Algorithms and implementations for exponential decay models

Hans Petter Langtangen[1,2]

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

Oct 17, 2015

Contents

List of Exercises and Problems

Throughout industry and science it is common today to study nature or technological devices through models on a computer. With such models the computer acts as a virtual lab where experiments can be done in a fast, reliable, safe, and cheap way. In some fields, e.g., aerospace engineering, the computer models are now so sophisticated that they can replace physical experiments to a large extent.

A vast amount of computer models are based on ordinary and partial differential equations. This book is an introduction to the various scientific ingredients we need for reliable computing with such type of models. A key theme is to solve differential equations *numerically* on a computer. Many methods are available for this purpose, but the focus here is on *finite difference methods*, because these are simple, yet versatile, for solving a wide range of ordinary and partial differential equations. The present chapter first presents the mathematical ideas of finite difference methods and derives algorithms, i.e., formulations of the methods ready for computer programming. Then we create programs and learn how we can be sure that the programs really work correctly.

1 Finite difference methods

This section explains the basic ideas of finite difference methods via the simple ordinary differential equation $u' = -au$. Emphasis is put on the reasoning behind problem discretizing and introduction of key concepts such as mesh, mesh function, finite difference approximations, averaging in a mesh, derivation of algorithms, and discrete operator notation.

1.1 A basic model for exponential decay

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

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

In this equation, $u(t)$ is a scalar function of time t, a is a constant (in this book we mostly work with $a > 0$), and $u'(t)$ means differentiation with respect to t. This type of equation arises in a number of widely different phenomena where some quantity u undergoes exponential reduction (provided $a > 0$). Examples include radioactive decay, population decay, investment decay, cooling of an object, pressure decay in the atmosphere, and retarded motion in fluids. Some models with growth, $a < 0$, are treated as well. We have chosen this particular ODE not only because its applications are relevant, but even more because studying numerical solution methods for this particular ODE gives important insight that can be reused in far more complicated settings, in particular when solving diffusion-type partial differential equations.

The exact solution. Although our interest is in *approximate* numerical solutions of $u' = -au$, it is convenient to know the exact analytical solution of the problem so we can compute the error in numerical approximations. The

analytical solution of this ODE is found by separation of variables, which results in

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

for any arbitrary constant C. To obtain a unique solution, we need a condition to fix the value of C. This condition is known as the *initial condition* and stated as $u(0) = I$. That is, we know that the value of u is I when the process starts at $t = 0$. With this knowledge, the exact solution becomes $u(t) = Ie^{-at}$. The initial condition is also crucial for numerical methods: without it, we can never start the numerical algorithms!

A complete problem formulation. Besides an initial condition for the ODE, we also need to specify a time interval for the solution: $t \in (0, T]$. The point $t = 0$ is not included since we know that $u(0) = I$ and assume that the equation governs u for $t > 0$. Let us now summarize the information that is required to state the complete problem formulation: find $u(t)$ such that

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

This is known as a *continuous problem* because the parameter t varies continuously from 0 to T. For each t we have a corresponding $u(t)$. There are hence infinitely many values of t and $u(t)$. The purpose of a numerical method is to formulate a corresponding *discrete* problem whose solution is characterized by a finite number of values, which can be computed in a finite number of steps on a computer. Typically, we choose a finite set of time values $t_0, t_1, \ldots, t_{N_t}$, and create algorithms that generate the corresponding u values $u_0, u_1, \ldots, u_{N_t}$.

1.2 The Forward Euler scheme

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

1. discretizing the domain,
2. requiring fulfillment of 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 a finite number of $N_t + 1$ points

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

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

We want the solution u at the mesh points: $u(t_n)$, $n = 0, 1, \ldots, N_t$. A notational short-form for $u(t_n)$, which will be used extensively, is u^n. More precisely, we let u^n be the *numerical approximation* to the exact solution $u(t_n)$ at $t = t_n$.

