

# Introduction to computing with finite difference methods

Hans Petter Langtangen<sup>1,2</sup>

<sup>1</sup>Center for Biomedical Computing, Simula Research Laboratory

<sup>2</sup>Department of Informatics, University of Oslo

Jul 14, 2014

## Contents

### 1 Introduction to difference methods

A basic model for exponential decay . . . . .	
The Forward Euler scheme . . . . .	
The Backward Euler scheme . . . . .	
The Crank-Nicolson scheme . . . . .	
The unifying $\theta$ -rule . . . . .	
Constant time step . . . . .	
Compact operator notation for finite differences . . . . .	

### 2 Implementation

Making a solver function . . . . .	
Verifying the implementation . . . . .	
Computing the numerical error as a mesh function . . . . .	
Computing the norm of the numerical error . . . . .	
Plotting solutions . . . . .	
Memory-saving implementation . . . . .	

### 3 Analysis of finite difference equations

Experimental investigation of oscillatory solutions . . . . .	
Exact numerical solution . . . . .	
Stability . . . . .	
Comparing amplification factors . . . . .	
Series expansion of amplification factors . . . . .	
The fraction of numerical and exact amplification factors . . . . .	
The global error at a point . . . . .	
Integrated errors . . . . .	
Truncation error . . . . .	
Consistency, stability, and convergence . . . . .	

### 4 Exercises

## Model extensions

5.1 Generalization: including a variable coefficient . . . . .	
5.2 Generalization: including a source term . . . . .	
5.3 Implementation of the generalized model problem . . . . .	
5.4 Verifying a constant solution . . . . .	
5.5 Verification via manufactured solutions . . . . .	
5.6 Extension to systems of ODEs . . . . .	

## General first-order ODEs

6.1 Generic form . . . . .	
6.2 The $\theta$ -rule . . . . .	
6.3 An implicit 2-step backward scheme . . . . .	
6.4 Leapfrog schemes . . . . .	
6.5 The 2nd-order Runge-Kutta scheme . . . . .	
6.6 A 2nd-order Taylor-series method . . . . .	
6.7 The 2nd- and 3rd-order Adams-Bashforth schemes . . . . .	
6.8 4th-order Runge-Kutta scheme . . . . .	
6.9 The Odespy software . . . . .	
6.10 Example: Runge-Kutta methods . . . . .	
6.11 Example: Adaptive Runge-Kutta methods . . . . .	

## Exercises

### Applications of exponential decay models

8.1 Scaling . . . . .	
8.2 Evolution of a population . . . . .	
8.3 Compound interest and inflation . . . . .	
8.4 Radioactive Decay . . . . .	
8.5 Newton's law of cooling . . . . .	
8.6 Decay of atmospheric pressure with altitude . . . . .	
8.7 Compaction of sediments . . . . .	
8.8 Vertical motion of a body in a viscous fluid . . . . .	
8.9 Decay ODEs from solving a PDE by Fourier expansions . . . . .	

## Exercises

## Exercises, Problems, and Projects

1	Visualize the accuracy of finite differences ...	p. 44
2	Explore the $\theta$ -rule for exponential ...	p. 45
3	Experiment with precision in tests and the ...	p. 60
4	Implement the 2-step backward scheme	p. 61
5	Implement the 2nd-order Adams-Bashforth scheme ...	p. 61
6	Implement the 3rd-order Adams-Bashforth scheme ...	p. 61
7	Analyze explicit 2nd-order methods	p. 61
8	Implement and investigate the Leapfrog scheme	p. 61
9	Make a unified implementation of many schemes	p. 63
10	Derive schemes for Newton's law of cooling	p. 75
11	Implement schemes for Newton's law of cooling	p. 75
12	Find time of murder from body temperature	p. 75
13	Simulate an oscillating cooling process	p. 76
14	Radioactive decay of Carbon-14	p. 76
15	Simulate stochastic radioactive decay	p. 76
16	Radioactive decay of two substances	p. 77
17	Simulate the pressure drop in the atmosphere	p. 77
18	Make a program for vertical motion in a fluid	p. 77
19	Simulate parachuting	p. 78
20	Formulate vertical motion in the atmosphere	p. 79
21	Simulate vertical motion in the atmosphere	p. 80
22	Compute $y =  x $ by solving an ODE	p. 80
23	Simulate growth of a fortune with random interest ...	p. 80
24	Simulate a population in a changing environment ...	p. 81
25	Simulate logistic growth	p. 81
26	Rederive the equation for continuous compound ...	p. 82

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

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

- How to think when constructing finite difference methods, with special focus on 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, lambda functions in functions (closures), doctests, unit tests, command-line interfaces
- How to perform array computing and understand the difference from scalar computing
- How to conduct and automate large-scale numerical experiments
- How to generate scientific reports
- How to uncover numerical artifacts in the computed solution
- How to analyze the numerical schemes mathematically to understand why they work
- How to derive mathematical expressions for various measures of the error of the numerical methods, frequently by using the `sympy` software for symbolic computation
- Introduce concepts such as finite difference operators, mesh (grid), mesh function, truncation error, consistency, and convergence
- Present additional methods for the general nonlinear ODE  $u' = f(u, t)$ , including the Runge-Kutta methods, and the 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 science and finance

---

## Exposition in a nutshell.

---

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

How do we solve a differential equation problem and produce numbers?

How do we trust the answer?

---

---

## Finite difference methods

---

plain the basic ideas of finite difference methods using a simple ordinary differential equation  $u' = -au$  as primary example. Emphasis is put on the reasoning when discretizing the problem and introduction of key concepts such as mesh, mesh function, finite difference approximations, averaging in a mesh, derivation of algorithms, and discrete operations.

---

### Basic model for exponential decay

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

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

where  $a$  is a constant and  $u'(t)$  means differentiation with respect to time  $t$ . This problem arises in a number of widely different phenomena where some quantity  $u$  undergoes exponential reduction. Examples include radioactive decay, population decay, investment of an object, pressure decay in the atmosphere, and retarded motion in fluids (for example,  $a$  can be negative as well), see Section 8 for details and motivation. We have chosen this simple ODE not only because its applications are relevant, but even more because a numerical solution method for this simple ODE gives important insight that can be generalized to more complicated settings, in particular when solving diffusion-type partial differential equations. The analytical solution of the ODE is found by the method of separation of variables:

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

where  $C$  is an arbitrary constant. To formulate a mathematical problem for which there is a unique solution, we need a condition to fix the value of  $C$ . This condition is known as the *initial condition* and is usually given as  $u(0) = I$ . That is, we know the value  $I$  of  $u$  when the process starts at  $t = 0$ . The solution 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 included in the domain now  $u$  here and assume that the equation governs  $u$  for  $t > 0$ . The complete ODE problem reads: find  $u(t)$  such that

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

This is known as a *continuous problem* because the parameter  $t$  varies continuously. For each  $t$  we have a corresponding  $u(t)$ . There are hence infinitely many values of  $u(t)$ . The purpose of a numerical method is to formulate a corresponding *discrete problem* which is characterized by a finite number of values, which can be computed in a few steps on a computer.

## 2 The Forward Euler scheme

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

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

**Step 1: Discretizing the domain.** The time domain  $[0, T]$  is represented by  $N_t + 1$  points

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

The collection of points  $t_0, t_1, \dots, t_{N_t}$  constitutes a *mesh* or *grid*. Often the mesh is uniformly spaced in the domain  $[0, T]$ , which means that the spacing  $t_{n+1} - t_n$  is constant. This spacing is often denoted by  $\Delta t$ , in this case  $t_n = n\Delta t$ .

We seek the solution  $u$  at the mesh points:  $u(t_n)$ ,  $n = 1, 2, \dots, N_t$ . Note that the exact solution is known as  $I$ . A notational short-form for  $u(t_n)$ , which will be used extensively, is  $u_n$ . Precisely, we let  $u^n$  be the *numerical approximation* to the exact solution  $u(t_n)$ . Since a numerical approximation is a *mesh function*, here defined only at the mesh points, to clearly distinguish between the numerical and the exact solution, we often plot the exact solution, as in  $u_e(t_n)$ . Figure 1 shows the  $t_n$  and  $u_n$  points for  $n = 1, \dots, N_t$  as well as  $u_e(t)$  as the dashed line. The goal of a numerical method for ODE is to approximate the mesh function by solving a finite set of *algebraic equations* derived from the problem.

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

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

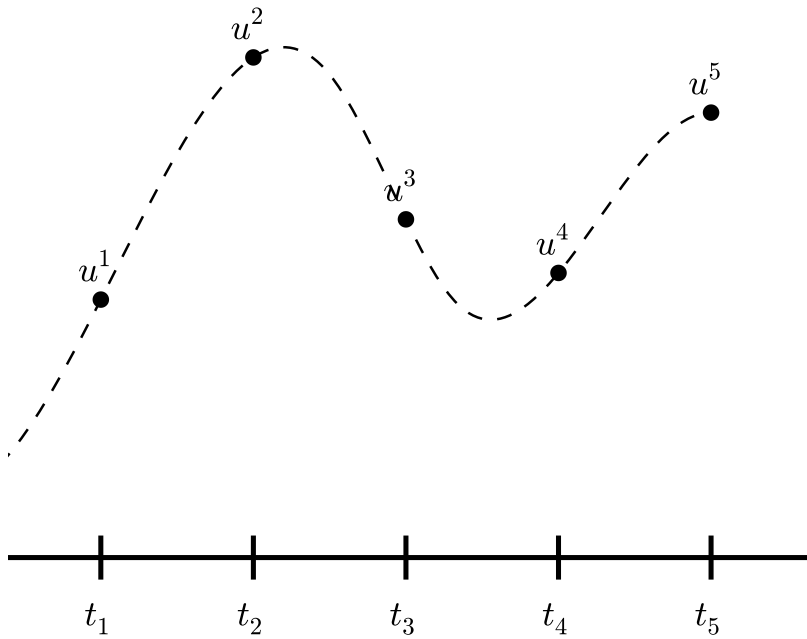


Figure 1: Time mesh with discrete solution values.

**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 require it is fulfilled at a finite set of discrete points in time. The mesh points  $t_0, t_1, \dots, t_N$  (but not the only) choice of points. The original ODE is then reduced to the following:

$$u'(t_n) = -au(t_n), \quad n = 0, \dots, N.$$

**Replacing derivatives by finite differences.** The next and most essential step is to replace the derivative  $u'$  by a finite difference approximation. Let us find a 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

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

It will be absolutely clear that if we want to compute the solution up to time level  $N$ , (4) to hold for  $n = 0, \dots, N-1$  since (6) for  $n = N-1$  creates an equation for  $u^N$ .

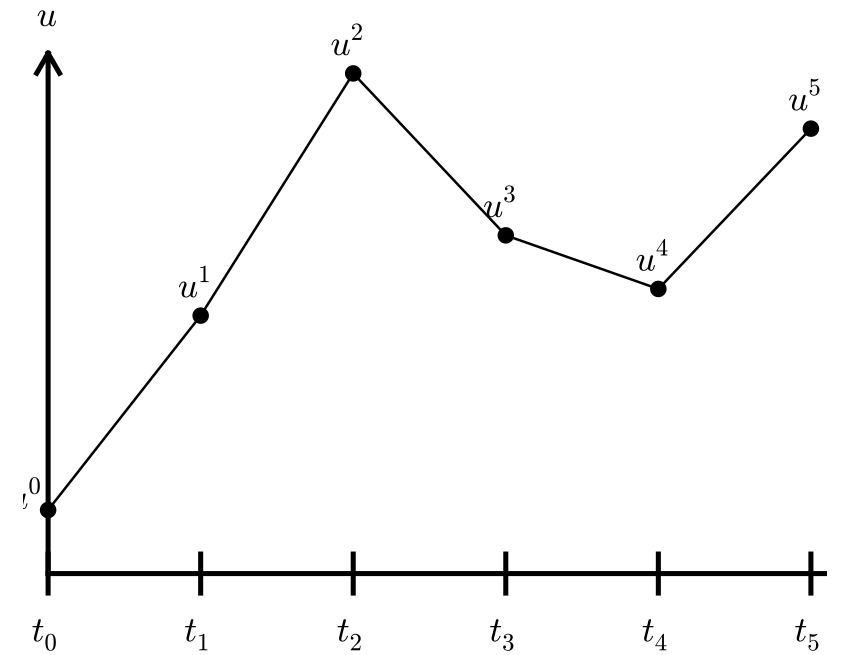


Figure 2: Linear interpolation between the discrete solution values (dashed line).

Equation (6) is the discrete counterpart to the original ODE problem (1), and can be viewed as a *finite difference scheme* or more generally as the *discrete equations* of the problem. A fundamental feature of these equations is that they are *algebraic* and can hence be solved to produce the mesh function, i.e., the values of  $u$  at the mesh points ( $u^n$ ,  $n = 0, \dots, N$ ).

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

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

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

Computation with (7) is straightforward:

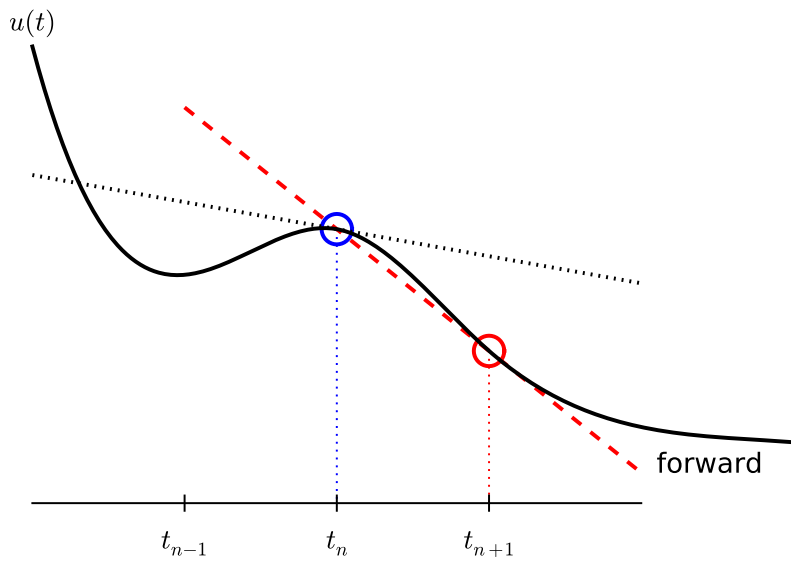


Figure 3: Illustration of a forward difference.

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

until we reach  $u^{N_t}$ . Very often,  $t_{n+1} - t_n$  is constant for all  $n$ , so we can introduce a symbol  $\Delta t$  for the time step:  $\Delta t = t_{n+1} - t_n$ ,  $n = 0, 1, \dots, N_t - 1$ . Using a constant  $\Delta t$  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 computer calculate the sequence  $u^1, u^2, u^3, \dots$ . However, finding such a formula for  $u^n$  is possible only for simple problems, so in general finite difference equations must be solved on a computer. In the next sections will show, the scheme (7) is just one out of many alternative (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 difference method presented in the previous section. Another alternative is

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

since this difference is based on going backward in time ( $t_{n-1}$ ) for information, we call it the Backward Euler difference. Figure 4 explains the idea.

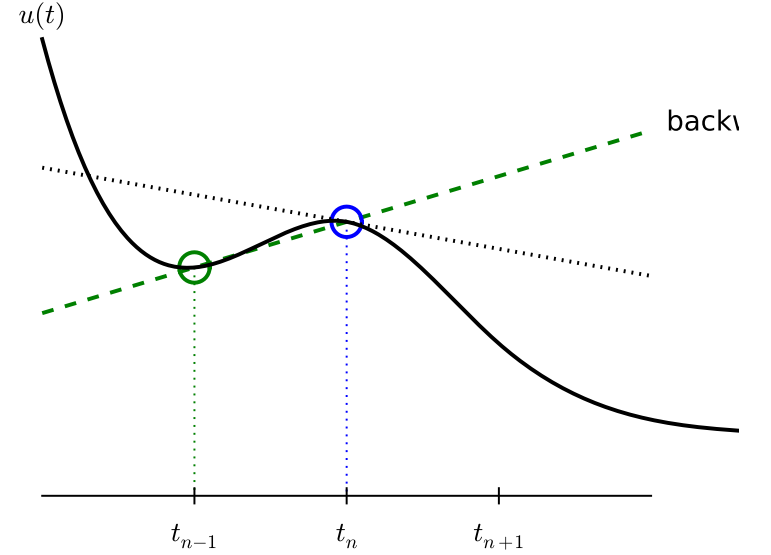


Figure 4: Illustration of a backward difference.

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

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

We assume, as explained under step 4 in Section 1.2, that we have computed  $u^{n-1}$  such that (9) can be used to compute  $u^n$ . For direct similarity with the Forward Euler scheme, 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

The finite difference approximations used to derive the schemes (7) and (10) are forward and backward differences, known to be less accurate than central (or midpoint) differences. To construct 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  $\Delta t$  is 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 approximation or Backward Euler approximation (8) (with  $n$  replaced by  $n+1$ ). The accuracy of an approximation to the derivative of  $u$  depends on *where* we seek the derivative of the interval  $[t_n, t_{n+1}]$  or at the end points.

