Python stores all the command-line argun the list `sys.argv`, and there are, in principle, two ways of programmi command-line arguments in Python:

- Decide upon a sequence of parameters on the command line a their values directly from the `sys.argv[1:]` list (`sys.argv[0]` is program name).
- Use option-value pairs (`--option value`) on the command line to default values of input parameters, and utilize the `argparse.Argume` tool to interact with the command line.

Both strategies will be illustrated next.

Reading a sequence of command-line arguments. The `decay_pl` `py`[15] program needs the following input data: I, a, T, an option to turn on or off (`makeplot`), and a list of Δt values.

[15] `http://tinyurl.com/jvzzcfn/decay/decay_plot_mpl.py`

The simplest way of reading this input from the command line is to say that ıe first four command-line arguments correspond to the first four points in the st above, in that order, and that the rest of the command-line arguments are ıe Δt values. The input given for `makeplot` can be a string among `'on'`, `'off'`, `True'`, and `'False'`. The code for reading this input is most conveniently put ı a function:

```
import sys

def read_command_line():
    if len(sys.argv) < 6:
        print 'Usage: %s I a T on/off dt1 dt2 dt3 ...' % \
              sys.argv[0]; sys.exit(1)  # abort

    I = float(sys.argv[1])
    a = float(sys.argv[2])
    T = float(sys.argv[3])
    makeplot = sys.argv[4] in ('on', 'True')
    dt_values = [float(arg) for arg in sys.argv[5:]]

    return I, a, T, makeplot, dt_values
```

One should note the following about the constructions in the program above:

- Everything on the command line ends up in a *string* in the list `sys.argv`. Explicit conversion to, e.g., a `float` object is required if the string as a number we want to compute with.
- The value of `makeplot` is determined from a boolean expression, which becomes `True` if the command-line argument is either `'on'` or `'True'`, and `False` otherwise.
- It is easy to build the list of Δt values: we simply run through the rest of the list, `sys.argv[5:]`, convert each command-line argument to `float`, and collect these `float` objects in a list, using the compact and convenient *list comprehension* syntax in Python.

he loops over θ and Δt values can be coded in a `main` function:

```
def main():
    I, a, T, makeplot, dt_values = read_command_line()
    for theta in 0, 0.5, 1:
        for dt in dt_values:
            E = explore(I, a, T, dt, theta, makeplot)
            print '%3.1f %6.2f: %12.3E' % (theta, dt, E)
```

he complete program can be found in `decay_cml.py`[16].

[16] `http://tinyurl.com/jvzzcfn/decay/decay_cml.py`

Working with an argument parser. Python's `ArgumentParser` to `argparse` module makes it easy to create a professional command-line i to any program. The documentation of `ArgumentParser`[17] demonstı versatile applications, so we shall here just list an example containir features. On the command line we want to specify option-value pairs and T, e.g., `--a 3.5 --I 2 --T 2`. Including `--makeplot` turns the and excluding this option turns the plot off. The Δt values can be g `--dt 1 0.5 0.25 0.1 0.01`. Each parameter must have a sensible defau so that we specify the option on the command line only when the defau is not suitable.

We introduce a function for defining the mentioned command-line c

```
def define_command_line_options():
    import argparse
    parser = argparse.ArgumentParser()
    parser.add_argument('--I', '--initial_condition', type=float,
                        default=1.0, help='initial condition, u(0
                        metavar='I')
    parser.add_argument('--a', type=float,
                        default=1.0, help='coefficient in ODE',
                        metavar='a')
    parser.add_argument('--T', '--stop_time', type=float,
                        default=1.0, help='end time of simulation
                        metavar='T')
    parser.add_argument('--makeplot', action='store_true',
                        help='display plot or not')
    parser.add_argument('--dt', '--time_step_values', type=float,
                        default=[1.0], help='time step values',
                        metavar='dt', nargs='+', dest='dt_values'
    return parser
```

Each command-line option is defined through the `parser.add_ar` method. Alternative options, like the short `--I` and the more explaining `--initial_condition` can be defined. Other arguments are `type` for the object type, a default value, and a help string, which gets printed if the co line argument `-h` or `--help` is included. The `metavar` argument spec value associated with the option when the help string is printed. For e the option for I has this help output:

```
Terminal> python decay_argparse.py -h
  ...
  --I I, --initial_condition I
                        initial condition, u(0)
  ...
```

The structure of this output is

```
    --I metavar, --initial_condition metavar
                          help-string
```

[17] `http://docs.python.org/library/argparse.html`

The `--makeplot` option is a pure flag without any value, implying a true alue if the flag is present and otherwise a false value. The `action='store_true'` akes an option for such a flag.

Finally, the `--dt` option demonstrates how to allow for more than one value eparated by blanks) through the `nargs='+'` keyword argument. After the ommand line is parsed, we get an object where the values of the options are ored as attributes. The attribute name is specified by the `dist` keyword rgument, which for the `--dt` option is `dt_values`. Without the `dest` argument, ne value of an option `--opt` is stored as the attribute `opt`.

The code below demonstrates how to read the command line and extract the alues for each option:

```
def read_command_line():
    parser = define_command_line_options()
    args = parser.parse_args()
    print 'I={}, a={}, T={}, makeplot={}, dt_values={}'.format(
        args.I, args.a, args.T, args.makeplot, args.dt_values)
    return args.I, args.a, args.T, args.makeplot, args.dt_values
```

The `main` function remains the same as in the `decay_cml.py` code based on eading from `sys.argv` directly. A complete program featuring the demo above f `ArgumentParser` appears in the file `decay_argparse.py`[18].

.7 Creating a graphical web user interface

he Python package Parampool[19] can be used to automatically generate a eb-based *graphical user interface* (GUI) for our simulation program. Although ne programming technique dramatically simplifies the efforts to create a GUI, ne forthcoming material on equipping our `decay_mod` module with a GUI is uite technical and of significantly less importance than knowing how to make a ommand-line interface (Section 2.6). There is no danger in jumping right to ection 2.8.

Making a compute function. The first step is to identify a function that erforms the computations and that takes the necessary input variables as rguments. This is called the *compute function* in Parampool terminology. We ay start with a copy of the basic file `decay_plot_mpl.py`[20], which has a `main` unction displayed in Section 2.5 for carrying out simulations and plotting for series of Δt values. Now we want to control and view the same experiments om a web GUI.

To tell Parampool what type of input data we have, we assign default values f the right type to all arguments in the main function and call it `main_GUI`:

```
def main_GUI(I=1.0, a=.2, T=4.0,
             dt_values=[1.25, 0.75, 0.5, 0.1],
             theta_values=[0, 0.5, 1]):
```

[18] http://tinyurl.com/jvzzcfn/decay/decay_argparse.py
[19] https://github.com/hplgit/parampool
[20] http://tinyurl.com/jvzzcfn/decay/decay_plot_mpl.py

The compute function must return the HTML code we want for dis the result in a web page. Here we want to show plots of the numerical ar solution for different methods and Δt values. The plots can be organi table with θ (methods) varying through the columns and Δt varying thro rows. Assume now that a new version of the `explore` function not only the error `E` but also HTML code containing the plot. Then we can w `main_GUI` function as

```
def main_GUI(I=1.0, a=.2, T=4.0,
             dt_values=[1.25, 0.75, 0.5, 0.1],
             theta_values=[0, 0.5, 1]):
    # Build HTML code for web page. Arrange plots in columns
    # corresponding to the theta values, with dt down the rows
    theta2name = {0: 'FE', 1: 'BE', 0.5: 'CN'}
    html_text = '<table>\n'
    for dt in dt_values:
        html_text += '<tr>\n'
        for theta in theta_values:
            E, html = explore(I, a, T, dt, theta, makeplot=True)
            html_text += """
<td>
<center><b>%s, dt=%g, error: %s</b></center><br>
%s
</td>
""" % (theta2name[theta], dt, E, html)
        html_text += '</tr>\n'
    html_text += '</table>\n'
    return html_text
```

Rather than creating plot files and showing the plot on the screen, version of the `explore` function makes a string with the PNG code of and embeds that string in HTML code. This action is conveniently pe by Parampool's `save_png_to_str` function:

```
import matplotlib.pyplot as plt
...
# plot
plt.plot(t, u, 'r-')
plt.xlabel('t')
plt.ylabel('u')
...
from parampool.utils import save_png_to_str
html_text = save_png_to_str(plt, plotwidth=400)
```

Note that we now write `plt.plot`, `plt.xlabel`, etc. The `html_text` s long and contains all the characters that build up the PNG file of the plot. The new `explore` function can make use of the above code snip return `html_text` along with `E`.

Generating the user interface. The web GUI is automatically ge by the following code, placed in a file `decay_GUI_generate.py`[21]

[21] http://tinyurl.com/jvzzcfn/decay/decay_GUI_generate.py

```
from parampool.generator.flask import generate
from decay_GUI import main
generate(main,
         output_controller='decay_GUI_controller.py',
         output_template='decay_GUI_view.py',
         output_model='decay_GUI_model.py')
```

unning the `decay_GUI_generate.py` program results in three new files whose ames are specified in the call to `generate`:

1. `decay_GUI_model.py` defines HTML widgets to be used to set input data in the web interface,
2. `templates/decay_GUI_views.py` defines the layout of the web page,
3. `decay_GUI_controller.py` runs the web application.

/e only need to run the last program, and there is no need to look into these les.

unning the web application. The web GUI is started by

```
erminal> python decay_GUI_controller.py
```

pen a web browser at the location `127.0.0.1:5000`. Input fields for `I`, `a`, , `dt_values`, and `theta_values` are presented. Setting the latter two to `1.25, 0.5]` and `[1, 0.5]`, respectively, and pressing *Compute* results in four lots, see Figure 9. With the techniques demonstrated here, one can easily create tailored web GUI for a particular type of application and use it to interactively xplore physical and numerical effects.

.8 Computing convergence rates

/e expect that the error E in the numerical solution is reduced if the mesh size t is decreased. More specifically, many numerical methods obey a power-law elation between E and Δt:

$$E = C\Delta t^r, \tag{47}$$

here C and r are (usually unknown) constants independent of Δt. The formula 7) is viewed as an asymptotic model valid for sufficiently small Δt. How small normally hard to estimate without doing numerical estimations of r.

The parameter r is known as the *convergence rate*. For example, if the onvergence rate is 2, halving Δt reduces the error by a factor of 4. Diminishing t then has a greater impact on the error compared with methods that have = 1. For a given value of r, we refer to the method as of r-th order. First- and econd-order methods are most common in scientific computing.

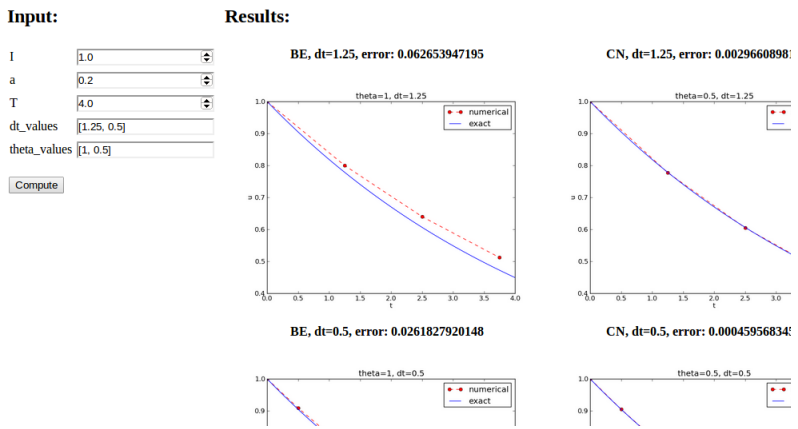


Figure 9: Automatically generated graphical web interface.

Estimating r. There are two alternative ways of estimating C and on a set of m simulations with corresponding pairs $(\Delta t_i, E_i)$, $i = 0, \ldots$ and $\Delta t_i < \Delta t_{i-1}$ (i.e., decreasing cell size).

1. Take the logarithm of (47), $\ln E = r \ln \Delta t + \ln C$, and fit a straigh the data points $(\Delta t_i, E_i)$, $i = 0, \ldots, m-1$.
2. Consider two consecutive experiments, $(\Delta t_i, E_i)$ and $(\Delta t_{i-1}, E_{i-}$ viding the equation $E_{i-1} = C\Delta t_{i-1}^r$ by $E_i = C\Delta t_i^r$ and solvi yields

$$r_{i-1} = \frac{\ln(E_{i-1}/E_i)}{\ln(\Delta t_{i-1}/\Delta t_i)}$$

for $i = 1, \ldots, m-1$.

The disadvantage of method 1 is that (47) might not be valid for the meshes (largest Δt values). Fitting a line to all the data points is then mis Method 2 computes convergence rates for pairs of experiments and allo see if the sequence r_i converges to some value as $i \to m-2$. The final $r_$ then be taken as the convergence rate. If the coarsest meshes have a c rate, the corresponding time steps are probably too large for (47) to l That is, those time steps lie outside the asymptotic range of Δt value the error behaves like (47).

Implementation. It is straightforward to extend the `main` functio program `decay_argparse.py` with statements for computing $r_0, r_1, \ldots$ from (47):

```
rom math import log

lef main():
    I, a, T, makeplot, dt_values = read_command_line()
    r = {}  # estimated convergence rates
    for theta in 0, 0.5, 1:
        E_values = []
        for dt in dt_values:
            E = explore(I, a, T, dt, theta, makeplot=False)
            E_values.append(E)

        # Compute convergence rates
        m = len(dt_values)
        r[theta] = [log(E_values[i-1]/E_values[i])/
                    log(dt_values[i-1]/dt_values[i])
                    for i in range(1, m, 1)]

    for theta in r:
        print '\nPairwise convergence rates for theta=%g:' % theta
        print ' '.join(['%.2f' % r_ for r_ in r[theta]])
    return r
```

he program containing this `main` function is called `decay_convrate.py`[22].

The `r` object is a *dictionary of lists*. The keys in this dictionary are the θ alues. For example, `r[1]` holds the list of the r_i values corresponding to $\theta = 1$. n the loop `for theta in r`, the loop variable `theta` takes on the values of the eys in the dictionary `r` (in an undetermined ordering). We could simply do a `rint r[theta]` inside the loop, but this would typically yield output of the onvergence rates with 16 decimals:

```
[1.331919482274763, 1.1488178494691532, ...]
```

Instead, we format each number with 2 decimals, using a list comprehension o turn the list of numbers, `r[theta]`, into a list of formatted strings. Then we oin these strings with a space in between to get a sequence of rates on one line n the terminal window. More generally, `d.join(list)` joins the strings in the st `list` to one string, with `d` as delimiter between `list[0]`, `list[1]`, etc.

Here is an example on the outcome of the convergence rate computations:

```
erminal> python decay_convrate.py --dt 0.5 0.25 0.1 0.05 0.025 0.01
..
airwise convergence rates for theta=0:
.33 1.15 1.07 1.03 1.02

airwise convergence rates for theta=0.5:
.14 2.07 2.03 2.01 2.01

airwise convergence rates for theta=1:
.98 0.99 0.99 1.00 1.00
```

[22] http://tinyurl.com/jvzzcfn/decay/decay_convrate.py

The Forward and Backward Euler methods seem to have an r valu stabilizes at 1, while the Crank-Nicolson seems to be a second-order with $r = 2$.

Very often, we have some theory that predicts what r is for a nu method. Various theoretical error measures for the θ-rule point to r $\theta = 0.5$ and $r = 1$ otherwise. The computed estimates of r are in ve agreement with these theoretical values.

Why convergence rates are important.
The strong practical application of computing convergence rates is fo fication: wrong convergence rates point to errors in the code, and c convergence rates brings evidence that the implementation is correct. rience shows that bugs in the code easily destroy the expected conver rate.

Debugging via convergence rates. Let us experiment with bugs the implication on the convergence rate. We may, for instance, forget to by `a` in the denominator in the updating formula for `u[n+1]`:

```
u[n+1] = (1 - (1-theta)*a*dt)/(1 + theta*dt)*u[n]
```

Running the same `decay_convrate.py` command as above gives the e convergence rates (!). Why? The reason is that we just specified the Δ are relied on default values for other parameters. The default value c Forgetting the factor `a` has then no effect. This example shows how imp is to avoid parameters that are 1 or 0 when verifying implementations. I the code `decay_v0.py` with $a = 2.1$ and $I = 0.1$ yields

```
Terminal> python decay_convrate.py --a 2.1 --I 0.1  \
          --dt 0.5 0.25 0.1 0.05 0.025 0.01
...
Pairwise convergence rates for theta=0:
1.49 1.18 1.07 1.04 1.02

Pairwise convergence rates for theta=0.5:
-1.42 -0.22 -0.07 -0.03 -0.01

Pairwise convergence rates for theta=1:
0.21 0.12 0.06 0.03 0.01
```

This time we see that the expected convergence rates for the Crank-N and Backward Euler methods are not obtained, while $r = 1$ for the I Euler method. The reason for correct rate in the latter case is that θ the wrong `theta*dt` term in the denominator vanishes anyway.

The error

```
ı[n+1] = ((1-theta)*a*dt)/(1 + theta*dt*a)*u[n]
```

ıanifests itself through wrong rates $r \approx 0$ for all three methods. About the ıme results arise from an erroneous initial condition, `u[0] = 1`, or wrong loop mits, `range(1,Nt)`. It seems that in this simple problem, most bugs we can ıink of are detected by the convergence rate test, provided the values of the ıput data do not hide the bug.

A `verify_convergence_rate` function could compute the dictionary of list ła `main` and check if the final rate estimates (r_{m-2}) are sufficiently close to the xpected ones. A tolerance of 0.1 seems appropriate, given the uncertainty in stimating r:

```
lef verify_convergence_rate():
    r = main()
    tol = 0.1
    expected_rates = {0: 1, 1: 1, 0.5: 2}
    for theta in r:
        r_final = r[theta][-1]
        diff = abs(expected_rates[theta] - r_final)
        if diff > tol:
            return False
    return True  # all tests passed
```

Ve remark that `r[theta]` is a list and the last element in any list can be xtracted by the index `-1`.

.9 Memory-saving implementation

he computer memory requirements of our implementations so far consists ıainly of the `u` and `t` arrays, both of length $N_t + 1$, plus some other temporary rrays that Python needs for intermediate results if we do array arithmetics ı our program (e.g., `I*exp(-a*t)` needs to store `a*t` before `-` can be applied) it and then `exp`). The extremely modest storage requirements of simple)DE problems put no restrictions on the formulations of the algorithm and nplementation. Nevertheless, when the methods for ODEs used here are applied) three-dimensional partial differential equation (PDE) problems, memory ;orage requirements suddenly become an issue.

The PDE counterpart to our model problem $u' = -a$ is a diffusion equation $_t = a\nabla^2 u$ posed on a space-time domain. The discrete representation of this omain may in 3D be a spatial mesh of M^3 points and a time mesh of N_t oints. A typical desired value for M is 100 in many applications, or even 1000. toring all the computed u values, like we have done in the programs so far, emands storage of some arrays of size $M^3 N_t$, giving a factor of M^3 larger ;orage demands compared to our ODE programs. Each real number in the rray for u requires 8 bytes (b) of storage. With $M = 100$ and $N_t = 1000$, ıere is a storage demand of $(10^3)^3 \cdot 1000 \cdot 8 = 8$ Gb for the solution array. ortunately, we can usually get rid of the N_t factor, resulting in 8 Mb of storage. elow we explain how this is done, and the technique is almost always applied ı implementations of PDE problems.

Let us critically evaluate how much we really need to store in the cor memory in our implementation of the θ method. To compute a new u^{n+} need is u^n. This implies that the previous $u^{n-1}, u^{n-2}, \ldots, u^0$ values do r to be stored in an array, although this is convenient for plotting and data in the program. Instead of the `u` array we can work with two variables numbers, `u` and `u_1`, representing u^{n+1} and u^n in the algorithm, resp At each time level, we update `u` from `u_1` and then set `u_1 = u` so t computed u^{n+1} value becomes the "previous" value u^n at the next tir The downside is that we cannot plot the solution after the simulation since only the last two numbers are available. The remedy is to store co values in a file and use the file for visualizing the solution later.

We have implemented this memory saving idea in the file `decay_m py`[23], which is a merge of the `decay_plot_mpl.py`[24] and `decay_argpar` programs, using module prefixes `np` for `numpy` and `plt` for `matplotlib.`

The following function demonstrates how we work with the two mos values of the unknown:

```
def solver_memsave(I, a, T, dt, theta, filename='sol.dat'):
    """
    Solve u'=-a*u, u(0)=I, for t in (0,T] with steps of dt.
    Minimum use of memory. The solution is stored in a file
    (with name filename) for later plotting.
    """
    dt = float(dt)         # avoid integer division
    Nt = int(round(T/dt))  # no of intervals

    outfile = open(filename, 'w')
    # u: time level n+1, u_1: time level n
    t = 0
    u_1 = I
    outfile.write('%.16E  %.16E\n' % (t, u_1))
    for n in range(1, Nt+1):
        u = (1 - (1-theta)*a*dt)/(1 + theta*dt*a)*u_1
        u_1 = u
        t += dt
        outfile.write('%.16E  %.16E\n' % (t, u))
    outfile.close()
    return u, t
```

This code snippet serves as a quick introduction to file writing in Python. the data in the file into arrays `t` and `u` are done by the function

```
def read_file(filename='sol.dat'):
    infile = open(filename, 'r')
    u = [];  t = []
    for line in infile:
        words = line.split()
        if len(words) != 2:
            print 'Found more than two numbers on a line!', words
            sys.exit(1)  # abort
```

[23] http://tinyurl.com/jvzzcfn/decay/decay_memsave.py
[24] http://tinyurl.com/jvzzcfn/decay/decay_plot_mpl.py
[25] http://tinyurl.com/jvzzcfn/decay/decay_argparse.py

```
        t.append(float(words[0]))
        u.append(float(words[1]))
    return np.array(t), np.array(u)
```

This type of file with numbers in rows and columns is very common, and `numpy` as a function `loadtxt` which loads such tabular data into a two-dimensional rray, say with name `data`. The number in row `i` and column `j` is then `data[i,j]`. he whole column number `j` can be extracted by `data[:,j]`. A version of `ead_file` using `np.loadtxt` reads

```
lef read_file_numpy(filename='sol.dat'):
    data = np.loadtxt(filename)
    t = data[:,0]
    u = data[:,1]
    return t, u
```

The present counterpart to the `explore` function from `decay_plot_mpl.py`[26] ıust run `solver_memsave` and then load data from file before we can compute ıe error measure and make the plot:

```
lef explore(I, a, T, dt, theta=0.5, makeplot=True):
    filename = 'u.dat'
    u, t = solver_memsave(I, a, T, dt, theta, filename)

    t, u = read_file(filename)
    u_e = exact_solution(t, I, a)
    e = u_e - u
    E = np.sqrt(dt*np.sum(e**2))
    if makeplot:
        plt.figure()
        ...
```

The `decay_memsave.py`[27] file also includes command-line options `--I`, `--a`, `-T`, `--dt`, `--theta`, and `--makeplot` for controlling input parameters and making single run. For example,

```
erminal> python decay_memsave.py --T 10 --theta 1 --dt 2
```

esults in the output

```
I=1.0, a=1.0, T=10.0, makeplot=True, theta=1.0, dt=2.0
theta=1.0 dt=2 Error=3.136E-01
```

Software engineering

[26] `http://tinyurl.com/jvzzcfn/decay/decay_plot_mpl.py`
[27] `http://tinyurl.com/jvzzcfn/decay/decay_memsave.py`

Goal.
Efficient use of differential equation models requires software that is to test and flexible for setting up extensive numerical experiments. section introduces three important concepts:

- Modules
- Testing frameworks
- Implementation with classes

The concepts are introduced using the differential equation problem $-au$, $u(0) = I$, as example.

3.1 Making a module

The DRY principle.
The previous sections have outlined numerous different programs, them having their own copy of the `solver` function. Such copies c same piece of code is against the important *Don't Repeat Yourself* (principle in programming. If we want to change the `solver` function should be *one and only one* place where the change needs to be perfor

To clean up the repetitive code snippets scattered among the `deca` files, we start by collecting the various functions we want to keep for th in one file, now called `decay_mod.py`[28] (`mod` stands for "module"). The f functions are copied to this file:

- `solver` for computing the numerical solution
- `verify_three_steps` for verifying the first three solution points hand calculations
- `verify_discrete_solution` for verifying the entire computed s against an exact formula for the numerical solution
- `explore` for computing and plotting the solution
- `define_command_line_options` for defining option-value pairs command line
- `read_command_line` for reading input from the command line, tended to work both with `sys.argv` directly and with an `Argument` object
- `main` for running experiments with $\theta = 0, 0.5, 1$ and a series of Δ and computing convergence rates

[28] `http://tinyurl.com/jvzzcfn/decay/decay_mod.py`

- `main_GUI` for doing the same as the `main` function, but modified for automatic GUI generation
- `verify_convergence_rate` for verifying the computed convergence rates against the theoretically expected values

/e use Matplotlib for plotting. A sketch of the `decay_mod.py` file, with complete ersions of the modified functions, looks as follows:

```
rom numpy import *
rom matplotlib.pyplot import *
import sys

def solver(I, a, T, dt, theta):
    ...

def verify_three_steps():
    ...

def verify_exact_discrete_solution():
    ...

def exact_solution(t, I, a):
    ...

def explore(I, a, T, dt, theta=0.5, makeplot=True):
    ...

def define_command_line_options():
    ...

def read_command_line(use_argparse=True):
    if use_argparse:
        parser = define_command_line_options()
        args = parser.parse_args()
        print 'I={}, a={}, makeplot={}, dt_values={}'.format(
            args.I, args.a, args.makeplot, args.dt_values)
        return args.I, args.a, args.makeplot, args.dt_values
    else:
        if len(sys.argv) < 6:
            print 'Usage: %s I a on/off dt1 dt2 dt3 ...' % \
                  sys.argv[0]; sys.exit(1)

        I = float(sys.argv[1])
        a = float(sys.argv[2])
        T = float(sys.argv[3])
        makeplot = sys.argv[4] in ('on', 'True')
        dt_values = [float(arg) for arg in sys.argv[5:]]

        return I, a, makeplot, dt_values

def main():
    ...
```

This `decay_mod.py` file is already a module such that we can import desired unctions in other programs. For example, we can in a file do


```
from decay_mod import solver
u, t = solver(I=1.0, a=3.0, T=3, dt=0.01, theta=0.5)
```

However, it should also be possible to both use `decay_mod.py` as a *and* execute the file as a program that runs `main()`. This is accompli ending the file with a *test block*:

```
if __name__ == '__main__':
    main()
```

When `decay_mod.py` is used as a module, `__name__` equals the modu `decay_mod`, while `__name__` equals `'__main__'` when the file is run a gram. Optionally, we could run the verification tests if the word `ve` present on the command line and `verify_convergence_rate` could be `verify_rates` is found on the command line. The `verify_rates` argume be removed before we read parameter values from the command line, ot the `read_command_line` function (called by `main`) will not work prope

```
if __name__ == '__main__':
    if 'verify' in sys.argv:
        if verify_three_steps() and verify_discrete_solution():
            pass # ok
        else:
            print 'Bug in the implementation!'
    elif 'verify_rates' in sys.argv:
        sys.argv.remove('verify_rates')
        if not '--dt' in sys.argv:
            print 'Must assign several dt values'
            sys.exit(1)  # abort
        if verify_convergence_rate():
            pass
        else:
            print 'Bug in the implementation!'
    else:
        # Perform simulations
        main()
```

3.2 Prefixing imported functions by the module na

Import statements of the form `from module import *` import functi variables in `module.py` into the current file. For example, when doing

```
from numpy import *
from matplotlib.pyplot import *
```

we get mathematical functions like `sin` and `exp` as well as MATLA functions like `linspace` and `plot`, which can be called by these well-know Unfortunately, it sometimes becomes confusing to know where a pa function comes from. Is it from `numpy`? Or `matplotlib.pyplot`? Or i own function?