When we need to clearly distinguish between the numerical and exact solution, we often place a subscript e on the exact solution, as in $u_{\mathrm{e}}(t_n)$. Figure 1 shows the t_n and u^n points for $n = 0, 1, \ldots, N_t = 7$ as well as $u_{\mathrm{e}}(t)$ as the dashed line.

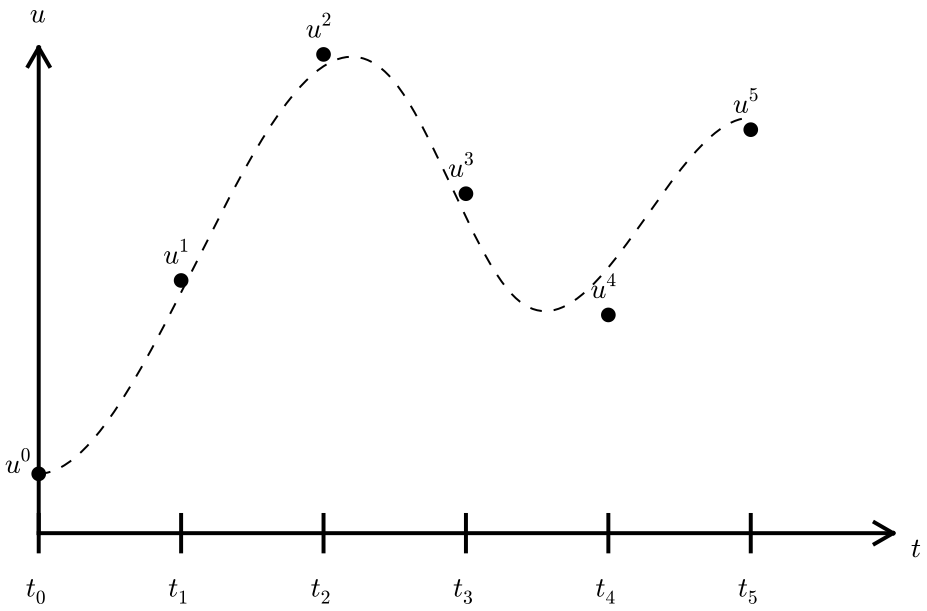


Figure 1: Time mesh with discrete solution values at points and a dashed line indicating the true solution.

We say that the numerical approximation, i.e., the collection of u^n values for $n = 0, \ldots, N_t$, constitutes a *mesh function*. A "normal" continuous function is a curve defined for all real t values in $[0, T]$, but a mesh function is only defined at discrete points in time. If you want to compute the mesh function *between* the mesh points, where it is not defined, an *interpolation method* must be used. Usually, linear interpolation, i.e., drawing a straight line between the mesh function values, see Figure 1, suffices. To compute the solution for some $t \in [t_n, t_{n+1}]$, we use the linear interpolation formula