In the formula (11), where  $u'$  is evaluated at  $t_{n+1/2}$ , it is natural to demand the derivative at the time points *between* the mesh points:

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

Substituting (12) in (12) results in

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

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

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

Substituting (13) in (13) results in

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

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

Assume that  $u^n$  is already computed so that  $u^{n+1}$  is the unknown, which we can solve for.

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

The difference scheme (16) is often called the Crank-Nicolson (CN) scheme or a second order scheme.

### The unifying $\theta$ -rule

Forward Euler, Backward Euler, and Crank-Nicolson schemes can be formulated with a varying parameter  $\theta$ :

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

For

$\theta = 0$  gives the Forward Euler scheme

$\theta = 1$  gives the Backward Euler scheme, and

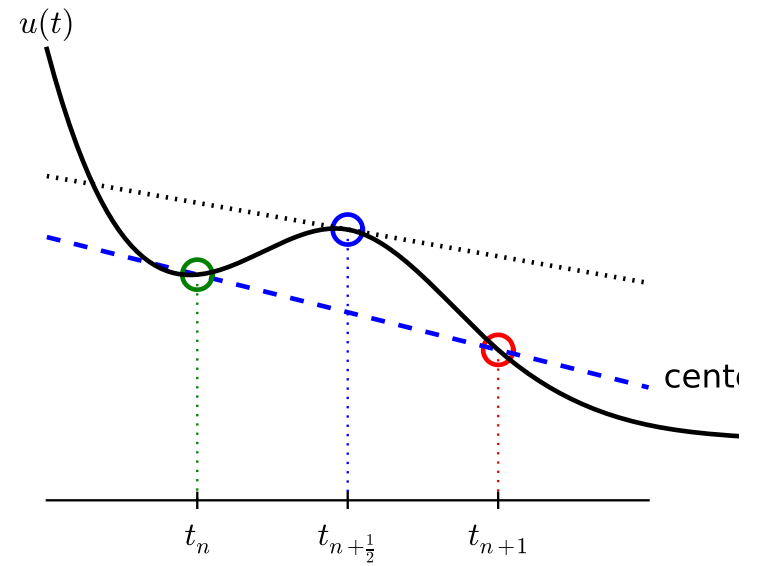


Figure 5: Illustration of a centered difference.

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

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

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

This scheme is known as the  $\theta$ -rule, or alternatively written as the "theta-rule".

#### Derivation.

We start with replacing  $u'$  by the fraction

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

as in the Forward Euler, Backward Euler, and Crank-Nicolson schemes. Then we see that the difference between the methods concerns which point this fraction approximates the derivative. Or in other words, at which point we sample the ODE. So far this has been at the end points or the midpoint of  $[t_n, t_{n+1}]$ . However, we may choose any point  $\tilde{t}$  in the interval. The difficulty is that evaluating the right-hand side  $-au$  at an arbitrary point  $\tilde{t}$  is a problem as in Section 1.4: the point value must be expressed by the discrete values that we compute by the scheme, i.e.,  $u^n$  and  $u^{n+1}$ . Following the averaging idea from Section 1.4, the value of  $u$  at an arbitrary point  $\tilde{t}$  can be calculated as a weighted average of  $u^n$  and  $u^{n+1}$ .

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

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

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

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

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

as 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  $t$   
where the spacing of the mesh points are determined as the computations proce  
er, a uniformly distributed set of mesh points is very common and sufficient fo  
ns. It therefore makes sense to present the finite difference schemes for a unifor  
on  $t_n = n\Delta t$ , where  $\Delta t$  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.

$u^{n+1} = (1 - a\Delta t)u^n$	Forward Euler
$u^{n+1} = \frac{1}{1 + a\Delta t}u^n$	Backward Euler
$u^{n+1} = \frac{1 - \frac{1}{2}a\Delta t}{1 + \frac{1}{2}a\Delta t}u^n$	Crank-Nicolson
$u^{n+1} = \frac{1 - (1-\theta)a\Delta t}{1 + \theta a\Delta t}u^n$	The $\theta$ - rule

urprisingly, 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  
ues 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  
rward 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 respec  
ent variable, here  $t$ . The operator is built of the symbol  $D$ , with the variable as  
perscript 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:

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

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  $[\bar{u}^t]^{n+\frac{1}{2}}$ . (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:

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

1 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 app  
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

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

ion.

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

rule can be specified by

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

where  $\bar{u}$  is a new time difference and a *weighted averaging operator*:

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

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

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

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

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

## Implementation

We will now make a computer program for solving

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

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

The programs referred to in this section are found in the `src/decay1` directory (we use the Unix term *directory* for what many others nowadays call *folder*).

**Initial problem.** We want to explore the Forward Euler scheme, the Backward Euler, and Crank-Nicolson schemes applied to our model problem. From an implementation point of view, it is advantageous to implement the  $\theta$ -rule

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

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

we can generate the three other schemes by various choices of  $\theta$ :  $\theta = 0$  for Forward Euler,  $\theta = 1$  for Backward Euler, and  $\theta = 1/2$  for Crank-Nicolson. Given  $a$ ,  $u^0 = I$ ,  $T$ , and  $\Delta t$ ,

we can use the  $\theta$ -rule to compute  $u^1, u^2, \dots, u^{N_t}$ , where  $t_{N_t} = N_t \Delta t$ , and  $N_t$  the number of time steps  $N_t = T/\Delta t$ .

**Computer Language: Python.** Any programming language can be used to compute the  $u^{n+1}$  values from the formula above. However, in this document we shall mainly use Python for several reasons:

- Python has a very clean, readable syntax (often known as "executable pseudocode").
- Python code is very similar to MATLAB code (and MATLAB has a particular strength in use for scientific computing).
- Python is a full-fledged, very powerful programming language.
- Python is similar to C++, but much simpler to work with and results in more readable code.
- Python has a rich set of modules for scientific computing, and its popularity in scientific computing is rapidly growing.
- Python was made for being combined with compiled languages (C, C++, Fortran) to reuse existing numerical software and to reach high computational performance.
- Python has extensive support for administrative tasks needed when doing large-scale computational investigations.
- Python has extensive support for graphics (visualization, user interfaces, web interfaces).
- FEniCS, a very powerful tool for solving PDEs by the finite element method, is implemented in Python.

Learning Python is easy. Many newcomers to the language will probably learn by following the examples to perform their own computer experiments. The examples show how to use Python code and gradually make use of more powerful constructs as we progress. If it is not inconvenient for the problem at hand, our Python code is made as close to MATLAB code for easy transition between the two languages.

Readers who feel the Python examples are too hard to follow will probably find reading a tutorial, e.g.,

- The Official Python Tutorial<sup>2</sup>
- Python Tutorial on tutorialspoint.com<sup>3</sup>
- Interactive Python tutorial site<sup>4</sup>
- A Beginner's Python Tutorial<sup>5</sup>

The author also has a comprehensive book [4] that teaches scientific programming from the ground up.

<sup>2</sup><http://docs.python.org/2/tutorial/>

<sup>3</sup><http://www.tutorialspoint.com/python/>

<sup>4</sup><http://www.learnpython.org/>

<sup>5</sup>[http://en.wikibooks.org/wiki/A\\_Beginner's\\_Python\\_Tutorial](http://en.wikibooks.org/wiki/A_Beginner's_Python_Tutorial)



## Making a solver function

we have an array  $u$  for storing the  $u^n$  values,  $n = 0, 1, \dots, N_t$ . The algorithm:

initialize  $u^0$

$t = t_n, n = 1, 2, \dots, N_t$ : compute  $u_n$  using the  $\theta$ -rule formula

**for computing the numerical solution.** The following Python function takes the data of the problem ( $I, a, T, \Delta t, \theta$ ) as arguments and returns two arrays with the solution  $u^i$  and the mesh points  $t_0, \dots, t_{N_t}$ , respectively:

```
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."""
    Nt = int(T/dt)          # no of time intervals
    T = Nt*dt              # adjust T to fit time step dt
    u = zeros(Nt+1)         # array of u[n] values
    t = linspace(0, T, Nt+1) # time mesh

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

The `numpy` library contains a lot of functions for array computing. Most of the functions are similar to what is found in the alternative scientific computing language MATLAB. We will use a few of

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

`range` deserves a comment, especially for newcomers to Python. The construction `range(s, e)` generates all integers from 0 to `Nt` in steps of `s`, *but not including Nt*. Omitting `e` gives `range(s)`. For example, `range(0, 6, 3)` gives 0 and 3, while `range(0, Nt)` generates 0, 1, ..., `Nt-1`, which implies the following assignments to `u[n+1]`: `u[1], u[2], ..., u[Nt]`, which is valid since `u` has length `Nt+1`. The first index in Python arrays or lists is *always* 0 and the last is `len(u)-1`. The length of an array `u` is obtained by `len(u)` or `u.size`.

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

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

**division.** The shown implementation of the `solver` may face problems and errors. For example, if `T, a, dt,` and `theta` are given as integers, see Exercises ?? and ??. The problem is *integer division* in Python (as well as in Fortran, C, C++, and many other languages). For example, `1/2` becomes 0, while `1.0/2`, `1/2.0`, or `1.0/2.0` all become 0.5. It is enough to have either the numerator or the denominator is a real number (i.e., a `float` object) to ensure floating point division. Inserting a conversion `dt = float(dt)` guarantees that `dt` is a `float`. See the problems in Exercise ??.

Another problem with computing  $N_t = T/\Delta t$  is that we should round  $N_t$  to the nearest integer. With `Nt = int(T/dt)` the `int` operation picks the largest integer smaller than `T/dt`. The 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 a doc string enclosed in triple double quotes `"""`. The purpose of this string object is to document what the function does and what the arguments are. In this case the necessary documentation is more than one line, but with triple double quoted strings the text may span several lines.

```
def solver(I, a, T, dt, theta):
    """
    Solve
        u'(t) = -a*u(t),
    with initial condition u(0)=I, for t in the time interval
    (0,T]. The time interval is divided into time steps of
    length dt.

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

Such documentation strings appearing right after the header of a function are called doc strings. There are tools that can automatically produce nicely formatted documentation from the definition of functions and the contents of doc strings.

It is strongly recommended to equip any function whose purpose is not obvious from its name. Nevertheless, the forthcoming text deviates from this rule if the function is a module or a class.

**formatting of numbers.** Having computed the discrete solution `u`, it is natural to print the numbers:

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

act **print** statement gives unfortunately quite ugly output because the **t** and **u** are not aligned in nicely formatted columns. To fix this problem, we recommend to use the `format` function supported in most programming languages inherited from C. Another choice is `format` with *mat string syntax*.

Printing `t[i]` and `u[i]` in two nicely formatted columns is done like this with the

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

entage signs signify "slots" in the text where the variables listed at the end
 are inserted. For each "slot" one must specify a format for how the variable is
 to be printed. The string: `s` for pure text, `d` for an integer, `g` for a real number written as
 compact decimal, `e` for scientific notation with three decimals in a field of width 9 characters,
 `9.3E` for scientific notation with three decimals in a field of width 9 characters,
 `2f` for standard decimal notation with two decimals formatted with minimum width 2,
 or `.2f` for standard decimal notation with two decimals formatted with minimum width 1.
 The `printf` syntax provides a quick way of formatting tabular output of numbers
 and text.

Alternative *format string syntax* looks like

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

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

**the program.** The function and main program shown above must be placed in a file with name `decay_v1.py`<sup>6</sup> (v1 stands for "version 1" - we shall make numerous versions of this program). Make sure you write the code with a suitable text editor (Gedit, Emacs, 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 shell which will look different in your terminal window, depending on the terminal application it is set up, commands like `python decay_v1.py` can be issued. These commands are executed 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` is executed by the command `run decay_v1.py`:

```
ipython
un decay_v1.py
u=1
```

---

[/tinyurl.com/jvzzcfn/decay/decay\\_v1.py](http://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 commands called with the up arrow, `%pdb` turns on debugging so that variables can be inspected, program aborts due to an exception, output of commands are stored in variable `_`, statements can be profiled, any operating system command can be executed, loaded automatically and other customizations can be performed when starting IPython. I mention a few of the most useful features.

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

```
terminal> python programname
```

The reason is that such typesetting makes the text more compact in the vertical direction, allowing sessions with IPython syntax.

## .2 Verifying the implementation

It is easy to make mistakes while deriving and implementing numerical algorithms. One should never believe in the printed  $u$  values before they have been thoroughly verified. The best idea is to compare the computed solution with the exact solution, when that exists. There will always be a discrepancy between these two solutions because of the numerical approximation. The challenging question is whether we have the mathematically correct discrepancy or not. One may have another, maybe small, discrepancy due to both an approximation error and a round-off error in the implementation.

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

**Running a few algorithmic steps by hand.** The simplest approach to preference for the discrete solution  $u$  of finite difference equations is to compute a few steps of the algorithm by hand. Then we can compare the hand calculations with numbers computed by a program.

A straightforward approach is to use a calculator and compute  $u^1$ ,  $u^2$ , and  $u^3$ . For  $\Delta t = 0.8$ , 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():
    """Compare three steps with known manual computations."""
    a = 0.8; I = 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 `False`.

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 `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])
```

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

```
ver_three_steps()
```

verification test is always done, future errors introduced accidentally in the p  
od chance of being detected.

complete program including the verification above is found in the file `decay_ver`

[/tinyurl.com/jvzzcfn/decay/decay\\_verf.py](http://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  
ie 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  $u_e(t) = Ie^{-at}$  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:

$$e^n = u_e(t_n) - u^n, \quad n = 0, 1, \dots, N_t.$$

we may view  $u_e^n = u_e(t_n)$  as the representation of  $u_e(t)$  as a mesh function  
ontinuous function defined for all  $t \in [0, T]$  ( $u_e^n$  is often called the *representat*  
mesh). Then,  $e^n = u_e^n - u^n$  is clearly the difference of two mesh functions. This  
 $^n$  is natural when programming.

The error mesh function  $e^n$  can be computed by

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

ote that the mesh functions `u` and `u_e` are represented by arrays and associated  
i 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 th  
`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) i  
to this array returned from `exact_solution`. The expression `u_e - u` is t  
between two arrays, resulting in a new array referred to by `e`.

### 4 Computing the norm of the numerical error

instead of working with the error  $e^n$  on the entire mesh, we often want one nu  
ie 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

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

orm (35) ("L-two norm") has nice mathematical properties and is the most s a generalization of the well-known Euclidian 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  $\ell^2$ . The error in  $\|f\|_{\ell^2}^2$  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  $\Delta t \rightarrow 0$ . 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  $L^\infty$  fined above, are defined by

$$\begin{aligned}\|f^n\|_{\ell^2} &= \left( \Delta t \sum_{n=0}^{N_t} (f^n)^2 \right)^{1/2}, \\ \|f^n\|_{\ell^1} &= \Delta t \sum_{n=0}^{N_t} |f^n| \\ \|f^n\|_{\ell^\infty} &= \max_{0 \leq n \leq N_t} |f^n|.\end{aligned}$$

Note that the  $L^2$ ,  $L^1$ ,  $\ell^2$ , and  $\ell^1$  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 mesh points. Then we can replace  $\Delta t$  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  $f = (f_0, \dots, f_N)$ :

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

ere we have used the common vector component notation with subscripts ( $f_n$ ). e will mostly work with mesh functions and use the discrete  $\ell^2$  norm (39) or tl 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:

```
a = len(u)      # length of u array (alt: u.size)
i_e = zeros(m)
s = 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

ie  $t$  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 between points  $(t[n], u_e[n])$  while  $u_e(t)$  varies as an exponential function between them. The technique for showing the "exact" variation of  $u_e(t)$  between the mesh points is to use 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 legend. Appropriate names on the axis, a title, and a file containing the plot as an input for reports. The Matplotlib package (`matplotlib.pyplot`) contains functions for this. The names of the functions are similar to the plotting functions known from MATLAB. The plot session then becomes

```
plotlib.pyplot import *  
  
nspace(0, T, 1001)      # create new plot  
act_solution(t_e, I, a)  # fine mesh for u_e  
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  $\theta$  and  $\Delta t$  so as to distinguish files from different runs with  $\theta$  and  $\Delta t$ .

A more sophisticated and easy-to-read filename can be generated by mapping the names for the three common schemes: FE (Forward Euler,  $\theta = 0$ ), BE (Backward Euler,  $\theta = 0.5$ ), and CN (Crank-Nicolson,  $\theta = 0.5$ ). A Python dictionary is ideal for such a mapping. The strings:

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

**Results with computing and plotting.** Let us wrap up the computation of  $u$  and all the plotting statements in a function `explore`. This function can be called with  $\theta$  and  $\Delta t$  values to see how the error varies with the method and the mesh resolution.