An alternative import is

```
import numpy
import matplotlib.pyplot
```

nd such imports require functions to be prefixed by the module name, e.g.,

```
 = numpy.linspace(0, T, Nt+1)
_e = I*numpy.exp(-a*t)
atplotlib.pyplot.plot(t, u_e)
```

his is normally regarded as a better habit because it is explicitly stated from hich module a function comes from.

The modules `numpy` and `matplotlib.pyplot` are so frequently used, and eir full names quite tedious to write, so two standard abbreviations have volved in the Python scientific computing community:

```
import numpy as np
import matplotlib.pyplot as plt

 = np.linspace(0, T, Nt+1)
_e = I*np.exp(-a*t)
lt.plot(t, u_e)
```

version of the `decay_mod` module where we use the `np` and `plt` prefixes is ound in the file decay_mod_prefix.py[29].

The downside of prefixing functions by the module name is that mathematical xpressions like $e^{-at}\sin(2\pi t)$ get cluttered with module names,

```
umpy.exp(-a*t)*numpy.sin(2(numpy.pi*t)
# or
np.exp(-a*t)*np.sin(2*np.pi*t)
```

uch an expression looks like `exp(-a*t)*sin(2*pi*t)` in most other programming languages. Similarly, `np.linspace` and `plt.plot` look less familiar to people who are used to MATLAB and who have not adopted Python's prefix style. Whether to do `from module import *` or `import module` depends on personal aste and the problem at hand. In these writings we use `from module import` shorter programs where similarity with MATLAB could be an advantage, and here a one-to-one correspondence between mathematical formulas and Python xpressions is important. The style `import module` is preferred inside Python odules (see Exercise 11 for a demonstration).

.3 Doctests

We have emphasized how important it is to be able to run tests in the program at ny time. This was solved by calling various `verify*` functions in the previous xamples. However, there exists well-established procedures and corresponding ools for automating the execution of tests. We shall briefly demonstrate two nportant techniques: *doctest* and *unit testing*. The corresponding files are the odules decay_mod_doctest.py[30] and decay_mod_nosetest.py[31].

[29] http://tinyurl.com/jvzzcfn/decay/decay_mod_prefix.py
[30] http://tinyurl.com/jvzzcfn/decay/decay_mod_doctest.py
[31] http://tinyurl.com/jvzzcfn/decay/decay_mod_nosetest.py

A doc string (the first string after the function header) is used to dc the purpose of functions and their arguments. Very often it is instru include an example on how to use the function. Interactive example Python shell are most illustrative as we can see the output resulting from calls. For example, we can in the `solver` function include an example o this function and printing the computed `u` and `t` arrays:

```
def solver(I, a, T, dt, theta):
    """
    Solve u'=-a*u, u(0)=I, for t in (0,T] with steps of dt.


    >>> u, t = solver(I=0.8, a=1.2, T=4, dt=0.5, theta=0.5)
    >>> for t_n, u_n in zip(t, u):
    ...     print 't=%.1f, u=%.14f' % (t_n, u_n)
    t=0.0, u=0.80000000000000
    t=0.5, u=0.43076923076923
    t=1.0, u=0.23195266272189
    t=1.5, u=0.12489758761948
    t=2.0, u=0.06725254717972
    t=2.5, u=0.03621291001985
    t=3.0, u=0.01949925924146
    t=3.5, u=0.01049960113002
    t=4.0, u=0.00565363137770
    """
    ...
```

When such interactive demonstrations are inserted in doc strings, P doctest[32] module can be used to automate running all commands in int sessions and compare new output with the output appearing in the do All we have to do in the current example is to write

```
Terminal> python -m doctest decay_mod_doctest.py
```

This command imports the `doctest` module, which runs all tests. No ad command-line argument is allowed when running doctests. If any test f problem is reported, e.g.,

```
Terminal> python -m doctest decay_mod_doctest.py
**********************************************************
File "decay_mod_doctest.py", line 12, in decay_mod_doctest....
Failed example:
    for t_n, u_n in zip(t, u):
        print 't=%.1f, u=%.14f' % (t_n, u_n)
Expected:
    t=0.0, u=0.80000000000000
    t=0.5, u=0.43076923076923
    t=1.0, u=0.23195266272189
    t=1.5, u=0.12489758761948
    t=2.0, u=0.06725254717972
Got:
    t=0.0, u=0.80000000000000
```

[32] http://docs.python.org/library/doctest.html

```
   t=0.5, u=0.43076923076923
   t=1.0, u=0.23195266272189
   t=1.5, u=0.12489758761948
   t=2.0, u=0.06725254718756
**********************************************************
 items had failures:
  1 of   2 in decay_mod_doctest.solver
**Test Failed*** 1 failures.
```

Note that in the output of `t` and `u` we write `u` with 14 digits. Writing all 16 igits is not a good idea: if the tests are run on different hardware, round-off rors might be different, and the `doctest` module detects that the numbers are ot precisely the same and reports failures. In the present application, where $< u(t) \leq 0.8$, we expect round-off errors to be of size 10^{-16}, so comparing 15 igits would probably be reliable, but we compare 14 to be on the safe side.

Doctests are highly encouraged as they do two things: 1) demonstrate how a nction is used and 2) test that the function works.

Here is an example on a doctest in the `explore` function:

```
def explore(I, a, T, dt, theta=0.5, makeplot=True):
    """
    Run a case with the solver, compute error measure,
    and plot the numerical and exact solutions (if makeplot=True).

    >>> for theta in 0, 0.5, 1:
    ...     E = explore(I=1.9, a=2.1, T=5, dt=0.1, theta=theta,
    ...                 makeplot=False)
    ...     print '%.10E' % E
    ...
    7.3565079236E-02
    2.4183893110E-03
    6.5013039886E-02
    """
    ...
```

his time we limit the output to 10 digits.

Caution.
Doctests requires careful coding if they use command-line input or print results to the terminal window. Command-line input must be simulated by filling `sys.argv` correctly, e.g., `sys.argv = '--I 1.0 --a 5'.split`. The output lines of print statements must be copied exactly as they appear when running the statements in an interactive Python shell.

.4 Unit testing with nose

he unit testing technique consists of identifying small units of code, usually nctions (or classes), and write one or more tests for each unit. One test should, leally, not depend on the outcome of other tests. For example, the doctest ı function `solver` is a unit test, and the doctest in function `explore` as well, ut the latter depends on a working `solver`. Putting the error computation

and plotting in `explore` in two separate functions would allow independ tests. In this way, the design of unit tests impacts the design of functio recommended practice is actually to design and write the unit tests fi *then* implement the functions!

In scientific computing it is not always obvious how to best perfo testing. The units is naturally larger than in non-scientific software. Ve the solution procedure of a mathematical problem identifies a unit.

Basic use of nose. The `nose` package is a versatile tool for implement tests in Python. Here is a short explanation of the usage of nose:

1. Implement tests in functions with names starting with `test_`. Su tions cannot have any arguments.
2. The test functions perform assertions on computed results using functions from the `nose.tools` module.
3. The test functions can be in the source code files or be collected in s files with names `test*.py`.

Here comes a very simple illustration of the three points. Assume that this function in a module `mymod`:

```
def double(n):
    return 2*n
```

Either in this file, or in a separate file `test_mymod.py`, we implemer function whose purpose is to test that the function `double` works as int

```
import nose.tools as nt

def test_double():
    result = double(4)
    nt.assert_equal(result, 8)
```

Notice that `test_double` has no arguments. We need to do an `import` or `from mymod import double` if this test resides in a separate file. Ru

```
Terminal> nosetests -s mymod
```

makes the `nose` tool run all functions with names matching `test` `mymod.py`. Alternatively, if the test functions are in some `test_mymod` we can just write `nosetests -s`. The nose tool will then look for all fi names mathching `test*.py` and run all functions `test_*()` in these fil

When you have nose tests in separate test files with names `test*` common to collect these files in a subdirectory `tests`, or `*_tests` if y several test subdirectories. Running `nosetests -s` will then recursively all `tests` and `*_tests` subdirectories and run all functions `test_*()` in

`est_*.py` in these directories. Just one command can then launch a series of sts in a directory tree!

An example of a `tests` directory with different types of `test*.py` files are ound in src/decay/tests[33]. Note that these perform imports of modules in the arent directory. These imports works well because the tests are supposed to be n by `nosetests -s` executed in the parent directory (`decay`).

Tip.
The `-s` option to `nosetests` assures that any print statement in the `test_*` functions appears in the output. Without this option, `nosetests` suppressed whatever the tests writes to the terminal window (standard output). Such behavior is annoying, especially when developing and testing tests.

The number of failed tests and their details are reported, or an `OK` is printed all tests passed.

The advantage with the `nose` package is two-fold:

1. tests are written and collected in a structured way, and
2. large collections of tests, scattered throughout a tree of directories, can be executed with one command `nosetests -s`.

lternative assert statements. In case the `nt.assert_equal` function finds at the two arguments are equal, the test is a success, otherwise it is a failure nd an exception of type `AssertionError` is raised. The particular exception is e indicator that a test has failed.

Instead of calling the convenience function `nt.assert_equal`, we can use ython's plain `assert` statement, which tests if a boolean expression is true and ises an `AssertionError` otherwise. Here, the statement is `assert result == 8`.

A completely manual alternative is to explicitly raise an `AssertionError` xception if the computed result is wrong:

```
if result != 8:
    raise AssertionError()
```

pplying nose. Let us illustrate how to use the `nose` tool for testing key nctions in the `decay_mod` module. Or more precisely, the module is called ecay_mod_unittest with all the `verify*` functions removed as these now are utdated by the unit tests.

We design three unit tests:

1. A comparison between the computed u^n values and the exact discrete solution.
2. A comparison between the computed u^n values and precomputed, verified reference values.
3. A comparison between observed and expected convergence rates.

[33] http://tinyurl.com/jvzzcfn/decay/tests

These tests follow very closely the code in the previously shown `verify*` fu We start with comparing u^n, as computed by the function `solver`, to the for the exact discrete solution:

```
import nose.tools as nt
import decay_mod_unittest as decay_mod
import numpy as np

def exact_discrete_solution(n, I, a, theta, dt):
    """Return exact discrete solution of the theta scheme."""
    dt = float(dt)  # avoid integer division
    factor = (1 - (1-theta)*a*dt)/(1 + theta*dt*a)
    return I*factor**n

def test_exact_discrete_solution():
    """
    Compare result from solver against
    formula for the discrete solution.
    """
    theta = 0.8; a = 2; I = 0.1; dt = 0.8
    N = int(8/dt)  # no of steps
    u, t = decay_mod.solver(I=I, a=a, T=N*dt, dt=dt, theta=theta)
    u_de = np.array([exact_discrete_solution(n, I, a, theta, dt)
                     for n in range(N+1)])
    diff = np.abs(u_de - u).max()
    nt.assert_almost_equal(diff, 0, delta=1E-14)
```

The `nt.assert_almost_equal` is the relevant function for compar real numbers. The `delta` argument specifies a tolerance for the com Alternatively, one can specify a `places` argument for the number of places to be used in the comparison.

After having carefully verified the implementation, we may store c computed numbers in the test program or in files for use in future tests. an example on how the outcome from the `solver` function can be com what is considered to be correct results:

```
def test_solver():
    """
    Compare result from solver against
    precomputed arrays for theta=0, 0.5, 1.
    """
    I=0.8; a=1.2; T=4; dt=0.5  # fixed parameters
    precomputed = {
        't': np.array([ 0. ,  0.5,  1. ,  1.5,  2. ,  2.5,
                        3. ,  3.5,  4. ]),
        0.5: np.array(
            [ 0.8        ,  0.43076923,  0.23195266, 0.12489759,
              0.06725255,  0.03621291,  0.01949926, 0.0104996 ,
              0.00565363]),
        0: np.array(
            [  8.00000000e-01,   3.20000000e-01,
               1.28000000e-01,   5.12000000e-02,
               2.04800000e-02,   8.19200000e-03,
               3.27680000e-03,   1.31072000e-03,
```

```
             5.24288000e-04]),
      1: np.array(
          [ 0.8       ,  0.5       ,  0.3125    ,  0.1953125 ,
            0.12207031,  0.07629395,  0.04768372,  0.02980232,
            0.01862645]),
      }
  for theta in 0, 0.5, 1:
      u, t = decay_mod.solver(I, a, T, dt, theta=theta)
      diff = np.abs(u - precomputed[theta]).max()
      # Precomputed numbers are known to 8 decimal places
      nt.assert_almost_equal(diff, 0, places=8,
                             msg='theta=%s' % theta)
```

he `precomputed` object is a dictionary with four keys: `'t'` for the time mesh, nd three θ values for u^n solutions corresponding to $\theta = 0, 0.5, 1$.

Testing for special type of input data that may cause trouble constitutes common way of constructing unit tests. For example, the updating formula or u^{n+1} may be incorrectly evaluated because of unintended integer divisions. /ith

```
heta = 1; a = 1; I = 1; dt = 2
```

ıe nominator and denominator in the updating expression,

```
(1 - (1-theta)*a*dt)
(1 + theta*dt*a)
```

valuate to 1 and 3, respectively, and the fraction `1/3` will call up integer division nd consequently lead to `u[n+1]=0`. We construct a unit test to make sure `olver` is smart enough to avoid this problem:

```
def test_potential_integer_division():
    """Choose variables that can trigger integer division."""
    theta = 1; a = 1; I = 1; dt = 2
    N = 4
    u, t = decay_mod.solver(I=I, a=a, T=N*dt, dt=dt, theta=theta)
    u_de = np.array([exact_discrete_solution(n, I, a, theta, dt)
                     for n in range(N+1)])
    diff = np.abs(u_de - u).max()
    nt.assert_almost_equal(diff, 0, delta=1E-14)
```

The final test is to see that the convergence rates corresponding to $\theta = 0, 0.5, 1$ re 1, 2, and 1, respectively:

```
def test_convergence_rates():
    """Compare empirical convergence rates to exact ones."""
    # Set command-line arguments directly in sys.argv
    import sys
    sys.argv[1:] = '--I 0.8 --a 2.1 --T 5 '\
                   '--dt 0.4 0.2 0.1 0.05 0.025'.split()
    r = decay_mod.main()
    for theta in r:
        nt.assert_true(r[theta])  # check for non-empty list
```



```
    expected_rates = {0: 1, 1: 1, 0.5: 2}
    for theta in r:
        r_final = r[theta][-1]
        # Compare to 1 decimal place
        nt.assert_almost_equal(expected_rates[theta], r_final,
                               places=1, msg='theta=%s' % theta)
```

Nothing more is needed in the `test_decay_nose.py`[34] file where t reside. Running `nosetests -s` will report `Ran 3 tests` and an `OK` for Everytime we modify the `decay_mod_unittest` module we can run `nos` to quickly see if the edits have any impact on the verification tests.

Installation of nose. The `nose` package does not come with a st Python distribution and must therefore be installed separately. The proc standard and described on Nose's web pages[35]. On Debian-based Linux the command is `sudo apt-get install python-nose`, and with MacP run `sudo port install py27-nose`.

Using nose to test modules with doctests. Assume that `mod` is tl of some module that contains doctests. We may let `nose` run these docte report errors in the standard way using the code set-up

```
import doctest
import mod

def test_mod():
    failure_count, test_count = doctest.testmod(m=mod)
    nt.assert_equal(failure_count, 0,
                    msg='%d tests out of %d failed' %
                    (failure_count, test_count))
```

The call to `doctest.testmod` runs all doctests in the module file `mod` returns the number of failures (`failure_count`) and the total number (`test_count`). A real example is found in the file `test_decay_doctest`

3.5 Classical class-based unit testing

The classical way of implementing unit tests derives from the JUnit tool where all tests are methods in a class for testing. Python comes with a `unittest` for doing this type of unit tests. While `nose` allows simple func unit tests, `unittest` requires deriving a class `Test*` from `unittest.Te` and implementing each test as methods with names `test_*` in that strongly recommend to use `nose` over `unittest`, because it is much sim more convenient, but class-based unit testing is a very classical subj computational scientists should have some knowledge about. That is why introduction to `unittest` is included below.

[34] `http://tinyurl.com/jvzzcfn/decay/tests/test_decay_nose.py`
[35] `http://nose.readthedocs.org/en/latest/`
[36] `http://tinyurl.com/jvzzcfn/decay/tests/test_decay_doctest.py`

Basic use of unittest. We apply the `double` function in the `mymod` module introduced in the previous section as example. Unit testing with the aid of the `unittest` module consists of writing a file `test_mymod.py` with the content

```
import unittest
import mymod

class TestMyCode(unittest.TestCase):
    def test_double(self):
        result = mymod.double(4)
        self.assertEqual(result, 8)

if __name__ == '__main__':
    unittest.main()
```

The test is run by executing the test file `test_mymod.py` as a standard Python program. There is no support in `unittest` for automatically locating and running all tests in all test files in a directory tree.

Those who have experience with object-oriented programming will see that the difference between using `unittest` and `nose` is minor.

Demonstration of unittest. The same tests as shown for the nose framework are reimplemented with the `TestCase` classes in the file `test_decay_unittest.py`[37]. The tests are identical, the only difference being that with `unittest` we must write the tests as methods in a class and the assert functions have slightly different names.

```
import unittest
import decay_mod_unittest as decay
import numpy as np

def exact_discrete_solution(n, I, a, theta, dt):
    factor = (1 - (1-theta)*a*dt)/(1 + theta*dt*a)
    return I*factor**n

class TestDecay(unittest.TestCase):

    def test_exact_discrete_solution(self):
        ...
        diff = np.abs(u_de - u).max()
        self.assertAlmostEqual(diff, 0, delta=1E-14)

    def test_solver(self):
        ...
        for theta in 0, 0.5, 1:
            ...
            self.assertAlmostEqual(diff, 0, places=8,
                                   msg='theta=%s' % theta)

    def test_potential_integer_division():
        ...
        self.assertAlmostEqual(diff, 0, delta=1E-14)

    def test_convergence_rates(self):
        ...
        for theta in r:
            ...
            self.assertAlmostEqual(...)

if __name__ == '__main__':
    unittest.main()
```

[37] http://tinyurl.com/jvzzcfn/decay/tests/test_decay_nose.py

3.6 Implementing simple problem and solver classes

The θ-rule was compactly and conveniently implemented in a function in Section 2.1. In more complicated problems it might be beneficial classes and introduce a class `Problem` to hold the definition of the problem, a class `Solver` to hold the data and methods needed to numerically solve the problem, and a class `Visualizer` to make plots. This idea will be illustrated, resulting in code that represents an alternative to the `solver` and `explore` functions found in the `decay_mod` module.

Explaining the details of class programming in Python is considered beyond the scope of this text. Readers who are unfamiliar with Python class programming should first consult one of the many electronic Python tutorials or textbooks to come up to speed with concepts and syntax of Python classes before reading on. The author has a gentle introduction to class programming for scientific applications in [**?**], see Chapter 7 and 9 and Appendix E. Other useful references are

- The Python Tutorial: http://docs.python.org/2/tutorial/classes.html
- Wiki book on Python Programming: http://en.wikibooks.org/wiki/Python_Programming/Classes
- tutorialspoint.com: http://www.tutorialspoint.com/python/python_classes_objects.htm

The problem class. The purpose of the problem class is to store all information about the mathematical model. This usually means all the physical parameters in the problem. In the current example with exponential decay we may also add the exact solution of the ODE to the problem class. The simplest form of a problem class is therefore

```
from numpy import exp

class Problem:
    def __init__(self, I=1, a=1, T=10):
        self.T, self.I, self.a = I, float(a), T

    def exact_solution(self, t):
        I, a = self.I, self.a
        return I*exp(-a*t)
```

We could in the `exact_solution` method have written `self.I*exp(-self.a*t)`, but using local variables `I` and `a` allows the formula `I*exp(-a*t)` which looks closer to the mathematical expression Ie^{-at}. This is not an important issue with the current compact formula, but is beneficial in more complicated problems with longer formulas to obtain the closest possible relationship between code and mathematics. My coding style is to strip off the `self` prefix when the code expresses mathematical formulas.

The class data can be set either as arguments in the constructor or at any time later, e.g.,

```
problem = Problem(T=5)
problem.T = 8
problem.dt = 1.5
```

Some programmers prefer `set` and `get` functions for setting and getting data in classes, often implemented via *properties* in Python, but I consider that overkill when we just have a few data items in a class.)

It would be convenient if class `Problem` could also initialize the data from the command line. To this end, we add a method for defining a set of command-line options and a method that sets the local attributes equal to what was found on the command line. The default values associated with the command-line options are taken as the values provided to the constructor. Class `Problem` now becomes

```
class Problem:
    def __init__(self, I=1, a=1, T=10):
        self.T, self.I, self.a = I, float(a), T

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

        parser.add_argument(
            '--I', '--initial_condition', type=float,
            default=self.I, help='initial condition, u(0)',
            metavar='I')
        parser.add_argument(
            '--a', type=float, default=self.a,
            help='coefficient in ODE', metavar='a')
        parser.add_argument(
            '--T', '--stop_time', type=float, default=self.T,
            help='end time of simulation', metavar='T')
        return parser

    def init_from_command_line(self, args):
        self.I, self.a, self.T = args.I, args.a, args.T

    def exact_solution(self, t):
        I, a = self.I, self.a
        return I*exp(-a*t)
```

Observe that if the user already has an `ArgumentParser` object it can be supplied, but if she does not have any, class `Problem` makes one. Python's `None` object is used to indicate that a variable is not initialized with a proper value.

The solver class. The solver class stores data related to the numerical method and provides a function `solve` for solving the problem. A problem must be given to the constructor so that the solver can easily look up data. In the present example, the data related to the numerical solution consists of Δt and θ. We add, as in the problem class, functionality for Δt and θ from the command line:

```
class Solver:
    def __init__(self, problem, dt=0.1, theta=0.5):
        self.problem = problem
        self.dt, self.theta = float(dt), theta

    def define_command_line_options(self, parser):
        parser.add_argument(
            '--dt', '--time_step_value', type=float,
            default=0.5, help='time step value', metavar='dt')
        parser.add_argument(
            '--theta', type=float, default=0.5,
            help='time discretization parameter', metavar='dt')
        return parser

    def init_from_command_line(self, args):
        self.dt, self.theta = args.dt, args.theta

    def solve(self):
        from decay_mod import solver
        self.u, self.t = solver(
            self.problem.I, self.problem.a, self.problem.T,
            self.dt, self.theta)

    def error(self):
        u_e = self.problem.exact_solution(self.t)
        e = u_e - self.u
        E = sqrt(self.dt*sum(e**2))
        return E
```

Note that we here simply reuse the implementation of the numerical meth the `decay_mod` module. The `solve` function is just a *wrapper* of the pr developed stand-alone `solver` function.

The visualizer class. The purpose of the visualizer class is to plot the ical solution stored in class `Solver`. We also add the possibility to plot t solution. Access to the problem and solver objects is required when plots so the constructor must hold references to these objects:

```
class Visualizer:
    def __init__(self, problem, solver):
        self.problem, self.solver = problem, solver

    def plot(self, include_exact=True, plt=None):
        """
        Add solver.u curve to the plotting object plt,
        and include the exact solution if include_exact is True.
        This plot function can be called several times (if
        the solver object has computed new solutions).
        """
```

```
        if plt is None:
            import scitools.std  as plt # can use matplotlib as well

        plt.plot(self.solver.t, self.solver.u, '--o')
        plt.hold('on')
        theta2name = {0: 'FE', 1: 'BE', 0.5: 'CN'}
        name = theta2name.get(self.solver.theta, '')
        legends = ['numerical %s' % name]
        if include_exact:
            t_e = linspace(0, self.problem.T, 1001)
            u_e = self.problem.exact_solution(t_e)
            plt.plot(t_e, u_e, 'b-')
            legends.append('exact')
        plt.legend(legends)
        plt.xlabel('t')
        plt.ylabel('u')
        plt.title('theta=%g, dt=%g' %
                  (self.solver.theta, self.solver.dt))
        plt.savefig('%s_%g.png' % (name, self.solver.dt))
        return plt
```

The `plt` object in the `plot` method is worth a comment. The idea is that `lot` can add a numerical solution curve to an existing plot. Calling `plot` with `plt` object (which has to be a `matplotlib.pyplot` or `scitools.std` object ı this implementation), will just add the curve `self.solver.u` as a dashed ne with circles at the mesh points (leaving the color of the curve up to the lotting tool). This functionality allows plots with several solutions: just make loop where new data is set in the problem and/or solver classes, the solver's `olve()` method is called, and the most recent numerical solution is plotted by ıe `plot(plt)` method in the visualizer object Exercise 12 describes a problem :tting where this functionality is explored.

'ombing the objects. Eventually we need to show how the classes `Problem`, `olver`, and `Visualizer` play together:

```
lef main():
    problem = Problem()
    solver = Solver(problem)
    viz = Visualizer(problem, solver)

    # Read input from the command line
    parser = problem.define_command_line_options()
    parser = solver. define_command_line_options(parser)
    args = parser.parse_args()
    problem.init_from_command_line(args)
    solver. init_from_command_line(args)

    # Solve and plot
    solver.solve()
    import matplotlib.pyplot as plt
    #import scitools.std as plt
    plt = viz.plot(plt=plt)
    E = solver.error()
    if E is not None:
        print 'Error: %.4E' % E
    plt.show()
```


The file `decay_class.py`[38] constitutes a module with the three clas the `main` function.

Test the understanding.
Implement the problem in Exercise 29 in terms of problem, solver visualizer classes. Equip the classes and their methods with doc st with tests. Also include nose tests.

3.7 Improving the problem and solver classes

The previous `Problem` and `Solver` classes containing parameters soon g repetitive code when the number of parameters increases. Much of this c be parameterized and be made more compact. For this purpose, we d collect all parameters in a dictionary, `self.prms`, with two associated dict `self.types` and `self.help` for holding associated object types and help Provided a problem, solver, or visualizer class defines these three dictior the constructor, using default or user-supplied values of the parameters, create a super class `Parameters` with general code for defining comma options and reading them as well as methods for setting and getting a pa A `Problem` or `Solver` class will then inherit command-line functionality set/get methods from the `Parameters` class.

A generic class for parameters. A simplified version of the parame looks as follows:

```
class Parameters:
    def set(self, **parameters):
        for name in parameters:
            self.prms[name] = parameters[name]

    def get(self, name):
        return self.prms[name]

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

        for name in self.prms:
            tp = self.types[name] if name in self.types 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.prms:
            self.prms[name] = getattr(args, name)
```

[38] http://tinyurl.com/jvzzcfn/decay/decay_class.py

he file class_decay_oo.py[39] contains a slightly more advanced version of class arameters where we in the set and get functions test for valid parameter names nd raise exceptions with informative messages if any name is not registered.