$$u(t) \approx u^n + \frac{u^{n+1} - u^n}{t_{n+1} - t_n}(t - t_n)\,. \tag{3}$$

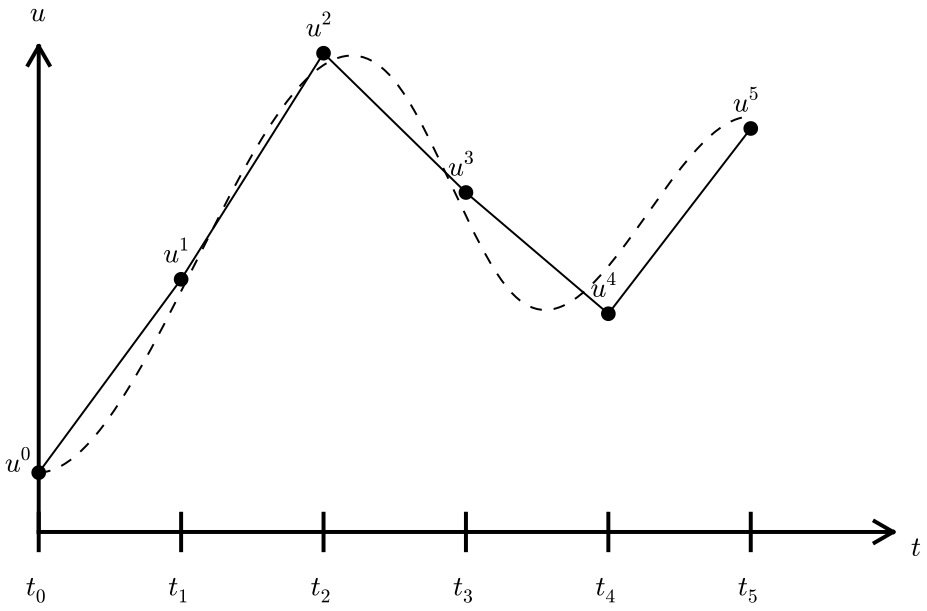


Figure 2: Linear interpolation between the discrete solution values (dashed curve is exact solution).

Notice.

The goal of a numerical solution method for ODEs is to compute the mesh function by solving a finite set of *algebraic equations* derived from the original ODE problem.

Step 2: Fulfilling the equation at discrete time points. The ODE is supposed to hold for all $t \in (0, T]$, i.e., at an infinite number of points. Now we relax that requirement and require that the ODE is fulfilled at a finite set of discrete points in time. The mesh points $t_0, t_1, \ldots, t_{N_t}$ are a natural (but not the only) choice of points. The original ODE is then reduced to the following equations:

$$u'(t_n) = -au(t_n), \quad n = 0, \ldots, N_t, \quad u(0) = I\,. \tag{4}$$

Even though the original ODE is not stated to be valid at $t = 0$, it is valid as close to $t = 0$ as we like, and it turns out that it is useful for construction of numerical methods to have (4) valid for $n = 0$. The next two steps show that we need (4) for $n = 0$.

Step 3: Replacing derivatives by finite differences. The next and most essential step of the method is to replace the derivative u' by a finite difference approximation. Let us first try a *forward* difference approximation (see Figure 3),

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

The name forward relates to the fact that we use a value forward in time, u^{n+1}, together with the value u^n at the point t_n, where we seek the derivative, to approximate $u'(t_n)$. Inserting this approximation in (4) results in

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

Note that if we want to compute the solution up to time level N_t, we only need (4) to hold for $n = 0, \ldots, N_t - 1$ since (6) for $n = N_t - 1$ creates an equation for the final value u^{N_t}.

Also note that we use the approximation symbol $\approx$ in (5), but not in (6). Instead, we view (6) as an equation that is not mathematically equivalent to (5), but represents an approximation to the equation (5).

Equation (6) is the discrete counterpart to the original ODE problem (1), and often referred to as a *finite difference scheme* or more generally as the *discrete equations* of the problem. The fundamental feature of these equations is that they are *algebraic* and can hence be straightforwardly solved to produce the mesh function, i.e., the approximate values of u at the mesh points: u^n, $n = 1, 2, \ldots, N_t$.

