oduction to computing with finite differe methods

Hans Petter Langtangen[1,2]

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

Jul 14, 2014

nts

Exercises, Problems, and Projects

Finite difference methods for partial differential equations (PDEs) employ a r
nd tools that can be introduced and illustrated in the context of simple ordi
quation (ODE) examples. This is what we do in the present document. By fi
DEs, we keep the mathematical problems to be solved as simple as possible (
ereby allowing full focus on understanding the key concepts and tools. The
the forthcoming treatment of ODEs is therefore solely dominated by what
umerical methods for PDEs.

Theory and practice are primarily illustrated by solving the very simple
$(0) = I$, where $a > 0$ is a constant, but we also address the generalized problem u
nd the nonlinear problem $u' = f(u, t)$. The following topics are introduced:

- How to think when constructing finite difference methods, with special focu
  Euler, Backward Euler, and Crank-Nicolson (midpoint) schemes
- How to formulate a computational algorithm and translate it into Python
- 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
- How to structure code in terms of functions, classes, and modules
- How to work with Python concepts such as arrays, lists, dictionaries, la
  functions in functions (closures), doctests, unit tests, command-line inte
  user interfaces
- How to perform array computing and understand the difference from scal
- 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 wh
- How to derive mathematical expressions for various measures of the er
  methods, frequently by using the `sympy` software for symbolic computatio
- Introduce concepts such as finite difference operators, mesh (grid), mesh fu
  truncation error, consistency, and convergence
- Present additional methods for the general nonlinear ODE $u' = f(u, t)$,
  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 phenomena in
  and finance

xposition in a nutshell.

hing we cover is put into a practical, hands-on context. All mathematics is trans
rking computing codes, and all the mathematical theory of finite difference met
ed here is motivated from a strong need to understand strange behavior of prog
ndamental questions saturate the text:

low to we solve a differential equation problem and produce numbers?

low to we trust the answer?

nite difference methods

lain the basic ideas of finite difference methods using a simple ordinary differe
n $u' = -au$ as primary example. Emphasis is put on the reasoning when discret
blem and introduction of key concepts such as mesh, mesh function, finite diffe
imations, averaging in a mesh, derivation of algorithms, and discrete ope
n.

basic model for exponential decay

l problem is perhaps the simplest ordinary differential equation (ODE):

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

0 is a constant and $u'(t)$ means differentiation with respect to time t. This
arises in a number of widely different phenomena where some quantity u un
al reduction. Examples include radioactive decay, population decay, investmer
an object, pressure decay in the atmosphere, and retarded motion in fluids (for
lels, a can be negative as well), see Section 8 for details and motivation. We have
ular ODE not only because its applications are relevant, but even more because s
solution methods for this simple ODE gives important insight that can be re
re complicated settings, in particular when solving diffusion-type partial diff
.

nalytical solution of the ODE is found by the method of separation of variables

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

bitrary constant C. To formulate a mathematical problem for which there is a
ve need a condition to fix the value of C. This condition is known as the *initial c*
l as $u(0) = I$. That is, we know the value I of u when the process starts at $t =$
tion is then $u(t) = Ie^{-at}$.

We seek the solution $u(t)$ of the ODE for $t \in (0, T]$. The point $t = 0$ is not in
now u here and assume that the equation governs u for $t > 0$. The complete OI
eads: find $u(t)$ such that

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

his is known as a *continuous problem* because the parameter t varies continu
'. For each t we have a corresponding $u(t)$. There are hence infinitely many
(t). The purpose of a numerical method is to formulate a corresponding *discret*
olution is characterized by a finite number of values, which can be computed in
f steps on a computer.

.2 The Forward Euler scheme

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

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 by
f $N_t + 1$ points

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

he collection of points $t_0, t_1, \ldots, t_{N_t}$ constitutes a *mesh* or *grid*. Often the mes
niformly spaced in the domain $[0, T]$, which means that the spacing $t_{n+1} - t_n$ is
. This spacing is often denoted by Δt, in 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 th
nown as I. A notational short-form for $u(t_n)$, which will be used extensive
recisely, we let u^n be the *numerical approximation* to the exact solution $u(t_n$
umerical approximation is a *mesh function*, here defined only at the mesh points
o clearly distinguish between the numerical and the exact solution, we often pla
n the exact solution, as in $u_{\mathrm{e}}(t_n)$. Figure 1 shows the t_n and u_n points for $n =$
s well as $u_{\mathrm{e}}(t)$ as the dashed line. The goal of a numerical method for ODE
ne mesh function by solving a finite set of *algebraic equations* derived from th
roblem.

Since finite difference methods produce solutions at the mesh points only, it is a
hat the solution is between the mesh points. One can use methods for interpola
ne value of u between mesh points. The simplest (and most widely used) interp
to assume that u varies linearly between the mesh points, see Figure 2. Giv
ne value of u at some $t \in [t_n, t_{n+1}]$ is by linear interpolation

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

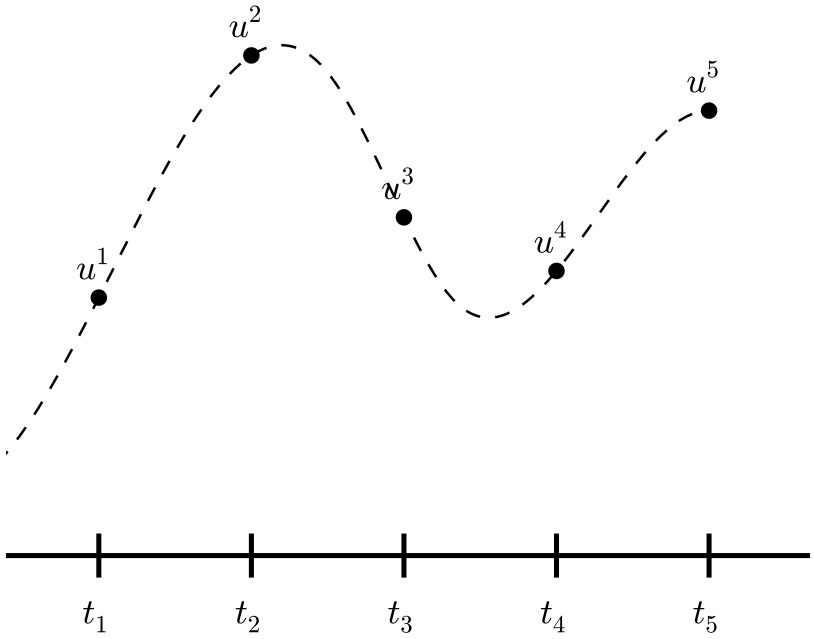


Figure 1: Time mesh with discrete solution values.

Fulfilling the equation at discrete time points. The ODE is supposed to
$T]$, i.e., at an infinite number of points. Now we relax that requirement and requ
is fulfilled at a finite set of discrete points in time. The mesh points $t_0, t_1, \ldots, t$
ɔut not the only) choice of points. The original ODE is then reduced to the fc
ons:

$$u'(t_n) = -au(t_n), \quad n = 0, \ldots, N_t\,.$$

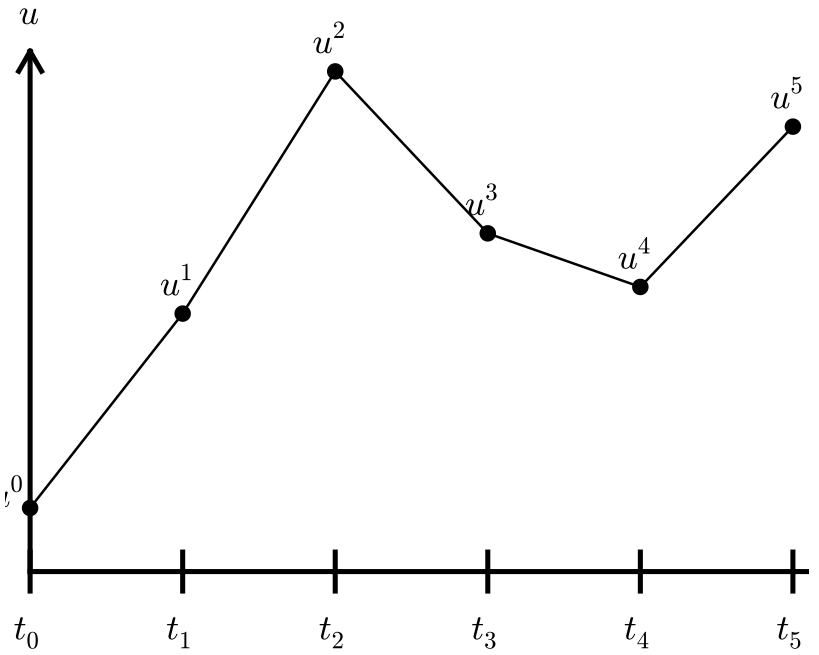
Replacing derivatives by finite differences. The next and most essentia
ɔd is to replace the derivative u' by a finite difference approximation. Let us fi
difference approximation (see Figure 3),

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

this approximation in (4) results in

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

ɾill be absolutely clear that if we want to compute the solution up to time leve
(4) to hold for $n = 0, \ldots, N_t - 1$ since (6) for $n = N_t - 1$ creates an equation
ɜ u^{N_t}.



igure 2: Linear interpolation between the discrete solution values (dashec
ɔlution).

Equation (6) is the discrete counterpart to the original ODE problem (1), ar
ɔ as *finite difference scheme* or more generally as the *discrete equations* of th
ndamental feature of these equations is that they are *algebraic* and can hence be s
ɔlved to produce the mesh function, i.e., the values of u at the mesh points $(u^n$,

tep 4: Formulating a recursive algorithm. The final step is to identify th
lgorithm to be implemented in a program. The key observation here is to realize
sed to compute u^{n+1} if u^n is known. Starting with $n = 0$, u^0 is known since u^0
ɔ) gives an equation for u^1. Knowing u^1, u^2 can be found from (6). In general,
ssumed known, and then we can easily solve for the unknown u^{n+1}:

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

Ve shall refer to (7) as the Forward Euler (FE) scheme for our model pr
ıathematical point of view, equations of the form (7) are known as *difference*
ıey express how differences in u, like $u^{n+1} - u^n$, evolve with n. The finite differ
e viewed as a method for turning a differential equation into a difference equa
Computation with (7) is straightforward:

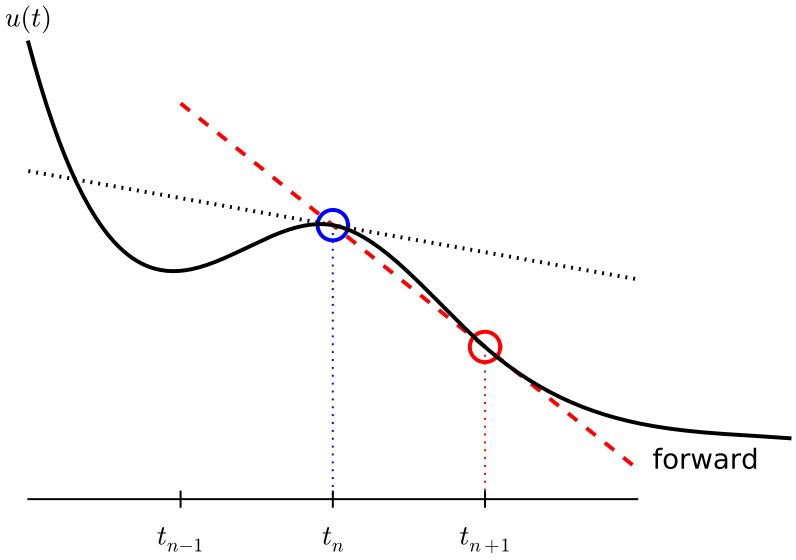


Figure 3: Illustration of a forward difference.

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

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

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

as that we have found a closed formula for u^n, and there is no need to let a cc he sequence $u^1, u^2, u^3, \ldots$. However, finding such a formula for u^n is possible or simple problems, so in general finite difference equations must be solved on a cc e next sections will show, the scheme (7) is just one out of many alternati (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 diffe resented in the previous section. Another alternative is

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

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

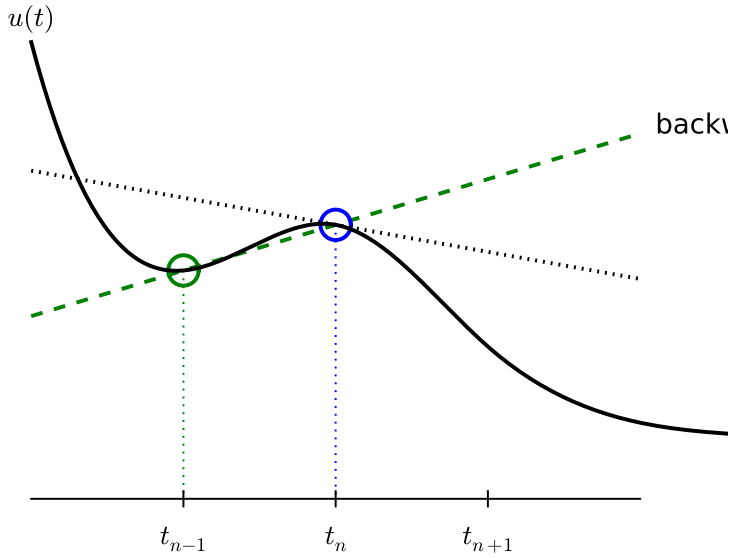


Figure 4: Illustration of a backward difference.

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

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

Ve assume, as explained under step 4 in Section 1.2, that we have computed ıch that (9) can be used to compute u^n. For direct similarity with the Forwa ') we replace n by $n+1$ in (9) and solve for the unknown value u^{n+1}:

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

.4 The Crank-Nicolson scheme

he finite difference approximations used to derive the schemes (7) and (10) ar ifferences, known to be less accurate than central (or midpoint) differences onstruct a central difference at $t_{n+1/2} = \frac{1}{2}(t_n + t_{n+1})$, or $t_{n+1/2} = (n + \frac{1}{2})\Delta t$ if t uniform in time. The approximation reads

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

the fraction on the right-hand side is the same as for the Forward Euler approx
ıe Backward Euler approximation (8) (with n replaced by $n+1$). The accurac
s an approximation to the derivative of u depends on *where* we seek the deriva
· of the interval $[t_n, t_{n+1}]$ or at the end points.
the formula (11), where u' is evaluated at $t_{n+1/2}$, it is natural to demand the
d at the time points *between* the mesh points:

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

) in (12) results in

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

$^{-\frac{1}{2}}$ is a short form for $u(t_{n+\frac{1}{2}})$. The problem is that we aim to compute u^n for i
that $u^{n+\frac{1}{2}}$ is not a quantity computed by our method. It must therefore be e
antities that we actually produce, i.e., the numerical solution at the mesh poi
· is to approximate $u^{n+\frac{1}{2}}$ as an arithmetic mean of the u values at the neighbori

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

) in (13) results in

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

sketches the geometric interpretation of such a centered difference.
sume that u^n is already computed so that u^{n+1} is the unknown, which we can s

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

difference scheme (16) is often called the Crank-Nicolson (CN) scheme or a n
d scheme.

he unifying θ-rule

ard Euler, Backward Euler, and Crank-Nicolson schemes can be formulated
ith a varying parameter θ:

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

ve:

0 gives the Forward Euler scheme

1 gives the Backward Euler scheme, and

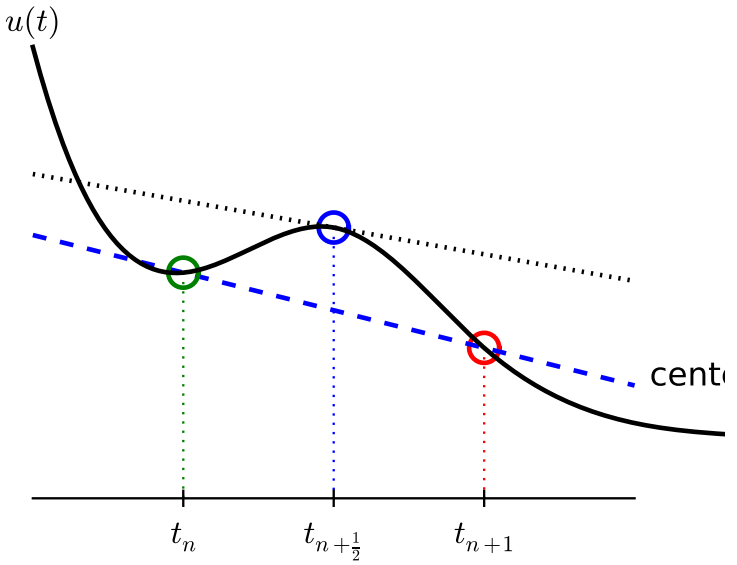


Figure 5: Illustration of a centered difference.

- $\theta = \frac{1}{2}$ gives the Crank-Nicolson scheme.
- We may alternatively choose any other value of θ in $[0, 1]$.

s before, u^n is considered known and u^{n+1} unknown, so we solve for the latte

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

his scheme is known as the θ-rule, or alternatively written as the "theta-rule".

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. Then we
the difference between the methods concerns which point this fraction appr
derivative. Or in other words, at which point we sample the ODE. So far this
end points or the midpoint of $[t_n, t_{n+1}]$. However, we may choose any point
The difficulty is that evaluating the right-hand side $-au$ at an arbitrary point t
problem as in Section 1.4: the point value must be expressed by the discret
that we compute by the scheme, i.e., u^n and u^{n+1}. Following the averagi
Section 1.4, the value of u at an arbitrary point $\tilde{t}$ can be calculated as a *wei*

generalizes the arithmetic mean $\frac{1}{2}u^n + \frac{1}{2}u^{n+1}$. If we express $\tilde{t}$ as a weighted av

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

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

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

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

onstant time step

es up to now have been formulated for a general non-uniform mesh in time: $t_0, t_1,$
rm meshes are highly relevant since one can use many points in regions where
nd save points in regions where u is slowly varying. This is the key idea of
where the spacing of the mesh points are determined as the computations proc
er, a uniformly distributed set of mesh points is very common and sufficient f
ns. It therefore makes sense to present the finite difference schemes for a unifor
on $t_n = n\Delta t$, where Δt is the constant spacing between the mesh points, also
time step. The resulting formulas look simpler and are perhaps more well kno

ary of schemes for constant time step.

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

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

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

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

rprisingly, we present these three alternative schemes because they have differ
both for the simple ODE in question (which can easily be solved as accur
and for more advanced differential equation problems.

he understanding.

point it can be good training to apply the explained finite difference discretiz
ques to a slightly different equation. Exercise 10 is therefore highly recommend
hat the key concepts are understood.

.7 Compact operator notation for finite differences

inite difference formulas can be tedious to write and read, especially for differ
ith many terms and many derivatives. To save space and help the reader o
uickly see the nature of the difference approximations, we introduce a comp
orward difference approximation is denoted by the D_t^+ operator:

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

he notation consists of an operator that approximates differentiation with respe
ent variable, here t. The operator is built of the symbol D, with the variable as
uperscript denoting the type of difference. The superscript $^+$ indicates a forwar
lace square brackets around the operator and the function it operates on and s
oint, where the operator is acting, by a superscript.

The corresponding operator notation for a centered difference and a backward

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

nd

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

ote that the superscript $^-$ denotes the backward difference, while no super
entral difference.

An averaging operator is also convenient to have:

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

he superscript t indicates that the average is taken along the time coordinat
verage $(u^n + u^{n+1})/2$ can now be expressed as $[\overline{u}^t]^{n+\frac{1}{2}}$. (When also spatial c
ne problem, we need the explicit specification of the coordinate after the bar.)