```
def explore(I, a, T, dt, theta=0.5, makeplot=True):  
    """  
    Run a case with the solver, compute error measure,  
    and plot the numerical and exact solutions (if makeplot=True).  
    """  
    u, t = solver(I, a, T, dt, theta)      # Numerical solution  
    u_e = exact_solution(t, I, a)  
    e = u_e - u  
    E = sqrt(dt*sum(e**2))  
    if makeplot:  
        figure()                          # create new plot  
        t_e = linspace(0, T, 1001)        # fine mesh for u_e  
        u_e = exact_solution(t_e, I, a)  
        plot(t, u, 'r--o')                 # red dashes w/circles  
        plot(t_e, u_e, 'b-')               # blue line for exact 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 curves in the same plot window, while we want the different pairs to appear in separate files. Calling `figure()` ensures this.

The `explore` function stores the plot in three different image file formats: PDF (Encapsulated PostScript). The PNG format is aimed at being included in LaTeX documents, and the EPS format in LaTeX documents. Viewers for these image files on Unix systems are `gv` (comes with Ghostscript) for PDF formats and `display` (from the ImageMagick) suite for PNG files:

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

The complete code containing the functions above resides in the file `decay_plot_mpl.py`. Running this program results in

```
terminal> 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
```

We observe that reducing  $\Delta t$  by a factor of 10 increases the accuracy for all three schemes. We also see that the combination of  $\theta = 0.5$  and a small time step  $\Delta t$  yields a much more accurate solution, and that  $\theta = 0$  and  $\theta = 1$  with  $\Delta t = 0.4$  result in large errors.

<sup>8</sup>[http://tinyurl.com/jvzzcfn/decay/decay\\_plot\\_mpl.py](http://tinyurl.com/jvzzcfn/decay/decay_plot_mpl.py)

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

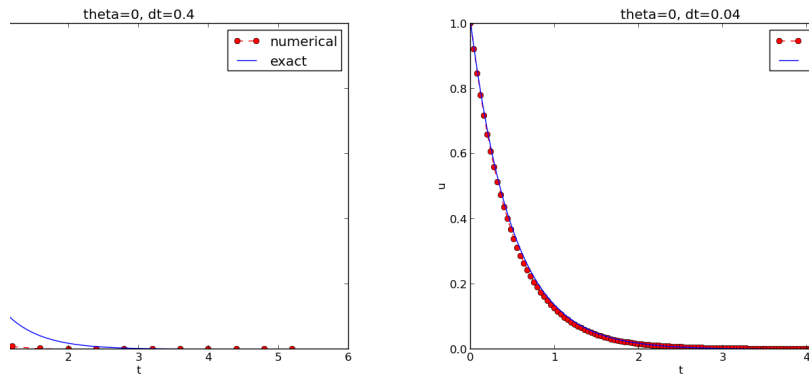


Figure 6: The Forward Euler scheme for two values of the time step.

**Mounting plot files.** Mounting two PNG files, as done in the figure, is easily done with the `montage` 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
```

---

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

TeX reports it is not recommended to use `montage` and PNG files as the result of a plot. Instead, plots should be made in the PDF format and combined using the `pdftk` and `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
```

---

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

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

[www.imagemagick.org/script/montage.php](http://www.imagemagick.org/script/montage.php)

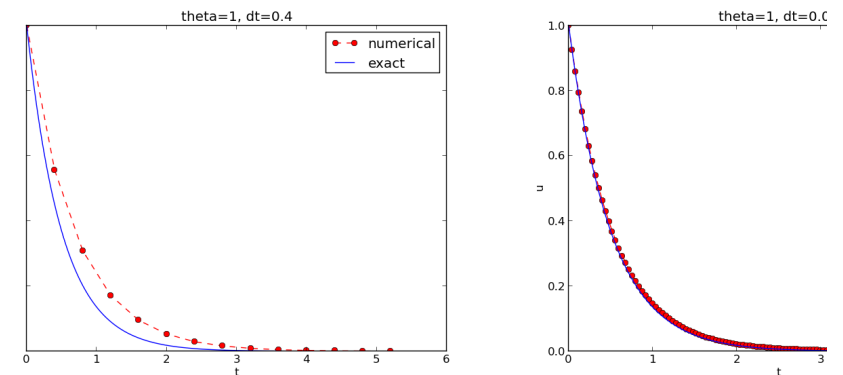


Figure 7: The Backward Euler scheme for two values of the time step.

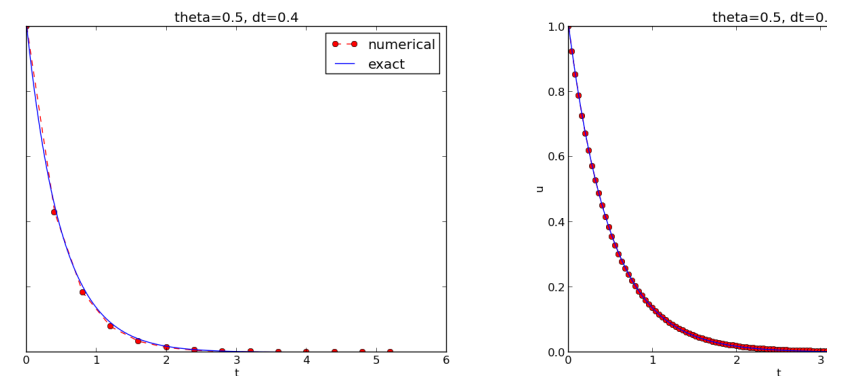


Figure 8: The Crank-Nicolson scheme for two values of the time step.

**Plotting with SciTools.** The SciTools package<sup>10</sup> provides a unified plotting interface, to many different plotting packages, including Matplotlib, Gnuplot, Gd, Tk, OpenDX, and VisIt. The syntax is very similar to that of Matplotlib and, in fact, the plotting commands shown above look the same in SciTools's Easyviz interface. The import statement, which reads

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

This statement performs a `from numpy import *` as well as an import of the most common functions of the Easyviz (`scitools.easyviz`) package, along with some additional numerical functions.

With Easyviz one can merge several plotting commands into a single one, as shown in the following example:

---

<sup>10</sup><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<sup>11</sup> file contains such a demo.

fault, Easyviz employs Matplotlib for plotting, but Gnuplot<sup>12</sup> and Grace<sup>13</sup> are also options:

---

```

python decay_plot_st.py --SCIT00LS_easyviz_backend gnuplot
python decay_plot_st.py --SCIT00LS_easyviz_backend grace

```

---

and used for creating plots (and numerous other options) can be permanently saved in a configuration file.

With Gnuplot windows are launched without any need to kill one before the next one (as in the case with Matplotlib) and one can press the key 'q' anywhere in a plot window to quit. Another advantage of Gnuplot is the automatic choice of sensible and distinguishable line styles, black-and-white PDF and PostScript files.

For more functionality for annotating plots with title, labels on the axis, legends, etc., see the documentation of Matplotlib and SciTools for more detailed information. We hope is that the programming syntax explained so far suffices for understanding the code. Learning more from a combination of the forthcoming examples and other resources, books and web pages.

---

## The understanding.

---

Exercise 11 asks you to implement a solver for a problem that is slightly different from the previous one. You may use the `solver` and `explore` functions explained above as a starting point. Apply the new solver to Exercise 12.

---

## Memory-saving implementation

---

The outer memory requirements of our implementations so far consists mainly of arrays of length  $N_t + 1$ , plus some other temporary arrays that Python creates. This is not a problem if we do array arithmetics in our program (e.g., `I*exp(-a*t)` needs to be computed and then `exp` can be applied to it and then `exp`). Regardless of how we implement simple operations, storage requirements are very modest and put not restriction on how we choose the numerical methods and algorithms. Nevertheless, when the methods for ODEs used here are extended to dimensional partial differential equation (PDE) problems, memory storage requirements become a challenging issue.

[/tinyurl.com/jvzzcfn/decay/decay\\_plot\\_st.py](http://tinyurl.com/jvzzcfn/decay/decay_plot_st.py)  
[/www.gnuplot.info/](http://www.gnuplot.info/)  
[/plasma-gate.weizmann.ac.il/Grace/](http://plasma-gate.weizmann.ac.il/Grace/)

The PDE counterpart to our model problem  $u' = -a$  is a diffusion equation  $u_t = D \nabla^2 u$  in a space-time domain. The discrete representation of this domain may involve a mesh of  $M^3$  points and a time mesh of  $N_t$  points. A typical desired value for  $M$  is 100, or even 1000. Storing all the computed  $u$  values, like we have done for ODEs, far, demands storage of some arrays of size  $M^3 N_t$ , giving a factor of  $M^3$  larger storage compared to our ODE programs. Each real number in the array for  $u$  requires 8 bytes of storage. With  $M = 100$  and  $N_t = 1000$ , there is a storage demand of  $(10^3)^3 \cdot 8$  bytes for the solution array. Fortunately, we can usually get rid of the  $N_t$  factor, reducing the storage. Below we explain how this is done, and the technique is almost always applicable to implementations of PDE problems.

Let us critically evaluate how much we really need to store in the computer memory for our implementation of the  $\theta$  method. To compute a new  $u^{n+1}$ , all we need is to know the previous  $u^{n-1}, u^{n-2}, \dots, u^0$  values do not need to be stored in an array, but only convenient for plotting and data analysis in the program. Instead of the `u` array we use two variables for real numbers, `u` and `u_1`, representing  $u^{n+1}$  and  $u^n$  in the algorithm. At each time level, we update `u` from `u_1` and then set `u_1 = u` so that the computer stores the "previous" value  $u^n$  at the next time level. The downside is that we lose the solution after the simulation is done since only the last two numbers are available to store computed values in a file and use the file for visualizing the solution later.

We have implemented this memory saving idea in the file `decay_memsave.py`. It is a slight modification of `decay_plot_mpl.py`<sup>15</sup> program.

The following function demonstrates how we work with the two most recent values of `u` and `u_1` known:

```

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

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

```

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

```

def read_file(filename='sol.dat'):
    infile = open(filename, 'r')
    u = []; t = []
    for line in infile:

```

<sup>14</sup>[http://tinyurl.com/jvzzcfn/decay/decay\\_memsave.py](http://tinyurl.com/jvzzcfn/decay/decay_memsave.py)

<sup>15</sup>[http://tinyurl.com/jvzzcfn/decay/decay\\_plot\\_mpl.py](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]))
return np.array(t), np.array(u)

```

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

```

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

```

represent the counterpart to the `explore` function from `decay_plot_mpl.py`<sup>16</sup> in `memsave` and then load data from file before we can compute the error measure as

```

def _explore(I, a, T, dt, theta=0.5, makeplot=True):
    name = 'u.dat'
    _memsave(I, a, T, dt, theta, filename=name)