'he problem class. A class Problem for the problem $u' = -au$, $u(0) = I$, $\in (0, T]$, with parameters input a, I, and T can now be coded as

```
class Problem(Parameters):
    """
    Physical parameters for the problem u'=-a*u, u(0)=I,
    with t in [0,T].
    """
    def __init__(self):
        self.prms = dict(I=1, a=1, T=10)
        self.types = dict(I=float, a=float, T=float)
        self.help = dict(I='initial condition, u(0)',
                         a='coefficient in ODE',
                         T='end time of simulation')

    def exact_solution(self, t):
        I, a = self.get('I'), self.get('a')
        return I*np.exp(-a*t)
```

'he solver class. Also the solver class is derived from class Parameters nd works with the prms, types, and help dictionaries in the same way as lass Problem. Otherwise, the code is very similar to class Solver in the ecay_class.py file:

```
class Solver(Parameters):
    def __init__(self, problem):
        self.problem = problem
        self.prms = dict(dt=0.5, theta=0.5)
        self.types = dict(dt=float, theta=float)
        self.help = dict(dt='time step value',
                         theta='time discretization parameter')

    def solve(self):
        from decay_mod import solver
        self.u, self.t = solver(
            self.problem.get('I'),
            self.problem.get('a'),
            self.problem.get('T'),
            self.get('dt'),
            self.get('theta'))

    def error(self):
        try:
            u_e = self.problem.exact_solution(self.t)
            e = u_e - self.u
            E = np.sqrt(self.get('dt')*np.sum(e**2))
        except AttributeError:
            E = None
        return E
```

[39] http://tinyurl.com/jvzzcfn/decay/class_decay_oo.py

The visualizer class. Class Visualizer can be identical to the on decay_class.py file since the class does not need any parameters. Ho few adjustments in the plot method is necessary since parameters are as, e.g., problem.get('T') rather than problem.T. The details are f the file class_decay_oo.py.

Finally, we need a function that solves a real problem using the Problem, Solver, and Visualizer. This function can be just like mai decay_class.py file.

The advantage with the Parameters class is that it scales to proble a large number of physical and numerical parameters: as long as the par are defined once via a dictionary, the compact code in class Paramet handle any collection of parameters of any size.

4 Performing scientific experiments

Goal.
This section explores the behavior of a numerical method for a differ equation through computer experiments. In particular, it is shown scientific experiments can be set up and reported. We address the problem

$$u'(t) = -au(t), \quad u(0) = I, \quad t \in (0, T],$$

numerically discretized by the θ-rule:

$$u^{n+1} = \frac{1 - (1 - \theta)a\Delta t}{1 + \theta a\Delta t} u^n, \quad u^0 = I\,.$$

Our aim is to plot $u^0, u^1, \ldots, u^N$ together with the exact solution $u_e =$ for various choices of the parameters in this numerical problem: I, and θ. We are especially interested in how the discrete solution com with the exact solution when the Δt parameter is varied and θ tak the three values corresponding to the Forward Euler, Backward Eule Crank-Nicolson schemes ($\theta = 0, 1, 0.5$, respectively).

4.1 Software

A verified implementation for computing the numerical solution u^n and it together with the exact solution u_e is found in the file decay_mod.py program admits command-line arguments to specify a series of Δt val will run a loop over these values and $\theta = 0, 0.5, 1$. We make a slight edi the plots are designed: the numerical solution is specified with line type (dashed red lines with dots at the mesh points), and the show() com removed to avoid a lot of plot windows popping up on the computer scr hardcopies of the plot are still stored in files via savefig). The slightly r

[40] http://tinyurl.com/jvzzcfn/decay/decay_mod.py

rogram has the name experiments/decay_mod.py[41]. All files associated with ıe scientific investigation are collected in a subdirectory experiments.

Running the experiments is easy since the decay_mod.py program already as the loops over θ and Δt implemented. An experiment with $I = 1$, $a = 2$, $= 5$, and $dt = 0.5, 0.25, 0.1, 0.05$ is run by

```
erminal> python decay_mod.py --I 1 --a 2 --makeplot \
         --T 5 --dt 0.5 0.25 0.1 0.05
```

.2 Combining plot files

he decay_mod.py program generates a lot of image files, e.g., FE_*.png, E_*.png, and CN_*.png. We want to combine all the FE_*.png files in a able fashion in one file, with two images in each row, starting with the largest t in the upper left corner and decreasing the value as we go to the right and own. This can be done using the montage[42] program. The often occurring white reas around the plots can be cropped away by the convert -trim command. he remaining white can be made transparent for HTML pages with a non-white ackground by the command convert -transparent white.

Also plot files in the PDF format with names FE_*.pdf, BE_*.pdf, and N_*.pdf are generated and these should be combined using other tools: pdftk o combine individual plots into one file with one plot per page, and pdfnup o combine the pages into a table with multiple plots per page. The resulting nage often has some extra surrounding white space that can be removed by the dfcrop program. The code snippets below contain all details about the usage f montage, convert, pdftk, pdfnup, and pdfcrop.

Running manual commands is boring, and errors may easily sneak in. Both or automating manual work and documenting the operating system commands e actually issued in the experiment, we should write a *script* (little program). n alternative is to write the commands into an IPython notebook and use ıe notebook as the script. A plain script as a standard Python program in a eparate text file will be used here.

Reproducible science.
A script that automates running our computer experiments will ensure that the experiments can easily be rerun by ourselves or others in the future, either to check the results or redo the experiments with other input data. Also, whatever we did to produce the results is documented in every detail in the script. Automating scripts are therefore essential to making our research *reproducible*, which is a fundamental principle in science.

The script takes a list of Δt values on the command line as input and akes three combined images, one for each θ value, displaying the quality of the umerical solution as Δt varies. For example,

[41] http://tinyurl.com/jvzzcfn/decay/experiments/decay_mod.py
[42] http://www.imagemagick.org/script/montage.php


```
Terminal> python decay_exper0.py 0.5 0.25 0.1 0.05
```

results in images FE.png, CN.png, BE.png, FE.pdf, CN.pdf, and BE.pc with four plots corresponding to the four Δt values. Each plot compa numerical solution with the exact one. The latter image is shown in Fi

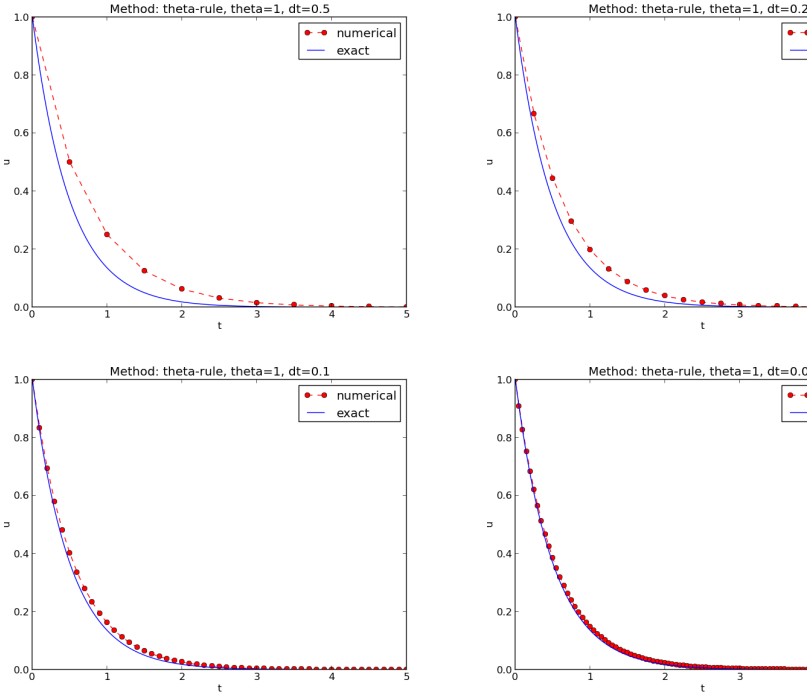


Figure 10: Illustration of the Backward Euler method for four time ste

Ideally, the script should be scalable in the sense that it works for any of Δt values, which is the case for this particular implementation:

```
import os, sys

def run_experiments(I=1, a=2, T=5):
    # The command line must contain dt values
    if len(sys.argv) > 1:
        dt_values = [float(arg) for arg in sys.argv[1:]]
    else:
        print 'Usage: %s dt1 dt2 dt3 ...' %  sys.argv[0]
        sys.exit(1)  # abort
```

```
    # Run module file as a stand-alone application
    cmd = 'python decay_mod.py --I %g --a %g --makeplot --T %g' % \
          (I, a, T)
    dt_values_str = ' '.join([str(v) for v in dt_values])
    cmd += ' --dt %s' % dt_values_str
    print cmd
    failure = os.system(cmd)
    if failure:
        print 'Command failed:', cmd; sys.exit(1)

    # Combine images into rows with 2 plots in each row
    image_commands = []
    for method in 'BE', 'CN', 'FE':
        pdf_files = ' '.join(['%s_%g.pdf' % (method, dt)
                              for dt in dt_values])
        png_files = ' '.join(['%s_%g.png' % (method, dt)
                              for dt in dt_values])
        image_commands.append(
            'montage -background white -geometry 100%' +
            ' -tile 2x %s %s.png' % (png_files, method))
        image_commands.append(
            'convert -trim %s.png %s.png' % (method, method))
        image_commands.append(
            'convert %s.png -transparent white %s.png' %
            (method, method))
        image_commands.append(
            'pdftk %s output tmp.pdf' % pdf_files)
        num_rows = int(round(len(dt_values)/2.0))
        image_commands.append(
            'pdfnup --nup 2x%d tmp.pdf' % num_rows)
        image_commands.append(
            'pdfcrop tmp-nup.pdf %s.pdf' % method)

    for cmd in image_commands:
        print cmd
        failure = os.system(cmd)
        if failure:
            print 'Command failed:', cmd; sys.exit(1)

    # Remove the files generated above and by decay_mod.py
    from glob import glob
    filenames = glob('*_*.png') + glob('*_*.pdf') + \
                glob('*_*.eps') + glob('tmp*.pdf')
    for filename in filenames:
        os.remove(filename)

if __name__ == '__main__':
    run_experiments()
```

his file is available as experiments/decay_exper0.py[43].

We may comment upon many useful constructs in this script:

- `[float(arg) for arg in sys.argv[1:]]` builds a list of real numbers from all the command-line arguments.
- `failure = os.system(cmd)` runs an operating system command, e.g., another program. The execution is successful only if `failure` is zero.

[43] http://tinyurl.com/jvzzcfn/decay/experiments/decay_exper0.py

- Unsuccessful execution usually makes it meaningless to continue gram, and therefore we abort the program with `sys.exit(1)`. A ment different from 0 signifies to the computer's operating system program stopped with a failure.
- `['%s_%s.png' % (method, dt) for dt in dt_values]` builds filenames from a list of numbers (`dt_values`).
- All `montage`, `convert`, `pdftk`, `pdfnup`, and `pdfcrop` commands for composite figures are stored in a list and later executed in a loop.
- `glob('*_*.png')` returns a list of the names of all files in the directory where the filename matches the Unix wildcard notation[44] * (meaning any text, underscore, any text, and then `.png`).
- `os.remove(filename)` removes the file with name `filename`.

4.3 Interpreting output from other programs

Programs that run other programs, like `decay_exper0.py` does, will oft to interpret output from those programs. Let us demonstrate how this in Python by extracting the relations between θ, Δt, and the error E as to the terminal window by the `decay_mod.py` program, when being exec `decay_exper0.py`. We will

- read the output from the `decay_mod.py` program
- interpret this output and store the E values in arrays for each θ v
- plot E versus Δt, for each θ, in a log-log plot

The simple `os.system(cmd)` call does not allow us to read the outp running `cmd`. Instead we need to invoke a bit more involved procedure:

```
from subprocess import Popen, PIPE, STDOUT
p = Popen(cmd, shell=True, stdout=PIPE, stderr=STDOUT)
output, dummy = p.communicate()
failure = p.returncode
if failure:
    print 'Command failed:', cmd; sys.exit(1)
```

The command stored in `cmd` is run and all text that is written to the s output *and* the standard error is available in the string `output`. Or i words, the text in `output` is what appeared in the terminal window while `cmd`.

Our next task is to run through the `output` string, line by line, an current line prints θ, Δt, and E, we split the line into these three pie

[44] http://en.wikipedia.org/wiki/Glob_(programming)

ore the data. The chosen storage structure is a dictionary `errors` with keys `t` to hold the Δt values in a list, and three θ keys to hold the corresponding E alues in a list. The relevant code lines are

```
errors = {'dt': dt_values, 1: [], 0: [], 0.5: []}
for line in output.splitlines():
    words = line.split()
    if words[0] in ('0.0', '0.5', '1.0'):  # line with E?
        # typical line: 0.0   1.25:    7.463E+00
        theta = float(words[0])
        E = float(words[2])
        errors[theta].append(E)
```

ote that we do not bother to store the Δt values as we read them from `output`, ecause we already have these values in the `dt_values` list.

We are now ready to plot E versus Δt for $\theta = 0, 0.5, 1$:

```
import matplotlib.pyplot as plt
plt.loglog(errors['dt'], errors[0], 'ro-')
plt.hold('on')
plt.loglog(errors['dt'], errors[0.5], 'b+-')
plt.loglog(errors['dt'], errors[1], 'gx-')
plt.legend(['FE', 'CN', 'BE'], loc='upper left')
plt.xlabel('log(time step)')
plt.ylabel('log(error)')
plt.title('Error vs time step')
plt.savefig('error.png')
plt.savefig('error.pdf')
```

lots occasionally need some manual adjustments. Here, the axis of the log-log lot look nicer if we adapt them strictly to the data, see Figure 11. To this end, e need to compute $\min E$ and $\max E$, and later specify the extent of the axes:

```
# Find min/max for the axis
E_min = 1E+20; E_max = -E_min
for theta in 0, 0.5, 1:
    E_min = min(E_min, min(errors[theta]))
    E_max = max(E_max, max(errors[theta]))

plt.loglog(errors['dt'], errors[0], 'ro-')
...
plt.axis([min(dt_values), max(dt_values), E_min, E_max])
...
```

The complete program, incorporating the code snippets above, is found in xperiments/decay_exper1.py[45]. This example can hopefully act as template r numerous other occasions where one needs to run experiments, extract data om the output of programs, make plots, and combine several plots in a figure le. The `decay_exper1.py` program is organized as a module, and other files an then easily extend the functionality, as illustrated in the next section.

[45] http://tinyurl.com/jvzzcfn/decay/experiments/decay_exper1.py

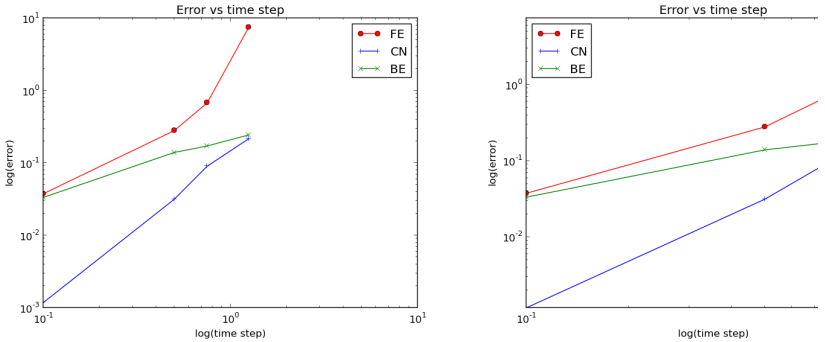


Figure 11: Default plot (left) and manually adjusted axes (right)

4.4 Making a report

The results of running computer experiments are best documented in report containing the problem to be solved, key code segments, and tl from a series of experiments. At least the part of the report contain plots should be automatically generated by the script that performs tl experiments, because in that script we know exactly which input data tl used to generate a specific plot, thereby ensuring that each figure is conn the right data. Take a look at an example at http://tinyurl.com/k3 writing_reports//sphinx-cloud/ to see what we have in mind.

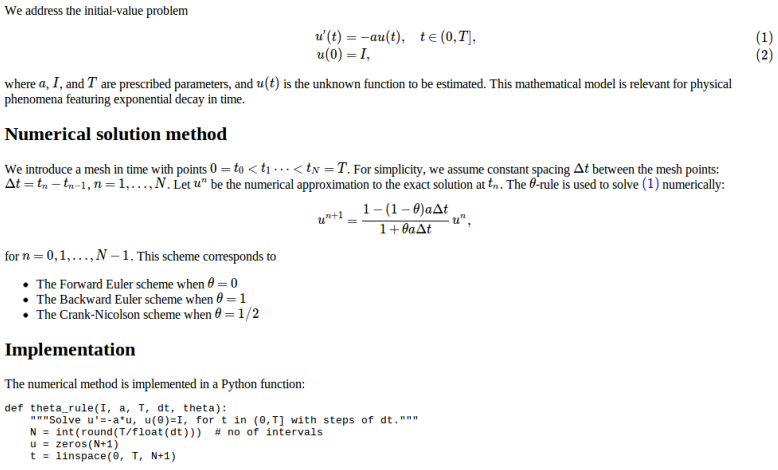
Plain HTML. Scientific reports can be written in a variety of forma we begin with the HTML[46] format which allows efficient viewing of experiments in any web browser. The program `decay_exper1_html.py` `decay_exper1.py` to perform the experiments and then runs statements ating an HTML file with a summary, a section on the mathematical pro section on the numerical method, a section on the `solver` function imple the method, and a section with subsections containing figures that sl results of experiments where Δt is varied for $\theta = 0, 0.5, 1$. The mentioned file contains all the details for writing this HTML report[48]. You can viev port on http://tinyurl.com/k3sdbuv/writing_reports//_static/r html.html.

HTML with MathJax. Scientific reports usually need mathemat mulas and hence mathematical typesetting. In plain HTML, as usec `decay_exper1_html.py` file, we have to use just the keyboard chara write mathematics. However, there is an extension to HTML, called Ma which allows formulas and equations to be typeset with LaTeX syntax an

[46] http://en.wikipedia.org/wiki/HTML
[47] http://tinyurl.com/jvzzcfn/decay/experiments/decay_exper1_html.py
[48] http://tinyurl.com/k3sdbuv/writing_reports//_static/report_html.html.ht
[49] http://www.mathjax.org/

ɔndered in web browsers, see Figure 12. A relatively small subset of LaTeX ɹvironments is supported, but the syntax for formulas is quite rich. Inline ɔrmulas are look like `\( u'=-au \)` while equations are surrounded by `$$` gns. Inside such signs, one can use `\[ u'=-au \]` for unnumbered equations, r `\begin{equation}` and `\end{equation}` surrounding `u'=-au` for numbered ɥuations, or `\begin{align}` and `\end{align}` for multiple aligned equations. ou need to be familiar with mathematical typesetting in LaTeX[50].

The file `decay_exper1_mathjax.py`[51] contains all the details for turning the revious plain HTML report into web pages with nicely typeset mathematics. he corresponding HTML code[52] be studied to see all details of the mathematical ɣpesetting.



We address the initial-value problem

$$u'(t) = -au(t), \quad t \in (0,T], \tag{1}$$
$$u(0) = I, \tag{2}$$

where a, I, and T are prescribed parameters, and $u(t)$ is the unknown function to be estimated. This mathematical model is relevant for physical phenomena featuring exponential decay in time.

Numerical solution method

We introduce a mesh in time with points $0 = t_0 < t_1 \cdots < t_N = T$. For simplicity, we assume constant spacing Δt between the mesh points: $\Delta t = t_n - t_{n-1}$, $n = 1, \ldots, N$. Let u^n be the numerical approximation to the exact solution at t_n. The θ-rule is used to solve (1) numerically:

$$u^{n+1} = \frac{1-(1-\theta)a\Delta t}{1+\theta a\Delta t}u^n,$$

for $n = 0, 1, \ldots, N-1$. This scheme corresponds to

- The Forward Euler scheme when $\theta = 0$
- The Backward Euler scheme when $\theta = 1$
- The Crank-Nicolson scheme when $\theta = 1/2$

Implementation

The numerical method is implemented in a Python function:

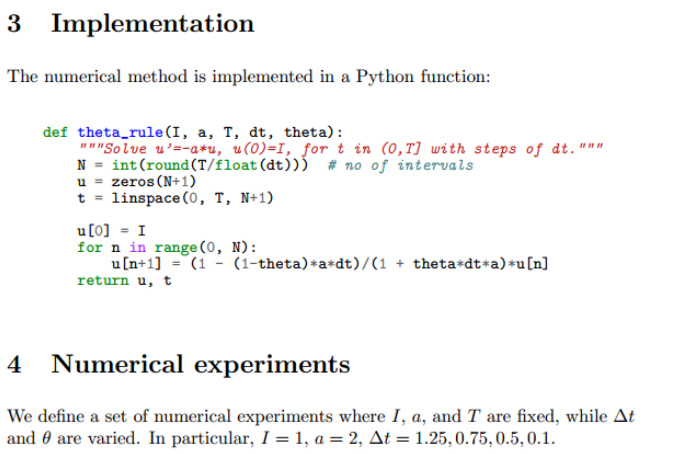
```
def theta_rule(I, a, T, dt, theta):
    """Solve u'=-a*u, u(0)=I, for t in (0,T] with steps of dt."""
    N = int(round(T/float(dt)))  # no of intervals
    u = zeros(N+1)
    t = linspace(0, T, N+1)
```

Figure 12: Report in HTML format with MathJax.

ɅTEX. The *de facto* language for mathematical typesetting and scientific ɛport writing is LaTeX[53]. A number of very sophisticated packages have been ɗdded to the language over a period of three decades, allowing very fine-tuned ɻyout and typesetting. For output in the PDF format[54], see Figure 13 for ɲ example, LaTeX is the definite choice when it comes to quality. The LaTeX ɯnguage used to write the reports has typically a lot of commands involving ackslashes and braces[55]. For output on the web, using HTML (and not the PDF irectly in the browser window), LaTeX struggles with delivering high quality ɣpesetting. Other tools, especially Sphinx, give better results and can also

[50] http://en.wikibooks.org/wiki/LaTeX/Mathematics
[51] http://tinyurl.com/jvzzcfn/decay/experiments/decay_exper1_html.py
[52] http://tinyurl.com/k3sdbuv/writing_reports//_static/report_mathjax.html.html
[53] http://en.wikipedia.org/wiki/LaTeX
[54] http://tinyurl.com/k3sdbuv/writing_reports//_static/report.pdf
[55] http://tinyurl.com/k3sdbuv/writing_reports//_static/report.tex.html

produce nice-looking PDFs. The file `decay_exper1_latex.py` shows generate the LaTeX source from a program.



3 Implementation

The numerical method is implemented in a Python function:

```
def theta_rule(I, a, T, dt, theta):
    """Solve u'=-a*u, u(0)=I, for t in (0,T] with steps of dt."""
    N = int(round(T/float(dt)))  # no of intervals
    u = zeros(N+1)
    t = linspace(0, T, N+1)

    u[0] = I
    for n in range(0, N):
        u[n+1] = (1 - (1-theta)*a*dt)/(1 + theta*dt*a)*u[n]
    return u, t
```

4 Numerical experiments

We define a set of numerical experiments where I, a, and T are fixed, while Δt and θ are varied. In particular, $I = 1$, $a = 2$, $\Delta t = 1.25, 0.75, 0.5, 0.1$.

Figure 13: Report in PDF format generated from LaTeX source.

Sphinx. Sphinx[56] is a typesetting language with similarities to HTl LaTeX, but with much less tagging. It has recently become very pop software documentation and mathematical reports. Sphinx can utilize L mathematical formulas and equations (via MathJax or PNG images). U nately, the subset of LaTeX mathematics supported is less than in full M (in particular, numbering of multiple equations in an `align` type enviror not supported). The Sphinx syntax[57] is an extension of the reStructu language. An attractive feature of Sphinx is its rich support for fancy l web pages[58]. In particular, Sphinx can easily be combined with variou *themes* that give a certain look and feel to the web site and that offers contents, navigation, and search facilities, see Figure 14.

Markdown. A recently popular format for easy writing of web pages down[59]. Text is written very much like one would do in email, using spac special characters to naturally format the code instead of heavily tagg text as in LaTeX and HTML. With the tool Pandoc[60] one can go from Ma to a variety of formats. HTML is a common output format, but LaTeX XML, OpenOffice, MediaWiki, and MS Word are some other possibiliti

Wiki formats. A range of wiki formats are popular for creating n the web, especially documents which allow groups of people to edit a

[56] http://sphinx.pocoo.org/
[57] http://tinyurl.com/k3sdbuv/writing_reports//_static/report_sphinx.rst.l
[58] http://tinyurl.com/k3sdbuv/writing_reports//_static/sphinx-cloud/index
[59] http://daringfireball.net/projects/markdown/
[60] http://johnmacfarlane.net/pandoc/

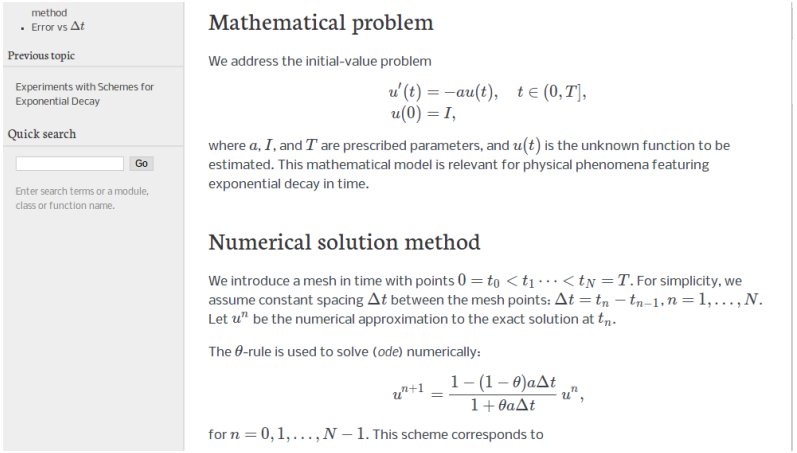


Figure 14: Report in HTML format generated from Sphinx source.

ontent. Apart from MediaWiki[61] (the wiki format used for Wikipedia), wiki rmats have no support for mathematical typesetting and also limited tools for isplaying computer code in nice ways. Wiki formats are therefore less suitable r scientific reports compared to the other formats mentioned here.

oconce. Since it is difficult to choose the right tool or format for writing a ientific report, it is advantageous to write the content in a format that easily anslates to LaTeX, HTML, Sphinx, Markdown, and various wikis. Doconce[62] is ch a tool. It is similar to Pandoc, but offers some special convenient features r writing about mathematics and programming. The tagging is modest[63], mewhere between LaTeX and Markdown. The program `decay_exper_do.py` monstrates how to generate (and write) Doconce code for a report.

Worked example. The HTML, LaTeX (PDF), Sphinx, and Doconce formats r the scientific report whose content is outlined above, are exemplified with urce codes and results at the web pages associated with this teaching material: `ttp://tinyurl.com/k3sdbuv/writing_reports/`.

.5 Publishing a complete project

report documenting scientific investigations should be accompanied by all the ftware and data used for the investigations so that others have a possibility to do the work and assess the qualify of the results. This possibility is important r *reproducible research* and hence reaching reliable scientific conclusions.

One way of documenting a complete project is to make a directory tree with ll relevant files. Preferably, the tree is published at some project hosting site like

[61] `http://www.mediawiki.org/wiki/MediaWiki`
[62] `https://github.com/hplgit/doconce`
[63] `http://tinyurl.com/k3sdbuv/writing_reports//_static/report.do.txt.html`

Bitbucket, GitHub, or Googlecode[64] so that others can download it as a zipfile, or clone the files directly using a version control system like M or Git. For the investigations outlined in Section 4.4, we can create a d tree with files

```
setup.py
./src:
   decay_mod.py
./doc:
   ./src:
      decay_exper1_mathjax.py
      make_report.sh
      run.sh
   ./pub:
      report.html
```

The `src` directory holds source code (modules) to be reused in other p the `setup.py` builds and installs such software, the `doc` directory cont documentation, with `src` for the source of the documentation and `pub` fo made, published documentation. The `run.sh` file is a simple Bash scrip the `python` command we used to run `decay_exper1_mathjax.py` to g the experiments and the `report.html` file.