The Backward Euler finite difference approximation to $u' = -au$ can be w
tilizing the compact notation:

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

n difference equations we often place the square brackets around the whole equa
t which mesh point the equation applies, since each term is supposed to be appr
ame point:

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

he Forward Euler scheme takes the form

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

hile the Crank-Nicolson scheme is written as

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

ion.

(25) and (27) and write out the expressions to see that (30) is indeed the C
ɔn scheme.

·rule can be specified by

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

ıe a new time difference and a *weighted averaging operator*:

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

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

$[0,1]$. Note that for $\theta = \frac{1}{2}$ we recover the standard centered difference and the s
c mean. The idea in (31) is to sample the equation at $t_{n+\theta}$, use a skew diffe
$[\bar{D}_t u]^{n+\theta}$, and a skew mean value. An alternative notation is

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

ıg at the various examples above and comparing them with the underlying diff
we see immediately which difference approximations that have been used and a
apply. Therefore, the compact notation effectively communicates the reasoning
differential equation into a difference equation.

ıplementation

nt make a computer program for solving

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

e difference methods. The program should also display the numerical solution
ɔn the screen, preferably together with the exact solution.

ograms referred to in this section are found in the `src/decay`[1] directory (we
Jnix term *directory* for what many others nowadays call *folder*).

atical problem. We want to explore the Forward Euler scheme, the Backwar
Crank-Nicolson schemes applied to our model problem. From an implementation
is advantageous to implement the θ-rule

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

ın generate the three other schemes by various of choices of θ: $\theta = 0$ for Forwar
Backward Euler, and $\theta = 1/2$ for Crank-Nicolson. Given a, $u^0 = I$, T, and Δt,

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

to use the θ-rule to compute $u^1, u^2, \ldots, u^{N_t}$, where $t_{N_t} = N_t\Delta t$, and N_t the c
$T/\Delta t$.

Computer Language: Python. Any programming language can be used
u^{n+1} values from the formula above. However, in this document we shall mai
ython of several reasons:

- Python has a very clean, readable syntax (often known as "executable pse
- Python code is very similar to MATLAB code (and MATLAB has a particu
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 re
C++.
- Python has a rich set of modules for scientific computing, and its popula
computing is rapidly growing.
- Python was made for being combined with compiled languages (C, C++
reuse existing numerical software and to reach high computational perf
implementations.
- Python has extensive support for administrative task needed when doing la
tational investigations.
- Python has extensive support for graphics (visualization, user interfaces, w
- FEniCS, a very powerful tool for solving PDEs by the finite element r
human-efficient to operate from Python.

earning Python is easy. Many newcomers to the language will probably learn
orthcoming examples to perform their own computer experiments. The exar
mple Python code and gradually make use of more powerful constructs as we p
s it is not inconvenient for the problem at hand, our Python code is made as clo
IATLAB code for easy transition between the two languages.

Readers who feel the Python examples are too hard to follow will probal
eading 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]

he author also has a comprehensive book [4] that teaches scientific programmi
om the ground up.

[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`

aking a solver function

e to have an array u for storing the u^n values, $n = 0, 1, \ldots, N_t$. The algorithm

alize u^0

$= t_n$, $n = 1, 2, \ldots, N_t$: compute u_n using the θ-rule formula

for computing the numerical solution. The following Python function t
a of the problem (I, a, T, Δt, θ) as arguments and returns two arrays with the
^{V_t} and the mesh points $t_0, \ldots, t_{N_t}$, respectively:

```
py import *

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

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

umpy library contains a lot of functions for array computing. Most of the functio
r to what is found in the alternative scientific computing language MATLAB.
of

os(Nt+1)` for creating an array of a size `Nt+1` and initializing the elements to z

space(0, T, Nt+1)` for creating an array with `Nt+1` coordinates uniformly dis
veen 0 and T

oop deserves a comment, especially for newcomers to Python. The construction `r
enerates all integers from `0` to `Nt` in steps of `s`, *but not including* `Nt`. Omitting
example, `range(0, 6, 3)` gives `0` and `3`, while `range(0, Nt)` generates `0, 1, .
implies the following assignments to `u[n+1]`: `u[1]`, `u[2]`, ..., `u[Nt]`, which is
e `u` has length `Nt+1`. The first index in Python arrays or lists is *always* `0` and th
u)-1`. The length of an array `u` is obtained by `len(u)` or `u.size`.
mpute with the `solver` function, we need to *call* it. Here is a sample call:

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

division. The shown implementation of the `solver` may face problems and
`T`, `a`, `dt`, and `theta` are given as integers, see Exercises ?? and ??. The pro
integer division in Python (as well as in Fortran, C, C++, and many other c
): `1/2` becomes `0`, while `1.0/2`, `1/2.0`, or `1.0/2.0` all become `0.5`. It is enough
nominator or the denominator is a real number (i.e., a `float` object) to ensure
ical division. Inserting a conversion `dt = float(dt)` guarantees that `dt` is `fl
oblems in Exercise ??.

Another problem with computing $N_t = T/\Delta t$ is that we should round N
nteger. With `Nt = int(T/dt)` the `int` operation picks the largest integer sm
Correct mathematical rounding as known from school is obtained by

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

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

```
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 there is
nclosed in triple double quotes `"""`. The purpose of this string object is to doc
nction does and what the arguments are. In this case the necessary documenta
ore than one line, but with triple double quoted strings the text may span sev

```
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.
    """
    ...
```

uch documentation strings appearing right after the header of a function are c
There are tools that can automatically produce nicely formatted documentatio
ne definition of functions and the contents of doc strings.

It is strongly recommended to equip any function whose purpose is not obv
ring. Nevertheless, the forthcoming text deviates from this rule if the functio
ne text.

Formatting of numbers. Having computed the discrete solution `u`, it is na
ne numbers:

```
out a table of t and u values:
 range(len(t)):
t t[i], u[i]
```

ract `print` statement gives unfortunately quite ugly output because the `t` and
igned in nicely formatted columns. To fix this problem, we recommend to use th
upported most programming languages inherited from C. Another choice is P
mat string syntax.
ıg `t[i]` and `u[i]` in two nicely formatted columns is done like this with th

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

entage signs signify "slots" in the text where the variables listed at the enc
are inserted. For each "slot" one must specify a format for how the variable is
the string: `s` for pure text, `d` for an integer, `g` for a real number written as co
e, `9.3E` for scientific notation with three decimals in a field of width 9 characte
2), or `.2f` for standard decimal notation with two decimals formatted with m
ıe printf syntax provides a quick way of formatting tabular output of numbers
the layout.
lternative *format string syntax* looks like

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

his format allows logical names in the "slots" where `t[i]` and `u[i]` are to be
" are surrounded by curly braces, and the logical name is followed by a colon a
-like specification of how to format real numbers, integers, or strings.

the program. The function and main program shown above must be pla
ith name `decay_v1.py`[6] (`v1` stands for "version 1" - we shall make numerous
f this program). Make sure you write the code with a suitable text editor (Gedit
epad++, or similar). The program is run by executing the file this way:

```
python decay_v1.py
```

`Terminal>` just indicates a prompt in a Unix/Linux or DOS terminal window. A
which will look different in your terminal window, depending on the terminal app
it is set up, commands like `python decay_v1.py` can be issued. These comma
d by the operating system.
ongly recommend to run Python programs within the IPython shell. First start
`ipython` in the terminal window. Inside the IPython shell, our program `decay`
the command `run decay_v1.py`:

```
ipython

un decay_v1.py
u=1
```

/tinyurl.com/jvzzcfn/decay/decay_v1.py


```
= 0.800 u=0.384615
= 1.600 u=0.147929
= 2.400 u=0.0568958
= 3.200 u=0.021883
= 4.000 u=0.00841653
= 4.800 u=0.00323713
= 5.600 u=0.00124505
= 6.400 u=0.000478865
= 7.200 u=0.000184179
= 8.000 u=7.0838e-05

n [2]:
```

The advantage of running programs in IPython are many: previous comn
ecalled with the up arrow, `%pdb` turns on debugging so that variables can be
rogram aborts due to an exception, output of commands are stored in variable
tatements can be profiled, any operating system command can be executed,
oaded automatically and other customizations can be performed when starti
ıention a few of the most useful features.

Although running programs in IPython is strongly recommended, most execu
ıe forthcoming text use the standard Python shell with prompt `>>>` and run pro
ypesetting like

```
erminal> python programname
```

he reason is that such typesetting makes the text more compact in the vertica
owing sessions with IPython syntax.

.2 Verifying the implementation

is easy to make mistakes while deriving and implementing numerical algorithı
ever believe in the printed u values before they have been thoroughly verified. T
lea is to compare the computed solution with the exact solution, when that exist
lways be a discrepancy between these two solutions because of the numerical
he challenging question is whether we have the mathematically correct disc
ave another, maybe small, discrepancy due to both an approximation error anc
nplementation.

The purpose of *verifying* a program is to bring evidence for the property tl
rors in the implementation. To avoid mixing unavoidable approximation erro
nplementation errors, we should try to make tests where we have some exact
ıe discrete solution or at least parts of it. Examples will show how this can be

tunning a few algorithmic steps by hand. The simplest approach to pı
eference for the discrete solution u of finite difference equations is to compute a
lgorithm by hand. Then we can compare the hand calculations with numbers
rogram.

A straightforward approach is to use a calculator and compute u^1, u^2, and u
$= 0.8$, and $\Delta t = 0.8$ we get

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

$$u^1 = AI = 0.0298245614035,$$
$$u^2 = Au^1 = 0.00889504462912,$$
$$u^3 = Au^2 = 0.00265290804728$$

arison of these manual calculations with the result of the `solver` function is car
ction

```
_solver_three_steps():
ompare three steps with known manual computations."""
a = 0.8; a = 2; I = 0.1; dt = 0.8
_hand = array([I,
               0.0298245614035,
               0.00889504462912,
               0.00265290804728])

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

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

`_solver_three_steps` function follows widely used conventions for *unit test*
such conventions we can at a later stage easily execute a big test suite for our s
entions are three-fold:

test function starts with `test_` and takes no arguments.

test ends up in a boolean expression that is `True` if the test passed and `Fa`
ed.

function runs `assert` on the boolean expression, resulting in program abortion
`AssertionError` exception) if the test failed.

program, where we call the `solver` function and print `u`, is now put in a s
`main`:

```
():
 = solver(I=1, a=2, T=8, dt=0.8, theta=1)
ite out a table of t and u values:
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])
```

ain program in the file may first run the verification test prior to going on with
n (`main()`):

```
ver_three_steps()
```

verification test is always done, future errors introduced accidentally in the p
od chance of being detected.
omplete program including the verification above is found in the file `decay_ve`

/tinyurl.com/jvzzcfn/decay/decay_verf.py

.3 Computing the numerical error as a mesh function

low that we have some evidence for a correct implementation, we are in a posi
ne computed u^n values in the `u` array with the exact u values at the mesh po
udy the error in the numerical solution.

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

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

A natural way to compare the exact and discrete solutions is to calculate the
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 functio
ontinuous function defined for all $t \in [0, T]$ (u^n_{e} is often called the *representat*
mesh). Then, $e^n = u^n_{\mathrm{e}} - u^n$ is clearly the difference of two mesh functions. This
^{n} is natural when programming.

The error mesh function e^n can be computed by

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

ote that the mesh functions `u` and `u_e` are represented by arrays and associated
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 points tha
`exact_solution`. This function evaluates `-a*t`, which is a scalar times an a
that the scalar is multiplied with each array element. The result is an array
`tmp1`. Then `exp(tmp1)` means applying the exponential function to each el
resulting an array, say `tmp2`. Finally, `I*tmp2` is computed (scalar times array)
to this array returned from `exact_solution`. The expression `u_e - u` is
between two arrays, resulting in a new array referred to by `e`.

.4 Computing the norm of the numerical error

nstead of working with the error e^n on the entire mesh, we often want one nu
ne size of the error. This is obtained by taking the norm of the error function.

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

$$||f||_{L^2} = \left(\int_0^T f(t)^2 dt \right)^{1/2},$$
$$||f||_{L^1} = \int_0^T |f(t)| dt,$$
$$||f||_{L^\infty} = \max_{t\in[0,T]} |f(t)|\,.$$

orm (35) ("L-two norm") has nice mathematical properties and is the most
s a generalization of the well-known Eucledian norm of vectors to functions. Th
l the max norm or the supremum norm. In fact, there is a whole family of nor

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

l. In particular, $p = 1$ corresponds to the L^1 norm above while $p = \infty$ is the L
rical computations involving mesh functions need corresponding norms. Given
alues, f^n, and some associated mesh points, t_n, a numerical integration rule can
te the L^2 and L^1 norms defined above. Imagining that the mesh function is e
nearly between the mesh points, the Trapezoidal rule is in fact an exact integrat
e modification of the L^2 norm for a mesh function f^n on a uniform mesh with
efore the well-known Trapezoidal integration formula

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

n approximation of this expression, motivated by the convenience of having a
s

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

lled the discrete L^2 norm and denoted by ℓ^2. The error in $||f||^2_{\ell^2}$ compared v
al integration formula is $\Delta t((f^0)^2 + (f^{N_t})^2)/2$, which means perturbed weight
s of the mesh function, and the error goes to zero as $\Delta t \to 0$. As long as
and stick to one kind of integration rule for the norm of a mesh function, the
acy of this rule is not of concern.
ree discrete norms for a mesh function f^n, corresponding to the L^2, L^1, and L^∞
fined 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 interval
f $f = 1$, then the norms are proportional to $\sqrt{T}$ or T). In some applications it
ink of a mesh function as just a vector of function values and neglect the inf
esh points. Then we can replace Δt by T/N_t and drop T. Moreover, it is conv
y the total length of the vector, $N_t + 1$, instead of N_t. This reasoning gives r
orms for a vector $f = (f_0, \ldots, f_N)$:

$$||f||_2 = \left(\frac{1}{N+1} \sum_{n=0}^{N} (f_n)^2 \right)^{1/2},$$
$$||f||_1 = \frac{1}{N+1} \sum_{n=0}^{N} |f_n|$$
$$||f||_{\ell^\infty} = \max_{0\le n\le N} |f_n|\,.$$

ere we have used the common vector component notation with subscripts (f_n)
Ve will mostly work with mesh functions and use the discrete ℓ^2 norm (39) or th
1), but the corresponding vector norms (42)-(44) are also much used in numeric
o it is important to know the different norms and the relations between them.

A single number that expresses the size of the numerical error will be take
alled E:

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

he corresponding Python code, using array arithmetics, reads

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

he `sum` function comes from `numpy` and computes the sum of the elements of ar
qrt function is from `numpy` and computes the square root of each element in the

calar computing. Instead of doing array computing `sqrt(dt*sum(e**2))`
ith 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,
nd runs much slower than what we can achieve with array computing.

lotting solutions

ıe `t` and `u` arrays, the approximate solution `u` is visualized by the intuitive co
`u)`:

```
plotlib.pyplot import *
u)
```

multiple curves. It will be illustrative to also plot $u_e(t)$ for comparison.
`u_e)` is not exactly what we want: the `plot` function draws straight lines betw
oints `(t[n], u_e[n])` while $u_e(t)$ varies as an exponential function between t
he technique for showing the "exact" variation of $u_e(t)$ between the mesh poi
a very fine mesh for $u_e(t)$:

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

more than one curve in the plot we need to associate each curve with a lege
appropriate names on the axis, a title, and a file containing the plot as an in
in reports. The Matplotlib package (`matplotlib.pyplot`) contains functions
The names of the functions are similar to the plotting functions known from M
te plot session then becomes

```
plotlib.pyplot import *

                        # create new plot
nspace(0, T, 1001)      # fine mesh for u_e
act_solution(t_e, I, a)
  u,   'r--o')          # red dashes w/circles
, u_e, 'b-')            # blue line for exact sol.
'numerical', 'exact'])
t')
u')
heta=%g, dt=%g' % (theta, dt))
'%s_%g.png' % (theta, dt))
```

`savefig` here creates a PNG file whose name reflects the values of θ and Δt so
distinguish files from different runs with θ and Δt.
more sophisticated and easy-to-read filename can be generated by mapping the
ms for the three common schemes: FE (Forward Euler, $\theta = 0$), BE (Backwar
N (Crank-Nicolson, $\theta = 0.5$). A Python dictionary is ideal for such a mappi
:o strings:

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

ents with computing and plotting. Let us wrap up the computation of t
ınd all the plotting statements in a function `explore`. This function can be c
and Δt values to see how the error varies with the method and the mesh resol


```
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 sol.
        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 will draw
ırves in the same plot window, while we want the different pairs to appear in s
nd files. Calling `figure()` ensures this.

The `explore` function stores the plot in three different image file formats: I
PS (Encapsulated PostScript). The PNG format is aimed at being included in
DF format in PDFLaTeX documents, and the EPS format in LaTeX documents.
iewers for these image files on Unix systems are `gv` (comes with Ghostscript) f
PS formats and `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`
unning 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 t
alues). We also see that the combination of $\theta = 0.5$ and a small time step Δ
uch more accurate solution, and that $\theta = 0$ and $\theta = 1$ with $\Delta t = 0.4$ result in t
olutions.

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

e 6 demonstrates that the numerical solution for $\Delta t = 0.4$ clearly lies below tl
t that the accuracy improves considerably by reducing the time step by a facto

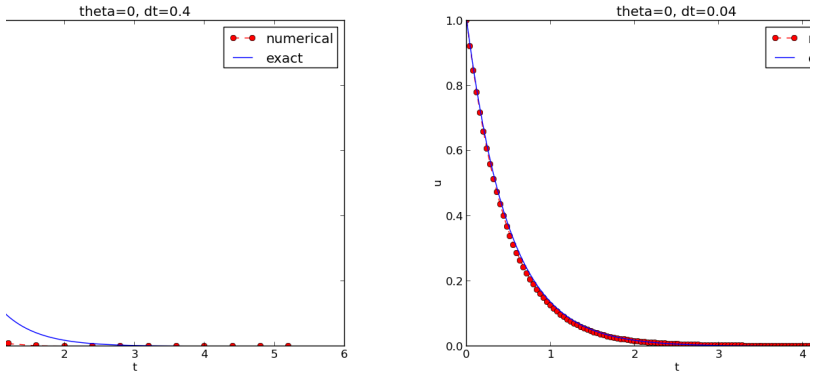


Figure 6: The Forward Euler scheme for two values of the time step.

ng plot files. Mounting two PNG files, as done in the figure, is easily done program from the ImageMagick suite:

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

`metry` argument is used to specify the size of the image, and here we prese sizes of the images. The `-tile HxV` option specifies `H` images in the horizontal d ages in the vertical direction. A series of image files to be combined are the name of the resulting combined image, here `FE1.png` at the end. The `convert` removes surrounding white areas in the figure (an operation usually known as manipulation programs).

LaTeX reports it is not recommended to use `montage` and PNG files as the result tion. Instead, plots should be made in the PDF format and combined using the nd `pdfcrop` tools (on Linux/Unix):

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

`tk` combines images into a multi-page PDF file, `pdfnup` combines the images in in table of images (pages), and `pdfcrop` removes white margins in the resulting c .

ehavior of the two other schemes is shown in Figures 7 and 8. Crank-Nic the most accurate scheme from this visual point of view.