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

```

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

## Analysis of finite difference equations

As the ODE for exponential decay,

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

where  $I$  and  $a$  are given constants. This problem is solved by the  $\theta$ -rule finite difference in the recursive equations

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

numerical solution  $u^{n+1}$ , which approximates the exact solution  $u_e$  at time point  $t_{n+1} = (n+1)\Delta t$ .

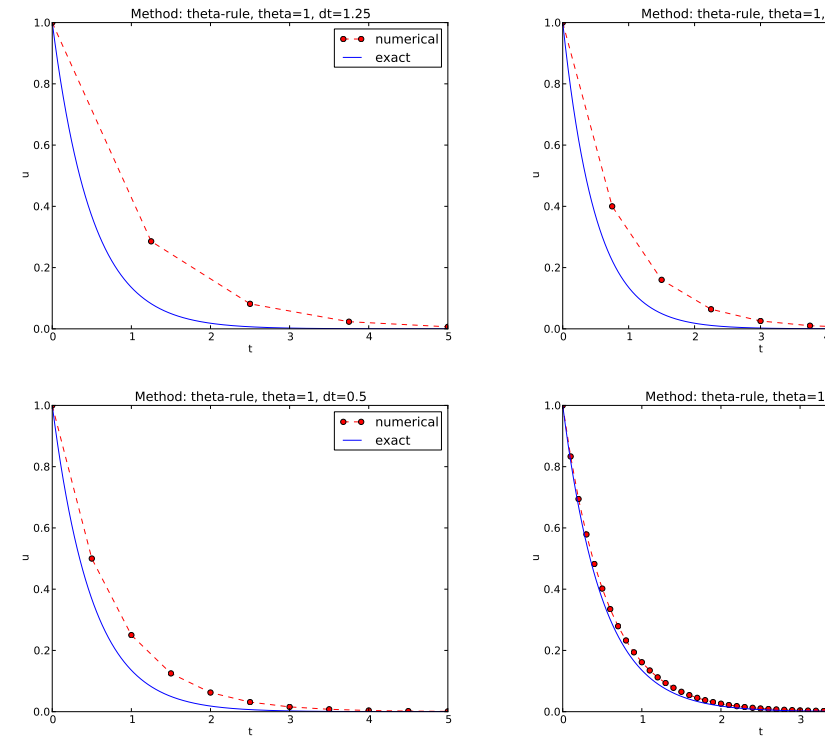


Figure 9: Backward Euler.

**Discouraging numerical solutions.** Choosing  $I = 1$ ,  $a = 2$ , and running `explore` with  $\Delta t = 1, 0.5, 0$  for  $\Delta t = 1.25, 0.75, 0.5, 0.1$ , gives the results in Figures 9, 10, and 11. The characteristics of the displayed curves can be summarized as follows:

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

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

<sup>16</sup>[http://tinyurl.com/jvzzcfn/decay/decay\\_plot\\_mpl.py](http://tinyurl.com/jvzzcfn/decay/decay_plot_mpl.py)



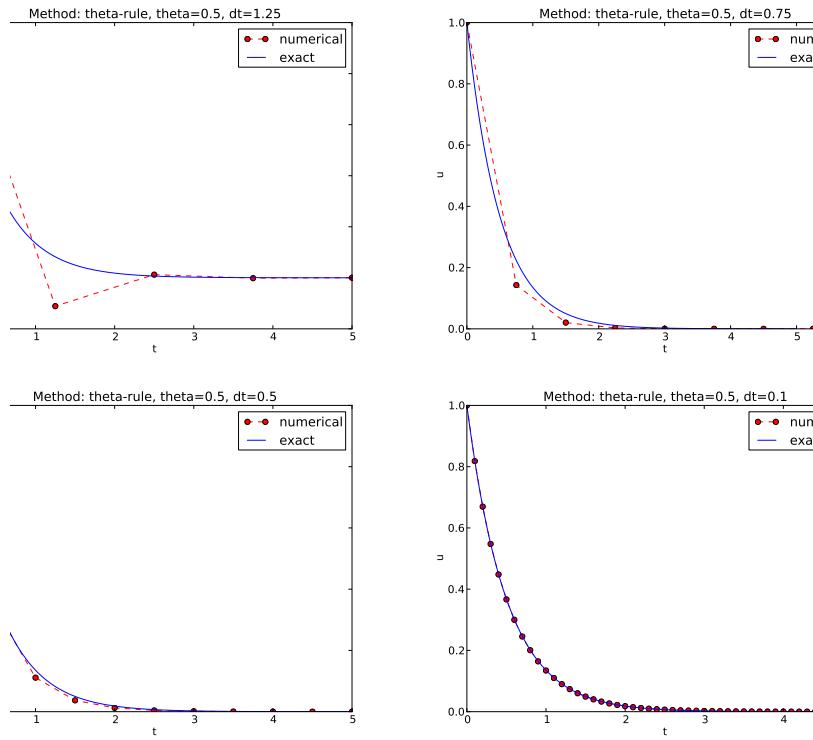


Figure 10: Crank-Nicolson.

the question

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

This question will be investigated both by numerical experiments and by precise mathematical analysis.

The latter will help establish general criteria on  $\Delta t$  for avoiding non-physical or growing solutions.

Another question to be raised is

How does  $\Delta t$  impact the error in the numerical solution?

For a simple model problem we can answer this question very precisely, but we will stick to simplified formulas for small  $\Delta t$  and touch upon important concepts such as *stability*, *accuracy*, *convergence rate* and *the order of a scheme*. Other fundamental concepts mentioned are *consistency*, *stability*, and *convergence*.

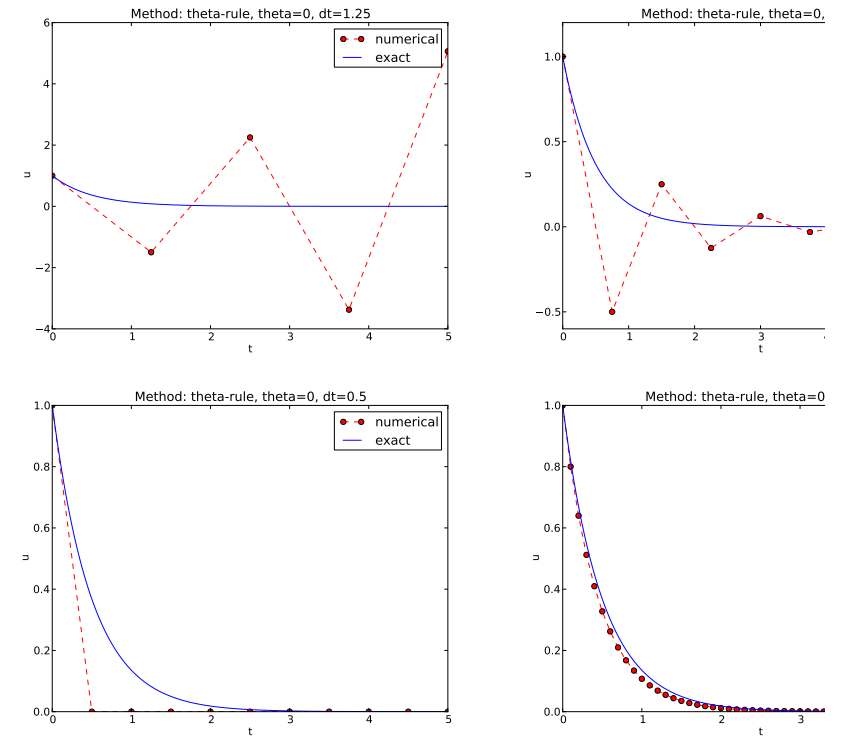


Figure 11: Forward Euler.

## 1.1 Experimental investigation of oscillatory solutions

To address the first question above, we may set up an experiment where we loop over different values of  $a$ ,  $\Delta t$ , and  $I$ . For each experiment, we flag the solution as oscillatory if

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

for some value of  $n$ , since we expect  $u^n$  to decay with  $n$ , but oscillations make it increase at some step. We will quickly see that oscillations are independent of  $I$ , but do depend on  $a$  and  $\Delta t$ . Therefore, we introduce a two-dimensional function  $B(a, \Delta t)$  which is 1 if oscillations occur and 0 otherwise. We can visualize  $B$  as a contour plot (lines for which  $B = \text{constant}$ ). The value  $B = 0.5$  corresponds to the borderline between oscillatory regions with  $B = 1$  and non-oscillatory regions with  $B = 0$  in the  $a, \Delta t$  plane.

The  $B$  function is defined at discrete  $a$  and  $\Delta t$  values. Say we have given  $a_0, \dots, a_{P-1}$ , and  $Q$   $\Delta t$  values,  $\Delta t_0, \dots, \Delta t_{Q-1}$ . These  $a_i$  and  $\Delta t_j$  values,  $i = 0, \dots, P-1$ ,  $j = 0, \dots, Q-1$ , form a rectangular mesh of  $P \times Q$  points in the plane. At each point we associate the corresponding value of  $B(a_i, \Delta t_j)$ , denoted  $B_{ij}$ . The  $B_{ij}$  values are stored in a two-dimensional array. We can thereafter create a plot of the contour

he oscillatory and monotone regions. The file `decay_osc_regions.py`<sup>17</sup> (`osc_1` or "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 void the undesired oscillations. This limit seems to be about 2 for Crank-Nicolsc rd Euler. We shall now establish a precise mathematical analysis of the discret explain the observations in our numerical experiments.

### Exact 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.

---

<sup>17</sup>[tinyurl.com/jvzzcfn/decay/decay\\_osc\\_regions.py](http://tinyurl.com/jvzzcfn/decay/decay_osc_regions.py)

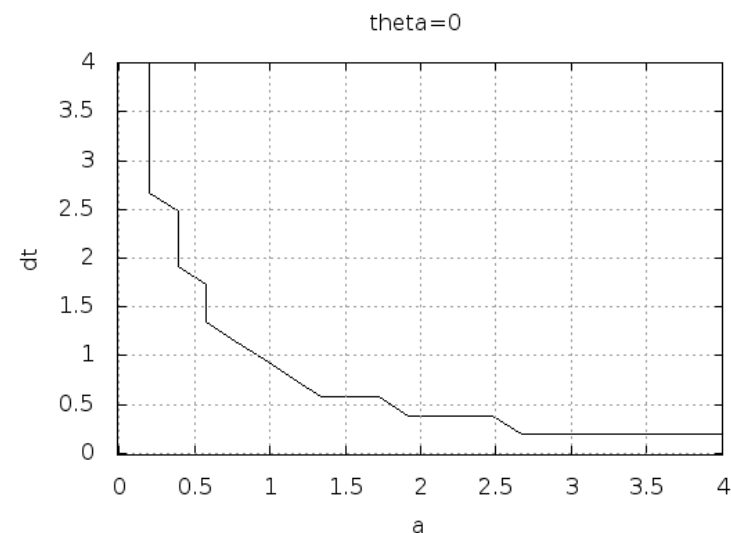


Figure 12: Forward Euler scheme: oscillatory solutions occur for points above

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

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

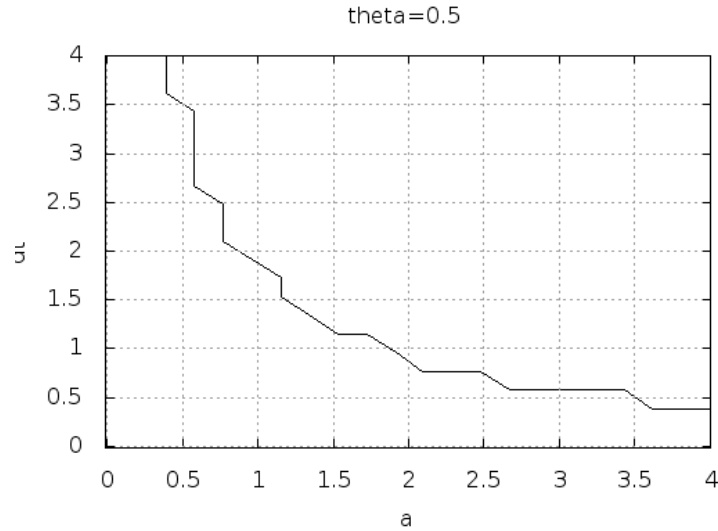
This is a solution technique of wider applicability than repeated use of the recursion.

Regardless of the solution approach, we have obtained a formula for  $u^n$ . To explain everything what we see in the figures above, but it also gives us a more accurate and stability properties of the three schemes.

### 3 Stability

Since  $u^n$  is a factor  $A$  raised to an integer power  $n$ , we realize that  $A < 0$  will imply  $u^n < 0$  and for even power result in  $u^n > 0$ . That is, the solution oscillates at mesh 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 curve.

$A > 0$  is a requirement for having a numerical solution with the same basic frequency as the exact solution, we may say that  $A > 0$  is a *stability criterion*. Expressing  $\Delta t$  the stability criterion reads

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

The backward Euler scheme is always stable since  $A < 0$  is impossible for  $\theta = 1$ . Computing solutions for Forward Euler and Crank-Nicolson demand  $\Delta t \leq 1/a$  and  $\Delta t \leq 2/a$  respectively. The relation between  $\Delta t$  and  $a$  looks reasonable: a larger  $a$  means faster decay and thus needs smaller time steps.

Looking at Figure 11, we see that with  $a\Delta t = 2 \cdot 1.25 = 2.5$ ,  $A = -1.5$ , and the numerical solution  $u^n = (-0.5)^n$  oscillates and grows. With  $a\Delta t = 2 \cdot 0.75 = 1.5$ ,  $A = -0.5$ ,  $u^n = (-0.5)^n$  oscillates and decays. The peculiar case  $\Delta t = 0.5$ , where the Forward Euler scheme produces a lock on the  $t$  axis, corresponds to  $A = 0$  and therefore  $u^0 = I = 1$  and  $u^n = 0$  for  $n > 0$ . The growing oscillations in the Crank-Nicolson scheme for  $\Delta t = 1.25$  are easily explained by  $A \approx -0.11 < 0$ .

The amplification factor  $A$  is called the *amplification factor* since the solution at a new time level is computed from the solution at the previous time level. For a decay process, we must obviously have  $|A| < 1$  for all  $\Delta t$  if  $\theta \geq 1/2$ . Arbitrarily large values of  $u$  can be generated when  $|A| > 1$ . The numerical solution is in such cases totally irrelevant to the exact solution of a decay process! To avoid this situation, we must for  $\theta < 1/2$  have

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

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

### Stability properties.

We may summarize the stability investigations as follows:

1. The Forward Euler method is a *conditionally stable* scheme because it requires  $\Delta t < 1/a$  for avoiding growing solutions and  $\Delta t < 2/a$  for avoiding oscillatory solutions.
2. The Crank-Nicolson is *unconditionally stable* with respect to growing solutions, but it is conditionally stable with the criterion  $\Delta t < 2/a$  for avoiding oscillations.
3. The Backward Euler method is unconditionally stable with respect to growing solutions and oscillatory solutions - any  $\Delta t$  will work.

Much literature on ODEs speaks about L-stable and A-stable methods. In our case, the Backward Euler method ensures non-growing solutions, while L-stable methods also avoid oscillations.

## 4 Comparing amplification factors

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

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

From this formula we see that the exact amplification factor is

$$A_e = e^{-a\Delta t}.$$

We realize that the exact and numerical amplification factors depend on the product  $a\Delta t$ . Therefore, it is convenient to introduce a symbol for this product,  $p$ . We define  $p = a\Delta t$ . Then we can view  $A$  and  $A_e$  as functions of  $p$ . Figure 14 shows these functions. Crank-Nicolson is the most accurate method, but that method has the unfortunate oscillations when  $p > 2$ .

## 5 Series expansion of amplification factors

As an alternative to the visual understanding inherent in Figure 14, there is a more analytical way to establish formulas for the approximation errors when the time step, here  $\Delta t$ , becomes small. In the present case we let  $p$  be our small parameter, and it makes sense to simplify the expressions for  $A$  and  $A_e$  by using Taylor series around  $p = 0$ . The Taylor polynomials are accurate for small  $p$  and greatly simplify the comparison of the analytical expressions since we then can compare polynomial approximations.

Calculating the Taylor series for  $A_e$  is easily done by hand, but the three values  $p = 0, 1, \frac{1}{2}$  lead to more cumbersome calculations. Nowadays, analytical computations

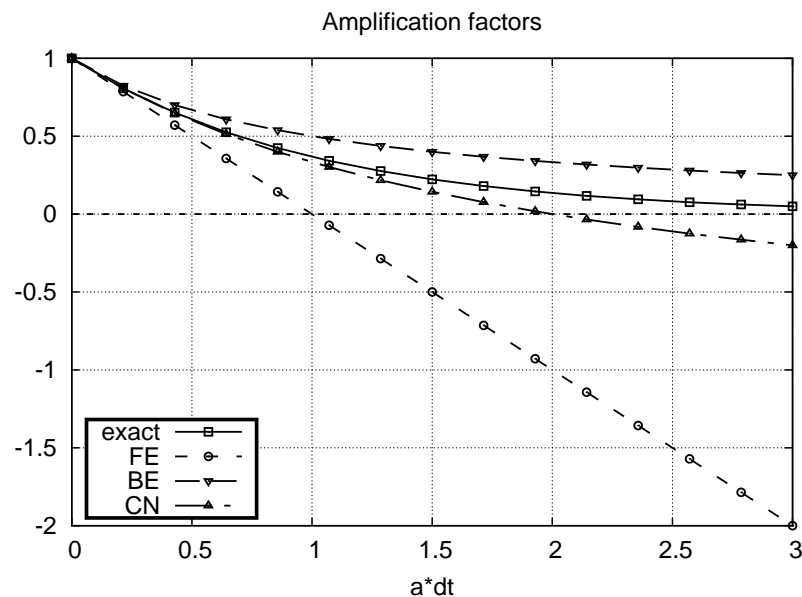


Figure 14: Comparison of amplification factors.

symbolic computer algebra software. The Python package `sympy` represents a computer algebra system, not yet as sophisticated as the famous Maple and Mathematica, but very easy to integrate with our numerical computations in Python.

Using `sympy`, it is convenient to enter the interactive Python mode where we can execute commands and statements and immediately see the results. Here is a simple example. We recommend to use `isympy` (or `ipython`) for such interactive sessions.

We will illustrate `sympy` with a standard Python shell syntax (`>>>` prompt) to compute a numerical approximation to  $e^{-p}$ :

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

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

The `>>>` represent input lines and lines without this prompt represents the results (note that `isympy` and `ipython` apply other prompts, but in this text we use the standard Python prompts for interactive Python computing). Apart from the order of the powers, the coefficients are easily recognized as the beginning of the Taylor series for  $e^{-p}$ .

We will now define the numerical amplification factor where  $p$  and  $\theta$  enter the formula as sym-

```
>>> 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 value

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

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

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

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

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

We are now in a position to compare Taylor series:

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

From these expressions we see that the error  $A - A_e \sim \mathcal{O}(p^2)$  for the Forward and Backward Euler schemes, while  $A - A_e \sim \mathcal{O}(p^3)$  for the Crank-Nicolson scheme. It is the *leading* term, the term of the lowest order (polynomial degree), that is of interest, because it is (much) bigger than the higher-order terms (think of  $p = 0.01$ :  $p$  is a hundred times smaller than  $p^2$ ).

Now,  $a$  is a given parameter in the problem, while  $\Delta t$  is what we can vary. We usually write the error expressions in terms of  $\Delta t$ . 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 factor that is half the two other schemes are of order  $\Delta t^2$  in the same quantity. What is the order expression? If we halve  $\Delta t$ , the error in amplification factor at  $t$  is reduced by a factor of 4 in the Forward and Backward Euler schemes, and by a factor of 8 in the Crank-Nicolson scheme. That is, as we reduce  $\Delta t$  to obtain more accurate results, the Crank-Nicolson scheme reduces the error more efficiently than the other schemes.

## 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  $A - A_e$ .

## 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  $p \rightarrow 0$  since  $t = n\Delta t$ .  
 $n$  must increase like  $\Delta t^{-1}$ . With  $n$  substituted by  $t/\Delta t$  in the leading-order  
ese become  $\frac{1}{2}na^2\Delta t^2 = \frac{1}{2}ta^2\Delta t$  for the Forward and Backward Euler sch  
=  $\frac{1}{12}ta^3\Delta t^2$  for the Crank-Nicolson scheme. The global error is therefore of  
 $\Delta t$ ) for the latter scheme and of first order for the former schemes.

the global error  $e^n \rightarrow 0$  as  $\Delta t \rightarrow 0$ , we say that the scheme is *convergent*. If  
numerical solution approaches the exact solution as the mesh is refined, and  
ired property of a numerical method.

## egrated errors

non to study the norm of the numerical error, as explained in detail in Section 4.  
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 so  
nse to compute the  $\ell^2$  error

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

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

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

ow,  $\sum_{n=0}^{N_t} n^2 \approx \frac{1}{3} N_t^3$ . Using this approximation, setting  $N_t = T/\Delta t$ , and 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  
ie 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  $\Delta t$  for the Forward Euler and Ba  
schemes.

## .9 Truncation error

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

illustrate the calculation of the truncation error for the Forward Euler scheme 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  $u_e(t)$  fulfills this equation. Since  $u_e(t)$  in the discrete equation, error in the discrete equation, called a *residual*, denoted

$$R^n = \frac{u_e(t_{n+1}) - u_e(t_n)}{\Delta t} + au_e(t_n).$$

is defined at each mesh point and is therefore a mesh function with a superinteresting feature of  $R^n$  is to see how it depends on the discretization parameter. One way for reaching this goal is to Taylor expand  $u_e$  around the point where the difference equation is supposed to hold, here  $t = t_n$ . We have that

$$u_e(t_{n+1}) = u_e(t_n) + u'_e(t_n)\Delta t + \frac{1}{2}u''_e(t_n)\Delta t^2 + \dots$$

this Taylor series in (55) gives

$$R^n = u'_e(t_n) + \frac{1}{2}u''_e(t_n)\Delta t + \dots + au_e(t_n).$$

fulfills the ODE  $u'_e = -au_e$  such that the first and last term cancels and we have

$$R^n \approx \frac{1}{2}u''_e(t_n)\Delta t.$$

is the *truncation error*, which for the Forward Euler is seen to be of first order. The above procedure can be repeated for the Backward Euler and the Crank-Nicolson scheme. In operator notation, write it out in detail, Taylor expand  $u_e$  around  $\tilde{t}$  at which the difference equation is defined, collect terms that correspond to the truncation error ( $\cdot au_e$ ), and identify the remaining terms as the residual  $R$ , which is the truncation error of the scheme. The Forward Euler scheme leads to

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

the Crank-Nicolson scheme gives

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

The order  $r$  of a finite difference scheme is often defined through the leading term of the truncation error. The above expressions point out that the Forward and Backward Euler schemes are of first order, while Crank-Nicolson is of second order. We have looked at other error measures in other sections, like the error in amplification factor and the error  $e^n = u_e(t_n) - u^n$ . The truncation error is more straightforward than deriving the expressions for the error measures and therefore the easiest way to establish the order of a scheme.