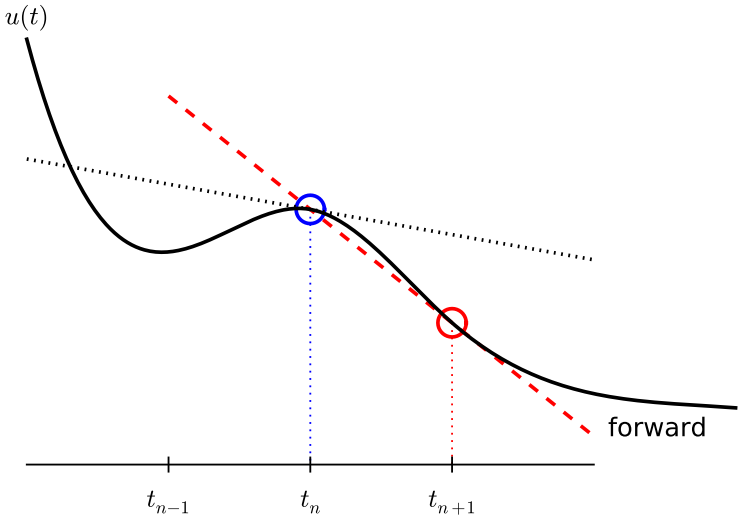


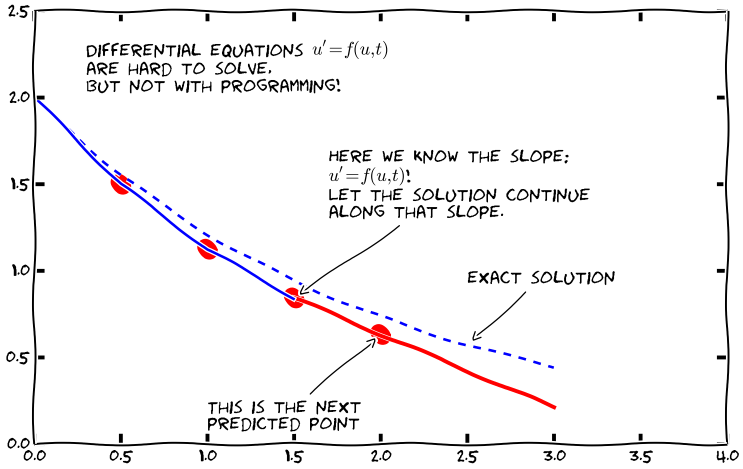
Figure 3: Illustration of a forward difference.

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

$$u^{n+1} = u^n - a(t_{n+1} - t_n)u^n \,. \tag{7}$$

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* since they express how differences in the dependent variable, here u, evolve with n. In our case, the differences in u are given by $u^{n+1} - u^n = -a(t_{n+1} - t_n)u^n$. The finite difference method can be viewed as a method for turning a differential equation into an algebraic difference equation that can be easily solved by repeated use of a formula like (7).

Interpretation. There is a very intuitive interpretation of the FE scheme, illustrated in the sketch below. We have computed some point values on the solution curve (small red disks), and the question is how we reason about the next point. Since we know u and t at the most recently computed point, the differential equation gives us the *slope* of the solution curve: $u' = -au$. We can draw this slope as a red line and continue the solution curve along that slope. As soon as we have chosen the next point on this line, we have a new t and u value and can compute a new slope and continue the process.



Computing with the recursive formula. Mathematical computation with (7) is straightforward:

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

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

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

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

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

1.3 The Backward Euler scheme

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

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

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

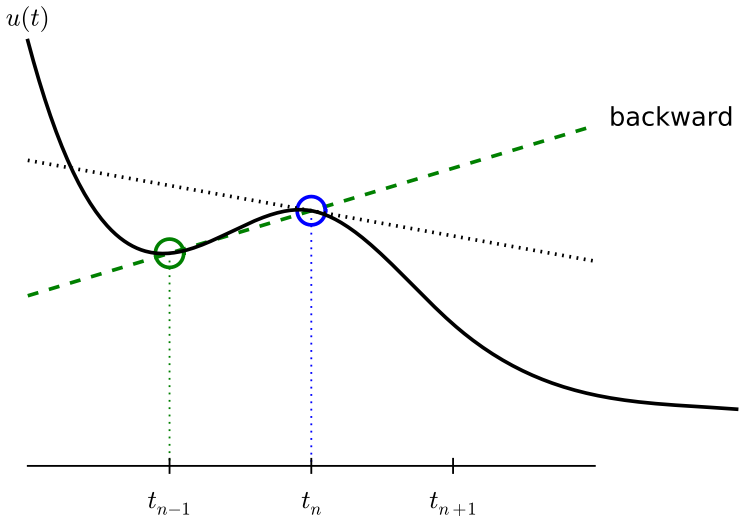


Figure 4: Illustration of a backward difference.