/www.imagemagick.org/script/montage.php

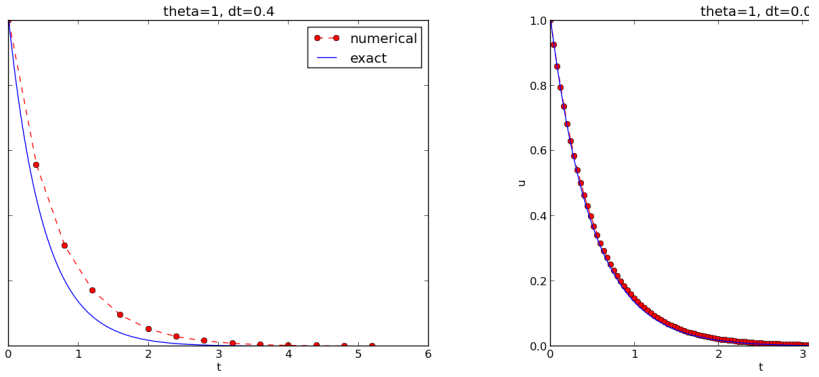


Figure 7: The Backward Euler scheme for two values of the time st

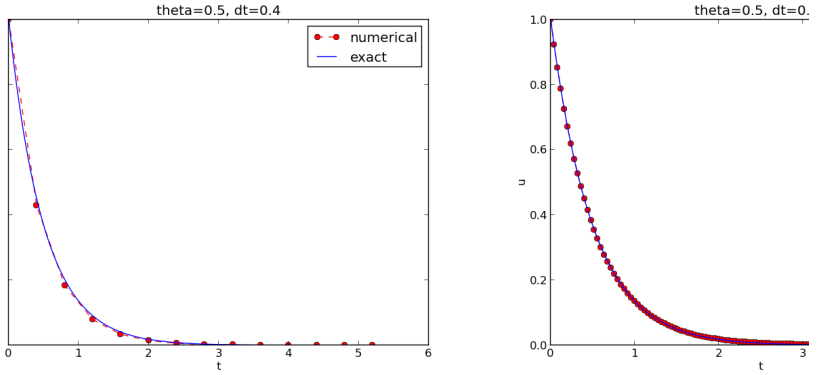


Figure 8: The Crank-Nicolson scheme for two values of the time st

lotting with SciTools. The SciTools package[10] provides a unified plotting asyviz, to many different plotting packages, including Matplotlib, Gnuplot, G TK, OpenDX, and VisIt. The syntax is very similar to that of Matplotlib an ct, the plotting commands shown above look the same in SciTool's Easyviz om the import statement, which reads

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

his statement performs a `from numpy import *` as well as an import of the mos f the Easyviz (`scitools.easyviz`) package, along with some additional numeri

With Easyviz one can merge several plotting commands into a single one rguments:

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

```
  u,   'r--o',           # red dashes w/circles
, u_e, 'b-',             # blue line for exact sol.
end=['numerical', 'exact'],
bel='t',
bel='u',
le='theta=%g, dt=%g' % (theta, dt),
efig='%s_%g.png' % (theta2name[theta], dt),
w=True)
```

y_plot_st.py[11] file contains such a demo.
fault, Easyviz employs Matplotlib for plotting, but Gnuplot[12] and Grace[13] ar
es:

```
python decay_plot_st.py --SCITOOLS_easyviz_backend gnuplot
python decay_plot_st.py --SCITOOLS_easyviz_backend grace
```

end used for creating plots (and numerous other options) can be permanentl
configuration file.
e Gnuplot windows are launched without any need to kill one before the next o
the case with Matplotlib) and one can press the key 'q' anywhere in a plot wi
nother advantage of Gnuplot is the automatic choice of sensible and distinguish
black-and-white PDF and PostScript files.
ding functionality for annotating plots with title, labels on the axis, legends,
he documentation of Matplotlib and SciTools for more detailed information
he hope is that the programming syntax explained so far suffices for understan
learning more from a combination of the forthcoming examples and other re
ooks and web pages.

he understanding.

e 11 asks you to implement a solver for a problem that is slightly different fro
ove. You may use the `solver` and `explore` functions explained above as a sta
Apply the new solver to Exercise 12.

emory-saving implementation

outer memory requirements of our implementations so far consists mainly
ays, both of length $N_t + 1$, plus some other temporary arrays that Python n
ate results if we do array arithmetics in our program (e.g., `I*exp(-a*t)` needs
e `-` can be applied to it and then `exp`). Regardless of how we implement simp
storage requirements are very modest and put not restriction on how we cho
tures and algorithms. Nevertheless, when the methods for ODEs used here are
imensional partial differential equation (PDE) problems, memory storage requi
become a challenging issue.

/tinyurl.com/jvzzcfn/decay/decay_plot_st.py
/www.gnuplot.info/
/plasma-gate.weizmann.ac.il/Grace/

The PDE counterpart to our model problem $u' = -a$ is a diffusion equation
n a space-time domain. The discrete representation of this domain may in
esh of M^3 points and a time mesh of N_t points. A typical desired value for M
pplications, or even 1000. Storing all the computed u values, like we have done
far, demands storage of some arrays of size $M^3 N_t$, giving a factor of M^3 larger
ompared to our ODE programs. Each real number in the array for u require
orage. With $M = 100$ and $N_t = 1000$, there is a storage demand of $(10^3)^3 \cdot$
r the solution array. Fortunately, we can usually get rid of the N_t factor, re
f storage. Below we explain how this is done, and the technique is almost al
nplementations of PDE problems.

Let us critically evaluate how much we really need to store in the comput
ur implementation of the θ method. To compute a new u^{n+1}, all we need is
hat the previous $u^{n-1}, u^{n-2}, \ldots, u^0$ values do not need to be stored in an array,
onvenient for plotting and data analysis in the program. Instead of the `u` array w
wo variables for real numbers, `u` and `u_1`, representing u^{n+1} and u^n in the algorit
t each time level, we update `u` from `u_1` and then set `u_1 = u` so that the comp
ecomes the "previous" value u^n at the next time level. The downside is that we
olution after the simulation is done since only the last two numbers are available
store computed values in a file and use the file for visualizing the solution lat

We have implemented this memory saving idea in the file `decay_memsave.`
ight modification of `decay_plot_mpl.py`[15] program.

The following function demonstrates how we work with the two most rece
nknown:

```
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
```

his code snippet serves as a quick introduction to file writing in Python. Rea
e 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:
```

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

```
words = line.split()
if len(words) != 2:
    print 'Found more than two numbers on a line!', words
    sys.exit(1)  # abort
t.append(float(words[0]))
u.append(float(words[1]))
rn np.array(t), np.array(u)
```

ype of file with numbers in rows and columns is very common, and `numpy` has a
which loads such tabular data into a two-dimensional array, say with name `da`
ı row `i` and column `j` is then `data[i,j]`. The whole column number `j` can be e:
`:,j]`. A version of `read_file` using `np.loadtxt` reads

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

resent counterpart to the `explore` function from `decay_plot_mpl.py`[16] m
`emsave` and then load data from file before we can compute the error measure a

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

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

from the internal implementation, where u^n values are stored in a file rather th
`ay_memsave.py` file works exactly as the `decay_plot_mpl.py` file.

alysis of finite difference equations

ss the ODE for exponential decay,

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

nd I are given constants. This problem is solved by the θ-rule finite difference
in the recursive equations

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

umerical solution u^{n+1}, which approximates the exact solution u_e at time poi
ant mesh spacing, which we assume here, $t_{n+1} = (n+1)\Delta t$.

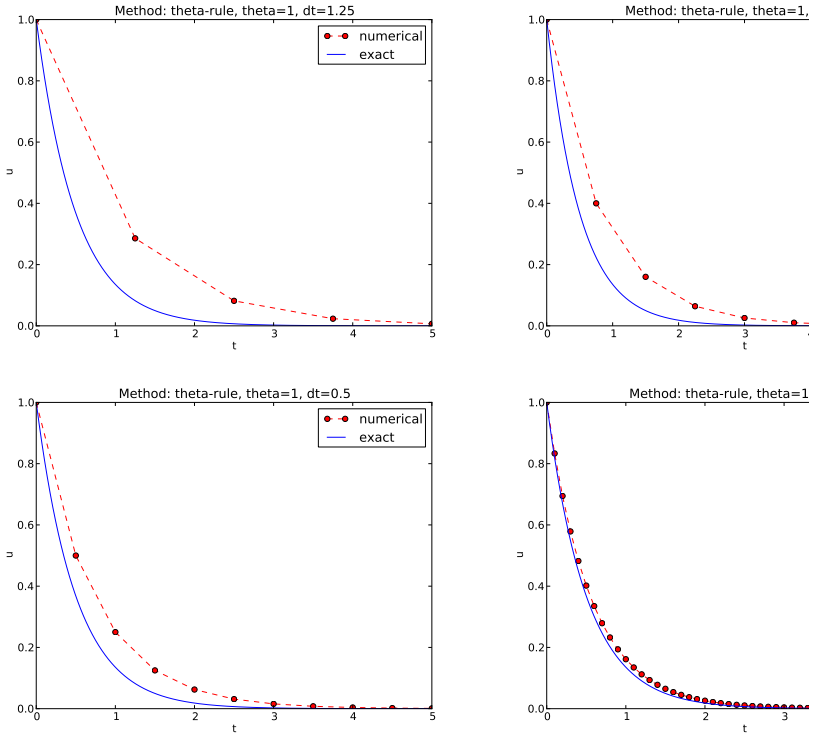


Figure 9: Backward Euler.

)iscouraging numerical solutions. Choosing $I = 1$, $a = 2$, and running e
$= 1, 0.5, 0$ for $\Delta t = 1.25, 0.75, 0.5, 0.1$, gives the results in Figures 9, 10, and 1
The characteristics of the displayed curves can be summarized as follows:

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

ince the exact solution of our model problem is a monotone function, $u(t) =$
ıese qualitatively wrong results are indeed alarming!

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

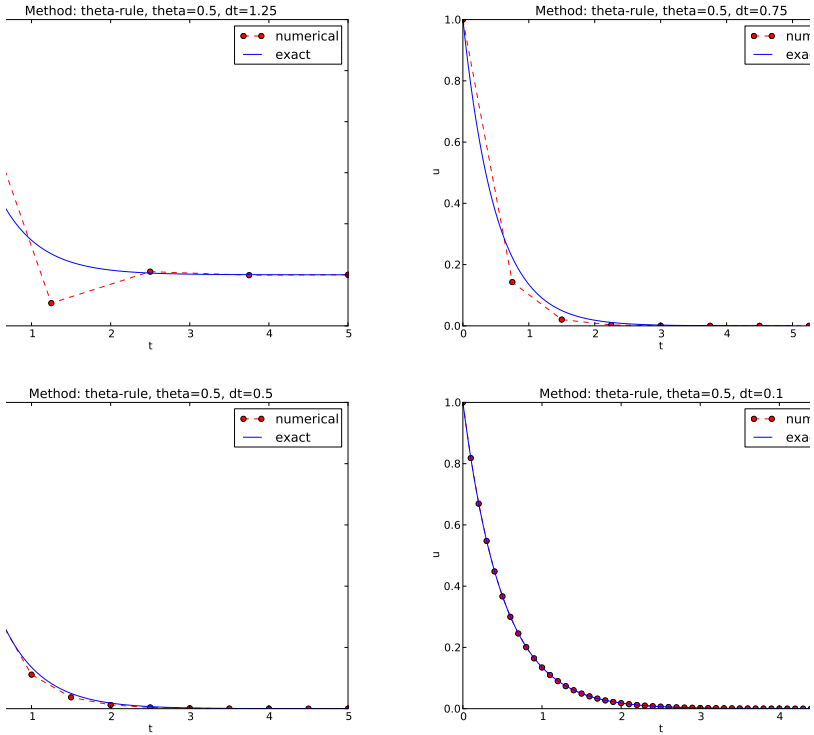


Figure 10: Crank-Nicolson.

: the question

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

estion will be investigated both by numerical experiments and by precise mathema
The latter will help establish general criteria on Δt for avoiding non-phy
ory or growing solutions.
other question to be raised is

How does Δt impact the error in the numerical solution?

r simple model problem we can answer this question very precisely, but we
ok at simplified formulas for small Δt and touch upon important concepts su
gence rate and *the order of a scheme*. Other fundamental concepts mentione
y, consistency, and convergence.

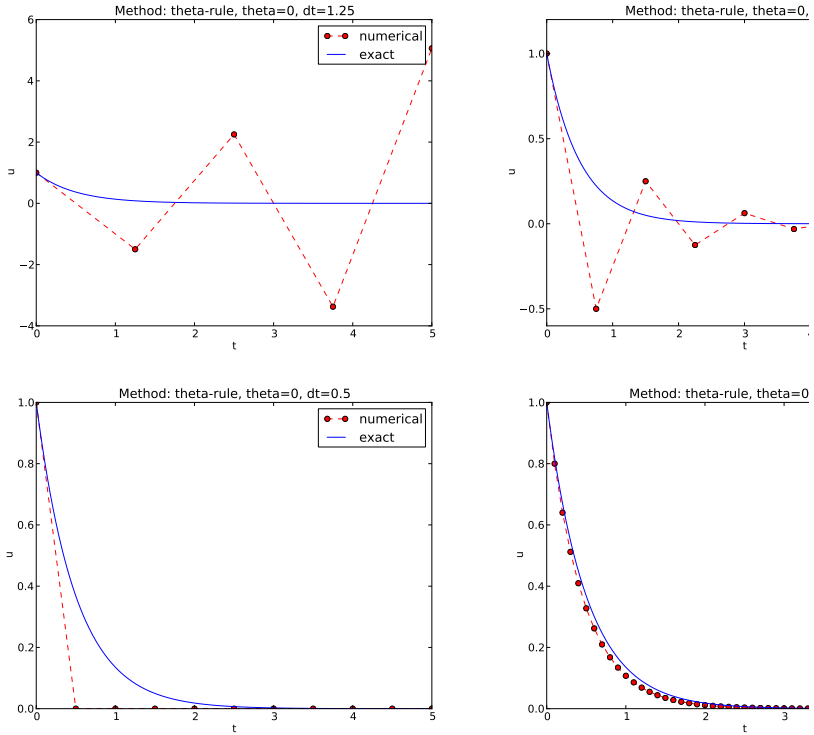


Figure 11: Forward Euler.

.1 Experimental investigation of oscillatory solutions

o address the first question above, we may set up an experiment where we loop
, and Δt. For each experiment, we flag the solution as oscillatory if

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

r some value of n, since we expect u^n to decay with n, but oscillations make u
me step. We will quickly see that oscillations are independent of I, but do d
t. Therefore, we introduce a two-dimensional function $B(a, \Delta t)$ which is 1 if o
nd 0 otherwise. We can visualize B as a contour plot (lines for which $B = \text{cons}$
$= 0.5$ corresponds to the borderline between oscillatory regions with $B = 1$
gions with $B = 0$ in the $a, \Delta t$ plane.

The B function is defined at discrete a and Δt values. Say we have gi
$0, \ldots, a_{P-1}$, and Q Δt values, $\Delta t_0, \ldots, \Delta t_{Q-1}$. These a_i and Δt_j values, i
$= 0, \ldots, Q-1$, form a rectangular mesh of $P \times Q$ points in the plane. At each
e associate the corresponding value of $B(a_i, \Delta t_j)$, denoted B_{ij}. The B_{ij} valu
ored in a two-dimensional array. We can thereafter create a plot of the contou

he oscillatory and monotone regions. The file `decay_osc_regions.py`[17] `osc_r` "oscillatory regions") contains all nuts and bolts to produce the $B = 0.5$ 2 and 13. The oscillatory region is above this line.

```
cay_mod import solver
numpy as np
scitools.std as st

_physical_behavior(I, a, T, dt, theta):

en lists/arrays a and dt, and numbers I, dt, and theta,
e a two-dimensional contour line B=0.5, where B=1>0.5
ns oscillatory (unstable) solution, and B=0<0.5 means
otone solution of u'=-au.

 np.asarray(a); dt = np.asarray(dt)  # must be arrays
 np.zeros((len(a), len(dt)))         # results
 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)
 dt_ = st.ndgrid(a, dt)  # make mesh of a and dt values
contour(a_, dt_, B, 1)
grid('on')
title('theta=%g' % theta)
xlabel('a'); st.ylabel('dt')
savefig('osc_region_theta_%s.png' % theta)
savefig('osc_region_theta_%s.pdf' % theta)