## 10 Consistency, stability, and convergence

Three fundamental concepts when solving differential equations by numerical methods are consistency, stability, and convergence. We shall briefly touch these concepts below the present model problem.

Consistency means that the error in the difference equation, measured through the truncation error, goes to zero as  $\Delta t \rightarrow 0$ . Since the truncation error tells how well the exact solution fulfills the difference equation, and the exact solution fulfills the differential equation, consistency means that the difference equation approaches the differential equation in the limit. The truncation errors in the previous section are all proportional to  $\Delta t$  or  $\Delta t^2$ , hence  $\Delta t \rightarrow 0$ , and all the schemes are consistent. Lack of consistency implies that a numerical solution does not approach the exact solution in the limit  $\Delta t \rightarrow 0$  than we aim at.

Stability means that the numerical solution exhibits the same qualitative properties as the exact solution. This is obviously a feature we want the numerical solution to have. For an exponential decay model, the exact solution is monotone and decaying. An increasing numerical solution is not in accordance with the decaying nature of the exact solution and we can also say that an oscillating numerical solution lacks the property of monotonicity. A numerical solution that is also unstable. We have seen that the Backward Euler scheme is monotone and decaying solutions, regardless of  $\Delta t$ , and is hence stable. The Forward Euler scheme can lead to increasing solutions and oscillating solutions if  $\Delta t$  is too large. The Crank-Nicolson scheme is stable unless  $\Delta t$  is sufficiently small. The Crank-Nicolson scheme can never lead to instability and has no problem to fulfill that stability property, but it can produce oscillations if  $\Delta t$  is not sufficiently small, unless  $\Delta t$  is sufficiently small.

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

Convergence is hard to establish theoretically, except in quite simple problems. Stability and consistency are much easier to calculate. A major milestone in the understanding of numerical methods for differential equations came in 1956 when Richtmyer established equivalence between convergence on one hand and stability on the other (the Lax equivalence theorem<sup>18</sup>). In practice it means that if a method is stable and consistent, and then it is automatically convergent (much harder to establish). The result holds for linear problems only, and in the case of nonlinear differential equations the relations between consistency, stability, and convergence are more complicated.

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

## Exercises

### Exercise 1: Visualize the accuracy of finite differences $u = e^{-at}$

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

<sup>18</sup>[http://en.wikipedia.org/wiki/Lax\\_equivalence\\_theorem](http://en.wikipedia.org/wiki/Lax_equivalence_theorem)

pecific

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

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

```
py import logspace, exp
py plt.plot(logspace(-6, 1, 101),
py exp(-1)/p,
py p, 'y')
```

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

Find the Taylor series expansions of the error fractions and find the leading order  $r$  in the error 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 $\theta$ -rule for exponential growth

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

Perform experiments with  $\theta = 0, 0.5, 1$  for various values of  $\Delta t$  to uncover numerical errors. The exact solution is a monotone, growing function when  $a < 0$ . Oscillations or wrong growth are signs of wrong qualitative behavior, which can be taken as a criterion.

Use the insight to select a few values of  $\Delta t$  and  $a$  fixed that demonstrate all types of numerical errors for the three different schemes ( $\theta = 0, 0.5, 1$ ).

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

`growth_exper.py`.

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

Modify the `decay_ampf_plot.py`<sup>19</sup> code.

`growth_ampf.py`.

Write a scientific report about the findings.

Use the examples from Section ?? to see how scientific reports can be written.

See: `growth_exper.pdf`, `growth_exper.html`.

[https://tinyurl.com/jvzzcfn/decay/decay\\_ampf\\_plot.py](https://tinyurl.com/jvzzcfn/decay/decay_ampf_plot.py)

## Model extensions

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

### 1.1 Generalization: including a variable coefficient

In the ODE for decay,  $u' = -au$ , we now consider the case where  $a$  depends on  $t$ :

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

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

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

The Backward Euler scheme becomes

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

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

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

Alternatively, we can use an average for the product  $au$ :

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

The  $\theta$ -rule unifies the three mentioned schemes. One version is to have  $a$  evaluated at  $t_{n+\theta}$ :

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

Another possibility is to apply a weighted average for the product  $au$ ,

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

With the finite difference operator notation the Forward Euler and Backward Euler schemes can be summarized as

$$\begin{aligned} [D_t^+ u] &= -au^n, \\ [D_t^- u] &= -au^n. \end{aligned}$$

The Crank-Nicolson and  $\theta$  schemes depend on whether we evaluate  $a$  at the same time as  $u$  or if we use an average. The various versions are written as

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

## Generalization: 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 evaluated. In the Runge-Kutta-Nicolson scheme and the  $\theta$ -rule we have a choice of whether to evaluate  $a(t)$  at the next point or use an average. The chosen strategy becomes particularly clear if we write the schemes in the operator notation:

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

## Implementation of the generalized model problem

**$\theta$ -rule formula.** Writing out the  $\theta$ -rule in (75), using (32) and (33), we

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

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

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

**Python code.** Here is a suitable implementation of (76) where  $a(t)$  and  $b(t)$  are Python functions:

```
def solver(I, a, b, T, dt, theta):
    # solve u' = -a(t)*u + b(t), u(0)=I,
    # t in (0,T] with steps of dt.
    # a, b are Python functions of t.

    Nt = int(round(T/dt))
    # avoid integer division
    # no of time intervals
    # adjust T to fit time step dt
    # array of u[n] values
    # time mesh

    u = zeros(Nt+1)
    t = linspace(0, T, Nt+1)

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

    return u, t
```

A solution is found in the file `decay_vc.py`<sup>20</sup> (vc stands for "variable coefficients").

[/tinyurl.com/jvzzcfn/decay/decay\\_vc.py](https://tinyurl.com/jvzzcfn/decay/decay_vc.py)

**Loading of variable coefficients.** The solver function shown above demands  $a$  and  $b$  to be Python functions of time  $t$ , say

```
def a(t):
    return a_0 if t < tp else k*a_0

def b(t):
    return 1
```

Here,  $a(t)$  has three parameters  $a_0$ ,  $tp$ , and  $k$ , which must be global variables. One implementation is to represent  $a$  by a class where the parameters are attributes and the method `__call__` evaluates  $a(t)$ :

```
class A:
    def __init__(self, a0=1, k=2):
        self.a0, self.k = a0, k

    def __call__(self, t):
        return self.a0 if t < self.tp else self.k*self.a0

a = A(a0=2, k=1) # a behaves as a function a(t)
```

For quick tests it is cumbersome to write a complete function or a class. The construction in Python is then convenient. For example,

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

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

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

is equivalent to

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

One 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)
```

A lambda function can appear anywhere where a variable can appear.

## 4 Verifying a constant solution

A very useful partial verification method is to construct a test problem with a very simple solution, usually  $u = \text{const}$ . Especially the initial debugging of a program code can benefit from such tests, because 1) all relevant numerical methods will exactly reproduce a constant solution, 2) many of the intermediate calculations are easy to control for a constant  $u$ , and 3) a constant  $u$  can uncover many bugs in an implementation.

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



```

ose.tools as nt

_constant_solution():
    """
    problem where u=u_const is the exact solution, to be
    reproduced (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

nsteps = 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)
difference = abs(u_e - u).max() # max deviation
nt.assert_almost_equal(difference, 0, places=14)

```

Interesting question is what type of bugs that will make the computed  $u^n$  deviate from the exact solution  $C$ . Fortunately, the updating formula and the initial condition must be able to reproduce the exact solution. Any attempt to make a wrong indexing in terms like  $a(t[n])$  or to introduce an erroneous factor in the formula creates a solution that is different from the exact solution.

## Verification via manufactured solutions

In the idea of the previous section, we can choose any formula as the exact solution, fit the data  $a(t)$ ,  $b(t)$ , and  $I$  to make the numerical solution fulfill the equation. This powerful technique for generating exact solutions is very useful for verification purposes and known as the *method of manufactured solutions*, often abbreviated as MMS.

A common choice of solution is a linear function in the independent variable (e.g.,  $u = ct + d$ ). Behind such a simple variation is that almost any relevant numerical solution method for ordinary differential equation problems is able to reproduce the linear function exactly to machine precision (e.g.,  $u = ct + d$  out of unity in size; precision is lost if  $u$  take on large values, see Exercise 3). This also makes some stronger demands to the numerical method and the implementation. A constant solution used in Section 5.4, at least in more complicated applications. The manufactured solution is often ideal for initial debugging before proceeding with a linear solution. Choose a linear solution  $u(t) = ct + d$ . From the initial condition it follows that the exact solution  $u$  in the ODE results in

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

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

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

With this choice of  $b(t)$  there are no restrictions on  $a(t)$  and  $c$ .

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

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

Clearly, all three finite difference approximations to the derivative are exact for the linear function counterpart  $u^n = t_n$ .

The difference equation for the Forward Euler scheme

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

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

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

which is always fulfilled. Similar calculations can be done for the Backward Euler scheme and the  $\theta$ -rule for that matter. In all cases,  $u^n = ct_n + I$  is an exact solution of the discrete equations. That is why we should expect that  $u^n - u_e(t_n) = 0$  modulo machine precision for  $n = 0, \dots, N$ .

The following function offers an implementation of this verification test for the Forward Euler scheme:

```

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)

```

Any error in the updating formula makes this test fail!

Choosing more complicated formulas as the exact solution, say  $\cos(t)$ , will not work. The numerical and exact solution coincide to machine precision, because finite difference approximations to the derivative are exact for the linear function counterpart.

exactly yield the exact derivative  $-\sin(t)$ . In such cases, the verification procedure measuring the convergence rates as exemplified in Section ???. Convergence is guaranteed as long as one has an exact solution of a problem that the solver can be tested against, which can always be obtained by the method of manufactured solutions.

## Extension to systems of ODEs

Models involving more than one unknown function and more than one equation comprise a system of two unknown functions  $u(t)$  and  $v(t)$ :

$$\begin{aligned} u' &= au + bv, \\ v' &= cu + dv, \end{aligned}$$

where  $a, b, c, d$ . Applying the Forward Euler method to each equation results in the following formula

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

Alternatively, the Crank-Nicolson or Backward Euler schemes result in a  $2 \times 2$  linear system of unknowns. The latter scheme gives

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

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

$$\begin{aligned} (1 - \Delta ta)u^{n+1} + \Delta tbv^{n+1} &= u^n, \\ \Delta tcu^{n+1} + (1 - \Delta td)v^{n+1} &= v^n, \end{aligned}$$

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

## General first-order ODEs

Turn the attention to general, nonlinear ODEs and systems of such ODEs. Our focus is on methods that can be readily reused for time-discretization PDEs, and diffusion problems. The methods are just briefly listed, and we refer to the rich literature for descriptions and analysis - the books [6, 1, 2, 3] are all excellent resources on numerical methods for ODEs. We also demonstrate the Odespy Python interface to a range of codes for general first-order ODE systems.

### Generic form

The system is commonly written in the generic form

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

where  $f(u, t)$  is some prescribed function. As an example, our most general example (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 function of  $t$  in case of a *system of ODEs* with  $m$  unknown components:

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

In that case, the right-hand side is a vector-valued function with  $m$  components,

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

Actually, any system of ODEs can be written in the form (89), but higher-order equations are reduced to 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 equation and systems of ODEs (vector  $u$ ). The choice of methods is inspired by the kind of problem, but is also popular also for partial differential equations.

## 2.2 The $\theta$ -rule

The  $\theta$ -rule scheme applied to  $u' = f(u, t)$  becomes

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

Bringing the unknown  $u^{n+1}$  to the left-hand side and the known terms on the right-hand side gives

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

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

## 2.3 An implicit 2-step backward scheme

The implicit backward method with 2 steps applies a three-level backward differentiation to  $u'(t)$ ,

$$u'(t_{n+1}) \approx \frac{3u^{n+1} - 4u^n + u^{n-1}}{2\Delta t},$$

which is an approximation of order  $\Delta t^2$  to the first derivative. The resulting scheme reads

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

der versions of the scheme (92) can be constructed by including more time levels. We know them as the Backward Differentiation Formulas (BDF), and the particular one often referred to as BDF2.

That the scheme (92) is implicit and requires solution of nonlinear equations within  $u$ . The standard 1st-order Backward Euler method or the Crank-Nicolson method is used for the first step.

## Leapfrog schemes

**Implicit Leapfrog scheme.** The derivative of  $u$  at some point  $t_n$  can be approximated as the central difference over two time steps,

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

an approximation of second order in  $\Delta t$ . The scheme can then be written as

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

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

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

That (94) is an explicit scheme, and that a nonlinear  $f$  (in  $u$ ) is trivial to handle gives the known  $u^n$  value. Some other scheme must be used as starter to compute  $u^1$  for the Forward Euler scheme since it is also explicit.

**Implicit Leapfrog scheme.** Unfortunately, the Leapfrog scheme (94) will develop oscillations with time (see Problem 8)!!! A remedy for such undesired oscillations is to introduce a *damping technique*. First, a standard Leapfrog step is taken, according to (94), and then the  $u^n$  value is adjusted according to

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

Oscillations will effectively damp in the solution, especially those with short wavelengths (near-to-point oscillations). A common choice of  $\gamma$  is 0.6 (a value used in the famous *Model*).

## The 2nd-order Runge-Kutta scheme

Implicit scheme

$$\begin{aligned} u^* &= u^n + \Delta t f(u^n, t_n), \\ u^{n+1} &= u^n + \Delta t \frac{1}{2} (f(u^n, t_n) + f(u^*, t_{n+1})), \end{aligned}$$

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

## 6 A 2nd-order Taylor-series method

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

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

From the equation  $u' = f(u, t)$  it follows that the derivatives of  $u$  can be expressed in terms of  $u$  and its derivatives:

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

resulting in the scheme

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

More terms in the series could be included in the Taylor polynomial to obtain an order higher than 2.

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

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

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

The scheme is explicit and requires another one-step scheme to compute  $u^1$  (the Forward Euler scheme or Heun's method, for instance). As the name implies, the scheme is of order 2.

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

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

The numerical error is of order  $\Delta t^3$ , and the scheme needs some method for computing  $u^1$ .

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

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

where  $\beta_j$  are known coefficients.

## 8 4th-order Runge-Kutta scheme

The perhaps most widely used method to solve ODEs is the 4th-order Runge-Kutta method, also called RK4. Its derivation is a nice illustration of common numerical approximation techniques. We go through the steps in detail.

The starting point is to integrate the ODE  $u' = f(u, t)$  from  $t_n$  to  $t_{n+1}$ :

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

to compute  $u(t_{n+1})$  and regard  $u(t_n)$  as known. The task is to find good approximation of the integral, since the integrand involves the unknown  $u$  between  $t_n$  and  $t_{n+1}$ . The integral can be approximated by the famous Simpson's rule<sup>21</sup>:

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

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

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

where  $\hat{f}^{n+\frac{1}{2}}$ ,  $\tilde{f}^{n+\frac{1}{2}}$ , and  $\bar{f}^{n+1}$  are approximations to  $f^{n+\frac{1}{2}}$  and  $f^{n+1}$  that can be based on quantities computed at  $t_n$ . For  $\hat{f}^{n+\frac{1}{2}}$  we can apply an approximation to  $u^{n+\frac{1}{2}}$  using the Forward Euler method with step  $\frac{1}{2}\Delta t$ :

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

This gives us a prediction of  $f^{n+\frac{1}{2}}$ , we can for  $\tilde{f}^{n+\frac{1}{2}}$  try a Backward Euler method to approximate  $u^{n+\frac{1}{2}}$ :

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

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

$$\bar{f}^{n+1} = f\left(u^n + \Delta t \bar{f}^{n+1}, t_{n+1}\right).$$

By now we have used the Forward and Backward Euler methods as well as the Crank-Nicolson method in the context of Simpson's rule. The hope is that the combination of these methods gives an overall time-stepping scheme from  $t_n$  to  $t_{n+1}$  that is much more accurate than the  $\mathcal{O}(\Delta t^2)$  of the individual steps. This is indeed true: the overall accuracy is  $\mathcal{O}(\Delta t^4)$ . To summarize, the 4th-order Runge-Kutta method becomes

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

where the quantities on the right-hand side are computed from (101)-(103). Note that the method is implicit so there is never any need to solve linear or nonlinear algebraic equations. If the function  $f$  is conditional and depends on  $f$ . There is a whole range of *implicit* Runge-Kutta methods.

<sup>21</sup>[https://en.wikipedia.org/wiki/Simpson's\\_rule](https://en.wikipedia.org/wiki/Simpson's_rule)

methods that are unconditionally stable, but require solution of algebraic equations at each time step.

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

## 9 The Odespy software

A wide range of the methods and software exist for solving (89). Many of them are available through a unified Python interface offered by the Odespy<sup>22</sup> package. Odespy provides Python implementations of the most fundamental schemes as well as Python interfaces to various packages for solving ODEs: ODEPACK<sup>23</sup>, Vode<sup>24</sup>, rkcf<sup>25</sup>, rkf45<sup>26</sup>, Radau<sup>27</sup>, and the ODE solvers in SciPy<sup>28</sup>, SymPy<sup>29</sup>, and odelab<sup>30</sup>.

The usage of Odespy follows this setup for the ODE  $u' = -au$ ,  $u(0) = I$ , solved 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  $\theta$ -rule: `ThetaRule`
- special cases of the  $\theta$ -rule: `ForwardEuler`, `BackwardEuler`, `CrankNicolson`
- the 2nd- and 4th-order Runge-Kutta methods: `RK2` and `RK4`
- The BDF methods and the Adam-Bashforth methods: `Vode`, `Lsode`, `Lsode`
- The Leapfrog scheme: `Leapfrog` and `LeapfrogFiltered`

## 10 Example: Runge-Kutta methods

Since all solvers have the same interface in Odespy, modulo different set of parameters, one can easily make a list of solver objects and run a loop (or a list of solvers). The code below, found in complete form in `decay_odespy.py`<sup>31</sup>, compares various Runge-Kutta methods of orders 2, 3, and 4 with the exact solution of the ODE  $u' = -au$ . Since we have quite long time steps, we have included the only relevant method, the Backward Euler scheme ( $\theta = 1$ ), as well. Figure 15 shows the re-

<sup>22</sup><https://github.com/hplgit/odespy>

<sup>23</sup>[https://computation.llnl.gov/casc/odepack/odepack\\_home.html](https://computation.llnl.gov/casc/odepack/odepack_home.html)

<sup>24</sup>[https://computation.llnl.gov/casc/odepack/odepack\\_home.html](https://computation.llnl.gov/casc/odepack/odepack_home.html)

<sup>25</sup><http://www.netlib.org/ode/rkcf.f>

<sup>26</sup><http://www.netlib.org/ode/rkf45.f>

<sup>27</sup><http://www.unige.ch/haier/software.html>

<sup>28</sup><http://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.ode.html>

<sup>29</sup><http://docs.sympy.org/dev/modules/mpmath/calculus/odes.html>

<sup>30</sup><http://olivierverdiier.github.com/odelab/>

<sup>31</sup>[http://tinyurl.com/jvzzcfn/decay/decay\\_odespy.py](http://tinyurl.com/jvzzcfn/decay/decay_odespy.py)