5 Exercises and Problems

Exercise 1: Derive schemes for Newton's law of cooli

Show in detail how we can apply the ideas of the Forward Euler, Ba Euler, Crank-Nicolson, and θ-rule discretizations to derive explicit compu formulas for new temperature values in Newton's law of cooling (see Sectio

$$\frac{dT}{dt} = -k(T - T_s), \quad T(0) = T_0\,.$$

Here, T is the temperature of the body, T_s is the temperature of the surro t is time, k is the heat transfer coefficient, and T_0 is the initial temper the body.

Filename: `schemes_cooling.pdf`.

Exercise 2: Implement schemes for Newton's law of c

Formulate a θ-rule for the three schemes in Exercise 1 such that you the three schemes from a single formula by varying the θ parameter. Im the θ scheme in a function `cooling(T0, k, T_s, t_end, dt, thet` where `T0` is the initial temperature, `k` is the heat transfer coefficient, `T_` temperature of the surroundings, `t_end` is the end time of the simulatio the time step, and `theta` corresponds to θ. The `cooling` function shoul the temperature as an array `T` of values at the mesh points and the time Construct verification examples to check that the implementation work

[64] `http://hplgit.github.com/teamods/bitgit/html/`

Hint. For verification, try to find an exact solution of the discrete equations. A trick is to introduce $u = T - T_s$, observe that $u^n = (T_0 - T_s)A^n$ for some amplification factor A, and then express this formula in terms of T^n.

Filename: `cooling.py`.

Exercise 3: Find time of murder from body temperature

A detective measures the temperature of a dead body to be 26.7 C at 2 pm. One hour later the temperature is 25.8 C. The question is when death occurred.

Assume that Newton's law of cooling (120) is an appropriate mathematical model for the evolution of the temperature in the body. First, determine k in (120) by formulating a Forward Euler approximation with one time steep from time 2 am to time 3 am, where knowing the two temperatures allows for finding k. Assume the temperature in the air to be 20 C. Thereafter, simulate the temperature evolution from the time of murder, taken as $t = 0$, when $T = 37$ C, until the temperature reaches 25.8 C. The corresponding time allows for answering when death occurred. Filename: `detective.py`.

Exercise 4: Experiment with integer division

Explain what happens in the following computations, where some are mathematically unexpected:

```
>>> dt = 3
>>> T = 8
>>> Nt = T/dt
>>> Nt
2
>>> theta = 1; a = 1
>>> (1 - (1-theta)*a*dt)/(1 + theta*dt*a)
1
```

Filename: `pyproblems.txt`.

Exercise 5: Experiment with wrong computations

Consider the `solver` function in the `decay_v1.py`[65] file and the following call:

```
u, t = solver(I=1, a=1, T=7, dt=2, theta=1)
```

The output becomes

```
t= 0.000 u=1
t= 2.000 u=0
t= 4.000 u=0
t= 6.000 u=0
```

Print out the result of all intermediate computations and use `type(v)` to see the object type of the result stored in `v`. Examine the intermediate calculations and explain why `u` is wrong and why we compute up to $t = 6$ only even though we specified $T = 7$. Filename: `decay_v1_err.py`.

[65] `http://tinyurl.com/jvzzcfn/decay/decay_v1.py`

Exercise 6: Plot the error function

Solve the problem $u' = -au$, $u(0) = I$, using the Forward Euler, Backwar and Crank-Nicolson schemes. For each scheme, plot the error functio $u_e(t_n) - u^n$ for Δt, $\frac{1}{4}\Delta t$, and $\frac{1}{8}\Delta t$, where u_e is the exact solution of the O u^n is the numerical solution at mesh point t_n. Filename: `decay_plot_er`

Exercise 7: Compare methods for a given time mesh

Make a program that imports the `solver` function from the `decay_mod` and offers a function `compare(dt, I, a)` for comparing, in a plot, the corresponding to $\theta = 0, 0.5, 1$ and the exact solution. This plot shows the of the methods for a given time mesh. Read input data for the proble the command line using appropriate functions in the `decay_mod` mod `--dt` option for giving several time step values can be reused: just use time step value for the computations). Filename: `decay_compare_the`

Exercise 8: Change formatting of numbers and debu

The `decay_memsave.py`[66] program writes the time values and solution v a file which looks like

```
0.0000000000000000E+00  1.0000000000000000E+00
2.0000000000000001E-01  8.3333333333333337E-01
4.0000000000000002E-01  6.9444444444444453E-01
6.0000000000000009E-01  5.7870370370370383E-01
8.0000000000000004E-01  4.8225308641975323E-01
1.0000000000000000E+00  4.0187757201646102E-01
1.2000000000000000E+00  3.3489797668038418E-01
1.3999999999999999E+00  2.7908164723365347E-01
```

Modify the file output such that it looks like

```
0.000  1.00000
0.200  0.83333
0.400  0.69444
0.600  0.57870
0.800  0.48225
1.000  0.40188
1.200  0.33490
1.400  0.27908
```

Run the modified program

```
Terminal> python decay_memsave_v2.py --T 10 --theta 1 \
          --dt 0.2 --makeplot
```

The program just prints `Bug in the implementation!` and does not s plot. What went wrong? Filename: `decay_memsave_v2.py`.

Problem 9: Write a doctest

Type in the following program and equip the `roots` function with a do

[66] `http://tinyurl.com/jvzzcfn/decay/decay_memsave.py`

```
import sys
# This sqrt(x) returns real if x>0 and complex if x<0
from numpy.lib.scimath import sqrt

def roots(a, b, c):
    """
    Return the roots of the quadratic polynomial
    p(x) = a*x**2 + b*x + c.

    The roots are real or complex objects.
    """
    q = b**2 - 4*a*c
    r1 = (-b + sqrt(q))/(2*a)
    r2 = (-b - sqrt(q))/(2*a)
    return r1, r2

a, b, c = [float(arg) for arg in sys.argv[1:]]
print roots(a, b, c)
```

Iake sure to test both real and complex roots. Write out numbers with 14 digits r less. Filename: `doctest_roots.py`.

'roblem 10: Write a nose test

Iake a nose test for the `roots` function in Problem 9. Filename: `test_roots.py`.

'roblem 11: Make a module

et

$$q(t) = \frac{RAe^{at}}{R + A(e^{at} - 1)}\,.$$

Iake a Python module `q_module` containing two functions `q(t)` and `dqdt(t)` for ɔmputing $q(t)$ and $q'(t)$, respectively. Perform a `from numpy import *` in this ıodule. Import `q` and `dqdt` in another file using the "star import" construction `rom q_module import *`. All objects available in this file is given by `dir()`. rint `dir()` and `len(dir())`. Then change the import of `numpy` in `q_module.py` ɔ `import numpy as np`. What is the effect of this import on the number of bjects in `dir()` in a file that does `from q_module import *`?

Filename: `q_module.py`.

:xercise 12: Make use of a class implementation

/e want to solve the exponential decay problem $u' = -au$, $u(0) = I$, for several .t values and $\theta = 0, 0.5, 1$. For each Δt value, we want to make a plot where the ıree solutions corresponding to $\theta = 0, 0.5, 1$ appear along with the exact solution. /rite a function `experiment` to accomplish this. The function should import the .asses `Problem`, `Solver`, and `Visualizer` from the `decay_class`[67] module and ıake use of these. A new command-line option `--dt_values` must be added to llow the user to specify the Δt values on the command line (the options `--dt` ıd `--theta` implemented by the `decay_class` module have then no effect when ınning the `experiment` function). Note that the classes in the `decay_class` ıodule should *not* be modified. Filename: `decay_class_exper.py`.

[67] `http://tinyurl.com/jvzzcfn/decay/decay_class.py`

Exercise 13: Generalize a class implementation

Consider the file `decay_class.py`[68] where the exponential decay probl $-au$, $u(0) = I$, is implemented via the classes `Problem`, `Solver`, and `Visu` Extend the classes to handle the more general problem

$$u'(t) = -a(t)u(t) + b(t), \quad u(0) = I,\ t \in (0, T],$$

using the θ-rule for discretization.

In the case with arbitrary functions $a(t)$ and $b(t)$ the problem cla longer guaranteed to provide an exact solution. Let the `exact_solu` class `Problem` return `None` if the exact solution for the particular proble available. Modify classes `Solver` and `Visualizer` accordingly.

Add test functions `test_*()` for the nose testing tool in the modu add a demo example where the environment suddenly changes (modele abrupt change in the decay rate a):

$$a(t) = \begin{cases} 1, & 0 \leq t \leq t_p, \\ k, & t > t_p, \end{cases}$$

where t_p is the point of time the environment changes. Take t_p = make plots that illustrate the effect of having $k \gg 1$ and $k \ll 1$. Fi `decay_class2.py`.

Exercise 14: Generalize an advanced class implement

Solve Exercise 13 by utilizing the class implementations in `decay_class_` Filename: `decay_class3.py`.

6 Analysis of finite difference equations

We address the ODE for exponential decay,

$$u'(t) = -au(t), \quad u(0) = I,$$

where a and I are given constants. This problem is solved by the θ-ru difference scheme, resulting in the recursive equations

$$u^{n+1} = \frac{1 - (1 - \theta)a\Delta t}{1 + \theta a\Delta t}u^n$$

for the numerical solution u^{n+1}, which approximates the exact solution u_e point t_{n+1}. For constant mesh spacing, which we assume here, $t_{n+1} = (n$

Discouraging numerical solutions. Choosing $I = 1$, $a = 2$, and experiments with $\theta = 1, 0.5, 0$ for $\Delta t = 1.25, 0.75, 0.5, 0.1$, gives the re Figures 15, 16, and 17.

The characteristics of the displayed curves can be summarized as fo

[68] `http://tinyurl.com/jvzzcfn/decay/decay_class.py`
[69] `http://tinyurl.com/jvzzcfn/decay/decay_class_oo.py`

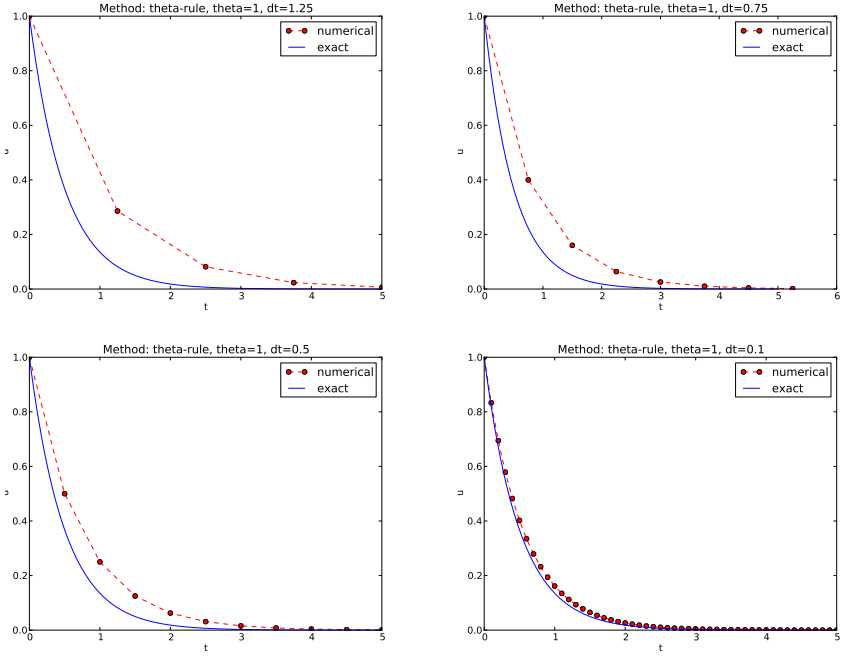


Figure 15: Backward Euler.

- The Backward Euler scheme always gives a monotone solution, lying above the exact curve.
- The Crank-Nicolson scheme gives the most accurate results, but for $\Delta t = 1.25$ the solution oscillates.
- The Forward Euler scheme gives a growing, oscillating solution for $\Delta t = 1.25$; a decaying, oscillating solution for $\Delta t = 0.75$; a strange solution $u^n = 0$ for $n \geq 1$ when $\Delta t = 0.5$; and a solution seemingly as accurate as the one by the Backward Euler scheme for $\Delta t = 0.1$, but the curve lies below the exact solution.

ince the exact solution of our model problem is a monotone function, $u(t) = e^{-at}$, some of these qualitatively wrong results are indeed alarming!

Goal.
We ask the question

- Under what circumstances, i.e., values of the input data I, a, and Δt will the Forward Euler and Crank-Nicolson schemes result in undesired oscillatory solutions?

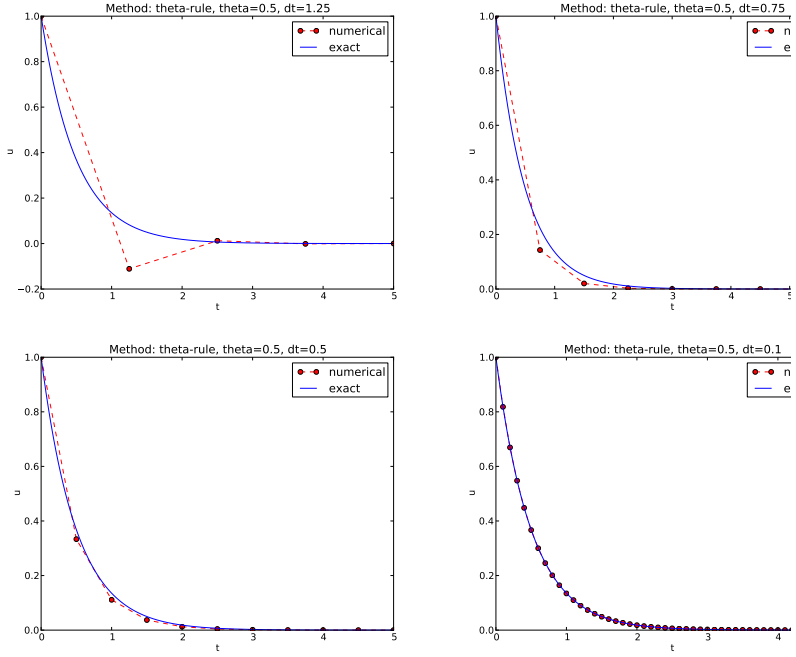


Figure 16: Crank-Nicolson.

The question will be investigated both by numerical experiments a precise mathematical theory. The latter will help establish general c on Δt for avoiding non-physical oscillatory or growing solutions.

Another question to be raised is

- How does Δt impact the error in the numerical solution?

For our simple model problem we can answer this question very pre but we will also look at simplified formulas for small Δt and touch important concepts such as *convergence rate* and *the order of a sc* Other fundamental concepts mentioned are stability, consistency, an vergence.

6.1 Experimental investigation of oscillatory solutic

To address the first question above, we may set up an experiment w loop over values of I, a, and Δt. For each experiment, we flag the sol oscillatory if

$$u^n > u^{n-1},$$

for some value of n, since we expect u^n to decay with n, but oscillations increase over a time step. We will quickly see that oscillations are inde

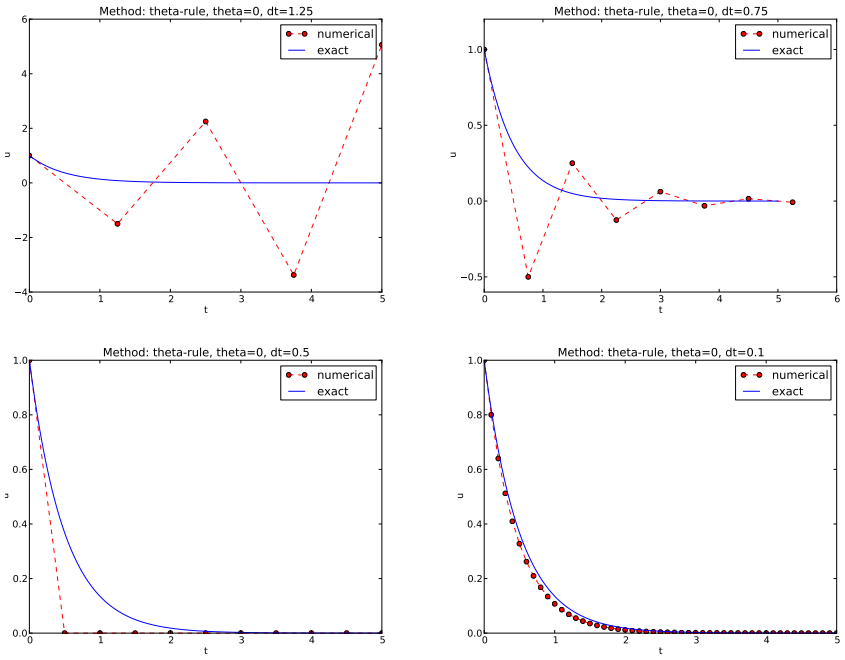


Figure 17: Forward Euler.

f I, but do depend on a and Δt. Therefore, we introduce a two-dimensional nction $B(a, \Delta t)$ which is 1 if oscillations occur and 0 otherwise. We can isualize B as a contour plot (lines for which $B = \text{const}$). The contour $B = 0.5$ orresponds to the borderline between oscillatory regions with $B = 1$ and onotone regions with $B = 0$ in the $a, \Delta t$ plane.

The B function is defined at discrete a and Δt values. Say we have given P a alues, $a_0, \ldots, a_{P-1}$, and Q Δt values, $\Delta t_0, \ldots, \Delta t_{Q-1}$. These a_i and Δt_j values, $= 0, \ldots, P-1$, $j = 0, \ldots, Q-1$, form a rectangular mesh of $P \times Q$ points in the lane. At each point $(a_i, \Delta t_j)$, we associate the corresponding value of $B(a_i, \Delta t_j)$, enoted B_{ij}. The B_{ij} values are naturally stored in a two-dimensional array. We an thereafter create a plot of the contour line $B_{ij} = 0.5$ dividing the oscillatory nd monotone regions. The file `decay_osc_regions.py`[70] `osc_regions` stands or "oscillatory regions") contains all nuts and bolts to produce the $B = 0.5$ line n Figures 18 and 19. The oscillatory region is above this line.

```
from decay_mod import solver
import numpy as np
import scitools.std as st
```

[70] `http://tinyurl.com/jvzzcfn/decay/decay_osc_regions.py`


```
def non_physical_behavior(I, a, T, dt, theta):
    """
    Given lists/arrays a and dt, and numbers I, dt, and theta,
    make a two-dimensional contour line B=0.5, where B=1>0.5
    means oscillatory (unstable) solution, and B=0<0.5 means
    monotone solution of u'=-au.
    """
    a = np.asarray(a); dt = np.asarray(dt)  # must be arrays
    B = np.zeros((len(a), len(dt)))         # results
    for i in range(len(a)):
        for j in range(len(dt)):
            u, t = solver(I, a[i], T, dt[j], theta)
            # Does u have the right monotone decay properties?
            correct_qualitative_behavior = True
            for n in range(1, len(u)):
                if u[n] > u[n-1]:  # Not decaying?
                    correct_qualitative_behavior = False
                    break  # Jump out of loop
            B[i,j] = float(correct_qualitative_behavior)
    a_, dt_ = st.ndgrid(a, dt)  # make mesh of a and dt values
    st.contour(a_, dt_, B, 1)
    st.grid('on')
    st.title('theta=%g' % theta)
    st.xlabel('a'); st.ylabel('dt')
    st.savefig('osc_region_theta_%s.png' % theta)
    st.savefig('osc_region_theta_%s.pdf' % theta)

non_physical_behavior(
    I=1,
    a=np.linspace(0.01, 4, 22),
    dt=np.linspace(0.01, 4, 22),
    T=6,
    theta=0.5)
```

By looking at the curves in the figures one may guess that $a\Delta t$ must than a critical limit to avoid the undesired oscillations. This limit seen about 2 for Crank-Nicolson and 1 for Forward Euler. We shall now e a precise mathematical analysis of the discrete model that can expl observations in our numerical experiments.

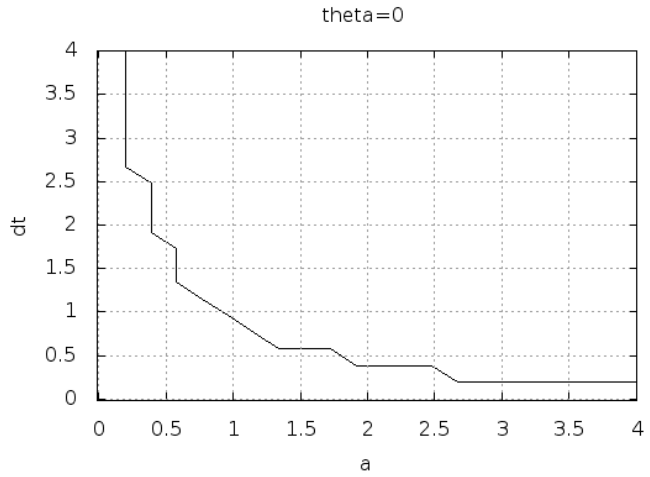
6.2 Exact numerical solution

Starting with $u^0 = I$, the simple recursion (52) can be applied repea times, with the result that

$$u^n = IA^n, \quad A = \frac{1 - (1-\theta)a\Delta t}{1 + \theta a \Delta t}.$$

Solving difference equations.

Difference equations where all terms are linear in u^{n+1}, u^n, and n u^{n-1}, u^{n-2}, etc., are called *homogeneous, linear* difference equations their solutions are generally of the form $u^n = A^n$. Inserting this expr and dividing by A^{n+1} gives a polynomial equation in A. In the presen



igure 18: Forward Euler scheme: oscillatory solutions occur for points above ıe curve.

we get

$$A = \frac{1-(1-\theta)a\Delta t}{1+\theta a\Delta t}\,.$$

This is a solution technique of wider applicability than repeated use of the recursion (52).

Regardless of the solution approach, we have obtained a formula for u^n. This ɔrmula can explain everything what we see in the figures above, but it also ives us a more general insight into accuracy and stability properties of the three :hemes.

.3 Stability

ince u^n is a factor A raised to an integer power n, we realize that $A < 0$ will ɔr odd powers imply $u^n < 0$ and for even power result in $u^n > 0$. That is, the ɔlution oscillates between the mesh points. We have oscillations due to $A < 0$ hen

$$(1-\theta)a\Delta t > 1\,. \tag{54}$$

ince $A > 0$ is a requirement for having a numerical solution with the same asic property (monotonicity) as the exact solution, we may say that $A > 0$ is a *tability criterion*. Expressed in terms of Δt the stability criterion reads

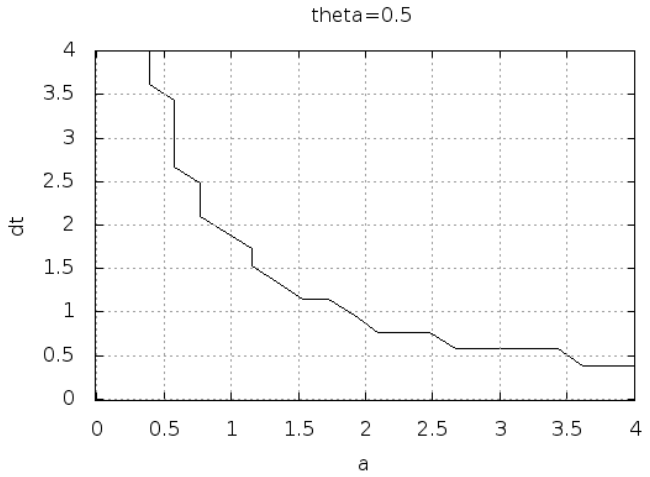


Figure 19: Crank-Nicolson scheme: oscillatory solutions occur for point the curve.

$$\Delta t < \frac{1}{(1-\theta)a}\,.$$

The Backward Euler scheme is always stable since $A < 0$ is impos: $\theta = 1$, while non-oscillating solutions for Forward Euler and Crank-N demand $\Delta t \leq 1/a$ and $\Delta t \leq 2/a$, respectively. The relation between Δ look reasonable: a smaller a means faster decay and hence a need for time steps.

Looking at Figure 17, we see that with $a\Delta t = 2 \cdot 1.25 = 2.5$, $A = -$ the solution $u^n = (-1.5)^n$ oscillates *and* grows. With $a\Delta t = 2 \cdot 0.7$ $A = -0.5$, $u^n = (-0.5)^n$ decays but oscillates. The peculiar case Δ where the Forward Euler scheme produces a solution that is stuck on th corresponds to $A = 0$ and therefore $u^0 = I = 1$ and $u^n = 0$ for $n \geq$ decaying oscillations in the Crank-Nicolson scheme for $\Delta t = 1.25$ ar explained by the fact that $A = -0.25$.

The factor A is called the *amplification factor* since the solution at a n level is A times the solution at the previous time level. For a decay pro must obviously have $|A| \leq 1$, which is fulfilled for all Δt if $\theta \geq 1/2$. Arl large values of u can be generated when $|A| > 1$ and n is large enou numerical solution is in such cases totally irrelevant to an ODE modelir processes! To avoid this situation, we must for $\theta < 1/2$ have

$$\Delta t \leq \frac{2}{(1-2\theta)a}, \tag{56}$$

hich means $\Delta t < 2/a$ for the Forward Euler scheme.

Stability properties.
We may summarize the stability investigations as follows:

1. The Forward Euler method is a *conditionally stable* scheme because it requires $\Delta t < 2/a$ for avoiding growing solutions and $\Delta t < 1/a$ for avoiding oscillatory solutions.

2. The Crank-Nicolson is *unconditionally stable* with respect to growing solutions, while it is conditionally stable with the criterion $\Delta t < 2/a$ for avoiding oscillatory solutions.

3. The Backward Euler method is unconditionally stable with respect to growing and oscillatory solutions - any Δt will work.

Much literature on ODEs speaks about L-stable and A-stable methods. In our case A-stable methods ensures non-growing solutions, while L-stable methods also avoids oscillatory solutions.

.4 Comparing amplification factors

fter establishing how A impacts the qualitative features of the solution, we shall ow look more into how well the numerical amplification factor approximates ıe exact one. The exact solution reads $u(t) = Ie^{-at}$, which can be rewritten as

$$u_{\mathrm{e}}(t_n) = Ie^{-an\Delta t} = I(e^{-a\Delta t})^n\,. \tag{57}$$

rom this formula we see that the exact amplification factor is

$$A_{\mathrm{e}} = e^{-a\Delta t}\,. \tag{58}$$

We realize that the exact and numerical amplification factors depend on a and $.t$ through the product $a\Delta t$. Therefore, it is convenient to introduce a symbol ›r this product, $p = a\Delta t$, and view A and A_{e} as functions of p. Figure 20 shows ıese functions. Crank-Nicolson is clearly closest to the exact amplification ıctor, but that method has the unfortunate oscillatory behavior when $p > 2$.

.5 Series expansion of amplification factors

s an alternative to the visual understanding inherent in Figure 20, there is a :rong tradition in numerical analysis to establish formulas for the approximation :rors when the discretization parameter, here Δt, becomes small. In the present ıse we let p be our small discretization parameter, and it makes sense to simplify ıe expressions for A and A_{e} by using Taylor polynomials around $p = 0$. The aylor polynomials are accurate for small p and greatly simplifies the comparison f the analytical expressions since we then can compare polynomials, term by :rm.