sical_behavior(
,
p.linspace(0.01, 4, 22),
np.linspace(0.01, 4, 22),
,
ta=0.5)
```

oking at the curves in the figures one may guess that $a\Delta t$ must be less than a oid the undesired oscillations. This limit seems to be about 2 for Crank-Nicols rd Euler. We shall now establish a precise mathematical analysis of the discret explain the observations in our numerical experiments.

xact numerical solution

with $u^0 = I$, the simple recursion (47) can be applied repeatedly n times, with tl

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

g difference equations.

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

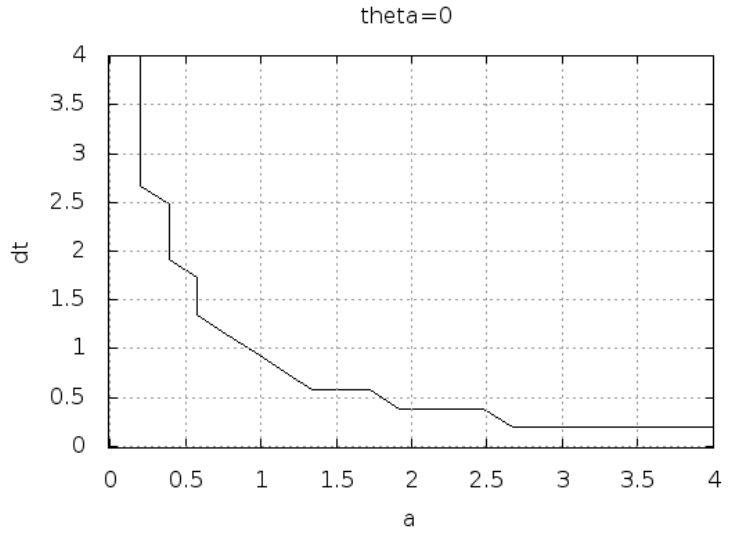


Figure 12: Forward Euler scheme: oscillatory solutions occur for points abo

Difference equations where all terms are linear in u^{n+1}, u^n, and maybe u^{n-} are called *homogeneous, linear* difference equations, and their solutions are ge form $u^n = A^n$. Inserting this expression and dividing by A^{n+1} gives a polyno in A. In the present case we get

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

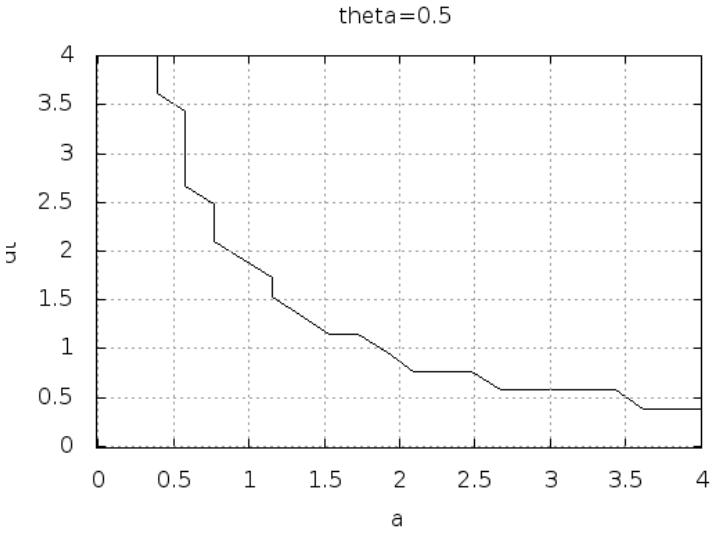
This is a solution technique of wider applicability than repeated use of the re

Regardless of the solution approach, we have obtained a formula for u^n. T xplain everything what we see in the figures above, but it also gives us a mor ito accuracy and stability properties of the three schemes.

.3 Stability

ince u^n is a factor A raised to an integer power n, we realize that $A < 0$ will nply $u^n < 0$ and for even power result in $u^n > 0$. That is, the solution oscilla esh points. We have oscillations due to $A < 0$ when

$$(1-\theta)a\Delta t > 1\,.$$



13: Crank-Nicolson scheme: oscillatory solutions occur for points above the c

> 0 is a requirement for having a numerical solution with the same basic p
city) as the exact solution, we may say that $A > 0$ is a *stability criterion*. Expr
Δt the stability criterion reads

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

Backward Euler scheme is always stable since $A < 0$ is impossible for $\theta =$
ating solutions for Forward Euler and Crank-Nicolson demand $\Delta t \leq 1/a$ and Δ
ly. The relation between Δt and a look reasonable: a larger a means faster de
eed for smaller time steps.
ng at Figure 11, we see that with $a\Delta t = 2 \cdot 1.25 = 2.5$, $A = -1.5$, and the
$.5)^n$ oscillates *and* grows. With $a\Delta t = 2 \cdot 0.75 = 1.5$, $A = -0.5$, $u^n = (-0.5)^n$
ates. The peculiar case $\Delta t = 0.5$, where the Forward Euler scheme produces a
ick on the t axis, corresponds to $A = 0$ and therefore $u^0 = I = 1$ and $u^n = 0$ fo
ring oscillations in the Crank-Nicolson scheme for $\Delta t = 1.25$ are easily explained
$A \approx -0.11 < 0$.
ctor A is called the *amplification factor* since the solution at a new time level is
on at the previous time level. For a decay process, we must obviously have
ulfilled for all Δt if $\theta \geq 1/2$. Arbitrarily large values of u can be generated when
large enough. The numerical solution is in such cases totally irrelevant to a
decay processes! To avoid this situation, we must for $\theta < 1/2$ have

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

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 requ
for avoiding growing solutions and $\Delta t < 1/a$ for avoiding oscillatory so
2. The Crank-Nicolson is *unconditionally stable* with respect to growing so
it is conditionally stable with the criterion $\Delta t < 2/a$ for avoiding oscilla
3. The Backward Euler method is unconditionally stable with respect to
oscillatory solutions - any Δt will work.

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

.4 Comparing amplification factors

fter establishing how A impacts the qualitative features of the solution, we shal
ito how well the numerical amplification factor approximates the exact one. Th
eads $u(t) = Ie^{-at}$, which can be rewritten as

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

rom this formula we see that the exact amplification factor is

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

We realize that the exact and numerical amplification factors depend on a
ie product $a\Delta t$. Therefore, it is convenient to introduce a symbol for this pr
nd view A and A_{e} as functions of p. Figure 14 shows these functions. Crank-N
losest to the exact amplification factor, but that method has the unfortunate osc
hen $p > 2$.

.5 Series expansion of amplification factors

s an alternative to the visual understanding inherent in Figure 14, there is a
numerical analysis to establish formulas for the approximation errors when th
arameter, here Δt, becomes small. In the present case we let p be our sma
arameter, and it makes sense to simplify the expressions for A and A_{e} by using Ta
round $p = 0$. The Taylor polynomials are accurate for small p and greatl
omparison of the analytical expressions since we then can compare polynomial
Calculating the Taylor series for A_{e} is easily done by hand, but the three
$= 0, 1, \frac{1}{2}$ lead to more cumbersome calculations. Nowadays, analytical computa

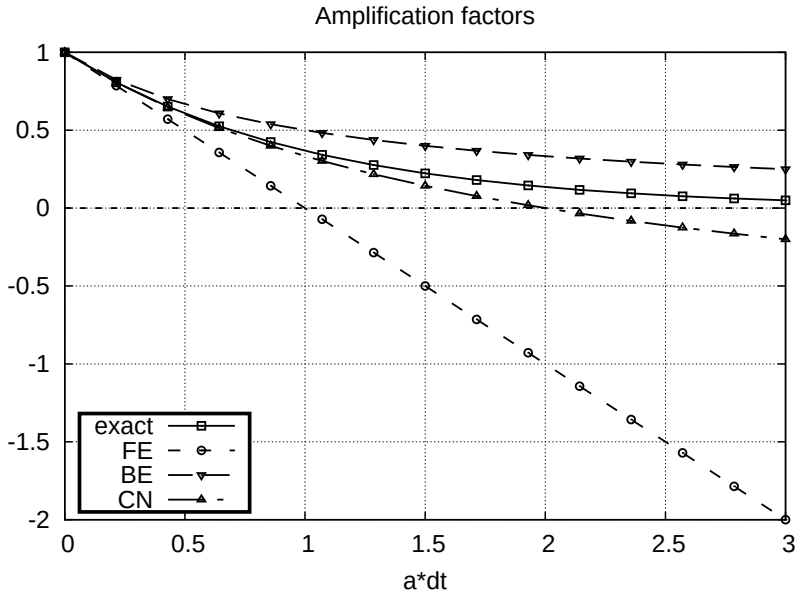


Figure 14: Comparison of amplification factors.

symbolic computer algebra software. The Python package `sympy` represents a p
algebra system, not yet as sophisticated as the famous Maple and Mathematica
nd very easy to integrate with our numerical computations in Python.
using `sympy`, it is convenient to enter the interactive Python mode where we ca
ıs and statements and immediately see the results. Here is a simple examp
ecommend to use `isympy` (or `ipython`) for such interactive sessions.
illustrate `sympy` with a standard Python shell syntax (`>>>` prompt) to compute
ıl approximation to e^{-p}:

```
 sympy import *
eate p as a mathematical symbol with name 'p'
Symbol('p')
eate a mathematical expression with p
= exp(-p)

nd the first 6 terms of the Taylor series of A_e
series(p, 0, 6)
)*p**2 - p - 1/6*p**3 - 1/120*p**5 + (1/24)*p**4 + O(p**6)
```

h `>>>` represent input lines and lines without this prompt represents the r
ions (note that `isympy` and `ipython` apply other prompts, but in this text we
for interactive Python computing). Apart from the order of the powers, the co
s easily recognized as the beginning of the Taylor series for e^{-p}.
define the numerical amplification factor where p and θ enter the formula as sy


```
>>> 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 valu

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

imilarly, we can replace `theta` by $1/2$ for Crank-Nicolson, preferably using a
epresentation 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 c
s

```
>>> 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
uler schemes, while $A - A_e \sim \mathcal{O}(p^3)$ for the Crank-Nicolson scheme. It is the *lea*
e., the term of the lowest order (polynomial degree), that is of interest, becaus
erm is (much) bigger than the higher-order terms (think of $p = 0.01$: p is a hund
ıan p^2).

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

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

We say that the Crank-Nicolson scheme has an error in the amplification fac
hile the two other schemes are of order Δt^2 in the same quantity. What is
f the order expression? If we halve Δt, the error in amplification factor at
e reduced by a factor of 4 in the Forward and Backward Euler schemes, and
ı the Crank-Nicolson scheme. That is, as we reduce Δt to obtain more accu
rank-Nicolson scheme reduces the error more efficiently than the other scheme

he fraction of numerical and exact amplification factors

ative comparison of the schemes is to look at the ratio A/A_e, or the error $1 - $

```
 1 - (A.subs(theta, 0)/A_e).series(p, 0, 4)
 1 - (A.subs(theta, 1)/A_e).series(p, 0, 4)
 1 - (A.subs(theta, half)/A_e).series(p, 0, 4)

*2 + (1/3)*p**3 + O(p**4)

2 + (1/3)*p**3 + O(p**4)

**3 + O(p**4)
```

ng-order terms have the same powers as in the analysis of $A - A_\mathrm{e}$.

he global error at a point

in the amplification factor reflects the error when progressing from time level t_n
gate the real error at a point, known as the *global error*, we look at $e^n = u^n$
n and Taylor expand the mathematical expressions as functions of $p = a\Delta t$:

```
Symbol('n')
= exp(-p*n)
= A**n
 u_e.series(p, 0, 4) - u_n.subs(theta, 0).series(p, 0, 4)
 u_e.series(p, 0, 4) - u_n.subs(theta, 1).series(p, 0, 4)
 u_e.series(p, 0, 4) - u_n.subs(theta, half).series(p, 0, 4)

p**2 - 1/2*n**2*p**3 + (1/3)*n*p**3 + O(p**4)

*2*p**3 - 1/2*n*p**2 + (1/3)*n*p**3 + O(p**4)

*p**3 + O(p**4)
```

d time t, the parameter n in these expressions increases as $p \to 0$ since $t = n\Delta t$
e n must increase like Δt^{-1}. With n substituted by $t/\Delta t$ in the leading-ord
ese become $\frac{1}{2}na^2\Delta t^2 = \frac{1}{2}ta^2\Delta t$ for the Forward and Backward Euler sche
$= \frac{1}{12}ta^3\Delta t^2$ for the Crank-Nicolson scheme. The global error is therefore o
Δt) for the latter scheme and of first order for the former schemes.
the global error $e^n \to 0$ as $\Delta t \to 0$, we say that the scheme is *convergent*. I
numerical solution approaches the exact solution as the mesh is refined, and
ired property of a numerical method.

tegrated errors

non to study the norm of the numerical error, as explained in detail in Section
can be computed by treating e^n as a function of t in `sympy` and performing s
n. For the Forward Euler scheme we have

```
 dt, t, T, theta = symbols('p n a dt t T 'theta')
1-theta)*p)/(1+theta*p)
p(-p*n)
*n
u_e.series(p, 0, 4) - u_n.subs(theta, 0).series(p, 0, 4)
```



```
# Introduce t and dt instead of n and p
error = error.subs('n', 't/dt').subs(p, 'a*dt')
error = error.as_leading_term(dt) # study only the first term
print error
error_L2 = sqrt(integrate(error**2, (t, 0, T)))
print 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 s
ense to compute the ℓ^2 error

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

We have obtained an exact analytical expressions for the error at $t = t_n$, but
eading-order error term only since we are mostly interested in how the err
olynomial in Δt, and then the leading order term will dominate. For the Forwa
$_\mathrm{e}(t_n) - u^n \approx \frac{1}{2}np^2$, and we have

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

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 taking
ives 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 sa
ne 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 order
Crank-Nicolson scheme and of first order in Δt for the Forward Euler and Ba
schemes.

.9 Truncation error

he truncation error is a very frequently used error measure for finite differe
defined as *the error in the difference equation that arises when inserting th*
ontrary to many other error measures, e.g., the true error $e^n = u_\mathrm{e}(t_n) - u^n$
ror is a quantity that is easily computable.

s illustrate the calculation of the truncation error for the Forward Euler sche
the difference equation on operator form,

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

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

is to see how well the exact solution $u_{\rm e}(t)$ fulfills this equation. Since $u_{\rm e}(t)$ in
bey the discrete equation, error in the discrete equation, called a *residual*, deno

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

ual is defined at each mesh point and is therefore a mesh function with a super
nteresting feature of R^n is to see how it depends on the discretization param
for reaching this goal is to Taylor expand $u_{\rm e}$ around the point where the di
is supposed to hold, here $t = t_n$. We have that

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

this Taylor series in (55) gives

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

ulfills the ODE $u'_{\rm e} = -au_{\rm e}$ such that the first and last term cancels and we ha

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

s the *truncation error*, which for the Forward Euler is seen to be of first order
bove procedure can be repeated for the Backward Euler and the Crank-Nicolson
with the scheme in operator notation, write it out in detail, Taylor expand $u_{\rm e}$
$\tilde{t}$ at which the difference equation is defined, collect terms that correspond to t
$au_{\rm e}$), and identify the remaining terms as the residual R, which is the truncati
ward Euler scheme leads to

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

Crank-Nicolson scheme gives

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

rder r of a finite difference scheme is often defined through the leading term
ation error. The above expressions point out that the Forward and Backwar
re of first order, while Crank-Nicolson is of second order. We have looked at oth
in other sections, like the error in amplification factor and the error $e^n = u_{\rm e}(t$
essed these error measures in terms of Δt to see the order of the method. N
g the truncation error is more straightforward than deriving the expressions f
sures and therefore the easiest way to establish the order of a scheme.

.10 Consistency, stability, and convergence

hree fundamental concepts when solving differential equations by numeric
nsistency, stability, and convergence. We shall briefly touch these concepts belo
f the present model problem.

Consistency means that the error in the difference equation, measured throug
rror, goes to zero as $\Delta t \to 0$. Since the truncation error tells how well the exac
he difference equation, and the exact solution fulfills the differential equation, cor
hat the difference equation approaches the differential equation in the limit. Th
he truncation errors in the previous section are all proportional to Δt or Δt^2, h
s $\Delta t \to 0$, and all the schemes are consistent. Lack of consistency implies that
different differential equation in the limit $\Delta t \to 0$ than we aim at.

Stability means that the numerical solution exhibits the same qualitative p
xact solution. This is obviously a feature we want the numerical solution to hav
xponential decay model, the exact solution is monotone and decaying. An incr
olution is not in accordance with the decaying nature of the exact solution and
Ve can also say that an oscillating numerical solution lacks the property of mo
xact solution and is also unstable. We have seen that the Backward Euler sche
monotone and decaying solutions, regardless of Δt, and is hence stable. Th
cheme can lead to increasing solutions and oscillating solutions if Δt is too large
nstable unless Δt is sufficiently small. The Crank-Nicolson can never lead to inc
nd has no problem to fulfill that stability property, but it can produce oscillati
unstable in that sense, unless Δt is sufficiently small.

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

Convergence is hard to establish theoretically, except in quite simple pr
resent one. Stability and consistency are much easier to calculate. A majo
the understanding of numerical methods for differential equations came in
nd Richtmeyer established equivalence between convergence on one hand and
tability on the other (the Lax equivalence theorem[18]). In practice it meant th
stablish that a method is stable and consistent, and then it is automatically con
uch harder to establish). The result holds for linear problems only, and in the w
ifferential equations the relations between consistency, stability, and convergenc
omplicated.

We have seen in the previous analysis that the Forward Euler, Backward E
icolson schemes are convergent ($e^n \to 0$), that they are consistent ($R^n \to 0$, a
table under certain conditions on the size of Δt. We have also derived explic
xpressions for e^n, the truncation error, and the stability criteria.

Exercises

Exercise 1: Visualize the accuracy of finite differences $u =$

he purpose of this exercise is to visualize the accuracy of finite difference appro
erivative of a given function. For any finite difference approximation, take th
ifference as an example, and any specific function, take $u = e^{-at}$, we may int

[18] http://en.wikipedia.org/wiki/Lax_equivalence_theorem

pecific

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

E as a function of Δt. We expect that $\lim_{\Delta t\to 0} E = 1$, while E may deviate sign
for large Δt. How the error depends on Δt is best visualized in a graph where
ic scale on for Δt, so we can cover many orders of magnitude of that quantity. I
ient creating an array of 100 intervals, on the logarithmic scale, ranging from 1
plotting E versus $p = a\Delta t$ with logarithmic scale on the Δt axis:

```
py import logspace, exp
plotlib.pyplot import plot
pace(-6, 1, 101)
p(-p)-1)/p
p, y)
```

such errors for the finite difference operators $[D_t^+ u]^n$ (forward), $[D_t^- u]^n$ (bac
^{n} (centered).
m a Taylor series expansions of the error fractions and find the leading order r i
of type $1+C\Delta t^r+\mathcal{O}(\Delta t^{r+1})$, where C is some constant. Filename: `decay_plot_f`

e 2: Explore the θ-rule for exponential growth

cise asks you to solve the ODE $u' = -au$ with $a < 0$ such that the ODE
ial growth instead of exponential decay. A central theme is to investigate nu
und non-physical solution behavior.

xperiments with $\theta = 0, 0.5, 1$ for various values of Δt to uncover numerical a
at the exact solution is a monotone, growing function when $a < 0$. Oscilla
tly wrong growth are signs of wrong qualitative behavior, which can be tak
criterion.
e insight to select a few values of Δt and a fixed that demonstrate all types of nu
or the three different schemes ($\theta = 0, 0.5, 1$).

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

up the amplification factor and plot it for $\theta = 0, 0.5, 1$ together with the exact
Use the plot to explain the observations made in the experiments.

Modify the `decay_ampf_plot.py`[19] code.
`growth_ampf.py`.

a scientific report about the findings.

se examples from Section **??** to see how scientific reports can be written.
s: `growth_exper.pdf`, `growth_exper.html`.

/tinyurl.com/jvzzcfn/decay/decay_ampf_plot.py

Model extensions

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

.1 Generalization: including a variable coefficient

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

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

A Forward Euler scheme consist of evaluating (56) at $t = t_n$ and approximati
ith a forward difference $[D_t^+ u]^n$:

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

he Backward Euler scheme becomes

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

he Crank-Nicolson method builds on sampling the ODE at $t_{n+\frac{1}{2}}$. We can eva
nd use an average for u at times t_n and t_{n+1}:

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

lternatively, we can use an average for the product au:

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

he θ-rule unifies the three mentioned schemes. One version is to have a evalua

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

nother 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 Backwar
an be summarized as

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

he Crank-Nicolson and θ schemes depend on whether we evaluate a at the sam
DE or if we use an average. The various versions are written as

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

eneralization: including a source term

extension of the model ODE is to include a source term $b(t)$:

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

. The time point where we sample the ODE determines where $b(t)$ is evaluat
-Nicolson scheme and the θ-rule we have a choice of whether to evaluate $a(t)$
rect point or use an average. The chosen strategy becomes particularly clear if
emes in the operator notation:

$$\begin{aligned}
[D_t^+ u &= -au + b]^n,\\
[D_t^- u &= -au + b]^n,\\
[D_t u &= -a\overline{u}^t + b]^{n+\frac{1}{2}},\\
[D_t u &= \overline{-au + b}^t]^{n+\frac{1}{2}},\\
[D_t u &= -a\overline{u}^{t,\theta} + b]^{n+\theta},\\
[D_t u &= \overline{-au + b}^{t,\theta}]^{n+\theta}\,.
\end{aligned}$$

mplementation of the generalized model problem

; the θ-rule formula. Writing out the θ-rule in (75), using (32) and (33), w

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

means evaluating a at $t = t_n$ and similar for a^{n+1}, b^n, and b^{n+1}. We solve for

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

hon code. Here is a suitable implementation of (76) where $a(t)$ and $b(t)$ are
nctions:

```
er(I, a, b, T, dt, theta):

e u'=-a(t)*u + b(t), u(0)=I,
t in (0,T] with steps of dt.
d b are Python functions of t.

 float(dt)            # avoid integer division
 int(round(T/dt))     # no of time intervals
Nt*dt                 # adjust T to fit time step dt
zeros(Nt+1)           # array of u[n] values
linspace(0, T, Nt+1)  # time mesh

 = I                  # assign initial condition
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]))
rn u, t
```

tion is found in the file `decay_vc.py`[20] (`vc` stands for "variable coefficients").