```

umpy as np
citoools.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 colors and markers, thus making curves easy to distinguish on screen and on black-and-white paper. Using SciTools, curve colors and markers are automatically given, thus making curves easy to distinguish on screen and on black-and-white paper. The automatic adding of markers is normal for a very fine mesh since all the markers get cluttered, but SciTools limits the number of markers in such cases. For the exact solution we use a very fine mesh, but above we specify the line type as a solid line (`-`), which means no markers are added. The color to be automatically determined by the backend used for plotting (Matplotlib or Grace).

note that the legends are based on the class names of the solvers, and the name of a class type (as a string) of an object `obj` is obtained by `obj.__class__.__name__`.

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

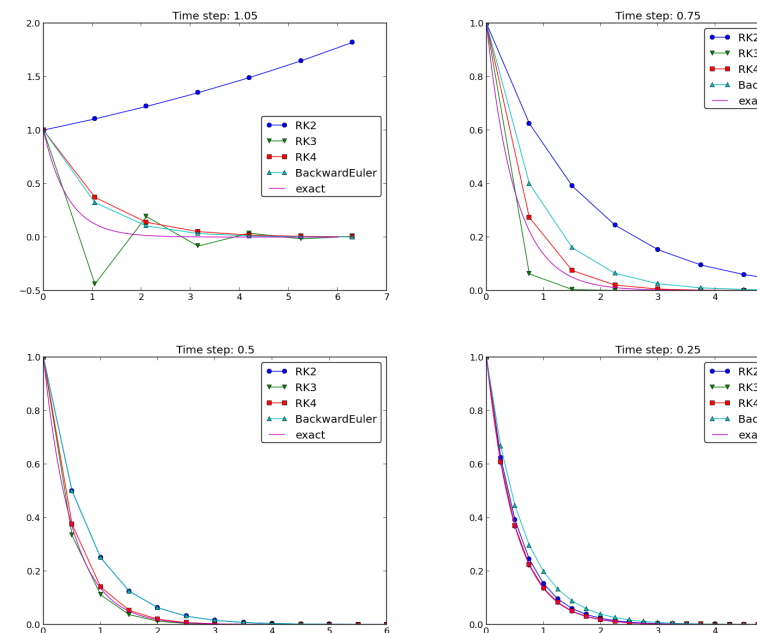


Figure 15: Behavior of different schemes for the decay equation.

Runge-Kutta method approaches the exact solution faster than the Backward Euler scheme. For larger  $\Delta t$ , while the higher order scheme is superior, this is a typical trend also for most schemes for ordinary and partial differential equations.

The 3rd-order Runge-Kutta method (RK3) has also artifacts in form of oscillations for larger  $\Delta t$  values, much like that of the Crank-Nicolson scheme. For finer  $\Delta t$  values, the 3rd-order Runge-Kutta method converges quickly to the exact solution.

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

**Remark about using the  $\theta$ -rule in Odespy.** The Odespy package assumes the equation is written as  $u' = f(u, t)$  with an  $f$  that is possibly nonlinear in  $u$ . The  $\theta$ -rule leads to

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

which is a *nonlinear equation* in  $u^{n+1}$ . Odespy's implementation of the  $\theta$ -rule uses the specialized Backward Euler (`BackwardEuler`) and Crank-Nicolson (`CrankNicolson`) methods for solving the nonlinear equation in  $u^{n+1}$ . This is only if the equation is linear in  $u$ , as in the model problem  $u' = -au$ , where we can easily solve for  $u^{n+1}$ . Therefore, we need to specify use of Newton's method to the equations. (Odespy also allows other methods than Newton's to be used, for instance Picard iteration, but that method is not implemented. The reason is that it applies the Forward Euler scheme to generate a start value for

Euler may give very wrong solutions for large  $\Delta t$  values. Newton's method, on the other hand, is insensitive to the start value in *linear problems*.)

### Example: Adaptive Runge-Kutta methods

Offers solution methods that can adapt the size of  $\Delta t$  with time to match a given tolerance in the solution. Intuitively, small time steps will be chosen in areas where the solution is changing rapidly, while larger time steps can be used where the solution is slowly varying. An *error estimator* is used to adjust the next time step at each time level.

One popular adaptive method for solving ODEs is the Dormand-Prince Runge-Kutta method of order 4 and 5. The 5th-order method is used as a reference solution and the difference between the 4th- and 5th-order methods is used as an indicator of the error in the numerical solution. The Dormand-Prince method is the default choice in MATLAB's widely used `ode45` function.

It is easy to set up Odespy to use the Dormand-Prince method and see how it selects time steps. To this end, we request only one time step from  $t = 0$  to  $t = T$  and then compute the necessary non-uniform time mesh to meet a certain error tolerance like

```

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

def f(t, u):
    return -a*u

def exact_solution(t):
    return I*np.exp(-a*t)

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

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

```

Running four cases with tolerances  $10^{-1}$ ,  $10^{-3}$ ,  $10^{-5}$ , and  $10^{-7}$ , gives the results in Figure 16. As expected, one would expect denser points in the beginning of the decay and larger time steps where the solution flattens out.

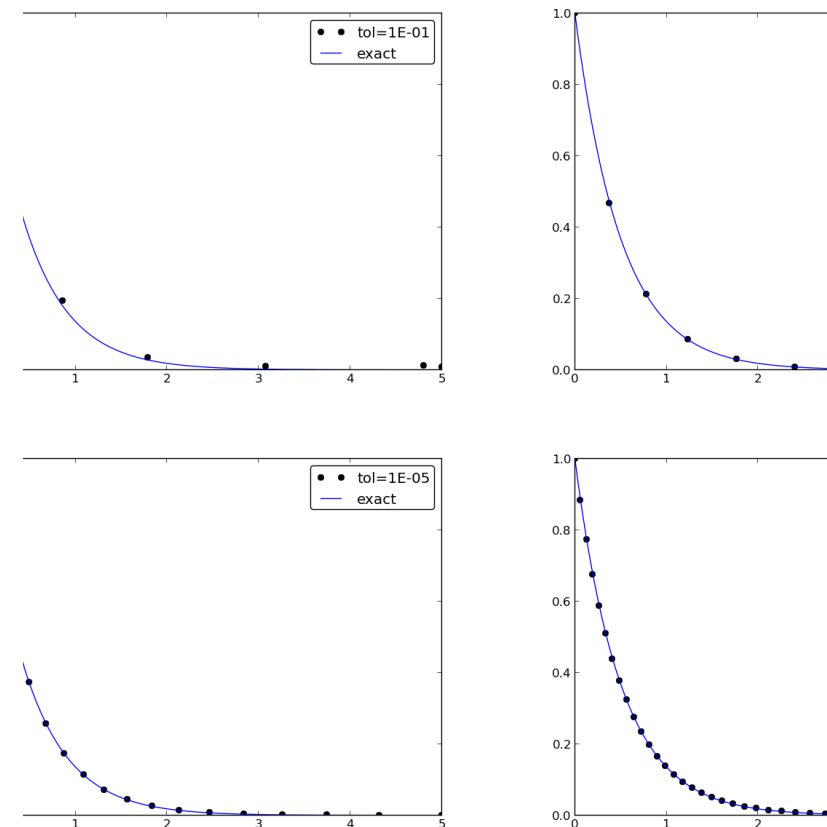


Figure 16: Choice of adaptive time mesh by the Dormand-Prince method for different tolerances.

## Exercises

### Exercise 3: Experiment with precision in tests and the size of the time step

It is claimed in Section 5.5 that most numerical methods will reproduce a linear function to machine precision. Test this assertion using the nose test function `test_linear_decay_vc.py`<sup>32</sup> program. Vary the parameter `c` from very small, via `c=1` to many large values. Print out the maximum difference between the numerical solution and the exact solution. What is the relevant value of the `places` (or `delta`) argument to `nose.tools.assert_almost_equal` in each case? Filename: `test_precision.py`.

<sup>32</sup>[http://tinyurl.com/jvzzcfn/decay/decay\\_vc.py](http://tinyurl.com/jvzzcfn/decay/decay_vc.py)

#### e 4: Implement the 2-step backward scheme

Use the 2-step backward method (92) for the model  $u'(t) = -a(t)u(t) + b(t)$ ,  $u(0) = u_0$ . The first step to be computed by either the Backward Euler scheme or the Crank-Nicolson scheme. Verify the implementation by choosing  $a(t)$  and  $b(t)$  such that the exact solution is known (see Section 5.5). Show mathematically that a linear solution is indeed a solution of the equations.

Compare convergence rates (see Section ??) in a test case  $a = \text{const}$  and  $b = 0$ , with an exact solution, and determine if the choice of a first-order scheme (Backward Euler) has any impact on the overall accuracy of this scheme. The expected error is  $\mathcal{O}(\Delta t^2)$ . Filename: `decay_backward2step.py`.

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

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

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

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

#### e 7: Analyze explicit 2nd-order methods

Compare the schemes (97) and (98) are identical in the case  $f(u, t) = -a(t)u$ , where  $a(t)$  is a constant. Assume that the numerical solution reads  $u^n = A^n$  for some unknown amplification factor  $A$  to be determined. Find  $A$  and derive stability criteria. Can the scheme be used for oscillatory solutions of  $u' = -au$ ? Plot the numerical and exact amplification factor. Filename: `F2_Taylor2.py`.

#### e 8: Implement and investigate the Leapfrog scheme

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

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

The Forward Euler method is needed to compute  $u^1$ . The Forward Euler scheme is a possible candidate to start the Leapfrog scheme for the model equation. Plot the solution in the case  $a = 1$ ,  $\Delta t = 0.01$ ,  $t \in [0, 4]$ . Compare with the exact solution  $u_e(t) = e^{-t}$ .

Verify 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 verify the implementation, and automate the verification through a nose test.