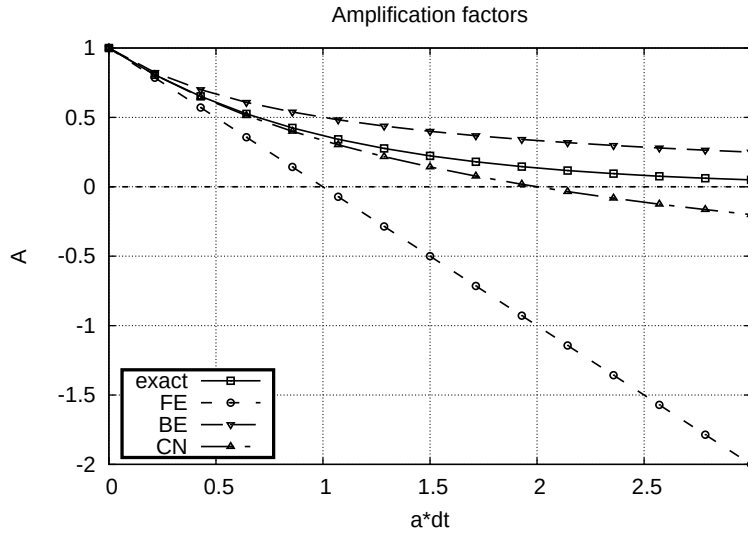


Figure 20: Comparison of amplification factors.

Calculating the Taylor series for A_{e} is easily done by hand, but tl versions of A for $\theta = 0, 1, \frac{1}{2}$ lead to more cumbersome calculations. No analytical computations can benefit greatly by symbolic computer algel ware. The Python package `sympy` represents a powerful computer algebra not yet as sophisticated as the famous Maple and Mathematica syste free and very easy to integrate with our numerical computations in Pyt

When using `sympy`, it is convenient to enter the interactive Pytho where we can write expressions and statements and immediately see the Here is a simple example. We strongly recommend to use `isympy` (or `i` for such interactive sessions.

Let us illustrate `sympy` with a standard Python shell syntax (`>>>` pro compute a Taylor polynomial approximation to e^{-p}:

```
>>> from sympy import *
>>> # Create p as a mathematical symbol with name 'p'
>>> p = Symbol('p')
>>> # Create a mathematical expression with p
>>> A_e = exp(-p)
>>>
>>> # Find the first 6 terms of the Taylor series of A_e
>>> A_e.series(p, 0, 6)
1 + (1/2)*p**2 - p - 1/6*p**3 - 1/120*p**5 + (1/24)*p**4 + O(p**6
```

Lines with `>>>` represent input lines and lines without this prompt rep the result of computations (note that `isympy` and `ipython` apply other p but in this text we always apply `>>>` for interactive Python computing)

om the order of the powers, the computed formula is easily recognized as the eginning of the Taylor series for e^{-p}.

Let us define the numerical amplification factor where p and θ enter the ormula as symbols:

```
>>> theta = Symbol('theta')
>>> A = (1-(1-theta)*p)/(1+theta*p)
```

o work with the factor for the Backward Euler scheme we can substitute the alue 1 for `theta`:

```
>>> A.subs(theta, 1)
1/(1 + p)
```

imilarly, we can replace `theta` by 1/2 for Crank-Nicolson, preferably using an xact rational representation of 1/2 in `sympy`:

```
>>> half = Rational(1,2)
>>> A.subs(theta, half)
1/(1 + (1/2)*p)*(1 - 1/2*p)
```

The Taylor series of the amplification factor for the Crank-Nicolson scheme an be computed as

```
>>> A.subs(theta, half).series(p, 0, 4)
1 + (1/2)*p**2 - p - 1/4*p**3 + O(p**4)
```

We are now in a position to compare Taylor series:

```
>>> FE = A_e.series(p, 0, 4) - A.subs(theta, 0).series(p, 0, 4)
>>> BE = A_e.series(p, 0, 4) - A.subs(theta, 1).series(p, 0, 4)
>>> CN = A_e.series(p, 0, 4) - A.subs(theta, half).series(p, 0, 4 )
>>> FE
(1/2)*p**2 - 1/6*p**3 + O(p**4)
>>> BE
-1/2*p**2 + (5/6)*p**3 + O(p**4)
>>> CN
(1/12)*p**3 + O(p**4)
```

rom these expressions we see that the error $A - A_e \sim \mathcal{O}(p^2)$ for the Forward nd Backward Euler schemes, while $A - A_e \sim \mathcal{O}(p^3)$ for the Crank-Nicolson cheme. It is the *leading order term*, i.e., the term of the lowest order (polynomial egree), that is of interest, because as $p \to 0$, this term is (much) bigger than ne higher-order terms (think of $p = 0.01$: p is a hundred times larger than p^2).

Now, a is a given parameter in the problem, while Δt is what we can vary. ne therefore usually writes the error expressions in terms Δt. When then have

$$A - A_e = \begin{cases} \mathcal{O}(\Delta t^2), & \text{Forward and Backward Euler,} \\ \mathcal{O}(\Delta t^3), & \text{Crank-Nicolson} \end{cases} \tag{59}$$

We say that the Crank-Nicolson scheme has an error in the ampli factor of order Δt^3, while the two other schemes are of order Δt^2 in tl quantity. What is the significance of the order expression? If we h the error in amplification factor at a time level will be reduced by a f 4 in the Forward and Backward Euler schemes, and by a factor of 8 Crank-Nicolson scheme. That is, as we reduce Δt to obtain more a results, the Crank-Nicolson scheme reduces the error more efficiently t other schemes.

6.6 The fraction of numerical and exact amplificatic tors

An alternative comparison of the schemes is to look at the ratio A/A_e error $1 - A/A_e$ in this ratio:

```
>>> FE = 1 - (A.subs(theta, 0)/A_e).series(p, 0, 4)
>>> BE = 1 - (A.subs(theta, 1)/A_e).series(p, 0, 4)
>>> CN = 1 - (A.subs(theta, half)/A_e).series(p, 0, 4)
>>> FE
(1/2)*p**2 + (1/3)*p**3 + O(p**4)
>>> BE
-1/2*p**2 + (1/3)*p**3 + O(p**4)
>>> CN
(1/12)*p**3 + O(p**4)
```

The leading-order terms have the same powers as in the analysis of A -

6.7 The global error at a point

The error in the amplification factor reflects the error when progressi time level t_n to t_{n-1}. To investigate the real error at a point, know *global error*, we look at $e^n = u^n - u_e(t_n)$ for some n and Taylor exp mathematical expressions as functions of $p = a\Delta t$:

```
>>> n = Symbol('n')
>>> u_e = exp(-p*n)
>>> u_n = A**n
>>> FE = u_e.series(p, 0, 4) - u_n.subs(theta, 0).series(p, 0, 4)
>>> BE = u_e.series(p, 0, 4) - u_n.subs(theta, 1).series(p, 0, 4)
>>> CN = u_e.series(p, 0, 4) - u_n.subs(theta, half).series(p, 0,
>>> FE
(1/2)*n*p**2 - 1/2*n**2*p**3 + (1/3)*n*p**3 + O(p**4)
>>> BE
(1/2)*n**2*p**3 - 1/2*n*p**2 + (1/3)*n*p**3 + O(p**4)
>>> CN
(1/12)*n*p**3 + O(p**4)
```

For a fixed time t, the parameter n in these expressions increases as $p \to$ $t = n\Delta t$ = const and hence n must increase like Δt^{-1}. With n sub by $t/\Delta t$ in the leading-order error terms, these become $\frac{1}{2}na^2\Delta t^2 = \frac{1}{2}ta$ the Forward and Backward Euler scheme, and $\frac{1}{12}na^3\Delta t^3 = \frac{1}{12}ta^3\Delta t^2$ Crank-Nicolson scheme. The global error is therefore of second order (in the latter scheme and of first order for the former schemes.

When the global error $e^n \to 0$ as $\Delta t \to 0$, we say that the scheme is *convergent*. means that the numerical solution approaches the exact solution as the mesh refined, and this is a much desired property of a numerical method.

.8 Integrated errors

is common to study the norm of the numerical error, as explained in detail in ection 2.4. The L^2 norm can be computed by treating e^n as a function of t in ympy and performing symbolic integration. For the Forward Euler scheme we ave

```
, n, a, dt, t, T, theta = symbols('p n a dt t T 'theta')
 = (1-(1-theta)*p)/(1+theta*p)
_e = exp(-p*n)
_n = A**n
rror = u_e.series(p, 0, 4) - u_n.subs(theta, 0).series(p, 0, 4)
 Introduce t and dt instead of n and p
rror = error.subs('n', 't/dt').subs(p, 'a*dt')
rror = error.as_leading_term(dt) # study only the first term
rint error
rror_L2 = sqrt(integrate(error**2, (t, 0, T)))
rint error_L2
```

he output reads

```
sqrt(30)*sqrt(T**3*a**4*dt**2*(6*T**2*a**2 - 15*T*a + 10))/60
```

hich means that the L^2 error behaves like $a^2\Delta t$.

Strictly speaking, the numerical error is only defined at the mesh points so it akes most sense to compute the ℓ^2 error

$$||e^n||_{\ell^2} = \sqrt{\Delta t \sum_{n=0}^{N_t} (u_{\rm e}(t_n) - u^n)^2}\,.$$

Ve have obtained an exact analytical expressions for the error at $t = t_n$, but ere we use the leading-order error term only since we are mostly interested in ow the error behaves as a polynomial in Δt, and then the leading order term ill dominate. For the Forward Euler scheme, $u_{\rm e}(t_n) - u^n \approx \frac{1}{2}np^2$, and we have

$$||e^n||_{\ell^2}^2 = \Delta t \sum_{n=0}^{N_t} \frac{1}{4} n^2 p^4 = \Delta t \frac{1}{4} p^4 \sum_{n=0}^{N_t} n^2\,.$$

ow, $\sum_{n=0}^{N_t} n^2 \approx \frac{1}{3}N_t^3$. Using this approximation, setting $N_t = T/\Delta t$, and aking the square root gives the expression

$$||e^n||_{\ell^2} = \frac{1}{2}\sqrt{\frac{T^3}{3}} a^2 \Delta t\,.$$

alculations for the Backward Euler scheme are very similar and provide the ame result, while the Crank-Nicolson scheme leads to

$$||e^n||_{\ell^2} = \frac{1}{12}\sqrt{\frac{T^3}{3}} a^3 \Delta t^2\,.$$

Summary of errors.
Both the point-wise and the time-integrated true errors are of second in Δt for the Crank-Nicolson scheme and of first order in Δt for the Fo Euler and Backward Euler schemes.

6.9 Truncation error

The truncation error is a very frequently used error measure for finite di methods. It is defined as *the error in the difference equation that aris inserting the exact solution*. Contrary to many other error measures, true error $e^n = u_{\rm e}(t_n) - u^n$, the truncation error is a quantity that i computable.

Let us illustrate the calculation of the truncation error for the Forwa scheme. We start with the difference equation on operator form,

$$[D_t u = -au]^n,$$

i.e.,

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

The idea is to see how well the exact solution $u_{\rm e}(t)$ fulfills this equatio $u_{\rm e}(t)$ in general will not obey the discrete equation, error in the discrete e called a *residual*, denoted here by R^n:

$$R^n = \frac{u_{\rm e}(t_{n+1}) - u_{\rm e}(t_n)}{\Delta t} + au_{\rm e}(t_n)\,.$$

The residual is defined at each mesh point and is therefore a mesh functi a superscript n.

The interesting feature of R^n is to see how it depends on the discre parameter Δt. The tool for reaching this goal is to Taylor expand $u_{\rm e}$ aro point where the difference equation is supposed to hold, here $t = t_n$. V that

$$u_{\rm e}(t_{n+1}) = u_{\rm e}(t_n) + u'_{\rm e}(t_n)\Delta t + \frac{1}{2}u''_{\rm e}(t_n)\Delta t^2 + \cdots$$

Inserting this Taylor series in (60) gives

$$R^n = u'_{\rm e}(t_n) + \frac{1}{2}u''_{\rm e}(t_n)\Delta t + \ldots + au_{\rm e}(t_n)\,.$$

Now, $u_{\rm e}$ fulfills the ODE $u'_{\rm e} = -au_{\rm e}$ such that the first and last term and we have

$$R^n \approx \frac{1}{2}u_{\rm e}''(t_n)\Delta t\,.$$

his R^n is the *truncation error*, which for the Forward Euler is seen to be of rst order in Δt.

The above procedure can be repeated for the Backward Euler and the Crank-icolson schemes. We start with the scheme in operator notation, write it out in etail, Taylor expand $u_{\rm e}$ around the point $\tilde{t}$ at which the difference equation is efined, collect terms that correspond to the ODE (here $u_{\rm e}' + au_{\rm e}$), and identify ıe remaining terms as the residual R, which is the truncation error. The ackward Euler scheme leads to

$$R^n \approx -\frac{1}{2}u_{\rm e}''(t_n)\Delta t,$$

hile the Crank-Nicolson scheme gives

$$R^{n+1/2} \approx \frac{1}{24}u_{\rm e}'''(t_{n+\frac{1}{2}})\Delta t^2\,.$$

The *order* r of a finite difference scheme is often defined through the leading erm Δt^r in the truncation error. The above expressions point out that the orward and Backward Euler schemes are of first order, while Crank-Nicolson of second order. We have looked at other error measures in other sections, ke the error in amplification factor and the error $e^n = u_{\rm e}(t_n) - u^n$, and xpressed these error measures in terms of Δt to see the order of the method. ormally, calculating the truncation error is more straightforward than deriving ıe expressions for other error measures and therefore the easiest way to establish ıe order of a scheme.

.10 Consistency, stability, and convergence

hree fundamental concepts when solving differential equations by numerical ıethods are consistency, stability, and convergence. We shall briefly touch these oncepts below in the context of the present model problem.

Consistency means that the error in the difference equation, measured through ıe truncation error, goes to zero as $\Delta t \to 0$. Since the truncation error ells how well the exact solution fulfills the difference equation, and the exact olution fulfills the differential equation, consistency ensures that the difference quation approaches the differential equation in the limit. The expressions for the uncation errors in the previous section are all proportional to Δt or Δt^2, hence ıey vanish as $\Delta t \to 0$, and all the schemes are consistent. Lack of consistency nplies that we actually solve a different differential equation in the limit $\Delta t \to 0$ ıan we aim at.

Stability means that the numerical solution exhibits the same qualitative roperties as the exact solution. This is obviously a feature we want the numerical olution to have. In the present exponential decay model, the exact solution is ıonotone and decaying. An increasing numerical solution is not in accordance ith the decaying nature of the exact solution and hence unstable. We can also

say that an oscillating numerical solution lacks the property of mono of the exact solution and is also unstable. We have seen that the Ba Euler scheme always leads to monotone and decaying solutions, regardles and is hence stable. The Forward Euler scheme can lead to increasing s and oscillating solutions if Δt is too large and is therefore unstable unle sufficiently small. The Crank-Nicolson can never lead to increasing soluti has no problem to fulfill that stability property, but it can produce osc solutions and is unstable in that sense, unless Δt is sufficiently small.

Convergence implies that the global (true) error mesh function $e^n = u^n \to 0$ as $\Delta t \to 0$. This is really what we want: the numerical solution close to the exact solution as we request by having a sufficiently fine m

Convergence is hard to establish theoretically, except in quite simple p like the present one. Stability and consistency are much easier to calcu major breakthrough in the understanding of numerical methods for diff equations came in 1956 when Lax and Richtmeyer established equi between convergence on one hand and consistency and stability on the ot Lax equivalence theorem[71]). In practice it meant that one can first establ a method is stable and consistent, and then it is automatically convergent is much harder to establish). The result holds for linear problems only the world of nonlinear differential equations the relations between cons stability, and convergence are much more complicated.

We have seen in the previous analysis that the Forward Euler, Ba Euler, and Crank-Nicolson schemes are convergent ($e^n \to 0$), that t consistent ($R^n \to 0$, and that they are stable under certain conditions size of Δt. We have also derived explicit mathematical expressions for truncation error, and the stability criteria.

7 Exercises

Exercise 15: Visualize the accuracy of finite differenc e^{-at}

The purpose of this exercise is to visualize the accuracy of finite di approximations of the derivative of a given function. For any finite di approximation, take the Forward Euler difference as an example, and any function, take $u = e^{-at}$, we may introduce an error fraction specific

$$E = \frac{[D_t^+ u]^n}{u'(t_n)} = \frac{\exp\left(-a(t_n + \Delta t)\right) - \exp\left(-at_n\right)}{-a\exp\left(-at_n\right)} = -\frac{1}{a\Delta t}\left(\exp\left(-a\Delta t\right)\right.$$

and view E as a function of Δt. We expect that $\lim_{\Delta t\to 0} E = 1$, while deviate significantly from unit for large Δt. How the error depends on Δ visualized in a graph where we use a logarithmic scale on for Δt, so we c many orders of magnitude of that quantity. Here is a code segment cre array of 100 intervals, on the logarithmic scale, ranging from 10^{-6} to 1 a plotting E versus $p = a\Delta t$ with logarithmic scale on the Δt axis:

[71] `http://en.wikipedia.org/wiki/Lax_equivalence_theorem`

```
from numpy import logspace, exp
from matplotlib.pyplot import plot
p = logspace(-6, 1, 101)
y = -(exp(-p)-1)/p
semilog(p, y)
```

lustrate such errors for the finite difference operators $[D_t^+ u]^n$ (forward), $[D_t^- u]^n$ ackward), and $[D_t u]^n$ (centered).

Perform a Taylor series expansions of the error fractions and find the leading rder r in the expressions of type $1 + C\Delta t^r + \mathcal{O}(\Delta t^{r+1})$, where C is some constant. ilename: `decay_plot_fd_exp_error.py`.

Exercise 16: Explore the θ-rule for exponential growth

his exercise asks you to solve the ODE $u' = -au$ with $a < 0$ such that the DE models exponential growth instead of exponential decay. A central theme to investigate numerical artifacts and non-physical solution behavior.

) Run experiments with θ and Δt to uncover numerical artifacts (the exact olution is a monotone, growing function). Use the insight to design a set of xperiments that aims to demonstrate all types of numerical artifacts for different hoices of Δt while a is fixed.

Hint. Modify the `decay_exper1.py` code to suit your needs.
Filename: `growth_exper.py`.

) Write a scientific report about the findings.

Hint. Use examples from Section 4.4 to see how scientific reports can be ritten.
Filenames: `growth_exper.pdf`, `growth_exper.html`.

) Plot the amplification factors for the various schemes together with the xact one for $a < 0$ and use the plot to explain the observations made in the xperiments.

Hint. Modify the `decay_ampf_plot.py`[72] code.
Filename: `growth_ampf.py`.

8 Model extensions

is time to consider generalizations of the simple decay model $u = -au$ and lso to look at additional numerical solution methods.

8.1 Generalization: including a variable coefficient

n the ODE for decay, $u' = -au$, we now consider the case where a depends on me:

$$u'(t) = -a(t)u(t), \quad t \in (0, T], \quad u(0) = I\,. \tag{61}$$

A Forward Euler scheme consist of evaluating (61) at $t = t_n$ and approximating the derivative with a forward difference $[D_t^+ u]^n$:

[72] `http://tinyurl.com/jvzzcfn/decay/decay_ampf_plot.py`

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

The Backward Euler scheme becomes

$$\frac{u^n - u^{n-1}}{\Delta t} = -a(t_n)u^n\,.$$

The Crank-Nicolson method builds on sampling the ODE at $t_{n+\frac{1}{2}}$. evaluate a at $t_{n+\frac{1}{2}}$ and use an average for u at times t_n and t_{n+1}:

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

Alternatively, we can use an average for the product au:

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

The θ-rule unifies the three mentioned schemes. One version is to evaluated at $t_{n+\theta}$,

$$\frac{u^{n+1} - u^n}{\Delta t} = -a((1-\theta)t_n + \theta t_{n+1})((1-\theta)u^n + \theta u^{n+1})\,.$$

Another possibility is to apply a weighted average for the product au,

$$\frac{u^{n+1} - u^n}{\Delta t} = -(1-\theta)a(t_n)u^n - \theta a(t_{n+1})u^{n+1}\,.$$

With the finite difference operator notation the Forward Euler and Ba Euler schemes can be summarized as

$$[D_t^+ u = -au]^n,$$
$$[D_t^- u = -au]^n\,.$$

The Crank-Nicolson and θ schemes depend on whether we evaluate sample point for the ODE or if we use an average. The various versi written as

$$[D_t u = -a\overline{u}^t]^{n+\frac{1}{2}},$$
$$[D_t u = -\overline{au}^t]^{n+\frac{1}{2}},$$
$$[D_t u = -a\overline{u}^{t,\theta}]^{n+\theta},$$
$$[D_t u = -\overline{au}^{t,\theta}]^{n+\theta}\,.$$

8.2 Generalization: including a source term

A further extension of the model ODE is to include a source term $b(t)$:

$$u'(t) = -a(t)u(t) + b(t), \quad t \in (0, T], \quad u(0) = I\,.$$

chemes. The time point where we sample the ODE determines where $b(t)$ is valuated. For the Crank-Nicolson scheme and the θ-rule we have a choice of hether to evaluate $a(t)$ and $b(t)$ at the correct point or use an average. The osen strategy becomes particularly clear if we write up the schemes in the perator notation:

$$[D_t^+ u = -au + b]^n, \tag{75}$$

$$[D_t^- u = -au + b]^n, \tag{76}$$

$$[D_t u = -a\overline{u}^t + b]^{n+\frac{1}{2}}, \tag{77}$$

$$[D_t u = \overline{-au + b}^t]^{n+\frac{1}{2}}, \tag{78}$$

$$[D_t u = -a\overline{u}^{t,\theta} + b]^{n+\theta}, \tag{79}$$

$$[D_t u = \overline{-au + b}^{t,\theta}]^{n+\theta}\,. \tag{80}$$

.3 Implementation of the generalized model problem

)eriving the θ-rule formula. Writing out the θ-rule in (80), using (32) and 3), we get

$$\frac{u^{n+1} - u^n}{\Delta t} = \theta(-a^{n+1}u^{n+1} + b^{n+1})) + (1-\theta)(-a^n u^n + b^n)), \tag{81}$$

here a^n means evaluating a at $t = t_n$ and similar for a^{n+1}, b^n, and b^{n+1}. We olve for u^{n+1}:

$$u^{n+1} = ((1 - \Delta t(1-\theta)a^n)u^n + \Delta t(\theta b^{n+1} + (1-\theta)b^n))(1 + \Delta t\theta a^{n+1})^{-1}\,. \tag{82}$$

'he Python code. Here is a suitable implementation of (81) where $a(t)$ and (t) are given as Python functions:

```
def solver(I, a, b, T, dt, theta):
    """
    Solve u'=-a(t)*u + b(t), u(0)=I,
    for t in (0,T] with steps of dt.
    a and b are Python functions of t.
    """
    dt = float(dt)            # avoid integer division
    Nt = int(round(T/dt))     # no of time intervals
    T = Nt*dt                 # adjust T to fit time step dt
    u = zeros(Nt+1)           # array of u[n] values
    t = linspace(0, T, Nt+1)  # time mesh

    u[0] = I                  # assign initial condition
    for n in range(0, Nt):    # n=0,1,...,Nt-1
        u[n+1] = ((1 - dt*(1-theta)*a(t[n]))*u[n] + \
                  dt*(theta*b(t[n+1]) + (1-theta)*b(t[n])))/\
                  (1 + dt*theta*a(t[n+1]))
    return u, t
```

his function is found in the file `decay_vc.py`[73] (`vc` stands for "variable coeffi-ents").

[73] `http://tinyurl.com/jvzzcfn/decay/decay_vc.py`

Coding of variable coefficients. The `solver` function shown above c the arguments `a` and `b` to be Python functions of time `t`, say

```
def a(t):
    return a_0 if t < tp else k*a_0

def b(t):
    return 1
```

Here, `a(t)` has three parameters `a0`, `tp`, and `k`, which must be global v A better implementation is to represent `a` by a class where the parame attributes and a *special method* `__call__` evaluates $a(t)$:

```
class A:
    def __init__(self, a0=1, k=2):
        self.a0, self.k = a0, k

    def __call__(self, t):
        return self.a0 if t < self.tp else self.k*self.a0

a = A(a0=2, k=1)  # a behaves as a function a(t)
```

For quick tests it is cumbersome to write a complete function or a cla *lambda function* construction in Python is then convenient. For examp

```
a = lambda t: a_0 if t < tp else k*a_0
```

is equivalent to the `def a(t):` definition above. In general,

```
f = lambda arg1, arg2, ...: expressin
```

is equivalent to

```
def f(arg1, arg2, ...):
    return expression
```

One can use lambda functions directly in calls. Say we want to solve $u' = u(0) = 2$:

```
u, t = solver(2, lambda t: 1, lambda t: 1, T, dt, theta)
```

A lambda function can appear anywhere where a variable can appear.

8.4 Verifying a constant solution

A very useful partial verification method is to construct a test proble a very simple solution, usually $u = \text{const}$. Especially the initial debug a program code can benefit greatly from such tests, because 1) all numerical methods will exactly reproduce a constant solution, 2) man intermediate calculations are easy to control for a constant u, and 3) constant u can uncover many bugs in an implementation.

The only constant solution for the problem $u' = -au$ is $u = 0$, but too many ugs can escape from that trivial solution. It is much better to search for a roblem where $u = C = \text{const} \neq 0$. Then $u' = -a(t)u + b(t)$ is more appropriate: ith $u = C$ we can choose any $a(t)$ and set $b = a(t)C$ and $I = C$. An appropriate ose test is

```
import nose.tools as nt

def test_constant_solution():
    """
    Test problem where u=u_const is the exact solution, to be
    reproduced (to machine precision) by any relevant method.
    """
    def exact_solution(t):
        return u_const

    def a(t):
        return 2.5*(1+t**3)  # can be arbitrary

    def b(t):
        return a(t)*u_const

    u_const = 2.15
    theta = 0.4; I = u_const; dt = 4
    Nt = 4  # enough with a few steps
    u, t = solver(I=I, a=a, b=b, T=Nt*dt, dt=dt, theta=theta)
    print u
    u_e = exact_solution(t)
    difference = abs(u_e - u).max()  # max deviation
    nt.assert_almost_equal(difference, 0, places=14)
```

An interesting question is what type of bugs that will make the computed u^n eviate from the exact solution C. Fortunately, the updating formula and the itial condition must be absolutely correct for the test to pass! Any attempt to ake a wrong indexing in terms like `a(t[n])` or any attempt to introduce an roneous factor in the formula creates a solution that is different from C.

.5 Verification via manufactured solutions

ollowing the idea of the previous section, we can choose any formula as the xact solution, insert the formula in the ODE problem and fit the data $a(t)$, $b(t)$, nd I to make the chosen formula fulfill the equation. This powerful technique or generating exact solutions is very useful for verification purposes and known s the *method of manufactured solutions*, often abbreviated MMS.

One common choice of solution is a linear function in the independent ariable(s). The rationale behind such a simple variation is that almost any elevant numerical solution method for differential equation problems is able to eproduce the linear function exactly to machine precision (if u is about unity n size; precision is lost if u take on large values, see Exercise 17). The linear olution also makes some stronger demands to the numerical method and the nplementation than the constant solution used in Section 8.4, at least in more omplicated applications. However, the constant solution is often ideal for initial ebugging before proceeding with a linear solution.