[20] /tinyurl.com/jvzzcfn/decay/decay_vc.py

oding of variable coefficients. The `solver` function shown above demand
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
```

ere, `a(t)` has three parameters `a0`, `tp`, and `k`, which must be global varia
nplementation is to represent `a` by a class where the parameters are attribut
ethod `__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 class. The
onstruction in Python is then convenient. For example,

```
a = lambda t: a_0 if t < tp else k*a_0
```

equivalent to the `def a(t):` definition above. In general,

```
f = lambda arg1, arg2, ...: expressin
```

equivalent to

```
def f(arg1, arg2, ...):
    return expression
```

ne can use lambda functions directly in calls. Say we want to solve $u' = -u +$

```
u, t = solver(2, lambda t: 1, lambda t: 1, T, dt, theta)
```

lambda function can appear anywhere where a variable can appear.

.4 Verifying a constant solution

very useful partial verification method is to construct a test problem with a very
sually $u = \text{const}$. Especially the initial debugging of a program code can bene
uch tests, because 1) all relevant numerical methods will exactly reproduce a c
) many of the intermediate calculations are easy to control for a constant u
onstant u can uncover many bugs in an implementation.

The only constant solution for the problem $u' = -au$ is $u = 0$, but too many
om that trivial solution. It is much better to search for a problem where $u =$
hen $u' = -a(t)u + b(t)$ is more appropriate: with $u = C$ we can choose any $a(t)$ a
nd $I = C$. An appropriate nose test is

```
ose.tools as nt

_constant_solution():

 problem where u=u_const is the exact solution, to be
oduced (to machine precision) by any relevant method.

exact_solution(t):
return u_const

a(t):
return 2.5*(1+t**3)  # can be arbitrary

b(t):
return a(t)*u_const

nst = 2.15
a = 0.4; I = u_const; dt = 4
 4  # enough with a few steps
 = solver(I=I, a=a, b=b, T=Nt*dt, dt=dt, theta=theta)
t u
= exact_solution(t)
erence = abs(u_e - u).max()  # max deviation
ssert_almost_equal(difference, 0, places=14)
```

eresting question is what type of bugs that will make the computed u^n deviate f
tion C. Fortunately, the updating formula and the initial condition must be ab
the test to pass! Any attempt to make a wrong indexing in terms like `a(t[n]`
o introduce an erroneous factor in the formula creates a solution that is differe

erification via manufactured solutions

the idea of the previous section, we can choose any formula as the exact s
formula in the ODE problem and fit the data $a(t)$, $b(t)$, and I to make the
ulfill the equation. This powerful technique for generating exact solutions is ver
ation purposes and known as the *method of manufactured solutions*, often abb

ommon choice of solution is a linear function in the independent variable(s
behind such a simple variation is that almost any relevant numerical solution me
l equation problems is able to reproduce the linear function exactly to machine p
out unity in size; precision is lost if u take on large values, see Exercise 3). Th
lso makes some stronger demands to the numerical method and the impleme
constant solution used in Section 5.4, at least in more complicated applications. I
ant solution is often ideal for initial debugging before proceeding with a linear s
oose a linear solution $u(t) = ct + d$. From the initial condition it follows tha
this u in the ODE results in

$$c = -a(t)u + b(t)\,.$$

ion $u = ct + I$ is then a correct solution if we choose

$$b(t) = c + a(t)(ct + I)\,.$$

$b(t)$ there are no restrictions on $a(t)$ and c.

Let prove that such a linear solution obeys the numerical schemes. To th
heck that $u^n = ca(t_n)(ct_n + I)$ fulfills the discrete equations. For these calcula
alculations involving linear solutions inserted in finite difference schemes, it
ompute the action of a difference operator on a linear function t:

$$[D_t^+ t]^n = \frac{t_{n+1} - t_n}{\Delta t} = 1,$$
$$[D_t^- t]^n = \frac{t_n - t_{n-1}}{\Delta t} = 1,$$
$$[D_t t]^n = \frac{t_{n+\frac{1}{2}} - t_{n-\frac{1}{2}}}{\Delta t} = \frac{(n+\frac{1}{2})\Delta t - (n-\frac{1}{2})\Delta t}{\Delta t} = 1\,.$$

learly, all three finite difference approximations to the derivative are exact fo
esh function counterpart $u^n = t_n$.

The difference equation for the Forward Euler scheme

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

ith $a^n = a(t_n)$, $b^n = c + a(t_n)(ct_n + I)$, and $u^n = ct_n + I$ then results in

$$c = -a(t_n)(ct_n + I) + c + a(t_n)(ct_n + I) = c$$

hich is always fulfilled. Similar calculations can be done for the Backward E
icolson schemes, or the θ-rule for that matter. In all cases, $u^n = ct_n + I$ is an
e discrete equations. That is why we should expect that $u^n - u_e(t_n) = 0$ mat
$u^n - u_e(t_n)|$ less than a small number about the machine precision for $n = 0,$.

The following function offers an implementation of this verification test b
xact 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 $\cos(t)$, wi
umerical and exact solution coincide to machine precision, because finite diffe

exactly yield the exact derivative $-\sin(t)$. In such cases, the verification procedu
on measuring the convergence rates as exemplified in Section **??**. Convergence r
ted as long as one has an exact solution of a problem that the solver can be te
an always be obtained by the method of manufactured solutions.

xtension to systems of ODEs

E models involves more than one unknown function and more than one equatic
nple of two unknown functions $u(t)$ and $v(t)$:

$$u' = au + bv,$$
$$v' = cu + dv,$$

unts a, b, c, d. Applying the Forward Euler method to each equation results ir
formula

$$u^{n+1} = u^n + \Delta t(au^n + bv^n),$$
$$v^{n+1} = u^n + \Delta t(cu^n + dv^n)\,.$$

her hand, the Crank-Nicolson or Backward Euler schemes result in a 2×2 linea
w unknowns. The latter schemes gives

$$u^{n+1} = u^n + \Delta t(au^{n+1} + bv^{n+1}),$$
$$v^{n+1} = v^n + \Delta t(cu^{n+1} + dv^{n+1})\,.$$

; u^{n+1} as well as v^{n+1} on the left-hand side results in

$$(1 - \Delta ta)u^{n+1} + bv^{n+1} = u^n,$$
$$cu^{n+1} + (1 - \Delta td)v^{n+1} = v^n,$$

system of two coupled, linear, algebraic equations in two unknowns.

eneral first-order ODEs

rn the attention to general, nonlinear ODEs and systems of such ODEs. Our fo
methods that can be readily reused for time-discretization PDEs, and diffusio
ular. The methods are just briefly listed, and we refer to the rich literature f
escriptions and analysis - the books [6, 1, 2, 3] are all excellent resources on m
for ODEs. We also demonstrate the Odespy Python interface to a range of
or general first-order ODE systems.

eneric form

commonly written in the generic form

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

here $f(u, t)$ is some prescribed function. As an example, our most general ex
odel (69) has $f(u, t) = -a(t)u(t) + b(t)$.

The unknown u in (89) may either be a scalar function of time t, or a vector
f t in case of a *system of ODEs* with m unknown components:

$$u(t) = (u^{(0)}(t), u^{(1)}(t), \ldots, u^{(m-1)}(t))\,.$$

n that case, the right-hand side is vector-valued function with m components,

$$\begin{aligned} f(u, t) = (&f^{(0)}(u^{(0)}(t), \ldots, u^{(m-1)}(t)),\\ &f^{(1)}(u^{(0)}(t), \ldots, u^{(m-1)}(t)),\\ &\vdots,\\ &f^{(m-1)}(u^{(0)}(t), \ldots, u^{(m-1)}(t)))\,. \end{aligned}$$

Actually, any system of ODEs can be written in the form (89), but higher-o
eed auxiliary unknown functions to enable conversion to a first-order system.

Next we list some well-known methods for $u' = f(u, t)$, valid both for a single
nd systems of ODEs (vector u). The choice of methods is inspired by the kind
re popular also for partial differential equations.

.2 The θ-rule

he θ-rule scheme applied to $u' = f(u, t)$ becomes

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

ringing the unknown u^{n+1} to the left-hand side and the known terms on the
ives

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

or a general f (not linear in u), this equation is *nonlinear* in the unknown u^{n}
or a scalar ODE ($m = 1$), we have to solve a single nonlinear algebraic equatio
or a system of ODEs, we get a system of coupled, nonlinear algebraic equat
ethod is a popular solution approach in both cases. Note that with the Forwa
$= 0$) we do not have to deal with nonlinear equations, because in that case we
pdating formula for u^{n+1}. This is known as an *explicit* scheme. With $\theta \neq 1$
ystems of algebraic equations, and the scheme is said to be *implicit*.

.3 An implicit 2-step backward scheme

he implicit backward method with 2 steps applies a three-level backward differ
ation 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 schem
eads

$$u^{n+1} = \frac{4}{3}u^n - \frac{1}{3}u^{n-1} + \frac{2}{3}\Delta t f(u^{n+1}, t_{n+1})\,.$$

der versions of the scheme (92) can be constructed by including more time levels
re known as the Backward Differentiation Formulas (BDF), and the particular
en referred to as BDF2.
hat the scheme (92) is implicit and requires solution of nonlinear equations w
in u. The standard 1st-order Backward Euler method or the Crank-Nicolson
ed for the first step.

eapfrog schemes

inary Leapfrog scheme. The derivative of u at some point t_n can be appro
cal difference over two time steps,

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

n approximation of second order in Δt. The scheme can then be written as

$$[D_{2t}u = f(u,t)]^n,$$

or notation. Solving for u^{n+1} gives

$$u^{n+1} = u^{n-1} + \Delta t f(u^n, t_n)\,.$$

hat (94) is an explicit scheme, and that a nonlinear f (in u) is trivial to handle
lves the known u^n value. Some other scheme must be used as starter to com
the Forward Euler scheme since it is also explicit.

red Leapfrog scheme. Unfortunately, the Leapfrog scheme (94) will develop
s with time (see Problem 8)[[[. A remedy for such undesired oscillations is to in
technique. First, a standard Leapfrog step is taken, according to (94), and t
u^n value is adjusted according to

$$u^n \;\leftarrow\; u^n + \gamma(u^{n-1} - 2u^n + u^{n+1})\,.$$

ns will effectively damp oscillations in the solution, especially those with short wa
-to-point oscillations). A common choice of γ is 0.6 (a value used in the famous
Model).

he 2nd-order Runge-Kutta scheme

tep scheme

$$u^* = u^n + \Delta t f(u^n, t_n),$$
$$u^{n+1} = u^n + \Delta t \frac{1}{2}\left(f(u^n, t_n) + f(u^*, t_{n+1})\right),$$

applies a Crank-Nicolson method (97) to the ODE, but replaces the term $f(u^{n+}$
iction $f(u^*, t_{n+1})$ based on a Forward Euler step (96). The scheme (96)-(97) is
method, but is also a 2nd-order Runge-Kutta method. The scheme is explicit,
pected to behave as Δt^2.

.6 A 2nd-order Taylor-series method

ne way to compute u^{n+1} given u^n is to use a Taylor polynomial. We may write
f 2nd degree:

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

rom the equation $u' = f(u,t)$ it follows that the derivatives of u can be express
nd its derivatives:

$$\begin{aligned} u'(t_n) &= f(u^n, t_n), \\ u''(t_n) &= \frac{\partial f}{\partial u}(u^n, t_n)u'(t_n) + \frac{\partial f}{\partial t} \\ &= f(u^n, t_n)\frac{\partial f}{\partial u}(u^n, t_n) + \frac{\partial f}{\partial t}, \end{aligned}$$

esulting 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 to obtain m
rder than 2.

.7 The 2nd- and 3rd-order Adams-Bashforth schemes

he following method is known as the 2nd-order Adams-Bashforth scheme:

$$u^{n+1} = u^n + \frac{1}{2}\Delta t\left(3f(u^n, t_n) - f(u^{n-1}, t_{n-1})\right)\,.$$

he scheme is explicit and requires another one-step scheme to compute u^1 (th
cheme or Heun's method, for instance). As the name implies, the scheme is of

Another explicit scheme, involving four time levels, is the 3rd-order Adams-B

$$u^{n+1} = u^n + \frac{1}{12}\left(23f(u^n, t_n) - 16f(u^{n-1}, t_{n-1}) + 5f(u^{n-2}, t_{n-2})\right)$$

he numerical error is of order Δt^3, and the scheme needs some method for comp

More general, higher-order Adams-Bashforth schemes (also called *explicit*
ompute u^{n+1} as a linear combination of f at k previous time steps:

$$u^{n+1} = u^n + \sum_{j=0}^{k}\beta_j f(u^{n-j}, t_{n-j}),$$

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-Kut
alled RK4. Its derivation is a nice illustration of common numerical approximati
t 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}:

$$u(t_{n+1}) - u(t_n) = \int_{t_n}^{t_{n+1}} f(u(t), t)dt\,.$$

:o compute $u(t_{n+1})$ and regard $u(t_n)$ as known. The task is to find good approxi
tegral, since the integrand involves the unknown u between t_n and t_{n+1}.
ıtegral can be approximated by the famous Simpson's rule[21]:

$$\int_{t_n}^{t_{n+1}} f(u(t), t)dt \approx \frac{\Delta t}{6}\left(f^n + 4f^{n+\frac{1}{2}} + f^{n+1}\right)\,.$$

lem now is that we do not know $f^{n+\frac{1}{2}} = f(u^{n+\frac{1}{2}}, t_{n+1/2})$ and $f^{n+1} = (u^{n+1}, t$
only u^n and hence f^n. The idea is to use various approximations for $f^{n+\frac{1}{2}}$ a
using well-known schemes for the ODE in the intervals $[t_n, t_{n+1/2}]$ and $[t_n, t_n$
ntegral approximation into four terms:

$$\int_{t_n}^{t_{n+1}} f(u(t), t)dt \approx \frac{\Delta t}{6}\left(f^n + 2\hat{f}^{n+\frac{1}{2}} + 2\tilde{f}^{n+\frac{1}{2}} + \bar{f}^{n+1}\right),$$

$^{-\frac{1}{2}}$, $\tilde{f}^{n+\frac{1}{2}}$, and $\bar{f}^{n+1}$ are approximations to $f^{n+\frac{1}{2}}$ and f^{n+1} that can be based on
quantities. For $\hat{f}^{n+\frac{1}{2}}$ we can apply an approximation to $u^{n+\frac{1}{2}}$ using the Forwa
ith step $\frac{1}{2}\Delta t$:

$$\hat{f}^{n+\frac{1}{2}} = f(u^n + \frac{1}{2}\Delta t f^n, t_{n+1/2})$$

; gives us a prediction of $f^{n+\frac{1}{2}}$, we can for $\tilde{f}^{n+\frac{1}{2}}$ try a Backward Euler me
ate $u^{n+\frac{1}{2}}$:

$$\tilde{f}^{n+\frac{1}{2}} = f(u^n + \frac{1}{2}\Delta t \hat{f}^{n+\frac{1}{2}}, t_{n+1/2})\,.$$

$^{\frac{1}{2}}$ as a hopefully good approximation to $f^{n+\frac{1}{2}}$, we can for the final term $\bar{f}^{n-}$
colson method to approximate u^{n+1}:

$$\bar{f}^{n+1} = f(u^n + \Delta t \hat{f}^{n+\frac{1}{2}}, t_{n+1})\,.$$

now used the Forward and Backward Euler methods as well as the Crank-N
ı the context of Simpson's rule. The hope is that the combination of these r
overall time-stepping scheme from t_n to t_n+1 that is much more accurate t
d $\mathcal{O}(\Delta t^2)$ of the individual steps. This is indeed true: the overall accuracy is (
nmarize, the 4th-order Runge-Kutta method becomes

$$u^{n+1} = u^n + \frac{\Delta t}{6}\left(f^n + 2\hat{f}^{n+\frac{1}{2}} + 2\tilde{f}^{n+\frac{1}{2}} + \bar{f}^{n+1}\right),$$

quantities on the right-hand side are computed from (101)-(103). Note that the
plicit so there is never any need to solve linear or nonlinear algebraic equations. I
ity is conditional and depends on f. There is a whole range of *implicit* Rung

/en.wikipedia.org/wiki/Simpson's_rule

ıethods that are unconditionally stable, but require solution of algebraic equat
t each time step.

The simplest way to explore more sophisticated methods for ODEs is to a
ıany high-quality software packages that exist, as the next section explains.

.9 The Odespy software

. wide range of the methods and software exist for solving (89). Many of metho
ırough a unified Python interface offered by the Odespy[22] package. Odespy
ython implementations of the most fundamental schemes as well as Python inte
ımous packages for solving ODEs: ODEPACK[23], Vode[24], rkc.f[25], rkf45.f[26], Rad
ıe ODE solvers in SciPy[28], SymPy[29], and odelab[30].

The usage of Odespy follows this setup for the ODE $u' = -au$, $u(0) = I$,
ɔlved by the famous 4th-order Runge-Kutta method, using $\Delta t = 1$ and $N_t = 6$

```
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`, `CrankNicols`
- the 2nd- and 4th-order Runge-Kutta methods: `RK2` and `RK4`
- The BDF methods and the Adam-Bashforth methods: `Vode`, `Lsode`, `Lsod`
- The Leapfrog scheme: `Leapfrog` and `LeapfrogFiltered`

.10 Example: Runge-Kutta methods

ince all solvers have the same interface in Odespy, modulo different set of pa
ɔlvers' constructors, one can easily make a list of solver objects and run a loop f
ɔt of) solvers. The code below, found in complete form in `decay_odespy.py`[3
ımous Runge-Kutta methods of orders 2, 3, and 4 with the exact solution of the
$' = -au$. Since we have quite long time steps, we have included the only relevan
me steps, the Backward Euler scheme ($\theta = 1$), as well. Figure 15 shows the re

[22] `https://github.com/hplgit/odespy`
[23] `https://computation.llnl.gov/casc/odepack/odepack_home.html`
[24] `https://computation.llnl.gov/casc/odepack/odepack_home.html`
[25] `http://www.netlib.org/ode/rkc.f`
[26] `http://www.netlib.org/ode/rkf45.f`
[27] `http://www.unige.ch/ hairer/software.html`
[28] `http://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.ode.html`
[29] `http://docs.sympy.org/dev/modules/mpmath/calculus/odes.html`
[30] `http://olivierverdier.github.com/odelab/`
[31] `http://tinyurl.com/jvzzcfn/decay/decay_odespy.py`

```
umpy as np
citools.std as plt
ys

 t):
rn -a*u

 = 2; T = 6
at(sys.argv[1]) if len(sys.argv) >= 2 else 0.75
(round(T/dt))
inspace(0, Nt*dt, Nt+1)

= [odespy.RK2(f),
   odespy.RK3(f),
   odespy.RK4(f),
   odespy.BackwardEuler(f, nonlinear_solver='Newton')]

= []
er in solvers:
er.set_initial_condition(I)
 = solver.solve(t)

plot(t, u)
hold('on')
nds.append(solver.__class__.__name__)

e with exact solution plotted on a very fine mesh
 np.linspace(0, T, 10001)
np.exp(-a*t_fine)
(t_fine, u_e, '-') # avoid markers by specifying line type
append('exact')

nd(legends)
e('Time step: %g' % dt)
()
```

ization tip.