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

```
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 discrete equations. However, if a Crank-Nicolson scheme is used for the first step, a second-order polynomial solution exists.

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

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

) 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 discrete equations has the form  $u^n = A^n$ , where  $A$  is an amplification factor to be determined. Determine  $A$  by inserting  $u^n = A^n$  in the Leapfrog scheme.
2. Compute  $A$  either by hand and/or with the aid of `sympy`. The polynomial equation for  $A$  has two 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 approximation.
5. How can  $C_1$  and  $C_2$  be determined?

the original Leapfrog scheme is unconditionally unstable as time grows, it dilution. 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  $\gamma$  can be determined by the filtered Leapfrog scheme and check that it can handle tests where the scheme is unstable.

∴ `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 can be written in the general form

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

choice of  $c_0, \dots, c_m$ . Find expressions for the  $c_j$  coefficients in case of the  $\theta$ -1 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. Assume that we just repeat Crank-Nicolson steps until (105) can be used. Use a subelist  $c_0, \dots, c_m$  for a particular method, and implement subclasses for all the methods. Verify the implementation by testing with a linear solution, which should be satisfied by all methods. Filename: `decay_schemes_oo.py`.

## Applications of exponential decay models

Section 8.8 presents many mathematical models that all end up with ODEs of the type  $u' = -au + b$ . The applications are taken from biology, finance, and physics, and cover population decay, compound interest and inflation, radioactive decay, cooling of objects, concentration media, pressure variations in the atmosphere, and air resistance on falling objects.

### Scaling

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

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

where  $u_m$  is a characteristic value of  $u$ ,  $u_c$  is a characteristic size of the range of  $u$  values,  $t_c$  is a characteristic size of the range of  $t$  where  $u$  varies significantly. Choosing  $u_m$ ,  $u_c$ ,  $t_c$  is often an art in complicated problems. We just state one choice first.

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

Inserting  $u = u_m + u_c \bar{u}$  and  $t = t_c \bar{t}$  in the problem  $u' = -au + b$ , assuming  $a$  and  $b$  are constants, results after some algebra in the *scaled problem*

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

where  $\beta$  is a dimensionless number

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

That is, only the special combination of  $b/(Ia)$  matters, not what the individual  $b$  and  $I$  are. Moreover, if  $b = 0$ , the scaled problem is independent of  $a$  and  $I$ ! This means that we can perform one numerical simulation of the scaled problem solution of any problem for a given  $a$  and  $I$  by stretching the axis in the plot:  $u = u_m + u_c \bar{u}$  or  $b \neq 0$ , we simulate the scaled problem for a few  $\beta$  values and recover the physical solution by translating and stretching the  $u$  axis and stretching the  $t$  axis.

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

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

For  $b = 0$ , 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 problem  $\bar{u}' = -\bar{u}$ , or  $u' = 1 - u$ ,  $u(0) = 0$ . Any implementation of the problem  $u' = -au + b$  can be reused for the scaled problem by setting  $a = 1$ ,  $b = 0$ , and  $I = 1 - \beta$  in the original problem. For one sets  $a = 1$ ,  $b = 1$ , and  $I = 0$  when the physical  $I$  is zero. Falling bodies described in Section 8.8, involves  $u' = -au + b$  with seven physical parameters. The scaled version of the problem if we start the motion from rest!

## 2 Evolution of a population

Let  $N$  be the number of individuals in a population occupying some spatial domain. If  $\bar{t}$  being an integer in this problem, we shall compute with  $N$  as a real number. If  $N$  is a continuous function of time. The basic model assumption is that in a time interval  $\Delta t$ , the number of newcomers to the populations (newborns) is proportional to  $N$ , with a proportionality constant  $\bar{b}$ . The amount of newcomers will increase the population and result in

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

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

If the number of removals from the population (deaths) is also proportional to  $N$ , with a proportionality constant  $d\Delta t$ , the population evolves according to

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

Dividing by  $\Delta t$  and letting  $\Delta t \rightarrow 0$ , we get the ODE



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

lation where the death rate ( $d$ ) is larger than then newborn rate ( $b$ ),  $a > 0$ , then experiences exponential decay rather than exponential growth.

If the populations there is an immigration of individuals into the spatial domain. If individuals are coming in per time unit, the equation for the population change becomes

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

The corresponding ODE reads

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

A simplification arises if we introduce a fractional measure of the population:  $u = N/N_0$ . The ODE problem now becomes

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

where  $r = (b - d)/N_0$  measures the net immigration per time unit as the fraction of the initial population. Very often,  $r$  is approximately constant, but  $f$  is usually a function of time. The growth rate  $r$  of a population decreases if the environment has limited resources. The environment can sustain at most  $N_{\max}$  individuals. We may then assume that the growth rate approaches zero as  $N$  approaches  $N_{\max}$ , i.e., as  $u$  approaches  $M = N_{\max}/N_0$ . The evolution of  $r$  is then a linear function:  $r(t) = r_0(1 - u(t)/M)$ , where  $r_0$  is the growth rate when the population is small relative to the maximum size and there is no immigration.

Using this  $r(t)$  in (108) results in the *logistic model* for the evolution of a population. For the moment that  $f = 0$ :

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

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

## Compound interest and inflation

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

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

If the interest rate is added every day. We therefore introduce a parameter  $m$  of periods per year when the interest is added. If  $n$  counts the periods, we have the differential model for compound interest:

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

This is a *difference equation*, but it can be transformed to a continuous differential equation in the limit process. The first step is to derive a formula for the growth of the investment over a time interval  $\Delta t$ . Starting with an investment  $u^0$ , and assuming that  $r$  is constant

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

Introducing time  $t$ , which here is a real-numbered counter for years, we have the continuous limit process. We can write

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

The second step is to assume *continuous compounding*, meaning that the investment grows continuously. This implies  $m \rightarrow \infty$ , and in the limit one gets the formula

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

which is nothing but the solution of the ODE problem

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

This is then taken as the ODE model for compound interest if  $r > 0$ . However, it applies equally well to inflation, which is just the case  $r < 0$ . One may also take the net growth of an investment, where  $r$  takes both compound interest and inflation into account. Note 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 process of exponential decay according to

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

## 4 Radioactive Decay

An atomic nucleus of an unstable atom may lose energy by emitting ionizing particles and is transformed to a nucleus with a different number of protons and neutrons. This is known as radioactive decay<sup>33</sup>. Actually, the process is stochastic when viewed for a single atom because it is impossible to predict exactly when a particular atom emits a particle. However, with a large number of atoms,  $N$ , one may view the process as deterministic and describe the mean behavior of the decay. Below we reason intuitively about an ODE for the decay. Hereafter, we show mathematically that a detailed stochastic model for single atoms converges to the same mean behavior.

**Deterministic model.** Suppose at time  $t$ , the number of the original atoms is  $N$ . The basic model assumption is that the transformation of the atoms of the original population over some time interval  $\Delta t$  is proportional to  $N$ , so that

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

<sup>33</sup>[http://en.wikipedia.org/wiki/Radioactive\\_decay](http://en.wikipedia.org/wiki/Radioactive_decay)

$\gamma$  is a constant. Introducing  $u = N(t)/N(0)$ , dividing by  $\Delta t$  and letting  $\Delta t \rightarrow 0$  gives the ODE:

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

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

**binomial model.** We have originally  $N_0$  atoms. Each atom may have decayed or survived at time  $t$ . We want to count how many original atoms that are left, i.e., how many have survived. The survival of a single atom at time  $t$  is a random event. Since there are  $N_0$  atoms, survival or decay, we have a Bernoulli trial<sup>34</sup>. Let  $p$  be the probability of that the probability of decay is  $1 - p$ . If each atom survives independently of the others, the probability of survival is the same for every atom, we have  $N_0$  statistically independent Bernoulli trials, known as a *binomial experiment* from probability theory. The probability  $P(N)$  that  $N$  out of  $N_0$  atoms have survived at time  $t$  is then given by the famous *binomial distribution*:

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

The (or expected) value  $E[P]$  of  $P(N)$  is known to be  $N_0 p$ . We want to estimate  $p$ . Let the interval  $[0, t]$  be divided into  $m$  small subintervals of length  $\Delta t = t/m$ . We make the assumption that the probability of decay of a single atom in an interval of length  $\Delta t$  is  $\tilde{p}$ , and that this probability is proportional to  $\Delta t$ :  $\tilde{p} = \lambda \Delta t$  (it sounds natural that the probability of decay increases with  $\Delta t$ ). The corresponding probability of survival is  $1 - \tilde{p}$ . Since  $\lambda$  is independent of time, we have, for each interval of length  $\Delta t$ , a Bernoulli trial: either survives or decays in that interval. Now,  $p$  should be the probability that an atom survives in all the intervals, i.e., that we have  $m$  successful Bernoulli trials in a row:

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

The expected number of atoms of the original type at time  $t$  is

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

To find the relation between the two types of Bernoulli trials and the ODE above, we let  $\Delta t \rightarrow 0$ ,  $m \rightarrow \infty$ . One can show that

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

This is the famous exponential waiting time (or arrival time) distribution for a Poisson process. The theory (obtained here, as often done, as the limit of a binomial experiment) says that the probability of decay,  $1 - e^{-\lambda t}$ , follows an exponential distribution<sup>35</sup>. The limit means that  $\Delta t$  is very small, and  $\tilde{p} = \lambda \Delta t$  is very small since the intensity of the decay is finite. This situation corresponds to a very small probability that an atom decays in a very short time interval, which is a reasonable model. The same model occurs in many other applications, e.g., when waiting for a taxi, or when finding defects along a road.

<sup>34</sup>[en.wikipedia.org/wiki/Bernoulli\\_trial](https://en.wikipedia.org/wiki/Bernoulli_trial)  
<sup>35</sup>[en.wikipedia.org/wiki/Exponential\\_distribution](https://en.wikipedia.org/wiki/Exponential_distribution)

**Relation between stochastic and deterministic models.** With  $p = e^{-\lambda \Delta t}$ , the expected number of original atoms at  $t$  is  $N_0 p = N_0 e^{-\lambda t}$ , which is exactly the ODE model  $N' = -\lambda N$ . This gives also an interpretation of  $a$  via  $\lambda$  or vice versa: the parameter  $a$  appearing here is that the ODE model captures the mean behavior of the underlying stochastic process. This is, however, not always the common relation between microscopic and macroscopic "averaged" models.

Also of interest is to see that a Forward Euler discretization of  $N' = -\lambda N$ ,  $N_{m+1} = N_m (1 - \lambda \Delta t)$  at time  $t_m = m \Delta t$ , which is exactly the expected value of the binomial experiment with  $N_0$  atoms and  $m$  small intervals of length  $\Delta t$ , where each atom has a probability  $\lambda \Delta t$  to decay in an interval.

A fundamental question is how accurate the ODE model is. The underlying process fluctuates around its expected value. A measure of the fluctuations is the standard deviation of the binomial experiment with  $N_0$  atoms, which can be shown to be  $\text{Std}[P] = \sqrt{N_0 p (1 - p)}$ . Compared to the size of the expectation, we get the normalized standard deviation:

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

showing that the normalized fluctuations are very small if  $N_0$  is very large, which is the case.

## 5 Newton's law of cooling

When a body at some temperature is placed in a cooling environment, the temperature of the body falls rapidly in the beginning, and then the changes in temperature become smaller. Newton carried out some experiments with a hot iron ball and found that the temperature evolved as a "geometric progression", meaning that the temperature decayed exponentially. This can be formulated as a differential equation: the rate of change of the temperature is proportional to the temperature difference between the body and its surroundings. This is known as *Newton's law of cooling*, which can be mathematically expressed as

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

where  $T$  is the temperature of the body,  $T_s$  is the temperature of the surroundings, and  $k$  is a positive constant. Equation (133) is primarily viewed as an empirical law: heat is efficiently convected away from the surface of the body by a flowing fluid at constant temperature  $T_s$ . The *heat transfer coefficient*  $k$  reflects the transfer of heat from the body 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 ODE.

## 6 Decay of atmospheric pressure with altitude

The vertical equilibrium of air in the atmosphere is governed by the equation

$$\frac{dp}{dz} = -\rho g.$$

Here,  $p(z)$  is the air pressure,  $\rho$  is the density of air, and  $g = 9.807 \text{ m/s}^2$  is a constant representing the acceleration of gravity. (Equation (116) follows directly from the general equations 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 constant (8.314 J/(mol K)), and  $T$  is the temperature. All variables  $p$ ,  $\varrho$ , and  $T$  vary with the depth  $z$ . Equation (117) in (116) results in an ODE with a variable coefficient:

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

**7.2.1 Atmospheric layers.** The atmosphere can be approximately modeled by seven layers, (118) is applied with a linear temperature of the form

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

where  $h_i$  denotes the bottom of layer number  $i$ , having temperature  $\bar{T}_i$ , and  $L_i$  is a constant temperature lapse rate for layer number  $i$ . The table below lists  $h_i$  (m),  $\bar{T}_i$  (K), and  $L_i$  (K/m) for the layers  $i = 1$  to 7.

$h_i$	$\bar{T}_i$	$L_i$
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

In this section, for simplification it might be convenient to write (118) on the form

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

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

**7.2.2 One-layer model.** One commonly used simplification is to assume that the temperature is constant within each layer. This means that  $L = 0$ .

**7.2.3 One-layer model.** Another commonly used approximation is to use a single layer instead of seven. This one-layer model<sup>36</sup> is based on  $T(z) = T_0 - Lz$ , with sea level temperature  $T_0 = 288$  K and temperature lapse rate  $L = 0.0065$  K/m.

## 8 Compaction of sediments

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

<sup>36</sup>[en.wikipedia.org/wiki/Density\\_of\\_air](https://en.wikipedia.org/wiki/Density_of_air)

Porosity reduces with depth because the weight of the sediments above and causes them to shrink and thereby increase the compaction.

A typical assumption is that the change in  $\phi$  at some depth  $z$  is negatively proportional to  $\phi$ . This assumption leads to the differential equation problem

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

where the  $z$  axis points downwards,  $z = 0$  is the surface with known porosity  $\phi_0$  constant.

The upper part of the Earth's crust consists of many geological layers stacked on top of each other, as indicated in Figure 17. The model (120) can be applied for each layer.

For each layer, we have the unknown porosity function  $\phi_i(z)$  fulfilling  $\phi_i'(z) = -c_i\phi_i$ , since the model (120) depends on the type of sediment in the layer. From the figure we see that new sediments are deposited on top of older ones as time progresses. The compaction of  $\phi$  is rapid in the beginning and then decreases (exponentially) with depth in each layer.

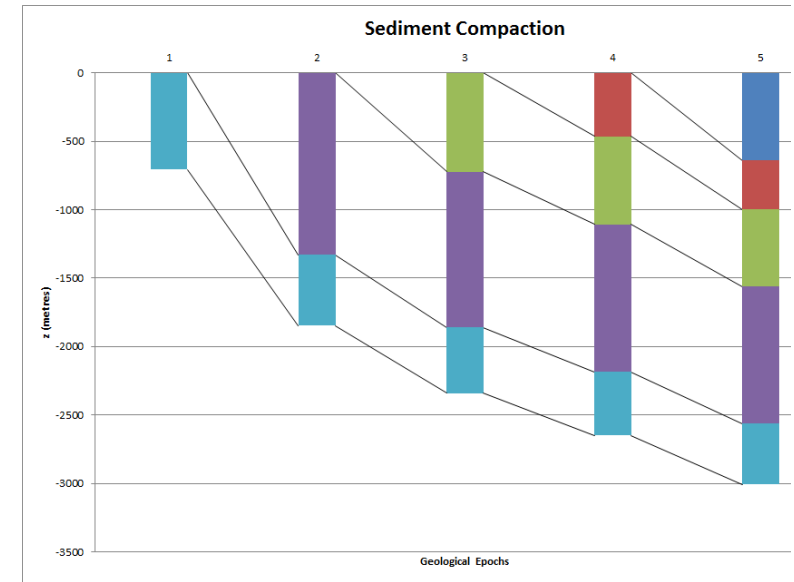


Figure 17: Illustration of the compaction of geological layers (with different colors).

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

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

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

## Vertical motion of a body in a viscous fluid

Moving vertically through a fluid (liquid or gas) is subject to three different forces: the gravity force, the drag force<sup>37</sup>, and the buoyancy force.

**Force of forces.** The gravity force is  $F_g = -mg$ , where  $m$  is the mass of the body,  $g$  is the acceleration of gravity. The uplift or buoyancy force ("Archimedes force") is  $F_b = \rho_b V g$ , where  $\rho_b$  is the density of the fluid and  $V$  is the volume of the body. Forces and other quantities are positive in the upward direction. The drag force is of two types, depending on the Reynolds number

$$\text{Re} = \frac{\rho_b d |v|}{\mu},$$

where  $d$  is the diameter of the body in the direction perpendicular to the flow,  $v$  is the velocity, and  $\mu$  is the dynamic viscosity of the fluid. When  $\text{Re} < 1$ , the drag force is given 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  $\text{Re} > 10^3$ , the drag force is quadratic in the velocity:

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

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

**Equation of motion.** All the mentioned forces act in the vertical direction. Newton's second law applied to the body says that the sum of these forces must equal the mass times its acceleration  $a$  in the vertical direction.

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

We have chosen to model the fluid resistance by the Stokes drag. Inserting the expressions yields

$$ma = -mg - 3\pi d \mu v + \rho_b V g.$$

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

$$m \frac{dv}{dt} = -mg - 3\pi d \mu v + \rho_b V g.$$

<sup>37</sup>[en.wikipedia.org/wiki/Drag\\_\(physics\)](https://en.wikipedia.org/wiki/Drag_(physics))

A small rewrite of this equation is handy: We express  $m$  as  $\rho_b V$ , where  $\rho_b$  is the density, and we divide by the mass to get

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

We may introduce the constants

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

so that the structure of the differential equation becomes obvious:

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

The corresponding initial condition is  $v(0) = v_0$  for some prescribed starting velocity.

This derivation can be repeated with the quadratic drag force  $F_d^{(q)}$ , leading

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

Defining

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

and  $b$  as above, we can write (127) as

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

**Terminal velocity.** An interesting aspect of (126) and (129) is whether  $v$  will approach a constant value, the so-called *terminal velocity*  $v_T$ , as  $t \rightarrow \infty$ . A constant  $v$  means  $v' = 0$  and therefore the terminal velocity  $v_T$  solves

$$0 = -av_T + b$$

and

$$0 = -a|v_T|v_T + b.$$

The former equation implies  $v_T = b/a$ , while the latter has solutions  $v_T = -\sqrt{|b/a|}$  (falling body,  $v_T < 0$ ) and  $v_T = \sqrt{|b/a|}$  for a rising body ( $v_T > 0$ ).

**Crank-Nicolson scheme.** Both governing equations, the Stokes' drag model and the quadratic drag model (129), can be readily solved by the Forward Euler scheme. However, for accuracy one can use the Crank-Nicolson method, but a straightforward application results in a nonlinear equation in the new unknown value  $v^{n+1}$  when applied to (129).

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

However, instead of approximating the term  $-|v|v$  by an arithmetic average or harmonic mean:

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

is of second order in  $\Delta t$ , just as for the arithmetic average and the center 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  $\mu$  are  $1.8 \cdot 10^{-5}$  Pas for air and  $8.9 \cdot 10^{-4}$  Pas for water.  $\rho$  can be taken as  $1.2 \text{ kg/m}^3$  for air and as  $1.0 \cdot 10^3 \text{ kg/m}^3$  for water. For convection in the atmosphere one should take into account that the density of air varies with altitude, see Section 8.6. One possible density variation arises from the one-layer model mentioned in section 8.7.

A density variation makes  $b$  time dependent and we need  $b^{n+\frac{1}{2}}$  in (132). To compute  $b^{n+\frac{1}{2}}$  we must also compute the vertical position  $z(t)$  of the body. For this, we can use a centered difference approximation:

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

$v^n$  is used in the expression for  $b$  to compute  $\rho(z^{n+\frac{1}{2}})$  and then  $b^{n+\frac{1}{2}}$ .

The drag coefficient<sup>38</sup>  $C_D$  depends heavily on the shape of the body. Some values are: 0.42 for a semi-sphere, 1.05 for a cube, 0.82 for a long cylinder (when the cylinder is perpendicular to the flow), 0.75 for a rocket, 1.0-1.3 for a man in upright position, 1.3 for a man in a horizontal position, and 0.04 for a streamlined, droplet-like body.

**ion.** To verify the program, one may assume a heavy body in air such that the air resistance can be neglected, and further assume a small velocity such that the air resistance can be neglected. This can be obtained by setting  $\mu$  and  $\rho$  to zero. The motion then reduces to  $v(t) = v_0 - gt$ , which is linear in  $t$  and therefore should be reproduced to a high accuracy (say tolerance  $10^{-15}$ ) by any implementation based on the Crank-Nicolson or Runge-Kutta methods.

Another verification, but not as powerful as the one above, can be based on computing the velocity and comparing with the exact expressions. The advantage of this verification is also the test situation  $\rho \neq 0$ .

Finally, the method of manufactured solutions can be applied to test the implementation. One chooses a solution, inserts it in the governing equation, but the solution then has no physical relevance in general.

Applying scaling, as described in Section 8.1, will for the linear case reduce the number of parameters down to choosing one value of a single dimensionless parameter.

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

<sup>38</sup>[en.wikipedia.org/wiki/Drag\\_coefficient](https://en.wikipedia.org/wiki/Drag_coefficient)

provided  $I \neq 0$ . If the motion starts from rest,  $I = 0$ , the scaled problem  $\bar{u}' = 1 - \beta \bar{u}$  is needed for estimating physical parameters. This means that there is a single universal problem of a falling body starting from rest:  $\bar{u}(t) = 1 - e^{-t}$ . All real physical problems are stretched the  $t$  axis and the  $\bar{u}$  axis in this dimensionless solution. More precisely, the velocity  $u(t)$  is related to the dimensionless velocity  $\bar{u}(\bar{t})$  through

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

## 9 Decay ODEs from solving a PDE by Fourier expansion

Suppose we have a partial differential equation

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2} + f(x, t),$$

with boundary conditions  $u(0, t) = u(L, t) = 0$  and initial condition  $u(x, 0) = u_0(x)$ . We express the solution as

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

for appropriate unknown functions  $A_k$ ,  $k = 1, \dots, m$ . We use the complex exponential for easy algebra, but the physical  $u$  is taken as the real part of any complex expansion. That the expansion in terms of  $e^{ikx\pi/L}$  is compatible with the boundary conditions means that  $e^{ikx\pi/L}$  vanishes for  $x = 0$  and  $x = L$ . Suppose we can express  $I(x)$  as

$$I(x) = \sum_{k=1}^m I_k e^{ikx\pi/L}.$$

Such an expansion can be computed by well-known Fourier expansion techniques. The coefficients are not important here. Also, suppose we can express the given  $f(x, t)$  as

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

Inserting the expansions for  $u$  and  $f$  in the differential equations demands that the coefficients corresponding to a given  $k$  must be equal. The calculations result in the following ordinary differential equations:

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

From the initial condition

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

it follows that  $A_k(0) = I_k$ ,  $k = 1, \dots, m$ . We then have  $m$  equations of the form  $A'_k(t) = -\alpha \frac{k^2 \pi^2}{L^2} A_k(t) + b_k(t)$ , for appropriate definitions of  $a$  and  $b$ . These ODE problems are independent, so that we can solve one problem at a time. The outline technique is a quite common one for solving partial differential equations.

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

## Exercises

### Exercise 10: Derive schemes for Newton's law of cooling

Detail how we can apply the ideas of the Forward Euler, Backward Euler, Crank-Nicolson, and implicit Euler discretizations to derive explicit computational formulas for new temperature values in Newton's law of cooling (see Section 8.5):

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

where  $T$  is the temperature of the body,  $T_s$  is the temperature of the surroundings,  $k$  is the heat transfer coefficient, and  $T_0$  is the initial temperature of the body. Filename: `cooling.py`.

### Exercise 11: Implement schemes for Newton's law of cooling

Write a  $\theta$ -rule for the three schemes in Exercise 10 such that you can get the three schemes from a single formula by varying the  $\theta$  parameter. Implement the  $\theta$  scheme in a function `newton_cooling(T0, k, T_s, t_end, dt, theta=0.5)`, where `T0` is the initial temperature, `k` is the heat transfer coefficient, `T_s` is the temperature of the surroundings, `t_end` is the end time, `dt` is the time step, and `theta` corresponds to  $\theta$ . The `cooling` function should return an array `T` of values at the mesh points and the time mesh `t`. Check your implementation with the examples in `cooling.py`.

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

Filename: `cooling.py`.

### Exercise 12: Find time of murder from body temperature

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

Assume that Newton's law of cooling (133) is an appropriate mathematical model for the temperature in the body. First, determine  $k$  in (133) by formulating a difference equation approximation with one time step from time 2 am to time 3 am, where knowing the temperature at these times allows for finding  $k$ . Assume the temperature in the air to be 20 C. Then use the temperature evolution from the time of murder, taken as  $t = 0$ , when the temperature reaches 25.8 C. The corresponding time allows for answering when the murder occurred. Filename: `detective.py`.

### Exercise 13: Simulate an oscillating cooling process

The surrounding temperature  $T_s$  in Newton's law of cooling (133) may vary in time. Assume the variations are periodic with period  $P$  and amplitude  $a$  around a constant mean value  $T_m$ :

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

Simulate a process with the following data:  $k = 20 \text{ min}^{-1}$ ,  $T(0) = 5 \text{ C}$ ,  $T_m = 20 \text{ C}$ , and  $P = 1 \text{ h}$ . Also experiment with  $P = 10 \text{ min}$  and  $P = 3 \text{ h}$ . Plot  $T$  and  $T_s$  in the same figure. Filename: `osc_cooling.py`.

### Exercise 14: Radioactive decay of Carbon-14

The Carbon-14<sup>39</sup> isotope, whose radioactive decay is used extensively in dating organic material that is tens of thousands of years old, has a half-life of 5,730 years. Determine the age of an organic material that contains 8.4 percent of its initial amount of Carbon-14. Use a time step of 1 year in the computations. The uncertainty in the half time of Carbon-14 is  $\pm 40$  years. Estimate the corresponding uncertainty in the estimate of the age?

**Hint.** Use simulations with  $5,730 \pm 40$  y as input and find the corresponding age. Filename: `carbon14.py`.

### Exercise 15: Simulate stochastic radioactive decay

The purpose of this exercise is to implement the stochastic model described in Section 8.6. Show that its mean behavior approximates the solution of the corresponding ODE.

The simulation goes on for a time interval  $[0, T]$  divided into  $N_t$  intervals of length  $\Delta t$ . In some time interval, we have  $N$  atoms that have survived the previous interval. In this interval, we have  $N$  independent Bernoulli trials with probability  $\lambda \Delta t$  in this interval by drawing  $N$  random numbers. If the number is less than  $\lambda \Delta t$ , the atom decays (survival) or 1 (decay), where the probability of getting 1 is  $\lambda \Delta t$ . We are interested in the number of decays,  $d$ , and the number of survived atoms in the next interval is  $N - d$ . The Bernoulli trials are simulated by drawing  $N$  uniformly distributed real numbers. If the number is less than  $\lambda \Delta t$ , the atom decays. This corresponds to a value less than  $\lambda \Delta t$ :