We choose a linear solution $u(t) = ct + d$. From the initial condition it that $d = I$. Inserting this u in the ODE results in

$$c = -a(t)u + b(t)\,.$$

Any function $u = ct + I$ is then a correct solution if we choose

$$b(t) = c + a(t)(ct + I)\,.$$

With this $b(t)$ there are no restrictions on $a(t)$ and c.

Let prove that such a linear solution obeys the numerical schemes. end, we must check that $u^n = ca(t_n)(ct_n + I)$ fulfills the discrete equatic these calculations, and later calculations involving linear solutions ins finite difference schemes, it is convenient to compute the action of a di operator on a linear function t:

$$\begin{aligned}
[D_t^+ t]^n &= \frac{t_{n+1} - t_n}{\Delta t} = 1,\\
[D_t^- t]^n &= \frac{t_n - t_{n-1}}{\Delta t} = 1,\\
[D_t t]^n &= \frac{t_{n+\frac{1}{2}} - t_{n-\frac{1}{2}}}{\Delta t} = \frac{(n+\frac{1}{2})\Delta t - (n-\frac{1}{2})\Delta t}{\Delta t} = 1\,.
\end{aligned}$$

Clearly, all three finite difference approximations to the derivative are e $u(t) = t$ or its mesh function counterpart $u^n = t_n$.

The difference equation for the Forward Euler scheme

$$[D_t^+ u = -au + b]^n,$$

with $a^n = a(t_n)$, $b^n = c + a(t_n)(ct_n + I)$, and $u^n = ct_n + I$ then results

$$c = -a(t_n)(ct_n + I) + c + a(t_n)(ct_n + I) = c$$

which is always fulfilled. Similar calculations can be done for the Ba Euler and Crank-Nicolson schemes, or the θ-rule for that matter. In a $u^n = ct_n + I$ is an exact solution of the discrete equations. That is should expect that $u^n - u_{\mathrm{e}}(t_n) = 0$ mathematically and $|u^n - u_{\mathrm{e}}(t_n)|$ l a small number about the machine precision for $n = 0, \ldots, N_t$.

The following function offers an implementation of this verification te on a linear exact solution:

```
def test_linear_solution():
    """
    Test problem where u=c*t+I is the exact solution, to be
    reproduced (to machine precision) by any relevant method.
    """
    def exact_solution(t):
        return c*t + I
```

```
def a(t):
    return t**0.5  # can be arbitrary

def b(t):
    return c + a(t)*exact_solution(t)

theta = 0.4; I = 0.1; dt = 0.1; c = -0.5
T = 4
Nt = int(T/dt)  # no of steps
u, t = solver(I=I, a=a, b=b, T=Nt*dt, dt=dt, theta=theta)
u_e = exact_solution(t)
difference = abs(u_e - u).max()  # max deviation
print difference
# No of decimal places for comparison depend on size of c
nt.assert_almost_equal(difference, 0, places=14)
```

ny error in the updating formula makes this test fail!

Choosing more complicated formulas as the exact solution, say $\cos(t)$, will not ıake the numerical and exact solution coincide to machine precision, because nite differencing of $\cos(t)$ does not exactly yield the exact derivative $-\sin(t)$. In ıch cases, the verification procedure must be based on measuring the convergence ıtes as exemplified in Section 2.8. Convergence rates can be computed as long s one has an exact solution of a problem that the solver can be tested on, but ıis can always be obtained by the method of manufactured solutions.

.6 Extension to systems of ODEs

Iany ODE models involves more than one unknown function and more than ne equation. Here is an example of two unknown functions $u(t)$ and $v(t)$:

$$u' = au + bv, \tag{86}$$
$$v' = cu + dv, \tag{87}$$

or constants a, b, c, d. Applying the Forward Euler method to each equation esults in simple updating formula

$$u^{n+1} = u^n + \Delta t(au^n + bv^n), \tag{88}$$
$$v^{n+1} = u^n + \Delta t(cu^n + dv^n)\,. \tag{89}$$

n the other hand, the Crank-Nicolson or Backward Euler schemes result in a $\times$ 2 linear system for the new unknowns. The latter schemes gives

$$u^{n+1} = u^n + \Delta t(au^{n+1} + bv^{n+1}), \tag{90}$$
$$v^{n+1} = v^n + \Delta t(cu^{n+1} + dv^{n+1})\,. \tag{91}$$

ollecting u^{n+1} as well as v^{n+1} on the left-hand side results in

$$(1 - \Delta ta)u^{n+1} + bv^{n+1} = u^n, \tag{92}$$
$$cu^{n+1} + (1 - \Delta td)v^{n+1} = v^n, \tag{93}$$

hich is a system of two coupled, linear, algebraic equations in two unknowns.

9 General first-order ODEs

We now turn the attention to general, nonlinear ODEs and systems ODEs. Our focus is on numerical methods that can be readily reused f discretization PDEs, and diffusion PDEs in particular. The methods briefly listed, and we refer to the rich literature for more detailed desc and analysis - the books [**?**, **?**, **?**, **?**] are all excellent resources on nu methods for ODEs. We also demonstrate the Odespy Python interfa range of different software for general first-order ODE systems.

9.1 Generic form

ODEs are commonly written in the generic form

$$u' = f(u, t), \quad u(0) = I,$$

where $f(u, t)$ is some prescribed function. As an example, our most exponential decay model (74) has $f(u, t) = -a(t)u(t) + b(t)$.

The unknown u in (94) may either be a scalar function of time t, or valued function of t in case of a *system of ODEs* with m unknown com

$$u(t) = (u^{(0)}(t), u^{(1)}(t), \ldots, u^{(m-1)}(t))\,.$$

In that case, the right-hand side is vector-valued function with m com

$$\begin{aligned} f(u, t) = (&f^{(0)}(u^{(0)}(t), \ldots, u^{(m-1)}(t)),\\ &f^{(1)}(u^{(0)}(t), \ldots, u^{(m-1)}(t)),\\ &\vdots,\\ &f^{(m-1)}(u^{(0)}(t), \ldots, u^{(m-1)}(t)))\,. \end{aligned}$$

Actually, any system of ODEs can be written in the form (94), but order ODEs then need auxiliary unknown functions to enable convers first-order system.

Next we list some well-known methods for $u' = f(u, t)$, valid bo single ODE (scalar u) and systems of ODEs (vector u). The choice of ı is inspired by the kind of schemes that are popular also for partial diff equations.

9.2 The θ-rule

The θ-rule scheme applied to $u' = f(u, t)$ becomes

$$\frac{u^{n+1} - u^n}{\Delta t} = \theta f(u^{n+1}, t_{n+1}) + (1 - \theta)f(u^n, t_n)\,.$$

Bringing the unknown u^{n+1} to the left-hand side and the known terms right-hand side gives

$$u^{n+1} - \Delta t\theta f(u^{n+1}, t_{n+1}) = u^n + \Delta t(1 - \theta)f(u^n, t_n)\,.$$

or a general f (not linear in u), this equation is *nonlinear* in the unknown u^{n+1} nless $\theta = 0$. For a scalar ODE ($m = 1$), we have to solve a single nonlinear lgebraic equation for u^{n+1}, while for a system of ODEs, we get a system of oupled, nonlinear algebraic equations. Newton's method is a popular solution pproach in both cases. Note that with the Forward Euler scheme ($\theta = 0$) we o not have to deal with nonlinear equations, because in that case we have an xplicit updating formula for u^{n+1}. This is known as an *explicit* scheme. With $\neq 1$ we have to solve systems of algebraic equations, and the scheme is said to e *implicit*.

.3 An implicit 2-step backward scheme

he implicit backward method with 2 steps applies a three-level backward ifference as approximation to $u'(t)$,

$$u'(t_{n+1}) \approx \frac{3u^{n+1} - 4u^n + u^{n-1}}{2\Delta t},$$

hich is an approximation of order Δt^2 to the first derivative. The resulting cheme for $u' = f(u,t)$ reads

$$u^{n+1} = \frac{4}{3}u^n - \frac{1}{3}u^{n-1} + \frac{2}{3}\Delta t f(u^{n+1}, t_{n+1})\,. \tag{97}$$

igher-order versions of the scheme (97) can be constructed by including more me levels. These schemes are known as the Backward Differentiation Formulas BDF), and the particular version (97) is often referred to as BDF2.

Note that the scheme (97) is implicit and requires solution of nonlinear quations when f is nonlinear in u. The standard 1st-order Backward Euler ethod or the Crank-Nicolson scheme can be used for the first step.

.4 Leapfrog schemes

he ordinary Leapfrog scheme. The derivative of u at some point t_n can e approximated by a central difference over two time steps,

$$u'(t_n) \approx \frac{u^{n+1} - u^{n-1}}{2\Delta t} = [D_{2t}u]^n \tag{98}$$

hich is an approximation of second order in Δt. The scheme can then be ritten as

$$[D_{2t}u = f(u,t)]^n,$$

ı operator notation. Solving for u^{n+1} gives

$$u^{n+1} = u^{n-1} + \Delta t f(u^n, t_n)\,. \tag{99}$$

bserve that (99) is an explicit scheme, and that a nonlinear f (in u) is trivial o handle since it only involves the known u^n value. Some other scheme must e used as starter to compute u^1, preferably the Forward Euler scheme since it also explicit.

The filtered Leapfrog scheme. Unfortunately, the Leapfrog sche will develop growing oscillations with time (see Problem 22)[[[. A rem such undesired oscillations is to introduce a *filtering technique*. First, a s Leapfrog step is taken, according to (99), and then the previous u^n adjusted according to

$$u^n \;\leftarrow\; u^n + \gamma(u^{n-1} - 2u^n + u^{n+1})\,.$$

The γ-terms will effectively damp oscillations in the solution, especiall with short wavelength (like point-to-point oscillations). A common choic 0.6 (a value used in the famous NCAR Climate Model).

9.5 The 2nd-order Runge-Kutta scheme

The two-step scheme

$$u^* = u^n + \Delta t f(u^n, t_n),$$
$$u^{n+1} = u^n + \Delta t \frac{1}{2}\left(f(u^n, t_n) + f(u^*, t_{n+1})\right),$$

essentially applies a Crank-Nicolson method (102) to the ODE, but ı the term $f(u^{n+1}, t_{n+1})$ by a prediction $f(u^*, t_{n+1})$ based on a Forwar step (101). The scheme (101)-(102) is known as Huen's method, but a 2nd-order Runge-Kutta method. The scheme is explicit, and the expected to behave as Δt^2.

9.6 A 2nd-order Taylor-series method

One way to compute u^{n+1} given u^n is to use a Taylor polynomial. We m up a polynomial of 2nd degree:

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

From the equation $u' = f(u,t)$ it follows that the derivatives of u can be ex in terms of f and its derivatives:

$$u'(t_n) = f(u^n, t_n),$$
$$u''(t_n) = \frac{\partial f}{\partial u}(u^n, t_n)u'(t_n) + \frac{\partial f}{\partial t}$$
$$= f(u^n, t_n)\frac{\partial f}{\partial u}(u^n, t_n) + \frac{\partial f}{\partial t},$$

resulting in the scheme

$$u^{n+1} = u^n + f(u^n, t_n)\Delta t + \frac{1}{2}\left(f(u^n, t_n)\frac{\partial f}{\partial u}(u^n, t_n) + \frac{\partial f}{\partial t}\right)\Delta t^2\,.$$

More terms in the series could be included in the Taylor polynomial tc methods of higher order than 2.

.7 The 2nd- and 3rd-order Adams-Bashforth schemes

he following method is known as the 2nd-order Adams-Bashforth scheme:

$$u^{n+1} = u^n + \frac{1}{2}\Delta t\left(3f(u^n,t_n) - f(u^{n-1},t_{n-1})\right)\,. \tag{104}$$

he scheme is explicit and requires another one-step scheme to compute u^1 (the orward Euler scheme or Heun's method, for instance). As the name implies, ıe scheme is of order Δt^2.

Another explicit scheme, involving four time levels, is the 3rd-order Adams-ashforth scheme

$$u^{n+1} = u^n + \frac{1}{12}\left(23f(u^n,t_n) - 16f(u^{n-1},t_{n-1}) + 5f(u^{n-2},t_{n-2})\right)\,. \tag{105}$$

he numerical error is of order Δt^3, and the scheme needs some method for ɔmputing u^1 and u^2.

More general, higher-order Adams-Bashforth schemes (also called *explicit dams methods*) compute u^{n+1} as a linear combination of f at k previous time :eps:

$$u^{n+1} = u^n + \sum_{j=0}^{k}\beta_j f(u^{n-j},t_{n-j}),$$

here β_j are known coefficients.

.8 4th-order Runge-Kutta scheme

he perhaps most widely used method to solve ODEs is the 4th-order Runge-:utta method, often called RK4. Its derivation is a nice illustration of common umerical approximation strategies, so let us go through the steps in detail.

The starting point is to integrate the ODE $u' = f(u,t)$ from t_n to t_{n+1}:

$$u(t_{n+1}) - u(t_n) = \int_{t_n}^{t_{n+1}} f(u(t),t)dt\,.$$

Ve want to compute $u(t_{n+1})$ and regard $u(t_n)$ as known. The task is to find ɔod approximations for the integral, since the integrand involves the unknown between t_n and t_{n+1}.

The integral can be approximated by the famous Simpson's rule[74]:

$$\int_{t_n}^{t_{n+1}} f(u(t),t)dt \approx \frac{\Delta t}{6}\left(f^n + 4f^{n+1/2} + f^{n+1}\right)\,.$$

[74] http://en.wikipedia.org/wiki/Simpson's_rule

The problem now is that we do not know $f^{n+1/2} = f(u^{n+1/2},t_{n+1}$ $f^{n+1} = (u^{n+1},t_{n+1})$ as we know only u^n and hence f^n. The idea is various approximations for $f^{n+1/2}$ and f^{n+1} based on using well-known for the ODE in the intervals $[t_n,t_{n+1/2}]$ and $[t_n,t_{n+1}]$. We split the approximation into four terms:

$$\int_{t_n}^{t_{n+1}} f(u(t),t)dt \approx \frac{\Delta t}{6}\left(f^n + 2\hat{f}^{n+1/2} + 2\tilde{f}^{n+1/2} + \bar{f}^{n+1}\right),$$

where $\hat{f}^{n+1/2}$, $\tilde{f}^{n+1/2}$, and $\bar{f}^{n+1}$ are approximations to $f^{n+1/2}$ and f^n can be based on already computed quantities. For $\hat{f}^{n+1/2}$ we can a approximation to $u^{n+1/2}$ using the Forward Euler method with step $\frac{1}{2}\Delta$

$$\hat{f}^{n+1/2} = f(u^n + \frac{1}{2}\Delta t f^n, t_{n+1/2})$$

Since this gives us a prediction of $f^{n+1/2}$, we can for $\tilde{f}^{n+1/2}$ try a Ba Euler method to approximate $u^{n+1/2}$:

$$\tilde{f}^{n+1/2} = f(u^n + \frac{1}{2}\Delta t\hat{f}^{n+1/2}, t_{n+1/2})\,.$$

With $\tilde{f}^{n+1/2}$ as a hopefully good approximation to $f^{n+1/2}$, we can for t term $\bar{f}^{n+1}$ use a Crank-Nicolson method to approximate u^{n+1}:

$$\bar{f}^{n+1} = f(u^n + \Delta t\hat{f}^{n+1/2}, t_{n+1})\,.$$

We have now used the Forward and Backward Euler methods as wel Crank-Nicolson method in the context of Simpson's rule. The hope is t combination of these methods yields an overall time-stepping scheme fro t_n+1 that is much more accurate than the $\mathcal{O}(\Delta t)$ and $\mathcal{O}(\Delta t^2)$ of the in steps. This is indeed true: the overall accuracy is $\mathcal{O}(\Delta t^4)$!

To summarize, the 4th-order Runge-Kutta method becomes

$$u^{n+1} = u^n + \frac{\Delta t}{6}\left(f^n + 2\hat{f}^{n+1/2} + 2\tilde{f}^{n+1/2} + \bar{f}^{n+1}\right),$$

where the quantities on the right-hand side are computed from (10 Note that the scheme is fully explicit so there is never any need to solv or nonlinear algebraic equations. However, the stability is conditio depends on f. There is a whole range of *implicit* Runge-Kutta methods unconditionally stable, but require solution of algebraic equations involv each time step.

The simplest way to explore more sophisticated methods for OD apply one of the many high-quality software packages that exist, as t section explains.

.9 The Odespy software

wide range of the methods and software exist for solving (94). Many of methods re accessible through a unified Python interface offered by the Odespy[75] package. Odespy features simple Python implementations of the most fundamental schemes s well as Python interfaces to several famous packages for solving ODEs: ODEPACK[76], Vode[77], rkc.f[78], rkf45.f[79], Radau5[80], as well as the ODE solvers n SciPy[81], SymPy[82], and odelab[83].

The usage of Odespy follows this setup for the ODE $u' = -au$, $u(0) = I$, $\in (0, T]$, here solved by the famous 4th-order Runge-Kutta method, using $t = 1$ and $N_t = 6$ steps:

```
def f(u, t):
    return -a*u

import odespy
import numpy as np

I = 1; a = 0.5; Nt = 6; dt = 1
solver = odespy.RK4(f)
solver.set_initial_condition(I)
t_mesh = np.linspace(0, Nt*dt, Nt+1)
u, t = solver.solve(t_mesh)
```

The previously listed methods for ODEs are all accessible in Odespy:

- the θ-rule: `ThetaRule`
- special cases of the θ-rule: `ForwardEuler`, `BackwardEuler`, `CrankNicolson`
- the 2nd- and 4th-order Runge-Kutta methods: `RK2` and `RK4`
- The BDF methods and the Adam-Bashforth methods: `Vode`, `Lsode`, `Lsoda`, `lsoda_scipy`
- The Leapfrog scheme: `Leapfrog` and `LeapfrogFiltered`

.10 Example: Runge-Kutta methods

ince all solvers have the same interface in Odespy, modulo different set of arameters to the solvers' constructors, one can easily make a list of solver bjects and run a loop for comparing (a lot of) solvers. The code below, found n complete form in `decay_odespy.py`[84], compares the famous Runge-Kutta ethods of orders 2, 3, and 4 with the exact solution of the decay equation

[75] https://github.com/hplgit/odespy
[76] https://computation.llnl.gov/casc/odepack/odepack_home.html
[77] https://computation.llnl.gov/casc/odepack/odepack_home.html
[78] http://www.netlib.org/ode/rkc.f
[79] http://www.netlib.org/ode/rkf45.f
[80] http://www.unige.ch/ hairer/software.html
[81] http://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.ode.html
[82] http://docs.sympy.org/dev/modules/mpmath/calculus/odes.html
[83] http://olivierverdier.github.com/odelab/
[84] http://tinyurl.com/jvzzcfn/decay/decay_odespy.py

$u' = -au$. Since we have quite long time steps, we have included t relevant θ-rule for large time steps, the Backward Euler scheme ($\theta = 1$), Figure 21 shows the results.

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

def f(u, t):
    return -a*u

I = 1; a = 2; T = 6
dt = float(sys.argv[1]) if len(sys.argv) >= 2 else 0.75
Nt = int(round(T/dt))
t = np.linspace(0, Nt*dt, Nt+1)

solvers = [odespy.RK2(f),
           odespy.RK3(f),
           odespy.RK4(f),
           odespy.BackwardEuler(f, nonlinear_solver='Newton')]

legends = []
for solver in solvers:
    solver.set_initial_condition(I)
    u, t = solver.solve(t)

    plt.plot(t, u)
    plt.hold('on')
    legends.append(solver.__class__.__name__)

# Compare with exact solution plotted on a very fine mesh
t_fine = np.linspace(0, T, 10001)
u_e = I*np.exp(-a*t_fine)
plt.plot(t_fine, u_e, '-') # avoid markers by specifying line typ
legends.append('exact')

plt.legend(legends)
plt.title('Time step: %g' % dt)
plt.show()
```

Visualization tip.
We use SciTools for plotting here, but importing `matplotlib.pypl` `plt` instead also works. However, plain use of Matplotlib as done results in curves with different colors, which may be hard to distin on black-and-white paper. Using SciTools, curves are automatically colors *and* markers, thus making curves easy to distinguish on screen colors and on black-and-white paper. The automatic adding of mark normally a bad idea for a very fine mesh since all the markers get clut but SciTools limits the number of markers in such cases. For the solution we use a very fine mesh, but in the code above we specif line type as a solid line (`-`), which means no markers and just a co be automatically determined by the backend used for plotting (Matp by default, but SciTools gives the opportunity to use other backen produce the plot, e.g., Gnuplot or Grace).

Also note the that the legends are based on the class names of the solvers, and in Python the name of a the class type (as a string) of an object `obj` is obtained by `obj.__class__.__name__`.

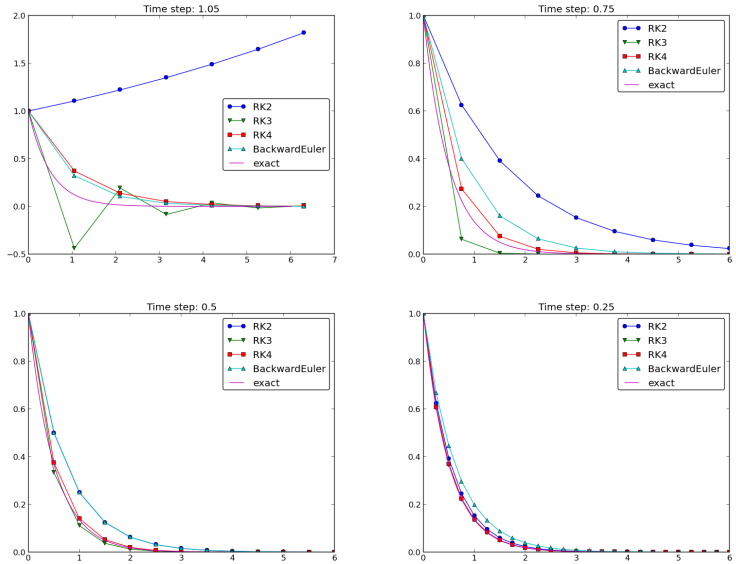


Figure 21: Behavior of different schemes for the decay equation.

The runs in Figure 21 and other experiments reveal that the 2nd-order unge-Kutta method (`RK2`) is unstable for $\Delta t > 1$ and decays slower than ıe Backward Euler scheme for large and moderate Δt (see Exercise 21 for ı analysis). However, for fine $\Delta t = 0.25$ the 2nd-order Runge-Kutta method pproaches the exact solution faster than the Backward Euler scheme. That is, ıe latter scheme does a better job for larger Δt, while the higher order scheme superior for smaller Δt. This is a typical trend also for most schemes for rdinary and partial differential equations.

The 3rd-order Runge-Kutta method (`RK3`) has also artifacts in form of scillatory behavior for the larger Δt values, much like that of the Crank-icolson scheme. For finer Δt, the 3rd-order Runge-Kutta method converges uickly to the exact solution.

The 4th-order Runge-Kutta method (`RK4`) is slightly inferior to the Backward uler scheme on the coarsest mesh, but is then clearly superior to all the other chemes. It is definitely the method of choice for all the tested schemes.

emark about using the θ-rule in Odespy. The Odespy package assumes ıat the ODE is written as $u' = f(u,t)$ with an f that is possibly nonlinear in u.

The θ-rule for $u' = f(u,t)$ leads to

$$u^{n+1} = u^n + \Delta t \left(\theta f(u^{n+1}, t_{n+1}) + (1-\theta) f(u^n, t_n)\right),$$

which is a *nonlinear equation* in u^{n+1}. Odespy's implementation of th (`ThetaRule`) and the specialized Backward Euler (`BackwardEuler`) and Nicolson (`CrankNicolson`) schemes must invoke iterative methods for the nonlinear equation in u^{n+1}. This is done even when f is linear in the model problem $u' = -au$, where we can easily solve for u^{n+1} b Therefore, we need to specify use of Newton's method to the equations. allows other methods than Newton's to be used, for instance Picard it but that method is not suitable. The reason is that it applies the Forwa scheme to generate a start value for the iterations. Forward Euler may g wrong solutions for large Δt values. Newton's method, on the other h insensitive to the start value in *linear problems*.)

9.11 Example: Adaptive Runge-Kutta methods

Odespy offers solution methods that can adapt the size of Δt with time t a desired accuracy in the solution. Intuitively, small time steps will be cl areas where the solution is changing rapidly, while larger time steps can where the solution is slowly varying. Some kind of *error estimator* is adjust the next time step at each time level.

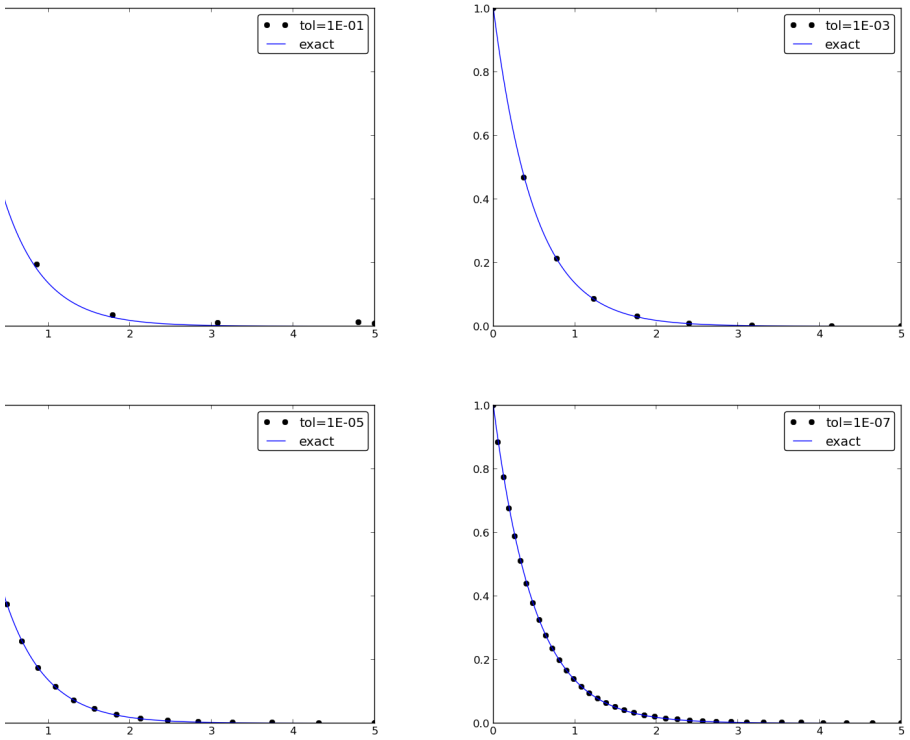
A very popular adaptive method for solving ODEs is the Dormanc Runge-Kutta method of order 4 and 5. The 5th-order method is us reference solution and the difference between the 4th- and 5th-order me used as an indicator of the error in the numerical solution. The Dormanc method is the default choice in MATLAB's widely used `ode45` routine.

We can easily set up Odespy to use the Dormand-Prince method how it selects the optimal time steps. To this end, we request only one ti from $t = 0$ to $t = T$ and ask the method to compute the necessary non- time mesh to meet a certain error tolerance. The code goes like

```
import odespy
import numpy as np
import decay_mod
import sys
#import matplotlib.pyplot as plt
import scitools.std as plt

def f(u, t):
    return -a*u

def exact_solution(t):
    return I*np.exp(-a*t)