SciTools for plotting here, but importing `matplotlib.pyplot` as `plt` instead
However, plain use of Matplotlib as done here results in curves with different c
may be hard to distinguish on black-and-white paper. Using SciTools, curve
atically given colors *and* markers, thus making curves easy to distinguish on s
olors and on black-and-white paper. The automatic adding of markers is norma
ea for a very fine mesh since all the markers get cluttered, but SciTools limit
r of markers in such cases. For the exact solution we use a very fine mesh, b
le above we specify the line type as a solid line (`-`), which means no markers
olor to be automatically determined by the backend used for plotting (Matplotl
, but SciTools gives the opportunity to use other backends to produce the plot
t or Grace).
o note the that the legends are based on the class names of the solvers,
hon the name of a the class type (as a string) of an object `obj` is obtaine
`.class__.__name__`.

ms in Figure 15 and other experiments reveal that the 2nd-order Runge-Kutta
nstable for $\Delta t > 1$ and decays slower than the Backward Euler scheme for la
Δt (see Exercise 7 for an analysis). However, for fine $\Delta t = 0.25$ the 2nd-orde

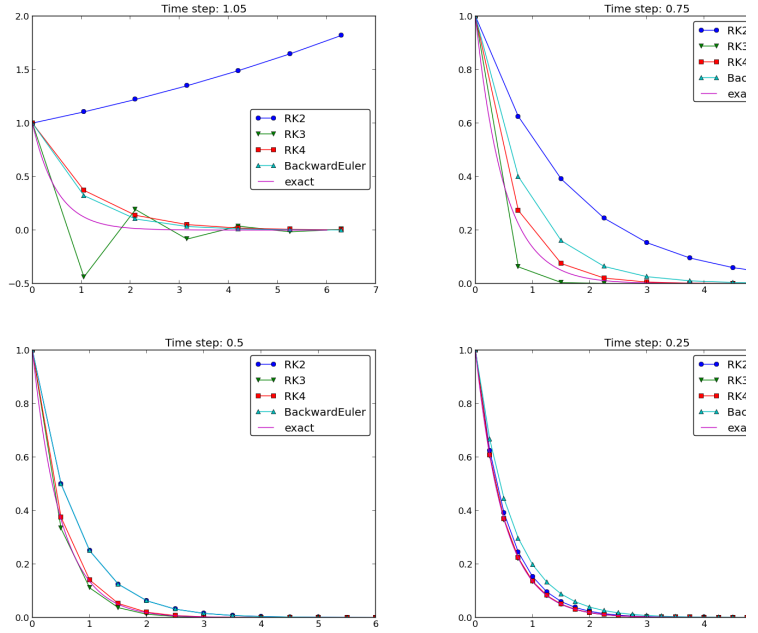


Figure 15: Behavior of different schemes for the decay equation.

utta method approaches the exact solution faster than the Backward Euler sche
tter scheme does a better job for larger Δt, while the higher order scheme is sup
t. This is a typical trend also for most schemes for ordinary and partial differ

The 3rd-order Runge-Kutta method (`RK3`) has also artifacts in form of oscillat
e larger Δt values, much like that of the Crank-Nicolson scheme. For finer Δ
unge-Kutta method converges quickly to the exact solution.

The 4th-order Runge-Kutta method (`RK4`) is slightly inferior to the Backwa
n the coarsest mesh, but is then clearly superior to all the other schemes. It
ethod of choice for all the tested schemes.

emark about using the θ-rule in Odespy. The Odespy package assume
written as $u' = f(u,t)$ with an f that is possibly nonlinear in u. The θ-rule
ads to

$$u^{n+1} = u^n + \Delta t\left(\theta f(u^{n+1}, t_{n+1}) + (1-\theta)f(u^n, t_n)\right),$$

hich is a *nonlinear equation* in u^{n+1}. Odespy's implementation of the θ-rule (
e specialized Backward Euler (`BackwardEuler`) and Crank-Nicolson (`CrankNi`
ust invoke iterative methods for solving the nonlinear equation in u^{n+1}. This is
is linear in u, as in the model problem $u' = -au$, where we can easily solve fo
herefore, we need to specify use of Newton's method to the equations. (Odes
ethods than Newton's to be used, for instance Picard iteration, but that metho
he reason is that it applies the Forward Euler scheme to generate a start value f

Euler may give very wrong solutions for large Δt values. Newton's method, on t
nsensitive to the start value in *linear problems*.)

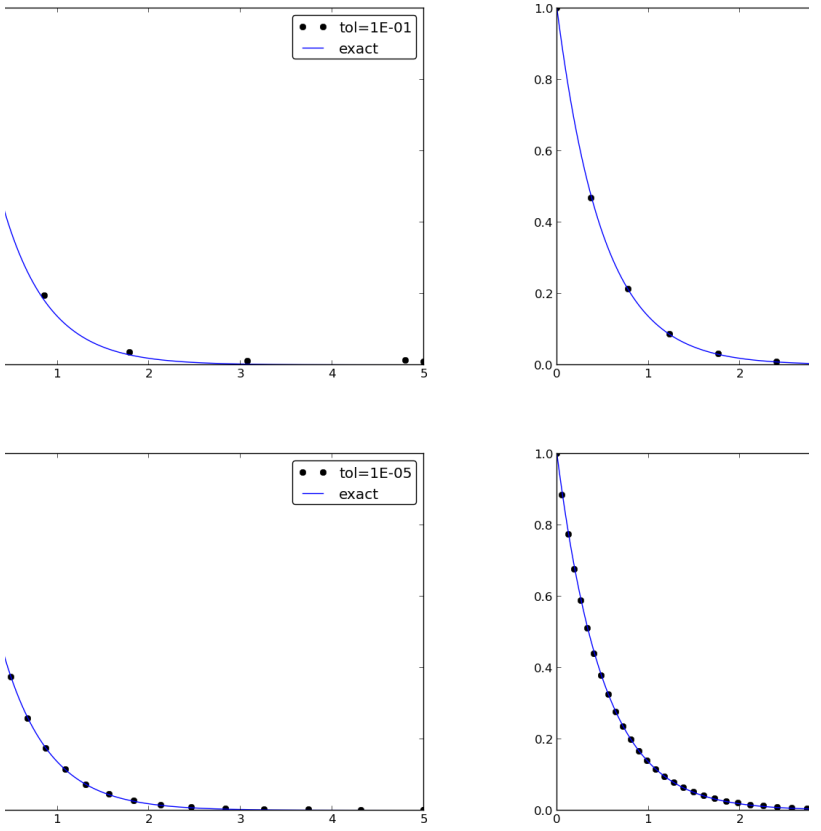
Example: Adaptive Runge-Kutta methods

ffers solution methods that can adapt the size of Δt with time to match a
n the solution. Intuitively, small time steps will be chosen in areas where the so
rapidly, while larger time steps can be used where the solution is slowly varyin
ror estimator is used to adjust the next time step at each time level.
y popular adaptive method for solving ODEs is the Dormand-Prince Rung
f order 4 and 5. The 5th-order method is used as a reference solution and the d
he 4th- and 5th-order methods is used as an indicator of the error in the nu
The Dormand-Prince method is the default choice in MATLAB's widely use

n easily set up Odespy to use the Dormand-Prince method and see how it sel
ime steps. To this end, we request only one time step from $t = 0$ to $t = T$ and
o compute the necessary non-uniform time mesh to meet a certain error toleran
like

```
odespy
numpy as np
decay_mod
sys
 matplotlib.pyplot as plt
scitools.std as plt

, t):
urn -a*u

ct_solution(t):
urn I*np.exp(-a*t)

a = 2; T = 5
loat(sys.argv[1])
= odespy.DormandPrince(f, atol=tol, rtol=0.1*tol)

 # just one step - let the scheme find its intermediate points
= np.linspace(0, T, Nt+1)
= np.linspace(0, T, 10001)

set_initial_condition(I)
solver.solve(t_mesh)

 t will only consist of [I, u^Nt] and [0,T]
r.u_all and solver.t_all contains all computed points
t(solver.t_all, solver.u_all, 'ko')
d('on')
t(t_fine, exact_solution(t_fine), 'b-')
end(['tol=%.0E' % tol, 'exact'])
efig('tmp_odespy_adaptive.png')
w()
```

ng four cases with tolerances 10^{-1}, 10^{-3}, 10^{-5}, and 10^{-7}, gives the results in F
y, one would expect denser points in the beginning of the decay and larger tin
solution flattens out.



igure 16: Choice of adaptive time mesh by the Dormand-Prince method for dif

Exercises

Exercise 3: Experiment with precision in tests and the size

is claimed in Section 5.5 that most numerical methods will reproduce a linear
achine precision. Test this assertion using the nose test function `test_linear_`
`ecay_vc.py`[32] program. Vary the parameter `c` from very small, via `c=1` to many l
rint out the maximum difference between the numerical solution and the exact
the relevant value of the `places` (or `delta`) argument to `nose.tools.assert`
n each case? Filename: `test_precision.py`.

[32] `http://tinyurl.com/jvzzcfn/decay/decay_vc.py`

e 4: Implement the 2-step backward scheme

ıt the 2-step backward method (92) for the model $u'(t) = -a(t)u(t) + b(t)$, u
first step to be computed by either the Backward Euler scheme or the Crank-I
Verify the implementation by choosing $a(t)$ and $b(t)$ such that the exact solution
Section 5.5). Show mathematically that a linear solution is indeed a solutioı
quations.
ute convergence rates (see Section **??**) in a test case $a = \text{const}$ and $b = 0$, w
e an exact solution, and determine if the choice of a first-order scheme (Backwar
st step has any impact on the overall accuracy of this scheme. The expected er
^{2}). Filename: `decay_backward2step.py`.

e 5: Implement the 2nd-order Adams-Bashforth scheme

t the 2nd-order Adams-Bashforth method (99) for the decay problem $u' = -a(t)$
$t \in (0, T]$. Use the Forward Euler method for the first step such that the overall
. Verify the implementation using an exact solution that is linear in time. Ana
searching for solutions $u^n = A^n$ when $a = \text{const}$ and $b = 0$. Compare this secoı
o the Crank-Nicolson scheme. Filename: `decay_AdamsBashforth2.py`.

e 6: Implement the 3rd-order Adams-Bashforth scheme

t the 3rd-order Adams-Bashforth method (100) for the decay problem $u' = -a(t)$
$t \in (0, T]$. Since the scheme is explicit, allow it to be started by two steps v
Euler method. Investigate experimentally the case where $b = 0$ and a is a consta
scillatory solutions for large Δt? Filename: `decay_AdamsBashforth3.py`.

e 7: Analyze explicit 2nd-order methods

t the schemes (97) and (98) are identical in the case $f(u, t) = -a$, where a
Assume that the numerical solution reads $u^n = A^n$ for some unknown ampli
to be determined. Find A and derive stability criteria. Can the scheme
solutions of $u' = -au$? Plot the numerical and exact amplification factor. F
`2_Taylor2.py`.

n 8: Implement and investigate the Leapfrog scheme

ıg scheme for the ODE $u'(t) = -a(t)u(t) + b(t)$ is defined by

$$[D_{2t}u = -au + b]^n .$$

e method is needed to compute u^1. The Forward Euler scheme is a possible ca

ıent the Leapfrog scheme for the model equation. Plot the solution in the cas
$= 1$, $\Delta t = 0.01$, $t \in [0, 4]$. Compare with the exact solution $u_{\text{e}}(t) = e^{-t}$.

mathematically that a linear solution in t fulfills the Forward Euler scheme for
the Leapfrog scheme for the subsequent steps. Use this linear solution to ve
tation, and automate the verification through a nose test.

Hint. It can be wise to automate the calculations such that it is easy to redo
ɔr other types of solutions. Here is a possible `sympy` function that takes a sym
(implemented as a Python function of `t`), fits the `b` term, and checks if `u` ful
quations:

```
import sympy as sp

def analyze(u):
    t, dt, a = sp.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 = sp.diff(u(t), t) + a*u(t)
    b = sp.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 = sp.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 = sp.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)
```

) Show that a second-order polynomial in t cannot be a solution of the dis
owever, if a Crank-Nicolson scheme is used for the first step, a second-order p
ıe equations exactly.

) Create a manufactured solution $u(t) = \sin(t)$ for the ODE $u' = -au + b$
ɔnvergence rate of the Leapfrog scheme using this manufactured solution.
ɔnvergence rate of the Leapfrog scheme is $\mathcal{O}(\Delta t^2)$. Does the use of a 1st-orde
rst step impact the convergence rate?

) Set up a set of experiments to demonstrate that the Leapfrog scheme (94) is
umerical artifacts (instabilities). Document the main results from this investig

) Analyze and explain the instabilities of the Leapfrog scheme (94):

1. Choose $a = \text{const}$ and $b = 0$. Assume that an exact solution of the discre
   the form $u^n = A^n$, where A is an amplification factor to be determined. De
   for A by inserting $u^n = A^n$ in the Leapfrog scheme.
2. Compute A either by hand and/or with the aid of `sympy`. The polynomi
   roots, A_1 and A_2. Let u^n be a linear combination $u^n = C_1 A_1^n + C_2 A_2^n$.
3. Show that one of the roots is the explanation of the instability.
4. Compare A with the exact expression, using a Taylor series approximatioı
5. How can C_1 and C_2 be determined?

the original Leapfrog scheme is unconditionally unstable as time grows, it d
ɔilization. This can be done by filtering, where we first find u^{n+1} from the
scheme and then replace u^n by $u^n + \gamma(u^{n-1} - 2u^n + u^{n+1})$, where γ can be t
ement the filtered Leapfrog scheme and check that it can handle tests where the
scheme is unstable.
s: `decay_leapfrog.py`, `decay_leapfrog.pdf`.

n 9: Make a unified implementation of many schemes

the linear ODE problem $u'(t) = -a(t)u(t) + b(t)$, $u(0) = I$. Explicit schemes
an be written in the general form

$$u^{n+1} = \sum_{j=0}^{m} c_j u^{n-j},$$

choice of $c_0, \ldots, c_m$. Find expressions for the c_j coefficients in case of the θ-r
l backward scheme, the Leapfrog scheme, the 2nd-order Runge-Kutta method,
Adams-Bashforth scheme.
a class `ExpDecay` that implements the general updating formula (105). The
applied for $n < m$, and for those n values, other schemes must be used. Ass
that we just repeat Crank-Nicolson steps until (105) can be used. Use a sub
e list $c_0, \ldots, c_m$ for a particular method, and implement subclasses for all the me
Verify the implementation by testing with a linear solution, which should be
d by all methods. Filename: `decay_schemes_oo.py`.

ɔplications of exponential decay models

on presents many mathematical models that all end up with ODEs of the ty
The applications are taken from biology, finance, and physics, and cover poɪ
decay, compound interest and inflation, radioactive decay, cooling of objects, con
cal media, pressure variations in the atmosphere, and air resistance on falling

caling

ications of a model $u' = -au + b$ will often involve a lot of parameters in the exp
b. It can be quite a challenge to find relevant values of all parameters. In
however, it turns out that it is not always necessary to estimate all parameters
mp them into one or a few *dimensionless* numbers by using a very attractive te
ling. It simply means to stretch the u and t axis is the present problem - and s
eters in the problem are lumped one parameter if $b \neq 0$ and no parameter whe
g means that we introduce a new function $\bar{u}(\bar{t})$, with

$$\bar{u} = \frac{u - u_m}{u_c}, \quad \bar{t} = \frac{t}{t_c},$$

is a characteristic value of u, u_c is a characteristic size of the range of u values
cteristic size of the range of t_c where u varies significantly. Choosing u_m, u_c, a
s easy and often an art in complicated problems. We just state one choice firs

$$u_c = I, \quad u_m = b/a, \quad t_c = 1/a\,.$$

ıserting $u = u_m + u_c\bar{u}$ and $t = t_c\bar{t}$ in the problem $u' = -au + b$, assuming a and
esults 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

$$\beta = \frac{b}{Ia}\,.$$

hat is, only the special combination of $b/(Ia)$ matters, not what the individu
nd I are. Moreover, if $b = 0$, the scaled problem is independent of a and I!
ıeans that we can perform one numerical simulation of the scaled problem
ɔlution of any problem for a given a and I by stretching the axis in the plot: $u =$
or $b \neq 0$, we simulate the scaled problem for a few β values and recover the ph
y translating and stretching the u axis and stretching the t axis.

The scaling breaks down if $I = 0$. In that case we may choose $u_m = 0$, $u_c = b$,
esulting in a slightly different scaled problem:

$$\frac{d\bar{u}}{d\bar{t}} = 1 - \bar{u}, \quad \bar{u}(0) = 0\,.$$

s with $b = 0$, the case $I = 0$ has a scaled problem with no physical parameters

It is common to drop the bars after scaling and write the scaled proble
$(0) = 1 - \beta$, or $u' = 1 - u$, $u(0) = 0$. Any implementation of the problem $u' = -$
an be reused for the scaled problem by setting $a = 1$, $b = 0$, and $I = 1 - \beta$ in tl
r one sets $a = 1$, $b = 1$, and $I = 0$ when the physical I is zero. Falling boc
escribed in Section 8.8, involves $u' = -au + b$ with seven physical parameters.
ı the scaled version of the problem if we start the motion from rest!

.2 Evolution of a population

et N be the number of individuals in a population occupying some spatial d
I being an integer in this problem, we shall compute with N as a real numbe
s a continuous function of time. The basic model assumption is that in a time
umber of newcomers to the populations (newborns) is proportional to N, with
ɔnstant $\bar{b}$. The amount of newcomers will increase the population and result ir

$$N(t + \Delta t) = N(t) + \bar{b}N(t)\,.$$

; is obvious that a long time interval Δt will result in more newcomers and l
herefore, we introduce $b = \bar{b}/\Delta t$: the number of newcomers per unit time an
Ve must then multiply b by the length of the time interval considered and by the
ɔ get the total number of new individuals, $b\Delta tN$.

If the number of removals from the population (deaths) is also proportic
roportionality constant $d\Delta t$, the population evolves according to

$$N(t + \Delta t) = N(t) + b\Delta tN(t) - d\Delta tN(t)\,.$$

ɔividing by Δt and letting $\Delta t \to 0$, we get the ODE

$$N' = (b-d)N, \quad N(0) = N_0\,.$$

ılation where the death rate (d) is larger than then newborn rate (b), $a > 0$,
n experiences exponential decay rather than exponential growth.
ıe populations there is an immigration of individuals into the spatial domain.
ls coming in per time unit, the equation for the population change becomes

$$N(t+\Delta t) = N(t) + b\Delta t N(t) - d\Delta t N(t) + \Delta t I\,.$$

sponding ODE reads

$$N' = (b-d)N + I, \quad N(0) = N_0\,.$$

simplification arises if we introduce a fractional measure of the population: u =
$= b - d$. The ODE problem now becomes

$$u' = ru + f, \quad u(0) = 1,$$

$= I/N_0$ measures the net immigration per time unit as the fraction of th
n. Very often, r is approximately constant, but f is usually a function of time
owth rate r of a population decreases if the environment has limited resources.
onment can sustain at most $N_{\max}$ individuals. We may then assume that the
oaches zero as N approaches $N_{\max}$, i.e., as u approaches $M = N_{\max}/N_0$. The
volution of r is then a linear function: $r(t) = r_0(1 - u(t)/M)$, where r_0 is th
ıte when the population is small relative to the maximum size and there is
Using this $r(t)$ in (108) results in the *logistic model* for the evolution of a po
for the moment that $f = 0$):

$$u' = r_0(1 - u/M)u, \quad u(0) = 1\,.$$

ı will grow at rate r_0, but the growth will decay as u approaches M, and then th
ıge in u, causing $u \to M$ as $t \to \infty$. Note that the logistic equation $u' = r_0(1 -$
ar because of the quadratic term $-u^2 r_0/M$.

ompound interest and inflation

ınnual interest rate is r percent and that the bank adds the interest once a year
ıt. If u^n is the investment in year n, the investment in year u^{n+1} grows to

$$u^{n+1} = u^n + \frac{r}{100}u^n\,.$$

the interest rate is added every day. We therefore introduce a parameter m
f periods per year when the interest is added. If n counts the periods, we h
tal model for compound interest:

$$u^{n+1} = u^n + \frac{r}{100m}u^n\,.$$

el is a *difference equation*, but it can be transformed to a continuous differential e
ı limit process. The first step is to derive a formula for the growth of the inv

ver a time t. Starting with an investment u^0, and assuming that r is constant

$$\begin{aligned} u^{n+1} &= \left(1 + \frac{r}{100m}\right) u^n \\ &= \left(1 + \frac{r}{100m}\right)^2 u^{n-1} \\ &\vdots \\ &= \left(1 + \frac{r}{100m}\right)^{n+1} u^0 \end{aligned}$$

ıtroducing time t, which here is a real-numbered counter for years, we have tha
ın write

$$u^{mt} = \left(1 + \frac{r}{100m}\right)^{mt} u^0\,.$$

he second step is to assume *continuous compounding*, meaning that the in
ontinuously. This implies $m \to \infty$, and in the limit one gets the formula

$$u(t) = u_0 e^{rt/100},$$

hich is nothing but the solution of the ODE problem

$$u' = \frac{r}{100}u, \quad u(0) = u_0\,.$$

his is then taken as the ODE model for compound interest if $r > 0$. Howeve
pplies equally well to inflation, which is just the case $r < 0$. One may also take
ıe net growth of an investemt, where r takes both compound interest and inflat
ote that for real applications we must use a time-dependent r in (112).