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

$$\frac{u^n - u^{n-1}}{t_n - t_{n-1}} = -au^n, \quad n = 1, \ldots, N_t\,. \tag{9}$$

We assume, as explained under step 4 in Section 1.2, that we have computed $u^0, u^1, \ldots, u^{n-1}$ such that (9) can be used to compute u^n. Note that (9) needs n to start at 1 (then it involves u^0, but no u^{-1}) and end at N_t.

For direct similarity with the formula for the Forward Euler scheme (7) we replace n by $n+1$ in (9) and solve for the unknown value u^{n+1}:

$$u^{n+1} = \frac{1}{1 + a(t_{n+1} - t_n)}u^n, \quad n = 0, \ldots, N_t - 1\,. \tag{10}$$

1.4 The Crank-Nicolson scheme

The finite difference approximations (5) and (8) used to derive the schemes (7) and (10), respectively, are both one-sided differences, i.e., we collect information

either forward or backward in time when approximating the derivative at a point. Such one-sided differences are known to be less accurate than central (or midpoint) differences, where we use information both forward and backward in time. A natural next step is therefore to construct a central difference approximation that will yield a more accurate numerical solution.

The central difference approximation to the derivative is sought at the point $t_{n+\frac{1}{2}} = \frac{1}{2}(t_n + t_{n+1})$ (or $t_{n+\frac{1}{2}} = (n+\frac{1}{2})\Delta t$ if the mesh spacing is uniform in time). The approximation reads

$$u'(t_{n+\frac{1}{2}}) \approx \frac{u^{n+1} - u^n}{t_{n+1} - t_n}\,. \tag{11}$$

Figure 5 sketches the geometric interpretation of such a centered difference. Note that the fraction on the right-hand side is the same as for the Forward Euler approximation (5) and the Backward Euler approximation (8) (with n replaced by $n+1$). The accuracy of this fraction as an approximation to the derivative of u depends on *where* we seek the derivative: in the center of the interval $[t_n, t_{n+1}]$ or at the end points. We shall later see that it is more accurate at the center point.

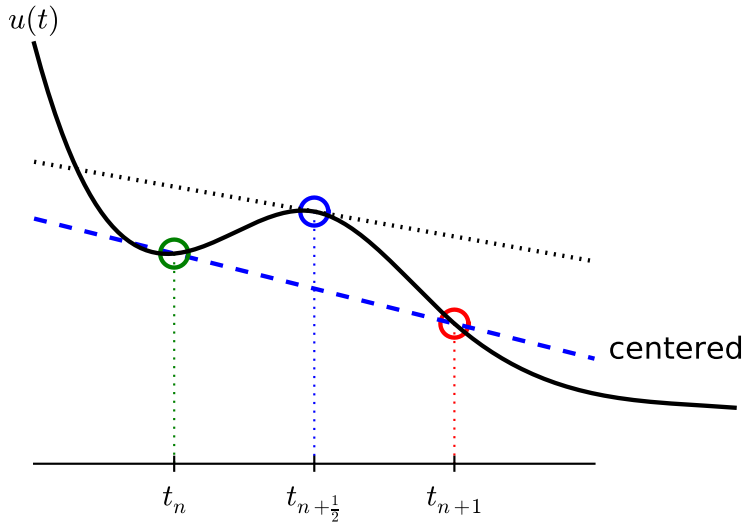


Figure 5: Illustration of a centered difference.

With the formula (11), where u' is evaluated at $t_{n+\frac{1}{2}}$, it is natural to demand the ODE to be fulfilled at the time points *between* the mesh points:

$$u'(t_{n+\frac{1}{2}}) = -au(t_{n+\frac{1}{2}}), \quad n = 0, \ldots, N_t - 1\,. \tag{12}$$

Using (11) in (12) results in the approximate discrete equation

$$\frac{u^{n+1}-u^n}{t_{n+1}-t_n} = -au^{n+\frac{1}{2}}, \quad n = 0, \ldots, N_t - 1, \tag{13}$$

where $u^{n+\frac{1}{2}}$ is a short form for the numerical approximation to $u(t_{n+\frac{1}{2}})$.