```
# Given lambda_, dt, N
import numpy as np
uniform = np.random.uniform(N)
bernoulli_trials = np.asarray(uniform < lambda_*dt, dtype=np.int)
d = bernoulli_trials.size
```

Observe that `uniform < lambda_*dt` is a boolean array whose true and false values correspond to 1 and 0, respectively, when converted to an integer array.

Repeat the simulation over  $[0, T]$  a large number of times, compute the average number of decays in each interval, and compare with the solution of the corresponding ODE in `stochastic_decay.py`.

<sup>39</sup><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 form type A nuclei with a half-life  $A_{1/2}$ , while substance B decay to form type B nuclei with a half-life  $B_{1/2}$ . Let  $u_A$  and  $u_B$  be the fractions of the initial amount of material in substance A and B, respectively. The following system of ODEs governs the evolution of  $u_A(t)$  and  $u_B(t)$ :

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

with  $u_A(0) = u_B(0) = 1$ .

Write a simulation program that solves for  $u_A(t)$  and  $u_B(t)$ . Verify the implementation by comparing the limiting values of  $u_A$  and  $u_B$  as  $t \rightarrow \infty$  (assume  $u'_A, u'_B \rightarrow 0$ ) with those obtained numerically.

Use the program for the case of  $A_{1/2} = 10$  minutes and  $B_{1/2} = 50$  minutes. Use a time step of 1 minute. Plot  $u_A$  and  $u_B$  versus time in the same plot. Filename: `radioactive_decay_2substances.py`.

## e 17: Simulate the pressure drop in the atmosphere

Implement the models for atmospheric pressure in Section 8.6. Make a program with the following functions:

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

2. `compute_pressure_seven_layer_constant(T)` computing  $p(z)$  using a seven-layer model, but with constant temperature in each layer,

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

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

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

Implement the Stokes' drag model (124) and the quadratic drag model (127) from Section 8.5 using the Crank-Nicolson scheme and a geometric mean for  $|v|v$  as explained, and assume constant density. At each time level, compute the Reynolds number  $Re$  and choose the Stokes' drag model if  $Re < 1$  and the quadratic drag model otherwise.

The computation of the numerical solution should take place either in a stand-alone function (see Section 2.1) or in a solver class that looks up a problem class for physical data (see Section 5.5). Create a module (see Section ??) and equip it with nose tests (see Section 5.5) for verifying the code.

How can the nose tests can be based on

1. the terminal velocity (see Section 8.8),

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

3. the method of manufactured solutions (see Section 5.5) combined with computing convergence rates (see Section ??).

For example, use a quadratic polynomial for the velocity in the method of manufactured solutions. The expected error is  $\mathcal{O}(\Delta t^2)$  from the centered finite difference approximation and the geometric mean approximation for  $|v|v$ .

A solution that is linear in  $t$  will also be an exact solution of the discrete equations for linear drag (by adding a source term that is linear in  $t$ ), but not for quadratic drag because of the geometric mean approximation. Use the method of manufactured solutions to add a source term *in the discrete equations* for quadratic drag so that a linear function of  $t$  is a solution. Add a nose test for checking that the numerical solution is reproduced 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 force balances the gravity force, but the drag will be significant and balance the other forces after some time. A ball has radius 11 cm and mass 0.43 kg. Start the motion from rest, set the density of water,  $\rho$ , to 1000 kg/m<sup>3</sup>, set the dynamic viscosity,  $\mu$ , to 10<sup>-3</sup> Pa s, and use a drag coefficient of 0.45. Plot the velocity of the rising ball. Filename: `vertical_motion.py`.

## Project 19: Simulate parachuting

The aim of this project is to develop a general solver for the vertical motion with linear and quadratic air drag, verify the solver, apply the solver to a skydiver in free fall, and use the solver to a complete parachute jump.

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

1. Set up the differential equation problem that governs the velocity of the motion. The skydiver is subject to the gravity force and a quadratic drag force. Assume constant density. An extra source term can be used for program verification. Identify the input data and the output.

2. Make a Python module for computing the velocity of the motion. Also equip the module with functionality for plotting the velocity.

**Hint 1.** Use the Crank-Nicolson scheme with a geometric mean of  $|v|v$  in time discretization. The expected error is  $\mathcal{O}(\Delta t^2)$  because we use a centered finite difference approximation with error  $\mathcal{O}(\Delta t^2)$  and a geometric mean approximation with error  $\mathcal{O}(\Delta t)$ .

**Hint 2.** You can either use functions or classes for implementation. If you choose functions, make a function `solver` that takes all the input data in the problem as arguments: the physical data (as a mesh function) and the time mesh. In case of a class-based implementation, introduce a problem class with the physical data and a solver class with the numerical solution 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 drag force.

3. Show that a linear function of  $t$  does not fulfill the discrete equations because of the geometric mean approximation used for the quadratic drag term. Fit a source term, as in the method of manufactured solutions, such that a linear function of  $t$  is a solution of the discrete equations. Add a nose test to check that this solution is reproduced to machine precision.

4. The expected error in this problem goes like  $\Delta t^2$  because we use a centered finite difference approximation with error  $\mathcal{O}(\Delta t^2)$  and a geometric mean approximation with error  $\mathcal{O}(\Delta t)$ . Use the method of manufactured solutions combined with computing convergence rates to verify the error. 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.

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

leade and Struthers [5] provide some data relevant to skydiving<sup>40</sup>. The mas  
dy and equipment can be set to 100 kg. A skydiver in spread-eagle formati  
ion of 0.5 m<sup>2</sup> in the horizontal plane. The density of air decreases varies altitu  
ken as constant, 1 kg/m<sup>3</sup>, 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  
as 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  
ally. Use the program to simulate a jump from  $z = 3000$  m to the ground  $z = 0$   
imum 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<sup>2</sup> 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  
1.8, 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 val  
ed to make plots more illustrative.

`skydiving.py`.

## e 20: Formulate vertical motion in the atmosphere

otion of a body in the atmosphere needs to take into account a varying air d  
of altitudes is many kilometers. In this case,  $\varrho$  varies with the altitude  $z$ . The  $\epsilon$   
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  
ation 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

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

<sup>40</sup>[en.wikipedia.org/wiki/Parachuting](http://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  
ie Crank-Nicolson methods. Filename: `falling_in_variable_density.pdf`.

## Exercise 21: Simulate vertical motion in the atmosphere

plement the Forward Euler or the Crank-Nicolson scheme derived in Exercise :  
ie 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.

**Remark.** In the Crank-Nicolson scheme one must solve a  $3 \times 3$  system of ec  
me level, since  $p$ ,  $\varrho$ , 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

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

hich has the solution  $y(x) = |x|$ . Using a mesh  $x_0 = -1$ ,  $x_1 = 0$ , and  $x_2 = 1$ , c  
 $y_1$  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  
 $T = 10$  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  
interest rate. Suppose that the inflation is constant at  $i$  percent per year and  
interest rate,  $p$ , changes randomly at each time step, starting at some value  $p$   
andom 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  
ceeds 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  
oney and the corresponding standard deviation. Plot the mean curve along  
hus one standard deviation and the mean minus one standard deviation. This v  
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
```

<sup>41</sup>[http://en.wikipedia.org/wiki/Felix\\_Baumgartner](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  $i$ ,  $i = 0, \dots, N-1$ , the value of the fortune is

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

standard deviation is

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

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

## e 24: Simulate a population in a changing environment

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

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

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

where the population growth is compensated by a sudden net immigration at time  $t_f$ :

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

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

We assume that the environmental conditions change periodically with time so

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

Study the combined birth and death rate oscillates around  $r_0$  with a maximum change over a period of length  $P$  in time. Set  $f = 0$  and experiment with the other parameters to observe typical features of the solution. Filename: `population_osc.py`.

## e 25: Simulate logistic growth

Solve the logistic ODE (109) using a Crank-Nicolson scheme where  $(u^{n+\frac{1}{2}})^2$  is approximated by the arithmetic mean:

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

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

## Exercise 26: Rederive the equation for continuous compounding

The ODE model (112) was derived under the assumption that  $r$  was constant. An alternative derivation without this assumption: 1) start with (110); 2) introduce  $\Delta t$  instead of  $m$ :  $\Delta t = 1/m$  if  $t$  is measured in years; 3) divide by  $\Delta t$  and let  $\Delta t \rightarrow 0$ . Simulate a case where the inflation is at a constant level  $I$  percent per year and the interest rate oscillates:  $r = -I/2 + r_0 \sin(2\pi t)$ . Compare solutions for  $r_0 = I, 3I$ . Filename: `interest_modeling.py`.

## References

- [1] D. Griffiths, F. David, and D. J. Higham. *Numerical Methods for Ordinary Differential Equations: Initial Value Problems*. Springer, 2010.
- [2] E. Hairer, S. P. Nørsett, and G. Wanner. *Solving Ordinary Differential Equations I: Nonstiff Problems*. Springer, 1993.
- [3] G. Hairer and E. Wanner. *Solving Ordinary Differential Equations II: Stiff Problems*. Springer, 1993.
- [4] H. P. Langtangen. *A Primer on Scientific Programming With Python*. Texts in Science and Engineering. Springer, third edition, 2012.
- [5] D. B. Meade and A. A. Struthers. Differential equations in the new millennium problem. *International Journal of Engineering Education*, 15(6):417–424, 1995.
- [6] L. Petzold and U. M. Ascher. *Computer Methods for Ordinary Differential and Algebraic Equations*, volume 61. SIAM, 1998.

- , 13, 52
- methods, 38
- ashforth scheme, 2nd-order, 54
- ashforth scheme, 3rd order, 54
- time stepping, 59
- equation, 8
- ion factor, 37
- hmetics, 22
- puting, 22
- metric, 11
- etric, 72
- difference, 10
- Euler scheme, 10
- scheme, 1-step, 10
- scheme, 2-step, 52
- eme, 52
- lifference, 10
- y, 44
- s function norms, 22
- ce, 44
- olson scheme, 10
- images, 27
- E, 5
- equation, 8
- 15
- quation, 8
- inction norms, 23
- s, 18
- Prince Runge-Kutta 4-5 method, 59
- 26
- ification factor, 41
- l, 41
- s, 24
- hemes, 52
- al decay, 5
- rence operator notation, 14
- rence scheme, 8
- rences, 7
- ward, 10
- centered, 10
- forward, 7
- folder, 15
- format string syntax (Python), 19
- forward difference, 7
- Forward Euler scheme, 8
- geometric mean, 72
- grid, 6
- Heun's method, 53
- implicit schemes, 52
- L-stable methods, 38
- lambda functions, 48
- Leapfrog scheme, 53
- Leapfrog scheme, filtered, 53
- logistic model, 65
- mesh, 6
- mesh function, 6
- mesh function norms, 23
- method of manufactured solutions, 49
- MMS (method of manufactured solutic
- montage program, 27
- norm
- continuous, 22
- discrete (mesh function), 23
- ode45, 59
- operator notation, finite differences, 14
- PDF plot, 26
- pdfcrop program, 27
- pdfnup program, 27
- pdftk program, 27
- plotting curves, 25
- PNG plot, 26
- population dynamics, 64
- printf format, 19
- radioactive decay, 66
- representative (mesh function), 22
- RK4, 54
- Runge-Kutta, 2nd-order scheme, 53
- Runge-Kutta, 4th-order scheme, 54

- calar computing, 24
- aling, 73
- ability, 36, 44
- aylor-series methods (for ODEs), 54
- rminal velocity, 72
- eta-rule, 11, 52
- iewing graphics files, 26
- isualizing curves, 25
- eighted average, 11