Introducing $a = \frac{r}{100}$, continuous inflation of an initial fortune I is then a pr
xponential decay according to

$$u' = -au, \quad u(0) = I\,.$$

.4 Radioactive Decay

ın atomic nucleus of an unstable atom may lose energy by emitting ionizing parti
e transformed to a nucleus with a different number of protons and neutrons.
nown as radioactive decay[33]. Actually, the process is stochastic when viewed f
ecause it is impossible to predict exactly when a particular atom emits a partic
ith a large number of atoms, N, one may view the process as deterministic a
ıean behavior of the decay. Below we reason intuitively about an ODE for the
hereafter, we show mathematically that a detailed stochastic model for single
ıme mean behavior.

eterministic model. Suppose at time t, the number of the original atom
asic model assumption is that the transformation of the atoms of the original
me interval Δt is proportional to N, so that

$$N(t+\Delta t) = N(t) - a\Delta t N(t),$$

[33] http://en.wikipedia.org/wiki/Radioactive_decay

· 0 is a constant. Introducing $u = N(t)/N(0)$, dividing by Δt and letting $\Delta t \to$
ing ODE:

$$u' = -au, \quad u(0) = 1\,.$$

neter a can for a given nucleus be expressed through the *half-life* $t_{1/2}$, which is t
the decay to reduce the initial amount by one half, i.e., $u(t_{1/2}) = 0.5$. With $u(t)$
$_2 = a^{-1}\ln 2$ or $a = \ln 2/t_{1/2}$.

ic model. We have originally N_0 atoms. Each atom may have decayed or sur
ar time t. We want to count how many original atoms that are left, i.e., how man
survived. The survival of a single atom at time t is a random event. Since there
mes, survival or decay, we have a Bernoulli trial[34]. Let p be the probability of
that the probability of decay is $1-p$). If each atom survives independently of th
robability of survival is the same for every atom, we have N_0 statistically B
own as a *binomial experiment* from probability theory. The probability $P(N)$
N_0 atoms have survived at time t is then given by the famous *binomial distri*

$$P(N) = \frac{N_0!}{N!(N_0-N)!}p^N(1-p)^{N_0-N}\,.$$

(or expected) value $\mathrm{E}[P]$ of $P(N)$ is known to be N_0p.
ains to estimate p. Let the interval $[0,t]$ be divided into m small subintervals o
nake the assumption that the probability of decay of a single atom in an int
is $\tilde{p}$, and that this probability is proportional to Δt: $\tilde{p} = \lambda\Delta t$ (it sounds natu
bility of decay increases with Δt). The corresponding probability of survival is
that λ is independent of time, we have, for each interval of length Δt, a Bernou
either survives or decays in that interval. Now, p should be the probability t
vives in all the intervals, i.e., that we have m successful Bernoulli trials in a r

$$p = (1-\lambda\Delta t)^m\,.$$

cted number of atoms of the original type at time t is

$$\mathrm{E}[P] = N_0p = N_0(1-\lambda\Delta t)^m, \quad m = t/\Delta t\,.$$

the relation between the two types of Bernoulli trials and the ODE above, we g
$\to t$, $m \to \infty$. One can show that

$$p = \lim_{m\to\infty}(1-\lambda\Delta t)^m = \lim_{m\to\infty}\left(1-\lambda\frac{t}{m}\right)^m = e^{-\lambda t}$$

e famous exponential waiting time (or arrival time) distribution for a Poisson pr
y theory (obtained here, as often done, as the limit of a binomial experimen
y of decay, $1-e^{-\lambda t}$, follows an exponential distribution[35]. The limit means tl
, hence Δt is very small, and $\tilde{p} = \lambda\Delta t$ is very small since the intensity of the
med finite. This situation corresponds to a very small probability that an at
very short time interval, which is a reasonable model. The same model occur
t applications, e.g., when waiting for a taxi, or when finding defects along a r

/en.wikipedia.org/wiki/Bernoulli_trial
/en.wikipedia.org/wiki/Exponential_distribution

telation between stochastic and deterministic models. With $p = e$
xpected number of original atoms at t as $N_0p = N_0e^{-\lambda t}$, which is exactly the
DE model $N' = -\lambda N$. This gives also an interpretation of a via λ or vice versa
nding here is that the ODE model captures the mean behavior of the under
odel. This is, however, not always the common relation between microscopic st
nd macroscopic "averaged" models.

Also of interest is to see that a Forward Euler discretization of $N' = -\lambda N$, N
$^m = N_0(1-\lambda\Delta t)^m$ at time $t_m = m\Delta t$, which is exactly the expected value o
xperiment with N_0 atoms and m small intervals of length Δt, where each atom
robability $\lambda\Delta t$ in an interval.

A fundamental question is how accurate the ODE model is. The underlying s
uctuates around its expected value. A measure of the fluctuations is the sta
f the binomial experiment with N_0 atoms, which can be shown to be $\mathrm{Std}[P]$
ompared to the size of the expectation, we get the normalized standard deviat

$$\frac{\sqrt{\mathrm{Var}[P]}}{\mathrm{E}[P]} = N_0^{-1/2}\sqrt{p^{-1}-1} = N_0^{-1/2}\sqrt{(1-e^{-\lambda t})^{-1}-1} \approx (N_0\lambda t)^{-}$$

owing that the normalized fluctuations are very small if N_0 is very large, whi
ase.

.5 Newton's law of cooling

hen a body at some temperature is placed in a cooling environment, experienc
emperature falls rapidly in the beginning, and then the changes in temperatur
e body's temperature equals that of the surroundings. Newton carried out some
ooling hot iron and found that the temperature evolved as a "geometric progres
rithmetic progression", meaning that the temperature decayed exponentially. I
as formulated as a differential equation: the rate of change of the temperatu
roportional to the temperature difference between the body and its surroundings
known as *Newton's law of cooling*, which can be mathematically expressed as

$$\frac{dT}{dt} = -k(T-T_s),$$

here T is the temperature of the body, T_s is the temperature of the surroun
nd k is a positive constant. Equation (133) is primarily viewed as an empirical
eat is efficiently convected away from the surface of the body by a flowing flu
onstant temperature T_s. The *heat transfer coefficient* k reflects the transfer c
ody to the surroundings and must be determined from physical experiments.

We must obviously have an initial condition $T(0) = T_0$ in addition to the co

.6 Decay of atmospheric pressure with altitude

ertical equilibrium of air in the atmosphere is governed by the equation

$$\frac{dp}{dz} = -\varrho g\,.$$

ere, $p(z)$ is the air pressure, ϱ is the density of air, and $g = 9.807\ \mathrm{m/s^2}$ is a
f the acceleration of gravity. (Equation (116) follows directly from the genera
quations for fluid motion, with the assumption that the air does not move.)

ressure is related to density and temperature through the ideal gas law

$$\varrho = \frac{Mp}{R^*T},$$

is the molar mass of the Earth's air (0.029 kg/mol), R^* is the universal gas c
ı/(mol K)), and T is the temperature. All variables p, ϱ, and T vary with the l
(117) in (116) results in an ODE with a variable coefficient:

$$\frac{dp}{dz} = -\frac{Mg}{R^*T(z)}p\,.$$

atmospheric layers. The atmosphere can be approximately modeled by seve
yer, (118) is applied with a linear temperature of the form

$$T(z) = \bar{T}_i + L_i(z - h_i),$$

h_i denotes the bottom of layer number i, having temperature $\bar{T}_i$, and L_i is a c
ımber i. The table below lists h_i (m), $\bar{T}_i$ (K), and L_i (K/m) for the layers $i =$

h_i	$\bar{T}_i$	L_i
0	288	-0.0065
11,000	216	0.0
20,000	216	0.001
32,000	228	0.0028
47,000	270	0.0
51,000	270	-0.0028
71,000	214	-0.002

mentation it might be convenient to write (118) on the form

$$\frac{dp}{dz} = -\frac{Mg}{R^*(\bar{T}(z) + L(z)(z - h(z)))}p,$$

$\bar{T}(z)$, $L(z)$, and $h(z)$ are piecewise constant functions with values given in the tal
he pressure at the sea level $z = 0$, $p_0 = p(0)$, is 101325 Pa.

ation: $L = 0$. One commonly used simplification is to assume that the temp
ıt within each layer. This means that $L = 0$.

ation: one-layer model. Another commonly used approximation is to wc
instead of seven. This one-layer model[36] is based on $T(z) = T_0 - Lz$, with s
temperature $T_0 = 288$ K and temperature lapse rate $L = 0.0065$ K/m.

ompaction of sediments

s, originally made from materials like sand and mud, get compacted through ge
he weight of new material that is deposited on the sea bottom. The porosity
s tells how much void (fluid) space there is between the sand and mud grai

/en.wikipedia.org/wiki/Density_of_air

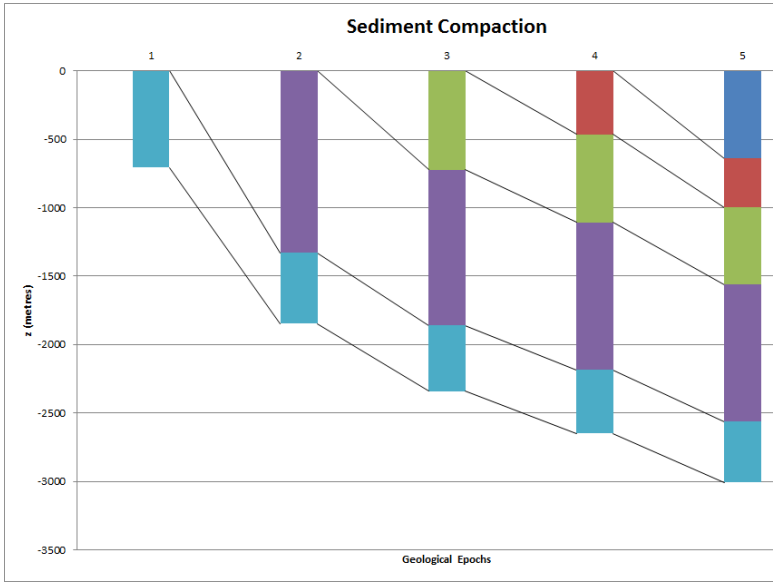
orosity reduces with depth because the weight of the sediments above and cause
› shrink and thereby increase the compaction.

A typical assumption is that the change in ϕ at some depth z is negatively p
his assumption leads to the differential equation problem

$$\frac{d\phi}{dz} = -c\phi, \quad \phi(0) = \phi_0,$$

here the z axis points downwards, $z = 0$ is the surface with known porosity
onstant.

The upper part of the Earth's crust consists of many geological layers stacke
ther, as indicated in Figure 17. The model (120) can be applied for each layer.
we have the unknown porosity function $\phi_i(z)$ fulfilling $\phi_i'(z) = -c_i z$, since the
ıodel (120) depends on the type of sediment in the layer. From the figure we see
f sediments are deposited on top of older ones as time progresses. The compacti
y ϕ, is rapid in the beginning and then decreases (exponentially) with depth in



igure 17: Illustration of the compaction of geological layers (with different colo

When we drill a well at present time through the right-most column of sedime
e can measure the thickness of the sediment in (say) the bottom layer. Let L_1 l
ssuming that the volume of sediment remains constant through time, we have
olume, $\int_0^{L_{1,0}} \phi_1 dz$, must equal the volume seen today, $\int_{\ell-L_1}^{\ell} \phi_1 dz$, where ℓ is t
ottom of the sediment in the present day configuration. After having solved for
f z, we can then find the original thickness $L_{1,0}$ of the sediment from the equa

$$\int_0^{L_{1,0}} \phi_1 dz = \int_{\ell-L_1}^{\ell} \phi_1 dz\,.$$

arbon exploration it is important to know $L_{1,0}$ and the compaction history of the
ediments.

ertical motion of a body in a viscous fluid

ioving vertically through a fluid (liquid or gas) is subject to three different f
e gravity force, the drag force[37], and the buoyancy force.

w of forces. The gravity force is $F_g = -mg$, where m is the mass of the bo
cceleration of gravity. The uplift or buoyancy force ("Archimedes force") is F_b
s the density of the fluid and V is the volume of the body. Forces and other qu
as positive in the upward direction.
rag force is of two types, depending on the Reynolds number

$$\mathrm{Re} = \frac{\varrho d|v|}{\mu},$$

the diameter of the body in the direction perpendicular to the flow, v is the ve
and μ is the dynamic viscosity of the fluid. When $\mathrm{Re} < 1$, the drag force is fa
by the so-called Stokes' drag, which for a spherical body of diameter d reads

$$F_d^{(S)} = -3\pi d\mu v\,.$$

Re, typically $\mathrm{Re} > 10^3$, the drag force is quadratic in the velocity:

$$F_d^{(q)} = -\frac{1}{2}C_D\varrho A|v|v,$$

is a dimensionless drag coefficient depending on the body's shape, and A is th
area as produced by a cut plane, perpendicular to the motion, through the thick
ly. The superscripts ^{q} and ^{S} in $F_d^{(S)}$ and $F_d^{(q)}$ indicate Stokes drag and quadra
ly.

n of motion. All the mentioned forces act in the vertical direction. Newton's
tion applied to the body says that the sum of these forces must equal the mas
es its acceleration a in the vertical direction.

$$ma = F_g + F_d^{(S)} + F_b\,.$$

ave chosen to model the fluid resistance by the Stokes drag. Inserting the exp
rces yields

$$ma = -mg - 3\pi d\mu v + \varrho gV\,.$$

nowns here are v and a, i.e., we have two unknowns but only one equation
s in physics we know that the acceleration is the time derivative of the
. This is our second equation. We can easily eliminate a and get a single diff
for v:

$$m\frac{dv}{dt} = -mg - 3\pi d\mu v + \varrho gV\,.$$

/en.wikipedia.org/wiki/Drag_(physics)

small rewrite of this equation is handy: We express m as $\varrho_b V$, where ϱ_b is th
ody, and we divide by the mass to get

$$v'(t) = -\frac{3\pi d\mu}{\varrho_b V}v + g\left(\frac{\varrho}{\varrho_b} - 1\right)\,.$$

Ve may introduce the constants

$$a = \frac{3\pi d\mu}{\varrho_b V}, \quad b = g\left(\frac{\varrho}{\varrho_b} - 1\right),$$

o that the structure of the differential equation becomes obvious:

$$v'(t) = -av(t) + b\,.$$

he corresponding initial condition is $v(0) = v_0$ for some prescribed starting vel
This derivation can be repeated with the quadratic drag force $F_d^{(q)}$, leading

$$v'(t) = -\frac{1}{2}C_D\frac{\varrho A}{\varrho_b V}|v|v + g\left(\frac{\varrho}{\varrho_b} - 1\right)\,.$$

efining

$$a = \frac{1}{2}C_D\frac{\varrho A}{\varrho_b V},$$

nd b as above, we can write (127) as

$$v'(t) = -a|v|v + b\,.$$

erminal velocity. An interesting aspect of (126) and (129) is whether v will
onstant value, the so-called *terminal velocity* v_T, as $t \to \infty$. A constant v mean
s $t \to \infty$ and therefore the terminal velocity v_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 $v_T = -\sqrt{|}$
ody ($v_T < 0$) and $v_T = \sqrt{b/a}$ for a rising body ($v_T > 0$).