I = 1; a = 2; T = 5
tol = float(sys.argv[1])
solver = odespy.DormandPrince(f, atol=tol, rtol=0.1*tol)

Nt = 1  # just one step - let the scheme find its intermediate p
t_mesh = np.linspace(0, T, Nt+1)
```

```
t_fine = np.linspace(0, T, 10001)

solver.set_initial_condition(I)
u, t = solver.solve(t_mesh)

# u and t will only consist of [I, u^Nt] and [0,T]
# solver.u_all and solver.t_all contains all computed points
plt.plot(solver.t_all, solver.u_all, 'ko')
plt.hold('on')
plt.plot(t_fine, exact_solution(t_fine), 'b-')
plt.legend(['tol=%.0E' % tol, 'exact'])
plt.savefig('tmp_odespy_adaptive.png')
plt.show()
```

Running four cases with tolerances 10^{-1}, 10^{-3}, 10^{-5}, and 10^{-7}, gives the esults in Figure 22. Intuitively, one would expect denser points in the beginning f the decay and larger time steps when the solution flattens out.



igure 22: Choice of adaptive time mesh by the Dormand-Prince method for ifferent tolerances.

10 Exercises

Exercise 17: Experiment with precision in tests an size of u

It is claimed in Section 8.5 that most numerical methods will reproduce a l act solution to machine precision. Test this assertion using the nose test f `test_linear_solution` in the `decay_vc.py`[85] program. Vary the para from very small, via `c=1` to many larger values, and print out the maximu ence between the numerical solution and the exact solution. What is the value of the `places` (or `delta`) argument to `nose.tools.assert_almost` in each case? Filename: `test_precision.py`.

Exercise 18: Implement the 2-step backward scheme

Implement the 2-step backward method (97) for the model $u'(t) = -a($ $b(t)$, $u(0) = I$. Allow the first step to be computed by either the Ba Euler scheme or the Crank-Nicolson scheme. Verify the implementa choosing $a(t)$ and $b(t)$ such that the exact solution is linear in t (see Sect Show mathematically that a linear solution is indeed a solution of the equations.

Compute convergence rates (see Section 2.8) in a test case $a = co$ $b = 0$, where we easily have an exact solution, and determine if the c a first-order scheme (Backward Euler) for the first step has any impact overall accuracy of this scheme. The expected error goes like $\mathcal{O}(\Delta t^2)$. F `decay_backward2step.py`.

Exercise 19: Implement the 2nd-order Adams-Basl scheme

Implement the 2nd-order Adams-Bashforth method (104) for the decay $u' = -a(t)u + b(t)$, $u(0) = I$, $t \in (0, T]$. Use the Forward Euler method first step such that the overall scheme is explicit. Verify the impleme using an exact solution that is linear in time. Analyze the scheme by se for solutions $u^n = A^n$ when $a = \text{const}$ and $b = 0$. Compare this secon secheme to the Crank-Nicolson scheme. Filename: `decay_AdamsBashfor`

Exercise 20: Implement the 3rd-order Adams-Basl scheme

Implement the 3rd-order Adams-Bashforth method (105) for the decay $u' = -a(t)u+b(t)$, $u(0) = I$, $t \in (0, T]$. Since the scheme is explicit, allow started by two steps with the Forward Euler method. Investigate experi the case where $b = 0$ and a is a constant: Can we have oscillatory solut large Δt? Filename: `decay_AdamsBashforth3.py`.

[85] `http://tinyurl.com/jvzzcfn/decay/decay_vc.py`

Exercise 21: Analyze explicit 2nd-order methods

how that the schemes (102) and (103) are identical in the case $f(u,t) = -a$, where $a > 0$ is a constant. Assume that the numerical solution reads $^n = A^n$ for some unknown amplification factor A to be determined. Find . and derive stability criteria. Can the scheme produce oscillatory solutions f $u' = -au$? Plot the numerical and exact amplification factor. Filename: `ecay_RK2_Taylor2.py`.

Problem 22: Implement and investigate the Leapfrog scheme

. Leapfrog scheme for the ODE $u'(t) = -a(t)u(t) + b(t)$ is defined by

$$[D_{2t}u = -au + b]^n .$$

. separate method is needed to compute u^1. The Forward Euler scheme is a ossible candidate.

) Implement the Leapfrog scheme for the model equation. Plot the solution ı the case $a = 1$, $b = 0$, $I = 1$, $\Delta t = 0.01$, $t \in [0, 4]$. Compare with the exact olution $u_e(t) = e^{-t}$.

) Show mathematically that a linear solution in t fulfills the Forward Euler cheme for the first step and the Leapfrog scheme for the subsequent steps. Use ıis linear solution to verify the implementation, and automate the verification ırough a nose test.

lint. It can be wise to automate the calculations such that it is easy to redo ıe calculations for other types of solutions. Here is a possible `sympy` function ıat takes a symbolic expression `u` (implemented as a Python function of `t`), fits ıe `b` term, and checks if `u` fulfills the discrete equations:

```
import sympy as sm

def analyze(u):
    t, dt, a = sm.symbols('t dt a')

    print 'Analyzing u_e(t)=%s' % u(t)
    print 'u(0)=%s' % u(t).subs(t, 0)

    # Fit source term to the given u(t)
    b = sm.diff(u(t), t) + a*u(t)
    b = sm.simplify(b)
    print 'Source term b:', b

    # Residual in discrete equations; Forward Euler step
    R_step1 = (u(t+dt) - u(t))/dt + a*u(t) - b
    R_step1 = sm.simplify(R_step1)
    print 'Residual Forward Euler step:', R_step1

    # Residual in discrete equations; Leapfrog steps
    R = (u(t+dt) - u(t-dt))/(2*dt) + a*u(t) - b
    R = sm.simplify(R)
    print 'Residual Leapfrog steps:', R

def u_e(t):
```



```
    return c*t + I

analyze(u_e)
# or short form: analyze(lambda t: c*t + I)
```

c) Show that a second-order polynomial in t cannot be a solution of the equations. However, if a Crank-Nicolson scheme is used for the first second-order polynomial solves the equations exactly.

d) Create a manufactured solution $u(t) = \sin(t)$ for the ODE $u' = -$ Compute the convergence rate of the Leapfrog scheme using this manuf solution. The expected convergence rate of the Leapfrog scheme is $\mathcal{O}(\Delta t^2$ the use of a 1st-order method for the first step impact the convergence

e) Set up a set of experiments to demonstrate that the Leapfrog sche is associated with numerical artifacts (instabilities). Document the mair from this investigation.

f) Analyze and explain the instabilities of the Leapfrog scheme (99):

1. Choose $a = \text{const}$ and $b = 0$. Assume that an exact solution of the equations has the form $u^n = A^n$, where A is an amplification fa be determined. Derive an equation for A by inserting $u^n = A^n$ Leapfrog scheme.

2. Compute A either by hand and/or with the aid of `sympy`. The pol for A has two roots, A_1 and A_2. Let u^n be a linear combinatic $C_1A_1^n + C_2A_2^n$.

3. Show that one of the roots is the explanation of the instability.

4. Compare A with the exact expression, using a Taylor series approx

5. How can C_1 and C_2 be determined?

g) Since the original Leapfrog scheme is unconditionally unstable grows, it demands some stabilization. This can be done by filtering we first find u^{n+1} from the original Leapfrog scheme and then replac $u^n + \gamma(u^{n-1} - 2u^n + u^{n+1})$, where γ can be taken as 0.6. Implement the Leapfrog scheme and check that it can handle tests where the original I scheme is unstable.

Filenames: `decay_leapfrog.py`, `decay_leapfrog.pdf`.

Problem 23: Make a unified implementation of many s

Consider the linear ODE problem $u'(t) = -a(t)u(t) + b(t)$, $u(0) = I$. schemes for this problem can be written in the general form

$$u^{n+1} = \sum_{j=0}^{m} c_j u^{n-j},$$

or some choice of $c_0, \ldots, c_m$. Find expressions for the c_j coefficients in case of he θ-rule, the three-level backward scheme, the Leapfrog scheme, the 2nd-order unge-Kutta method, and the 3rd-order Adams-Bashforth scheme.

Make a class `ExpDecay` that implements the general updating formula (110). he formula cannot be applied for $n < m$, and for those n values, other schemes ust be used. Assume for simplicity that we just repeat Crank-Nicolson steps ntil (110) can be used. Use a subclass to specify the list $c_0, \ldots, c_m$ for a articular method, and implement subclasses for all the mentioned schemes. erify the implementation by testing with a linear solution, which should be xactly reproduced by all methods. Filename: `decay_schemes_oo.py`.

.1 Applications of exponential decay models

his section presents many mathematical models that all end up with ODEs of he type $u' = -au + b$. The applications are taken from biology, finance, and hysics, and cover population growth or decay, compound interest and inflation, adioactive decay, cooling of objects, compaction of geological media, pressure ariations in the atmosphere, and air resistance on falling or rising bodies.

1.1 Scaling

eal applications of a model $u' = -au + b$ will often involve a lot of parameters n the expressions for a and b. It can be quite a challenge to find relevant values f all parameters. In simple problems, however, it turns out that it is not always ecessary to estimate all parameters because we can lump them into one or a few *imensionless* numbers by using a very attractive technique called scaling. It mply means to stretch the u and t axis is the present problem - and suddenly all arameters in the problem are lumped one parameter if $b \neq 0$ and no parameter hen $b = 0$!

Scaling means that we introduce a new function $\bar{u}(\bar{t})$, with

$$\bar{u} = \frac{u - u_m}{u_c}, \quad \bar{t} = \frac{t}{t_c},$$

here u_m is a characteristic value of u, u_c is a characteristic size of the range of u alues, and t_c is a characteristic size of the range of t_c where u varies significantly. hoosing u_m, u_c, and t_c is not always easy and often an art in complicated roblems. We just state one choice first:

$$u_c = I, \quad u_m = b/a, \quad t_c = 1/a\,.$$

nserting $u = u_m + u_c\bar{u}$ and $t = t_c\bar{t}$ in the problem $u' = -au + b$, assuming a nd b are constants, results after some algebra in the *scaled problem*

$$\frac{d\bar{u}}{d\bar{t}} = -\bar{u}, \quad \bar{u}(0) = 1 - \beta,$$

here β is a dimensionless number

$$\beta = \frac{b}{Ia}\,.$$

That is, only the special combination of $b/(Ia)$ matters, not what the in values of b, a, and I are. Moreover, if $b = 0$, the scaled problem is inde of a and I! In practice this means that we can perform one numerical sin of the scaled problem and recover the solution of any problem for a give I by stretching the axis in the plot: $u = I\bar{u}$ and $t = \bar{t}/a$. For $b \neq 0$, we s the scaled problem for a few β values and recover the physical soluti translating and stretching the u axis and stretching the t axis.

The scaling breaks down if $I = 0$. In that case we may choose $u_c = b/a$, and $t_c = 1/b$, resulting in a slightly different scaled problem:

$$\frac{d\bar{u}}{d\bar{t}} = 1 - \bar{u}, \quad \bar{u}(0) = 0\,.$$

As with $b = 0$, the case $I = 0$ has a scaled problem with no physical para

It is common to drop the bars after scaling and write the scaled p as $u' = -u$, $u(0) = 1 - \beta$, or $u' = 1 - u$, $u(0) = 0$. Any implementatio problem $u' = -au + b$, $u(0) = I$, can be reused for the scaled problem by $a = 1$, $b = 0$, and $I = 1 - \beta$ in the code, if $I \neq 0$, or one sets $a = 1$ and $I = 0$ when the physical I is zero. Falling bodies in fluids, as desc Section 11.8, involves $u' = -au + b$ with seven physical parameters. A vanish in the scaled version of the problem if we start the motion from

11.2 Evolution of a population

Let N be the number of individuals in a population occupying some domain. Despite N being an integer in this problem, we shall compute as a real number and view $N(t)$ as a continuous function of time. Th model assumption is that in a time interval Δt the number of newcomer populations (newborns) is proportional to N, with proportionality co The amount of newcomers will increase the population and result in to

$$N(t + \Delta t) = N(t) + \bar{b}N(t)\,.$$

It is obvious that a long time interval Δt will result in more newcom hence a larger $\bar{b}$. Therefore, we introduce $b = \bar{b}/\Delta t$: the number of new per unit time and per individual. We must then multiply b by the lengt time interval considered and by the population size to get the total nu new individuals, $b\Delta tN$.

If the number of removals from the population (deaths) is also prop to N, with proportionality constant $d\Delta t$, the population evolves accord

$$N(t + \Delta t) = N(t) + b\Delta tN(t) - d\Delta tN(t)\,.$$

Dividing by Δt and letting $\Delta t \to 0$, we get the ODE

$$N' = (b - d)N, \quad N(0) = N_0\,.$$

In a population where the death rate (d) is larger than then newborn $a > 0$, and the population experiences exponential decay rather than exp growth.

In some populations there is an immigration of individuals into the spatial omain. With I individuals coming in per time unit, the equation for the opulation change becomes

$$N(t+\Delta t) = N(t) + b\Delta t N(t) - d\Delta t N(t) + \Delta t I\,.$$

he corresponding ODE reads

$$N' = (b-d)N + I, \quad N(0) = N_0\,. \tag{112}$$

Some simplification arises if we introduce a fractional measure of the popula-on: $u = N/N_0$ and set $r = b - d$. The ODE problem now becomes

$$u' = ru + f, \quad u(0) = 1, \tag{113}$$

here $f = I/N_0$ measures the net immigration per time unit as the fraction of ıe initial population. Very often, r is approximately constant, but f is usually function of time.

The growth rate r of a population decreases if the environment has limited esources. Suppose the environment can sustain at most $N_{\max}$ individuals. We ıay then assume that the growth rate approaches zero as N approaches $N_{\max}$, e., as u approaches $M = N_{\max}/N_0$. The simplest possible evolution of r is ıen a linear function: $r(t) = r_0(1 - u(t)/M)$, where r_0 is the initial growth rate hen the population is small relative to the maximum size and there is enough esources. Using this $r(t)$ in (113) results in the *logistic model* for the evolution f a population (assuming for the moment that $f = 0$):

$$u' = r_0(1 - u/M)u, \quad u(0) = 1\,. \tag{114}$$

ıitially, u will grow at rate r_0, but the growth will decay as u approaches M, nd then there is no more change in u, causing $u \to M$ as $t \to \infty$. Note that ıe logistic equation $u' = r_0(1 - u/M)u$ is *nonlinear* because of the quadratic erm $-u^2 r_0/M$.

1.3 Compound interest and inflation

ay the annual interest rate is r percent and that the bank adds the interest nce a year to your investment. If u^n is the investment in year n, the investment ı year u^{n+1} grows to

$$u^{n+1} = u^n + \frac{r}{100}u^n\,.$$

ı reality, the interest rate is added every day. We therefore introduce a parameter ı for the number of periods per year when the interest is added. If n counts ıe periods, we have the fundamental model for compound interest:

$$u^{n+1} = u^n + \frac{r}{100m}u^n\,. \tag{115}$$

his model is a *difference equation*, but it can be transformed to a continuous ifferential equation through a limit process. The first step is to derive a formula

for the growth of the investment over a time t. Starting with an investı and assuming that r is constant in time, we get

$$\begin{aligned} u^{n+1} &= \left(1 + \frac{r}{100m}\right)u^n \\ &= \left(1 + \frac{r}{100m}\right)^2 u^{n-1} \\ &\vdots \\ &= \left(1 + \frac{r}{100m}\right)^{n+1} u^0 \end{aligned}$$

Introducing time t, which here is a real-numbered counter for years, ı that $n = mt$, so we can write

$$u^{mt} = \left(1 + \frac{r}{100m}\right)^{mt} u^0\,.$$

The second step is to assume *continuous compounding*, meaning that the is added continuously. This implies $m \to \infty$, and in the limit one g formula

$$u(t) = u_0 e^{rt/100},$$

which is nothing but the solution of the ODE problem

$$u' = \frac{r}{100}u, \quad u(0) = u_0\,.$$

This is then taken as the ODE model for compound interest if $r > 0$. I the reasoning applies equally well to inflation, which is just the case One may also take the r in (117) as the net growth of an investemt, takes both compound interest and inflation into account. Note that applications we must use a time-dependent r in (117).

Introducing $a = \frac{r}{100}$, continuous inflation of an initial fortune I is process exhibiting exponential decay according to

$$u' = -au, \quad u(0) = I\,.$$

11.4 Radioactive Decay

An atomic nucleus of an unstable atom may lose energy by emitting particles and thereby be transformed to a nucleus with a different nu protons and neutrons. This process is known as radioactive decay[86]. A the process is stochastic when viewed for a single atom, because it is im to predict exactly when a particular atom emits a particle. Neverthele a large number of atoms, N, one may view the process as determinis compute the mean behavior of the decay. Below we reason intuitivel an ODE for the mean behavior. Thereafter, we show mathematically detailed stochastic model for single atoms leads the same mean behavic

[86] `http://en.wikipedia.org/wiki/Radioactive_decay`

Deterministic model. Suppose at time t, the number of the original atom type is $N(t)$. A basic model assumption is that the transformation of the atoms of the original type in a small time interval Δt is proportional to N, so that

$$N(t+\Delta t) = N(t) - a\Delta t N(t),$$

where $a > 0$ is a constant. Introducing $u = N(t)/N(0)$, dividing by Δt and letting $\Delta t \to 0$ gives the following ODE:

$$u' = -au, \quad u(0) = 1\,. \tag{118}$$

The parameter a can for a given nucleus be expressed through the *half-life* $t_{1/2}$, which is the time taken for the decay to reduce the initial amount by one half, i.e., $u(t_{1/2}) = 0.5$. With $u(t) = e^{-at}$, we get $t_{1/2} = a^{-1}\ln 2$ or $a = \ln 2/t_{1/2}$.

Stochastic model. We have originally N_0 atoms. Each atom may have decayed or survived at a particular time t. We want to count how many original atoms that are left, i.e., how many atoms that have survived. The survival of a single atom at time t is a random event. Since there are only two outcomes, survival or decay, we have a Bernoulli trial[87]. Let p be the probability of survival (implying that the probability of decay is $1-p$). If each atom survives independently of the others, and the probability of survival is the same for every atom, we have N_0 statistically Bernoulli trials, known as a *binomial experiment* from probability theory. The probability $P(N)$ that N out of the N_0 atoms have survived at time t is then given by the famous *binomial distribution*

$$P(N) = \frac{N_0!}{N!(N_0-N)!}p^N(1-p)^{N_0-N}\,.$$

The mean (or expected) value $\mathrm{E}[P]$ of $P(N)$ is known to be $N_0 p$.

It remains to estimate p. Let the interval $[0,t]$ be divided into m small subintervals of length Δt. We make the assumption that the probability of decay of a single atom in an interval of length Δt is $\tilde{p}$, and that this probability is proportional to Δt: $\tilde{p} = \lambda\Delta t$ (it sounds natural that the probability of decay increases with Δt). The corresponding probability of survival is $1-\lambda\Delta t$. Believing that λ is independent of time, we have, for each interval of length Δt, a Bernoulli trial: the atom either survives or decays in that interval. Now, p should be the probability that the atom survives in all the intervals, i.e., that we have m successful Bernoulli trials in a row and therefore

$$p = (1-\lambda\Delta t)^m\,.$$

The expected number of atoms of the original type at time t is

$$\mathrm{E}[P] = N_0 p = N_0(1-\lambda\Delta t)^m, \quad m = t/\Delta t\,. \tag{119}$$

To see the relation between the two types of Bernoulli trials and the ODE above, we go to the limit $\Delta t \to t$, $m \to \infty$. One can show that

[87] http://en.wikipedia.org/wiki/Bernoulli_trial

$$p = \lim_{m\to\infty}(1-\lambda\Delta t)^m = \lim_{m\to\infty}\left(1-\lambda\frac{t}{m}\right)^m = e^{-\lambda t}$$

This is the famous exponential waiting time (or arrival time) distribution of a Poisson process in probability theory (obtained here, as often done, as the limit of a binomial experiment). The probability of decay, $1-e^{-\lambda t}$, follows the exponential distribution[88]. The limit means that m is very large, hence Δt is very small, and $\tilde{p} = \lambda\Delta t$ is very small since the intensity of the events is assumed finite. This situation corresponds to a very small probability that an atom will decay in a very short time interval, which is a reasonable model. The same model occurs in lots of different applications, e.g., when waiting for phone calls or when finding defects along a rope.

Relation between stochastic and deterministic models. With p from above we get the expected number of original atoms at t as $N_0 p = N_0 e^{-\lambda t}$, which is exactly the solution of the ODE model $N' = -\lambda N$. This gives a physical interpretation of a via λ or vice versa. Our important finding here is that the ODE model captures the mean behavior of the underlying stochastic experiment. This is, however, not always the common relation between microscopic stochastic models and macroscopic "averaged" models.

Also of interest is to see that a Forward Euler discretization of $N' = -\lambda N$, $N(0) = N_0$, gives $N^m = N_0(1-\lambda\Delta t)^m$ at time $t_m = m\Delta t$, which is nothing but the expected value of the stochastic experiment with N_0 atoms and m time intervals of length Δt, where each atom can decay with probability $\lambda\Delta t$ in each interval.

A fundamental question is how accurate the ODE model is. The underlying stochastic model fluctuates around its expected value. A measure of these fluctuations is the standard deviation of the binomial experiment with N_0 trials, which can be shown to be $\mathrm{Std}[P] = \sqrt{N_0 p(1-p)}$. Compared to the size of the expectation, we get the normalized standard deviation

$$\frac{\sqrt{\mathrm{Var}[P]}}{\mathrm{E}[P]} = N_0^{-1/2}\sqrt{p^{-1}-1} = N_0^{-1/2}\sqrt{(1-e^{-\lambda t})^{-1}-1} \approx (N_0\lambda t)^{-}$$

showing that the normalized fluctuations are very small if N_0 is very large, which is usually the case.

11.5 Newton's law of cooling

When a body at some temperature is placed in a cooling environment, experience shows that the temperature falls rapidly in the beginning, and then the changes in temperature levels off until the body's temperature equals that of the surroundings. Newton carried out some experiments on cooling bodies and found that the temperature evolved as a "geometric progression at times in arithmetic progression", meaning that the temperature decayed exponentially.

[88] http://en.wikipedia.org/wiki/Exponential_distribution

ater, this result was formulated as a differential equation: the rate of change of
ıe temperature in a body is proportional to the temperature difference between
ıe body and its surroundings. This statement is known as *Newton's law of
ɔoling*, which can be mathematically expressed as

$$\frac{dT}{dt} = -k(T - T_s), \tag{120}$$

here T is the temperature of the body, T_s is the temperature of the surroundings,
is time, and k is a positive constant. Equation (120) is primarily viewed as
ı empirical law, valid when heat is efficiently convected away from the surface
f the body by a flowing fluid such as air at constant temperature T_s. The
eat transfer coefficient k reflects the transfer of heat from the body to the
ırroundings and must be determined from physical experiments.

We must obviously have an initial condition $T(0) = T_0$ in addition to the
ɔoling law (120).

1.6 Decay of atmospheric pressure with altitude

ertical equilibrium of air in the atmosphere is governed by the equation

$$\frac{dp}{dz} = -\varrho g\,. \tag{121}$$

ere, $p(z)$ is the air pressure, ϱ is the density of air, and $g = 9.807$ m/s^2 is a
:andard value of the acceleration of gravity. (Equation (121) follows directly
om the general Navier-Stokes equations for fluid motion, with the assumption
ıat the air does not move.)

The pressure is related to density and temperature through the ideal gas law

$$\varrho = \frac{Mp}{R^*T}, \tag{122}$$

here M is the molar mass of the Earth's air (0.029 kg/mol), R^* is the universal
as constant (8.314 Nm/(mol K)), and T is the temperature. All variables p, ϱ,
ıd T vary with the height z. Inserting (122) in (121) results in an ODE with a
ariable coefficient:

$$\frac{dp}{dz} = -\frac{Mg}{R^*T(z)}p\,. \tag{123}$$

Multiple atmospheric layers. The atmosphere can be approximately mod-
ed by seven layers. In each layer, (123) is applied with a linear temperature of
ıe form

$$T(z) = \bar{T}_i + L_i(z - h_i),$$

here $z = h_i$ denotes the bottom of layer number i, having temperature $\bar{T}_i$, and
$_i$ is a constant in layer number i. The table below lists h_i (m), $\bar{T}_i$ (K), and L_i
K/m) for the layers $i = 0, \ldots, 6$.

i	h_i	$\bar{T}_i$	L_i
0	0	288	-0.0065
1	11,000	216	0.0
2	20,000	216	0.001
3	32,000	228	0.0028
4	47,000	270	0.0
5	51,000	270	-0.0028
6	71,000	214	-0.002

For implementation it might be convenient to write (123) on the form

$$\frac{dp}{dz} = -\frac{Mg}{R^*(\bar{T}(z) + L(z)(z - h(z)))}p,$$

where $\bar{T}(z)$, $L(z)$, and $h(z)$ are piecewise constant functions with values
the table. The value of the pressure at the sea level $z = 0$, $p_0 = p(0)$, is
Pa.

Simplification: $L = 0$. One commonly used simplification is to assu
the temperature is constant within each layer. This means that $L = 0$.

Simplification: one-layer model. Another commonly used approx
is to work with one layer instead of seven. This one-layer model[89] is b
$T(z) = T_0 - Lz$, with sea level standard temperature $T_0 = 288$ K and temp
lapse rate $L = 0.0065$ K/m.

11.7 Compaction of sediments

Sediments, originally made from materials like sand and mud, get con
through geological time by the weight of new material that is deposited
sea bottom. The porosity ϕ of the sediments tells how much void (fluic
there is between the sand and mud grains. The porosity reduces wit
because the weight of the sediments above and causes the void space t
and thereby increase the compaction.

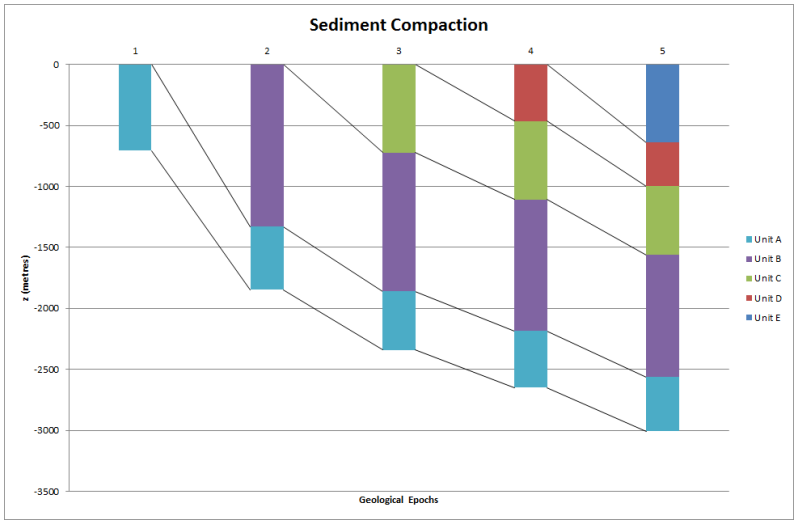
A typical assumption is that the change in ϕ at some depth z is ne
proportional to ϕ. This assumption leads to the differential equation pı

$$\frac{d\phi}{dz} = -c\phi, \quad \phi(0) = \phi_0,$$

where the z axis points downwards, $z = 0$ is the surface with known ı
and $c > 0$ is a constant.

The upper part of the Earth's crust consists of many geological layers
on top of each other, as indicated in Figure 23. The model (125) can be
for each layer. In layer number i, we have the unknown porosity functi
fulfilling $\phi_i'(z) = -c_i z$, since the constant c in the model (125) dep
the type of sediment in the layer. From the figure we see that new l
sediments are deposited on top of older ones as time progresses. The com
as measured by ϕ, is rapid in the beginning and then decreases (expon
with depth in each layer.

[89] `http://en.wikipedia.org/wiki/Density_of_air`



igure 23: Illustration of the compaction of geological layers (with different olors) through time.

When we drill a well at present time through the right-most column of ediments in Figure 23, we can measure the thickness of the sediment in (say) the ottom layer. Let L_1 be this thickness. Assuming that the volume of sediment emains constant through time, we have that the initial volume, $\int_0^{L_{1,0}} \phi_1 dz$, must qual the volume seen today, $\int_{\ell-L_1}^{\ell} \phi_1 dz$, where ℓ is the depth of the bottom of ne sediment in the present day configuration. After having solved for ϕ_1 as a unction of z, we can then find the original thickness $L_{1,0}$ of the sediment from ne equation

$$\int_0^{L_{1,0}} \phi_1 dz = \int_{\ell-L_1}^{\ell} \phi_1 dz\,.$$

n hydrocarbon exploration it is important to know $L_{1,0}$ and the compaction istory of the various layers of sediments.

1.8 Vertical motion of a body in a viscous fluid

body moving vertically through a fluid (liquid or gas) is subject to three ifferent types of forces: the gravity force, the drag force[90], and the buoyancy orce.

Overview of forces. The gravity force is $F_g = -mg$, where m is the mass f the body and g is the acceleration of gravity. The uplift or buoyancy force Archimedes force") is $F_b = \varrho g V$, where ϱ is the density of the fluid and V is

[90] http://en.wikipedia.org/wiki/Drag_(physics)

the volume of the body. Forces and other quantities are taken as positiv upward direction.

The drag force is of two types, depending on the Reynolds number

$$\mathrm{Re} = \frac{\varrho d|v|}{\mu},$$

where d is the diameter of the body in the direction perpendicular to the is the velocity of the body, and μ is the dynamic viscosity of the fluid $\mathrm{Re} < 1$, the drag force is fairly well modeled by the so-called Stokes' drag for a spherical body of diameter d reads

$$F_d^{(S)} = -3\pi d\mu v\,.$$

For large Re, typically $\mathrm{Re} > 10^3$, the drag force is quadratic in the velo

$$F_d^{(q)} = -\frac{1}{2} C_D \varrho A|v|v,$$

where C_D is a dimensionless drag coefficient depending on the body's and A is the cross-sectional area as produced by a cut plane, perpendi the motion, through the thickest part of the body. The superscripts ^{q} a $F_d^{(S)}$ and $F_d^{(q)}$ indicate Stokes drag and quadratic drag, respectively.

Equation of motion. All the mentioned forces act in the vertical d Newton's second law of motion applied to the body says that the sum forces must equal the mass of the body times its acceleration a in the direction.

$$ma = F_g + F_d^{(S)} + F_b\,.$$

Here we have chosen to model the fluid resistance by the Stokes drag. I the expressions for the forces yields

$$ma = -mg - 3\pi d\mu v + \varrho g V\,.$$

The unknowns here are v and a, i.e., we have two unknowns but o equation. From kinematics in physics we know that the acceleration is t derivative of the velocity: $a = dv/dt$. This is our second equation. We ca eliminate a and get a single differential equation for v:

$$m\frac{dv}{dt} = -mg - 3\pi d\mu v + \varrho g V\,.$$

A small rewrite of this equation is handy: We express m as $\varrho_b V$, where density of the body, and we divide by the mass to get

$$v'(t) = -\frac{3\pi d\mu}{\varrho_b V} v + g\left(\frac{\varrho}{\varrho_b} - 1\right)\,.$$

/e may introduce the constants

$$a = \frac{3\pi d\mu}{\varrho_b V}, \quad b = g\left(\frac{\varrho}{\varrho_b} - 1\right), \tag{130}$$

) that the structure of the differential equation becomes obvious:

$$v'(t) = -av(t) + b\,. \tag{131}$$

he corresponding initial condition is $v(0) = v_0$ for some prescribed starting elocity v_0.

This derivation can be repeated with the quadratic drag force $F_d^{(q)}$, leading) the result

$$v'(t) = -\frac{1}{2}C_D\frac{\varrho A}{\varrho_b V}|v|v + g\left(\frac{\varrho}{\varrho_b} - 1\right)\,. \tag{132}$$

efining

$$a = \frac{1}{2}C_D\frac{\varrho A}{\varrho_b V}, \tag{133}$$

nd b as above, we can write (132) as

$$v'(t) = -a|v|v + b\,. \tag{134}$$

erminal velocity. An interesting aspect of (131) and (134) is whether v will pproach a final constant value, the so-called *terminal velocity* v_T, as $t \to \infty$. A onstant v means that $v'(t) \to 0$ as $t \to \infty$ and therefore the terminal velocity $_T$ solves

$$0 = -av_T + b$$

nd

$$0 = -a|v_T|v_T + b\,.$$

he former equation implies $v_T = b/a$, while the latter has solutions $v_T = \sqrt{|b|/a}$ for a falling body ($v_T < 0$) and $v_T = \sqrt{b/a}$ for a rising body ($v_T > 0$).

. Crank-Nicolson scheme. Both governing equations, the Stokes' drag odel (131) and the quadratic drag model (134), can be readily solved by the orward Euler scheme. For higher accuracy one can use the Crank-Nicolson ethod, but a straightforward application this method results a nonlinear uation in the new unknown value v^{n+1} when applied to (134):

$$\frac{v^{n+1} - v^n}{\Delta t} = -a\frac{1}{2}(|v^{n+1}|v^{n+1} + |v^n|v^n) + b\,. \tag{135}$$

owever, instead of approximating the term $-|v|v$ by an arithmetic average, we an use a *geometric mean*:

$$(|v|v)^{n+\frac{1}{2}} \approx |v^n|v^{n+1}\,.$$

The error is of second order in Δt, just as for the arithmetic average centered finite difference approximation in (135). With this approximati the discrete equation

$$\frac{v^{n+1} - v^n}{\Delta t} = -a|v^n|v^{n+1} + b$$

becomes a *linear* equation in v^{n+1}, and we can therefore easily solve fo

$$v^{n+1} = \frac{v_n + \Delta t b^{n+\frac{1}{2}}}{1 + \Delta t a^{n+\frac{1}{2}}|v^n|}\,.$$

Physical data. Suitable values of μ are $1.8\cdot10^{-5}$ Pa s for air and $8.9\cdot1($ for water. Densities can be taken as 1.2 kg/m^3 for air and as $1.0\cdot10^3$ kg water. For considerable vertical displacement in the atmosphere one sho into account that the density of air varies with the altitude, see Secti One possible density variation arises from the one-layer model in the me section.

Any density variation makes b time dependent and we need $b^{n+\frac{1}{2}}$ i To compute the density that enters $b^{n+\frac{1}{2}}$ we must also compute the position $z(t)$ of the body. Since $v = dz/dt$, we can use a centered di approximation:

$$\frac{z^{n+\frac{1}{2}} - z^{n-\frac{1}{2}}}{\Delta t} = v^n \quad \Rightarrow \quad z^{n+\frac{1}{2}} = z^{n-\frac{1}{2}} + \Delta t\, v^n\,.$$

This $z^{n+\frac{1}{2}}$ is used in the expression for b to compute $\varrho(z^{n+\frac{1}{2}})$ and then

The drag coefficient[91] C_D depends heavily on the shape of the bod values are: 0.45 for a sphere, 0.42 for a semi-sphere, 1.05 for a cube, 0. long cylinder (when the center axis is in the vertical direction), 0.75 for a 1.0-1.3 for a man in upright position, 1.3 for a flat plate perpendicula flow, and 0.04 for a streamlined, droplet-like body.

Verification. To verify the program, one may assume a heavy bod such that the F_b force can be neglected, and further assume a small such that the air resistance F_d can also be neglected. This can be obta setting μ and ϱ to zero. The motion then leads to the velocity $v(t) =$ which is linear in t and therefore should be reproduced to machine p (say tolerance 10^{-15}) by any implementation based on the Crank-Nic Forward Euler schemes.

Another verification, but not as powerful as the one above, can be b computing the terminal velocity and comparing with the exact expressio advantage of this verification is that we can also the test situation $\varrho \neq$

[91] `http://en.wikipedia.org/wiki/Drag_coefficient`

As always, the method of manufactured solutions can be applied to test the nplementation of all terms in the governing equation, but the solution then has o physical relevance in general.

caling. Applying scaling, as described in Section 11.1, will for the linear case educe the need to estimate values for seven parameters down to choosing one alue of a single dimensionless parameter

$$\beta = \frac{\varrho_b g V \left(\frac{\varrho}{\varrho_b} - 1\right)}{3\pi d\mu I},$$

rovided $I \neq 0$. If the motion starts from rest, $I = 0$, the scaled problem $' = 1 - \bar{u}$, $\bar{u}(0) = 0$, has no need for estimating physical parameters. This means nat there is a single universal solution to the problem of a falling body starting om rest: $\bar{u}(t) = 1 - e^{-\bar{t}}$. All real physical cases correspond to stretching the $\bar{t}$ xis and the $\bar{u}$ axis in this dimensionless solution. More precisely, the physical elocity $u(t)$ is related to the dimensionless velocity $\bar{u}(\bar{t})$ through

$$u = \frac{\varrho_b g V \left(\frac{\varrho}{\varrho_b} - 1\right)}{3\pi d\mu} \bar{u}(t/(g(\varrho/\varrho_b - 1))) .$$

1.9 Decay ODEs from solving a PDE by Fourier expansions

uppose we have a partial differential equation

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2} + f(x,t),$$

ith boundary conditions $u(0,t) = u(L,t) = 0$ and initial condition $u(x,0) = (x)$. One may express the solution as

$$u(x,t) = \sum_{k=1}^{m} A_k(t) e^{ikx\pi/L},$$

or appropriate unknown functions A_k, $k = 1, \ldots, m$. We use the complex xponential $e^{ikx\pi/L}$ for easy algebra, but the physical u is taken as the real art of any complex expression. Note that the expansion in terms of $e^{ikx\pi/L}$ is ompatible with the boundary conditions: all functions $e^{ikx\pi/L}$ vanish for $x = 0$ nd $x = L$. Suppose we can express $I(x)$ as

$$I(x) = \sum_{k=1}^{m} I_k e^{ikx\pi/L} .$$

uch an expansion can be computed by well-known Fourier expansion techniques, ut the details are not important here. Also, suppose we can express the given (x,t) as

$$f(x,t) = \sum_{k=1}^{m} b_k(t) e^{ikx\pi/L} .$$

Inserting the expansions for u and f in the differential equations dema all terms corresponding to a given k must be equal. The calculations r the follow system of ODEs:

$$A_k'(t) = -\alpha \frac{k^2\pi^2}{L^2} + b_k(t), \quad k = 1, \ldots, m .$$

From the initial condition

$$u(x,0) = \sum_k A_k(0) e^{ikx\pi/L} = I(x) = \sum_k I_k e^{(ikx\pi/L)},$$

it follows that $A_k(0) = I_k$, $k = 1, \ldots, m$. We then have m equations of t $A_k' = -aA_k + b$, $A_k(0) = I_k$, for appropriate definitions of a and b. The problems independent each other such that we can solve one problem at The outline technique is a quite common approach for solving partial dif equations.

Remark. Since a_k depends on k and the stability of the Forward Euler demands $a_k \Delta t \leq 1$, we get that $\Delta t \leq \alpha^{-1} L^2 \pi^{-2} k^{-2}$. Usually, quite values are needed to accurately represent the given functions I and f and needs to be very small for these large values of k. Therefore, the Crank-N and Backward Euler schemes, which allow larger Δt without any growt solutions, are more popular choices when creating time-stepping algorit partial differential equations of the type considered in this example.

12 Exercises and Projects

Exercise 24: Simulate an oscillating cooling process

The surrounding temperature T_s in Newton's law of cooling (120) may time. Assume that the variations are periodic with period P and amp around a constant mean temperature T_m:

$$T_s(t) = T_m + a \sin\left(\frac{2\pi}{P} t\right) .$$

Simulate a process with the following data: $k = 20$ min^{-1}, $T(0) = 5$ C, C, $a = 2.5$ C, and $P = 1$ h. Also experiment with $P = 10$ min and I Plot T and T_s in the same plot. Filename: `osc_cooling.py`.

Exercise 25: Radioactive decay of Carbon-14

The Carbon-14[92] isotope, whose radioactive decay is used extensively i organic material that is tens of thousands of years old, has a half-life years. Determine the age of an organic material that contains 8.4 perce initial amount of Carbon-14. Use a time unit of 1 year in the computatio uncertainty in the half time of Carbon-14 is ±40 years. What is the corres uncertainty in the estimate of the age?

[92] `http://en.wikipedia.org/wiki/Carbon-14`

Hint. Use simulations with $5,730 \pm 40$ y as input and find the corresponding interval for the result.

Filename: `carbon14.py`.

Exercise 26: Simulate stochastic radioactive decay

The purpose of this exercise is to implement the stochastic model described in Section 11.4 and show that its mean behavior approximates the solution of the corresponding ODE model.

The simulation goes on for a time interval $[0, T]$ divided into N_t intervals of length Δt. We start with N_0 atoms. In some time interval, we have N atoms that have survived. Simulate N Bernoulli trials with probability $\lambda\Delta t$ in this interval by drawing N random numbers, each being 0 (survival) or 1 (decay), where the probability of getting 1 is $\lambda\Delta t$. We are interested in the number of decays, d, and the number of survived atoms in the next interval is then $N - d$. The Bernoulli trials are simulated by drawing N uniformly distributed real numbers on $[0, 1]$ and saying that 1 corresponds to a value less than $\lambda\Delta t$:

```
# Given lambda_, dt, N
import numpy as np
uniform = np.random.uniform(N)
Bernoulli_trials = np.asarray(uniform < lambda_*dt, dtype=np.int)
d = Bernoulli_trials.size
```

Observe that `uniform < lambda_*dt` is a boolean array whose true and false values become 1 and 0, respectively, when converted to an integer array.

Repeat the simulation over $[0, T]$ a large number of times, compute the average value of N in each interval, and compare with the solution of the corresponding ODE model. Filename: `stochastic_decay.py`.

Exercise 27: Radioactive decay of two substances

Consider two radioactive substances A and B. The nuclei in substance A decay to form nuclei of type B with a half-life $A_{1/2}$, while substance B decay to form type A nuclei with a half-life $B_{1/2}$. Letting u_A and u_B be the fractions of the initial amount of material in substance A and B, respectively, the following system of ODEs governs the evolution of $u_A(t)$ and $u_B(t)$:

$$\frac{1}{\ln 2}u'_A = u_B/B_{1/2} - u_A/A_{1/2}, \tag{139}$$

$$\frac{1}{\ln 2}u'_B = u_A/A_{1/2} - u_B/B_{1/2}, \tag{140}$$

with $u_A(0) = u_B(0) = 1$.

Make a simulation program that solves for $u_A(t)$ and $u_B(t)$. Verify the implementation by computing analytically the limiting values of u_A and u_B as $t \to \infty$ (assume $u'_A, u'_B \to 0$) and comparing these with those obtained numerically.

Run the program for the case of $A_{1/2} = 10$ minutes and $B_{1/2} = 50$ minutes. Use a time unit of 1 minute. Plot u_A and u_B versus time in the same plot. Filename: `radioactive_decay_2subst.py`.

Exercise 28: Simulate the pressure drop in the atmos

We consider the models for atmospheric pressure in Section 11.6. Make a p with three functions,

- one computing the pressure $p(z)$ using a seven-layer model and va
- one computing $p(z)$ using a seven-layer model, but with constant t ture in each layer, and
- one computing $p(z)$ based on the one-layer model.

How can these implementations be verified? Should ease of verification how you code the functions? Compare the three models in a plot. Fi `atmospheric_pressure.py`.

Exercise 29: Make a program for vertical motion in a

Implement the Stokes' drag model (129) and the quadratic drag mod from Section 11.8, using the Crank-Nicolson scheme and a geometric n $|v|v$ as explained, and assume constant fluid density. At each time level, c the Reynolds number Re and choose the Stokes' drag model if $\mathrm{Re} < 1$ quadratic drag model otherwise.

The computation of the numerical solution should take place either in alone function (as in Section 2.1) or in a solver class that looks up a p class for physical data (as in Section 3.6). Create a module (see Section 3 equip it with nose tests (see Section 3.4) for automatically verifying the

Verification tests can be based on

- the terminal velocity (see Section 11.8),
- the exact solution when the drag force is neglected (see Section 1
- the method of manufactured solutions (see Section 8.5) combin computing convergence rates (see Section 2.8).

Use, e.g., a quadratic polynomial for the velocity in the method of manu solutions. The expected error is $\mathcal{O}(\Delta t^2)$ from the centered finite di approximation and the geometric mean approximation for $|v|v$.

A solution that is linear in t will also be an exact solution of the equations in many problems. Show that this is true for linear drag (by a source term that depends on t), but not for quadratic drag becaus geometric mean approximation. Use the method of manufactured solu add a source term *in the discrete equations for quadratic drag* such that function of t is a solution. Add a nose test for checking that the linear f is reproduced to machine precision in the case of both linear and quadra

Apply the software to a case where a ball rises in water. The buoyan is here the driving force, but the drag will be significant and balance th forces after a short time. A soccer ball has radius 11 cm and mass 0.43 k the motion from rest, set the density of water, ϱ, to 1000 kg/m^3, set the c viscosity, μ, to 10^{-3} Pa s, and use a drag coefficient for a sphere: 0.45. velocity of the rising ball. Filename: `vertical_motion.py`.

roject 30: Simulate parachuting

he aim of this project is to develop a general solver for the vertical motion of a ody with quadratic air drag, verify the solver, apply the solver to a skydiver in ee fall, and finally apply the solver to a complete parachute jump.

All the pieces of software implemented in this project should be realized as ython functions and/or classes and collected in one module.

) Set up the differential equation problem that governs the velocity of the notion. The parachute jumper is subject to the gravity force and a quadratic rag force. Assume constant density. Add an extra source term be used for rogram verification. Identify the input data to the problem.

) Make a Python module for computing the velocity of the motion. Also quip the module with functionality for plotting the velocity.

lint 1. Use the Crank-Nicolson scheme with a geometric mean of $|v|v$ in time o linearize the equation of motion with quadratic drag.

lint 2. You can either use functions or classes for implementation. If you noose functions, make a function `solver` that takes all the input data in the roblem as arguments and that returns the velocity (as a mesh function) and ne time mesh. In case of a class-based implementation, introduce a problem lass with the physical data and a solver class with the numerical data and a `olve` method that stores the velocity and the mesh in the class.

Allow for a time-dependent area and drag coefficient in the formula for the rag force.

) Show that a linear function of t does not fulfill the discrete equations because f the geometric mean approximation used for the quadratic drag term. Fit source term, as in the method of manufactured solutions, such that a linear nction of t is a solution of the discrete equations. Make a nose test to check nat this solution is reproduced to machine precision.

) The expected error in this problem goes like Δt^2 because we use a cen- ered finite difference approximation with error $\mathcal{O}(\Delta t^2)$ and a geometric mean pproximation with error $\mathcal{O}(\Delta t^2)$. Use the method of manufactured solutions ombined with computing convergence rate to verify the code. Make a nose test or checking that the convergence rate is correct.

) Compute the drag force, the gravity force, and the buoyancy force as a nction of time. Create a plot with these three forces.

lint. You can either make a function `forces(v, t, plot=None)` that returns ne forces (as mesh functions) and `t` and shows a plot on the screen and also aves the plot to a file with name `plot` if `plot` is not `None`, or you can extend ne solver class with computation of forces and include plotting of forces in the isualization class.

) Compute the velocity of a skydiver in free fall before the parachute opens.

Hint. Meade and Struthers [?] provide some data relevant to skydiving mass of the human body and equipment can be set to 100 kg. A sky spread-eagle formation has a cross-section of 0.5 m^2 in the horizontal pla density of air decreases varies altitude, but can be taken as constant, 1 for altitudes relevant to skydiving (0-4000 m). The drag coefficient for a upright position can be set to 1.2. Start with a zero velocity. A free fall t has a terminating velocity of 45 m/s. (This value can be used to tur parameters.)

g) The next task is to simulate a parachute jumper during free fall a the parachute opens. At time t_p, the parachute opens and the drag co and the cross-sectional area change dramatically. Use the program to sir jump from $z = 3000$ m to the ground $z = 0$. What is the maximum acce measured in units of g, experienced by the jumper?

Hint. Following Meade and Struthers [?], one can set the cross-secti perpendicular to the motion to 44 m^2 when the parachute is open. that it takes 8 s to increase the area linearly from the original to the fin The drag coefficient for an open parachute can be taken as 1.8, but tune the known value of the typical terminating velocity reached before land m/s. One can take the drag coefficient as a piecewise constant functi an abrupt change at t_p. The parachute is typically released after $t_p = 6$ larger values of t_p can be used to make plots more illustrative.

Filename: `skydiving.py`.

Exercise 31: Formulate vertical motion in the atmos

Vertical motion of a body in the atmosphere needs to take into account a air density if the range of altitudes is many kilometers. In this case, ϱ var the altitude z. The equation of motion for the body is given in Section 1 us assume quadratic drag force (otherwise the body has to be very, very A differential equation problem for the air density, based on the informa the one-layer atmospheric model in Section 11.6, can be set up as

$$p'(z) = -\frac{Mg}{R^*(T_0 + Lz)}p,$$
$$\varrho = p\frac{M}{R^*T}.$$

To evaluate $p(z)$ we need the altitude z. From the principle that the ve the derivative of the position we have that

$$z'(t) = v(t),$$

where v is the velocity of the body.

Explain in detail how the governing equations can be discretized by ward Euler and the Crank-Nicolson methods. Filename: `falling_in_var`

[93] `http://en.wikipedia.org/wiki/Parachuting`

Exercise 32: Simulate vertical motion in the atmosphere

Implement the Forward Euler or the Crank-Nicolson scheme derived in Exercise 31. Demonstrate the effect of air density variation on a falling human, e.g., the famous fall of Felix Baumgartner[94]. The drag coefficient can be set to 1.2.

Remark. In the Crank-Nicolson scheme one must solve a 3×3 system of equations at each time level, since p, ϱ, and v are coupled, while each equation can be stepped forward at a time with the Forward Euler scheme. Filename: `falling_in_variable_density.py`.

Exercise 33: Compute $y = |x|$ by solving an ODE

Consider the ODE problem

$$y'(x) = \begin{cases} -1, & x < 0, \\ 1, & x \geq 0 \end{cases} \qquad x \in (-1, 1], \quad y(1-) = 1,$$

which has the solution $y(x) = |x|$. Using a mesh $x_0 = -1$, $x_1 = 0$, and $x_2 = 1$, calculate by hand y_1 and y_2 from the Forward Euler, Backward Euler, Crank-Nicolson, and Leapfrog methods. Use all of the former three methods for computing the y_1 value to be used in the Leapfrog calculation of y_2. Thereafter, visualize how these schemes perform for a uniformly partitioned mesh with $N = 10$ and $N = 11$ points. Filename: `signum.py`.

Exercise 34: Simulate growth of a fortune with random interest rate

The goal of this exercise is to compute the value of a fortune subject to inflation and a random interest rate. Suppose that the inflation is constant at i percent per year and that the annual interest rate, p, changes randomly at each time step, starting at some value p_0 at $t = 0$. The random change is from a value p^n at $t = t_n$ to $p_n + \Delta p$ with probability 0.25 and $p_n - \Delta p$ with probability 0.25. No change occurs with probability 0.5. There is also no change if p^{n+1} exceeds 15 or becomes below 1. Use a time step of one month, $p_0 = i$, initial fortune scaled to 1, and simulate 1000 scenarios of length 20 years. Compute the mean evolution of one unit of money and the corresponding standard deviation. Plot the mean curve along with the mean plus one standard deviation and the mean minus one standard deviation. This will illustrate the uncertainty in the mean curve.

Hint 1. The following code snippet computes p^{n+1}:

```
import random

def new_interest_rate(p_n, dp=0.5):
    r = random.random()  # uniformly distr. random number in [0,1)
    if 0 <= r < 0.25:
        p_np1 = p_n + dp
    elif 0.25 <= r < 0.5:
        p_np1 = p_n - dp
    else:
        p_np1 = p_n
    return (p_np1 if 1 <= p_np1 <= 15 else p_n)
```

[94] http://en.wikipedia.org/wiki/Felix_Baumgartner

Hint 2. If $u_i(t)$ is the value of the fortune in experiment number $0, \ldots, N-1$, the mean evolution of the fortune is

$$\bar{u}(t) = \frac{1}{N} \sum_{i=0}^{N-1} u_i(t),$$

and the standard deviation is

$$s(t) = \sqrt{\frac{1}{N-1} \left(-(\bar{u}(t))^2 + \sum_{i=0}^{N-1} (u_i(t))^2 \right)}.$$

Suppose $u_i(t)$ is stored in an array `u`. The mean and the standard deviation of the fortune is most efficiently computed by using two accumulation variables `sum_u` and `sum_u2`, and performing `sum_u += u` and `sum_u2 += u**2` after each experiment. This technique avoids storing all the $u_i(t)$ time series for computing the statistics.

Filename: `random_interest.py`.

Exercise 35: Simulate a population in a changing environment

We shall study a population modeled by (113) where the environment, represented by r and f, undergoes changes with time.

a) Assume that there is a sudden drop (increase) in the birth (death) rate at time $t = t_r$, because of limited nutrition or food supply:

$$a(t) = \begin{cases} r_0, & t < t_r, \\ r_0 - A, & t \geq t_r, \end{cases}$$

This drop in population growth is compensated by a sudden net immigration at time $t_f > t_r$:

$$f(t) = \begin{cases} 0, & t < t_f, \\ f_0, & t \geq t_a, \end{cases}$$

Start with r_0 and make $A > r_0$. Experiment with these and other parameters to illustrate the interplay of growth and decay in such a problem. Filename: `population_drop.py`.

b) Now we assume that the environmental conditions changes periodically in time so that we may take

$$r(t) = r_0 + A \sin\left(\frac{2\pi}{P} t\right).$$

hat is, the combined birth and death rate oscillates around r_0 with a maximum ıange of $\pm A$ repeating over a period of length P in time. Set $f = 0$ and xperiment with the other parameters to illustrate typical features of the solution. ilename: `population_osc.py`.

Exercise 36: Simulate logistic growth

olve the logistic ODE (114) using a Crank-Nicolson scheme where $(u^{n+1/2})^2$ is pproximated by a *geometric mean*:

$$(u^{n+1/2})^2 \approx u^{n+1}u^n\,.$$

his trick makes the discrete equation linear in u^{n+1}. Filename: `logistic_CN.py`.

Exercise 37: Rederive the equation for continuous compound interest

he ODE model (117) was derived under the assumption that r was constant. erform an alternative derivation without this assumption: 1) start with (115);) introduce a time step Δt instead of m: $\Delta t = 1/m$ if t is measured in ears; 3) divide by Δt and take the limit $\Delta t \to 0$. Simulate a case where the ıflation is at a constant level I percent per year and the interest rate oscillates: $= -I/2 + r_0\sin(2\pi t)$. Compare solutions for $r_0 = I, 3I/2, 2I$. Filename: `nterest_modeling.py`.

References

Index