Crank-Nicolson scheme. Both governing equations, the Stokes' drag mod
uadratic drag model (129), can be readily solved by the Forward Euler sche
ccuracy one can use the Crank-Nicolson method, but a straightforward applicat
esults a nonlinear equation in the new unknown value v^{n+1} when applied to (1

$$\frac{v^{n+1} - v^n}{\Delta t} = -a\frac{1}{2}(|v^{n+1}|v^{n+1} + |v^n|v^n) + b\,.$$

owever, instead of approximating the term $-|v|v$ by an arithmetic average
eometric mean:

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

is of second order in Δt, just as for the arithmetic average and the centere approximation in (130). With this approximation trick, the discrete equation

$$\frac{v^{n+1}-v^n}{\Delta t} = -a|v^n|v^{n+1} + b$$

a *linear* equation in v^{n+1}, and we can therefore easily solve for v^{n+1}:

$$v^{n+1} = \frac{v_n + \Delta t b^{n+\frac{1}{2}}}{1 + \Delta t a^{n+\frac{1}{2}}|v^n|}\,.$$

data. Suitable values of μ are $1.8\cdot 10^{-5}$ Pa s for air and $8.9\cdot 10^{-4}$ Pa s fo can be taken as 1.2 kg/m^3 for air and as $1.0\cdot 10^3$ kg/m^3 for water. For cons splacement in the atmosphere one should take into account that the density of a ltitude, see Section 8.6. One possible density variation arises from the one-laye ntioned section.
ensity variation makes b time dependent and we need $b^{n+\frac{1}{2}}$ in (132). To comp at enters $b^{n+\frac{1}{2}}$ we must also compute the vertical position $z(t)$ of the body , we can use a centered difference approximation:

$$\frac{z^{n+\frac{1}{2}} - z^{n-\frac{1}{2}}}{\Delta t} = v^n \quad \Rightarrow \quad z^{n+\frac{1}{2}} = z^{n-\frac{1}{2}} + \Delta t\, v^n\,.$$

is used in the expression for b to compute $\varrho(z^{n+\frac{1}{2}})$ and then $b^{n+\frac{1}{2}}$.
rag coefficient[38] C_D depends heavily on the shape of the body. Some values a re, 0.42 for a semi-sphere, 1.05 for a cube, 0.82 for a long cylinder (when the cer ertical direction), 0.75 for a rocket, 1.0-1.3 for a man in upright position, 1.3 f pendicular to the flow, and 0.04 for a streamlined, droplet-like body.

ion. To verify the program, one may assume a heavy body in air such that be neglected, and further assume a small velocity such that the air resistance eglected. This can be obtained by setting μ and ϱ to zero. The motion then ty $v(t) = v_0 - gt$, which is linear in t and therefore should be reproduced to a (say tolerance 10^{-15}) by any implementation based on the Crank-Nicolson or l emes.
er verification, but not as powerful as the one above, can be based on compu velocity and comparing with the exact expressions. The advantage of this verific an also the test situation $\varrho \neq 0$.
ays, the method of manufactured solutions can be applied to test the implem as in the governing equation, but the solution then has no physical relevance in

Applying scaling, as described in Section 8.1, will for the linear case reduce t te values for seven parameters down to choosing one value of a single dimen

$$\beta = \frac{\varrho_b g V\left(\frac{\varrho}{\varrho_b}-1\right)}{3\pi d\mu I},$$

/en.wikipedia.org/wiki/Drag_coefficient

rovided $I \neq 0$. If the motion starts from rest, $I = 0$, the scaled problem $\bar{u}' = 1 -$ o need for estimating physical parameters. This means that there is a single univ ne problem of a falling body starting from rest: $\bar{u}(t) = 1 - e^{-\bar{t}}$. All real physical o stretching the $\bar{t}$ axis and the $\bar{u}$ axis in this dimensionless solution. More precis elocity $u(t)$ is related to the dimensionless velocity $\bar{u}(\bar{t})$ through

$$u = \frac{\varrho_b g V\left(\frac{\varrho}{\varrho_b}-1\right)}{3\pi d\mu}\bar{u}(t/(g(\varrho/\varrho_b - 1)))\,.$$

.9 Decay ODEs from solving a PDE by Fourier expansio

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 $u(x,0) =$ xpress 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 exp or easy algebra, but the physical u is taken as the real part of any complex e at the expansion in terms of $e^{ikx\pi/L}$ is compatible with the boundary conditio $^{ikx\pi/L}$ vanish for $x = 0$ and $x = L$. 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 re not important here. Also, suppose we can express the given $f(x,t)$ as

$$f(x,t) = \sum_{k=1}^{m} b_k(t)e^{ikx\pi/L}\,.$$

serting the expansions for u and f in the differential equations demands orresponding to a given k must be equal. The calculations results in the f DEs:

$$A_k'(t) = -\alpha\frac{k^2\pi^2}{L^2} + b_k(t), \quad k = 1,\ldots,m\,.$$

rom the initial condition

$$u(x,0) = \sum_k A_k(0)e^{ikx\pi/L} = I(x) = \sum_k I_k e^{(ikx\pi/L)},$$

follows that $A_k(0) = I_k$, $k = 1,\ldots,m$. We then have m equations of the form $_k(0) = I_k$, for appropriate definitions of a and b. These ODE problems indepe ch that we can solve one problem at a time. The outline technique is a quite co or solving partial differential equations.

Since a_k depends on k and the stability of the Forward Euler scheme d
we get that $\Delta t \leq \alpha^{-1}L^2\pi^{-2}k^{-2}$. Usually, quite large k values are needed to ac
the given functions I and f and then Δt needs to be very small for these large v
ore, the Crank-Nicolson and Backward Euler schemes, which allow larger Δt
h in the solutions, are more popular choices when creating time-stepping algorit
fferential equations of the type considered in this example.

ercises

e 10: Derive schemes for Newton's law of cooling

etail how we can apply the ideas of the Forward Euler, Backward Euler, Crank-N
discretizations to derive explicit computational formulas for new temperature v
law of cooling (see Section 8.5):

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

s the temperature of the body, T_s is the temperature of the surroundings, t
heat transfer coefficient, and T_0 is the initial temperature of the body. Fi
`cooling.pdf`.

e 11: Implement schemes for Newton's law of cooling

a θ-rule for the three schemes in Exercise 10 such that you can get the three
ngle formula by varying the θ parameter. Implement the θ scheme in a f
`T0, k, T_s, t_end, dt, theta=0.5)`, where `T0` is the initial temperature,
sfer coefficient, `T_s` is the temperature of the surroundings, `t_end` is the end tim
, `dt` is the time step, and `theta` corresponds to θ. The `cooling` function shoul
erature as an array `T` of values at the mesh points and the time mesh `t`. C
n examples to check that the implementation works.

or verification, try to find an exact solution of the discrete equations. A tri
$u = T - T_s$, observe that $u^n = (T_0 - T_s)A^n$ for some amplification factor A, a
is formula in terms of T^n.
`cooling.py`.

e 12: Find time of murder from body temperature

ve measures the temperature of a dead body to be 26.7 C at 2 pm. One hour l
re is 25.8 C. The question is when death occurred.
e that Newton's law of cooling (133) is an appropriate mathematical model
of the temperature in the body. First, determine k in (133) by formulating a l
roximation with one time steep from time 2 am to time 3 am, where knowing
ures allows for finding k. Assume the temperature in the air to be 20 C. Th
the temperature evolution from the time of murder, taken as $t = 0$, when T
temperature reaches 25.8 C. The corresponding time allows for answering whe
Filename: `detective.py`.

xercise 13: Simulate an oscillating cooling process

he surrounding temperature T_s in Newton's law of cooling (133) may vary in tir
e variations are periodic with period P and amplitude a around a constant m
$_m$:

$$T_s(t) = T_m + a\sin\left(\frac{2\pi}{P}t\right)\,.$$

imulate a process with the following data: $k = 20$ min^{-1}, $T(0) = 5$ C, $T_m =$
nd $P = 1$ h. Also experiment with $P = 10$ min and $P = 3$ h. Plot T and T_s i
ilename: `osc_cooling.py`.

xercise 14: Radioactive decay of Carbon-14

he Carbon-14[39] isotope, whose radioactive decay is used extensively in dating
at is tens of thousands of years old, has a half-life of 5,730 years. Determi
rganic material that contains 8.4 percent of its initial amount of Carbon-14. U
year in the computations. The uncertainty in the half time of Carbon-14 is ±4
e corresponding uncertainty in the estimate of the age?

lint. Use simulations with $5,730 \pm 40$ y as input and find the corresponding
sult.
ilename: `carbon14.py`.

xercise 15: Simulate stochastic radioactive decay

he purpose of this exercise is to implement the stochastic model described in
ow that its mean behavior approximates the solution of the corresponding OI
The simulation goes on for a time interval $[0, T]$ divided into N_t intervals o
art with N_0 atoms. In some time interval, we have N atoms that have surviv
ernoulli trials with probability $\lambda\Delta t$ in this interval by drawing N random num
(survival) or 1 (decay), where the probability of getting 1 is $\lambda\Delta t$. We are i
umber of decays, d, and the number of survived atoms in the next interval is t
ernoulli trials are simulated by drawing N uniformly distributed real numb
aying 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
```

bserve that `uniform < lambda_*dt` is a boolean array whose true and false
nd 0, respectively, when converted to an integer array.
Repeat the simulation over $[0, T]$ a large number of times, compute the ave
each interval, and compare with the solution of the corresponding ODE m
`tochastic_decay.py`.

[39] `http://en.wikipedia.org/wiki/Carbon-14`

e 16: Radioactive decay of two substances

two radioactive substances A and B. The nuclei in substance A decay to forı
with a half-life $A_{1/2}$, while substance B decay to form type A nuclei with a
ting u_A and u_B be the fractions of the initial amount of material in substance A
ly, the following system of ODEs governs the evolution of $u_A(t)$ and $u_B(t)$:

$$\frac{1}{\ln 2}u'_A = u_B/B_{1/2} - u_A/A_{1/2},$$
$$\frac{1}{\ln 2}u'_B = u_A/A_{1/2} - u_B/B_{1/2},$$

) $= u_B(0) = 1$.
a simulation program that solves for $u_A(t)$ and $u_B(t)$. Verify the implementa
g analytically the limiting values of u_A and u_B as $t \to \infty$ (assume $u'_A, u'_B \to$
g these with those obtained numerically.
ıe program for the case of $A_{1/2} = 10$ minutes and $B_{1/2} = 50$ minutes. Use a time
lot u_A and u_B versus time in the same plot. Filename: `radioactive_decay_2su`

e 17: Simulate the pressure drop in the atmosphere

ler the models for atmospheric pressure in Section 8.6. Make a program wit

computing the pressure $p(z)$ using a seven-layer model and varying L,

computing $p(z)$ using a seven-layer model, but with constant temperature in ea

computing $p(z)$ based on the one-layer model.

these implementations be verified? Should ease of verification impact how you c
? Compare the three models in a plot. Filename: `atmospheric_pressure.py`.

e 18: Make a program for vertical motion in a fluid

ıt the Stokes' drag model (124) and the quadratic drag model (127) from Sec
Crank-Nicolson scheme and a geometric mean for $|v|v$ as explained, and assume c
ity. At each time level, compute the Reynolds number Re and choose the Stok
Re < 1 and the quadratic drag model otherwise.
ɔmputation of the numerical solution should take place either in a stand-alone
tion 2.1) or in a solver class that looks up a problem class for physical dat
?). Create a module (see Section **??**) and equip it with nose tests (see Section
ally verifying the code.
ation tests can be based on

terminal velocity (see Section 8.8),

exact solution when the drag force is neglected (see Section 8.8),

method of manufactured solutions (see Section 5.5) combined with computing
e rates (see Section **??**).

se, e.g., a quadratic polynomial for the velocity in the method of manufacture
xpected error is $\mathcal{O}(\Delta t^2)$ from the centered finite difference approximation an
ıean approximation for $|v|v$.

A solution that is linear in t will also be an exact solution of the discrete eq
roblems. Show that this is true for linear drag (by adding a source term tha
ut not for quadratic drag because of the geometric mean approximation. Us
ıanufactured solutions to add a source term *in the discrete equations for qua*
ıat a linear function of t is a solution. Add a nose test for checking that the li
eproduced to machine precision in the case of both linear and quadratic drag.

Apply the software to a case where a ball rises in water. The buoyancy f
riving force, but the drag will be significant and balance the other forces after
ɔccer ball has radius 11 cm and mass 0.43 kg. Start the motion from rest, se
ater, ϱ, to 1000 kg/m^3, set the dynamic viscosity, μ, to 10^{-3} Pa s, and use a dr
sphere: 0.45. Plot the velocity of the rising ball. Filename: `vertical_motior`

’roject 19: Simulate parachuting

he aim of this project is to develop a general solver for the vertical motion
uadratic air drag, verify the solver, apply the solver to a skydiver in free fall, a
ıe solver to a complete parachute jump.

All the pieces of software implemented in this project should be realized as F
nd/or classes and collected in one module.

) Set up the differential equation problem that governs the velocity of the motior
ımper is subject to the gravity force and a quadratic drag force. Assume consta
n extra source term be used for program verification. Identify the input data t

) Make a Python module for computing the velocity of the motion. Also equip
ınctionality for plotting the velocity.

Hint 1. Use the Crank-Nicolson scheme with a geometric mean of $|v|v$ in time
quation of motion with quadratic drag.

Hint 2. You can either use functions or classes for implementation. If you c
ıake a function `solver` that takes all the input data in the problem as arguments
ıe velocity (as a mesh function) and the time mesh. In case of a class-based
ıtroduce a problem class with the physical data and a solver class with the nur
`solve` 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

) Show that a linear function of t does not fulfill the discrete equations because
ıean approximation used for the quadratic drag term. Fit a source term, as i
ıanufactured solutions, such that a linear function of t is a solution of the dis
Iake a nose test to check that this solution is reproduced to machine precision

) The expected error in this problem goes like Δt^2 because we use a centered
pproximation with error $\mathcal{O}(\Delta t^2)$ and a geometric mean approximation with er
ıe method of manufactured solutions combined with computing convergence r
ɔde. Make a nose test for checking that the convergence rate is correct.

ute the drag force, the gravity force, and the buoyancy force as a function
plot with these three forces.

ou can either make a function `forces(v, t, plot=None)` that returns the fo
tions) and `t` and shows a plot on the screen and also saves the plot to a file wi
`lot` is not `None`, or you can extend the solver class with computation of for
otting of forces in the visualization class.

ıte the velocity of a skydiver in free fall before the parachute opens.

Ieade and Struthers [5] provide some data relevant to skydiving[40]. The mas
ody and equipment can be set to 100 kg. A skydiver in spread-eagle formatic
ion of 0.5 m^2 in the horizontal plane. The density of air decreases varies altitu
ken as constant, 1 kg/m^3, for altitudes relevant to skydiving (0-4000 m). T
for a man in upright position can be set to 1.2. Start with a zero velocity. A
ıas a terminating velocity of 45 m/s. (This value can be used to tune other para

ext task is to simulate a parachute jumper during free fall and after the parachut
$_p$, the parachute opens and the drag coefficient and the cross-sectional area
ully. Use the program to simulate a jump from $z = 3000$ m to the ground $z = 0$
ximum acceleration, measured in units of g, experienced by the jumper?

ollowing Meade and Struthers [5], one can set the cross-section area perpendi
n to 44 m^2 when the parachute is open. Assume that it takes 8 s to increase t
om the original to the final value. The drag coefficient for an open parachute
..8, but tuned using the known value of the typical terminating velocity reache
5.3 m/s. One can take the drag coefficient as a piecewise constant function
ange at t_p. The parachute is typically released after $t_p = 60$ s, but larger valu
ed to make plots more illustrative.
`skydiving.py`.

e 20: Formulate vertical motion in the atmosphere

ıotion of a body in the atmosphere needs to take into account a varying air d
of altitudes is many kilometers. In this case, ϱ varies with the altitude z. The e
for the body is given in Section 8.8. Let us assume quadratic drag force (other
to be very, very small). A differential equation problem for the air density, b
ıation for the one-layer atmospheric model in Section 8.6, can be set up as

$$p'(z) = -\frac{Mg}{R^*(T_0 + Lz)}p,$$
$$\varrho = p\frac{M}{R^*T}\,.$$

te $p(z)$ we need the altitude z. From the principle that the velocity is the deriv
on we have that

$$z'(t) = v(t),$$

/en.wikipedia.org/wiki/Parachuting

here v is the velocity of the body.
Explain in detail how the governing equations can be discretized by the For
ıe Crank-Nicolson methods. Filename: `falling_in_variable_density.pdf`.

Exercise 21: Simulate vertical motion in the atmosphere

nplement the Forward Euler or the Crank-Nicolson scheme derived in Exercise
ıe effect of air density variation on a falling human, e.g., the famous fall of Felix
he drag coefficient can be set to 1.2.

temark. In the Crank-Nicolson scheme one must solve a 3×3 system of eq
me level, since p, ϱ, and v are coupled, while each equation can be stepped fo
ith the Forward Euler scheme. Filename: `falling_in_variable_density.py`

Exercise 22: Compute $y = |x|$ by solving an ODE

onsider the ODE problem

$$y'(x) = \begin{cases} -1, & x < 0, \\ 1, & x \geq 0 \end{cases} \qquad x \in (-1, 1], \quad y(1-) = 1,$$

hich has the solution $y(x) = |x|$. Using a mesh $x_0 = -1$, $x_1 = 0$, and $x_2 = 1$, ca
$_1$ and y_2 from the Forward Euler, Backward Euler, Crank-Nicolson, and Leapfr
ll of the former three methods for computing the y_1 value to be used in the Lea
f y_2. Thereafter, visualize how these schemes perform for a uniformly partiti
$= 10$ and $N = 11$ points. Filename: `signum.py`.

Exercise 23: Simulate growth of a fortune with random int

he goal of this exercise is to compute the value of a fortune subject to inflatio
ıterest rate. Suppose that the inflation is constant at i percent per year and
ıterest rate, p, changes randomly at each time step, starting at some value p
ındom change is from a value p^n at $t = t_n$ to $p_n + \Delta p$ with probability 0.
ith probability 0.25. No change occurs with probability 0.5. There is also n
xceeds 15 or becomes below 1. Use a time step of one month, $p_0 = i$, initial f
, and simulate 1000 scenarios of length 20 years. Compute the mean evolutio
ıoney and the corresponding standard deviation. Plot the mean curve along
lus one standard deviation and the mean minus one standard deviation. This w
ncertainty 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
```

[41] http://en.wikipedia.org/wiki/Felix_Baumgartner

```
:
p_np1 = p_n
rn (p_np1 if 1 <= p_np1 <= 15 else p_n)
```

If $u_i(t)$ is the value of the fortune in experiment number i, $i = 0, \ldots, N-1$, tl
of the fortune is

$$\bar{u}(t) = \frac{1}{N}\sum_{i=0}^{N-1} u_i(t),$$

tandard deviation is

$$s(t) = \sqrt{\frac{1}{N-1}\left(-(\bar{u}(t))^2 + \sum_{i=0}^{N-1}(u_i(t))^2\right)}.$$

$u_i(t)$ is stored in an array `u`. The mean and the standard deviation of the fo
iently computed by using two accumulation arrays, `sum_u` and `sum_u2`, and per
`u` and `sum_u2 += u**2` after every experiment. This technique avoids storing
series for computing the statistics.
`random_interest.py`.

e 24: Simulate a population in a changing environment

study a population modeled by (108) where the environment, represented by
changes with time.

ıe that there is a sudden drop (increase) in the birth (death) rate at time $t = t_r$,
nutrition or food supply:

$$a(t) = \begin{cases} r_0, & t < t_r, \\ r_0 - A, & t \geq t_r, \end{cases}$$

in population growth is compensated by a sudden net immigration at time t_f

$$f(t) = \begin{cases} 0, & t < t_f, \\ f_0, & t \geq t_a, \end{cases}$$

ı r_0 and make $A > r_0$. Experiment with these and other parameters to illust
of growth and decay in such a problem. Filename: `population_drop.py`.

ve assume that the environmental conditions changes periodically with time so

$$r(t) = r_0 + A\sin\left(\frac{2\pi}{P}t\right).$$

he combined birth and death rate oscillates around r_0 with a maximum chang
over a period of length P in time. Set $f = 0$ and experiment with the other par
te typical features of the solution. Filename: `population_osc.py`.

e 25: Simulate logistic growth

logistic ODE (109) using a Crank-Nicolson scheme where $(u^{n+\frac{1}{2}})^2$ is approxim
ic mean:

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

makes the discrete equation linear in u^{n+1}. Filename: `logistic_CN.py`.

xercise 26: Rederive the equation for continuous compou

he ODE model (112) was derived under the assumption that r was consta
lternative derivation without this assumption: 1) start with (110); 2) introd
t instead of m: $\Delta t = 1/m$ if t is measured in years; 3) divide by Δt an
$t \to 0$. Simulate a case where the inflation is at a constant level I percent p
nterest rate oscillates: $r = -I/2 + r_0\sin(2\pi t)$. Compare solutions for $r_0 = I, 3I$,
`nterest_modeling.py`.

eferences

] D. Griffiths, F. David, and D. J. Higham. *Numerical Methods for Ordin
Equations: Initial Value Problems. Springer, 2010.

] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equa
Problems. Springer, 1993.

] G. Hairer and E. Wanner. *Solving Ordinary Differential Equations II*. Sprin

] H. P. Langtangen. *A Primer on Scientific Programming With Python*. Texts in
Science and Engineering. Springer, third edition, 2012.

] D. B. Meade and A. A. Struthers. Differential equations in the new millenium
problem. *International Journal of Engineering Education*, 15(6):417–424, 19

] L. Petzold and U. M. Ascher. *Computer Methods for Ordinary Differentia
Differential-Algebraic Equations, volume 61. SIAM, 1998.