There is a fundamental problem with the right-hand side of (13): we aim to compute u^n for integer n, which means that $u^{n+\frac{1}{2}}$ is not a quantity computed by our method. The quantity must therefore be expressed by the quantities that we actually produce, i.e., the numerical solution at the mesh points. One possibility is to approximate $u^{n+\frac{1}{2}}$ as an arithmetic mean of the u values at the neighboring mesh points:

$$u^{n+\frac{1}{2}} \approx \frac{1}{2}(u^n + u^{n+1}) \,. \tag{14}$$

Using (14) in (13) results in a new approximate discrete equation

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

There are three approximation steps leading to this formula: 1) the ODE is only valid at discrete points (between the mesh points), 2) the derivative is approximated by a finite difference, and 3) the value of u between mesh points is approximated by an arithmetic mean value. Despite one more approximation than for the Backward and Forward Euler schemes, the use of a centered difference leads to a more accurate method.

To formulate a recursive algorithm, we assume that u^n is already computed so that u^{n+1} is the unknown, which we can solve for:

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

The finite difference scheme (16) is often called the Crank-Nicolson (CN) scheme or a midpoint or centered scheme. Note that (16) as well as (7) and (10) apply whether the spacing in the time mesh, $t_{n+1} - t_n$, depends on n or is constant.

1.5 The unifying θ-rule

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

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

Observe that

- $\theta = 0$ gives the Forward Euler scheme
- $\theta = 1$ gives the Backward Euler scheme,

- $\theta = \frac{1}{2}$ gives the Crank-Nicolson scheme.

One may alternatively choose any other value of θ in $[0, 1]$, but this is not so common since the accuracy and stability of the scheme do not improve compared to the values $\theta = 0, 1, \frac{1}{2}$.

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

$$u^{n+1} = \frac{1 - (1-\theta)a(t_{n+1} - t_n)}{1 + \theta a(t_{n+1} - t_n)}\,. \tag{18}$$

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

Derivation.

We start with replacing u' by the fraction

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

in the Forward Euler, Backward Euler, and Crank-Nicolson schemes. Then we observe 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 the end points or the midpoint of $[t_n, t_{n+1}]$. However, we may choose any point $\tilde{t} \in [t_n, t_{n+1}]$. The difficulty is that evaluating the right-hand side $-au$ at an arbitrary point faces the same problem as in Section 1.4: the point value must be expressed by the discrete u quantities 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*, which generalizes the arithmetic mean $\frac{1}{2}u^n + \frac{1}{2}u^{n+1}$. The weighted average reads

$$u(\tilde{t}) \approx \theta u^{n+1} + (1-\theta)u^n, \tag{19}$$

where $\theta \in [0, 1]$ is a weighting factor. We can also express $\tilde{t}$ as a similar weighted average

$$\tilde{t} \approx \theta t_{n+1} + (1-\theta)t_n\,. \tag{20}$$

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

1.6 Constant time step

All schemes up to now have been formulated for a general non-uniform mesh in time: $t_0 < t_1 < \cdots < t_{N_t}$. Non-uniform meshes are highly relevant since

one can use many points in regions where u varies rapidly, and fewer points in regions where u is slowly varying. This idea saves the total number of points and therefore makes it faster to compute the mesh function u^n. Non-uniform meshes are used together with *adaptive* methods that are able to adjust the time mesh during the computations.

However, a uniformly distributed set of mesh points is not only convenient, but also sufficient for many applications. Therefore, it is a very common choice. We shall present the finite difference schemes for a uniform point distribution $t_n = n\Delta t$, where Δt is the constant spacing between the mesh points, also referred to as the *time step*. The resulting formulas look simpler and are more well known.

Summary of schemes for constant time step.

$$u^{n+1} = (1 - a\Delta t)u^n \qquad \text{Forward Euler} \tag{21}$$

$$u^{n+1} = \frac{1}{1 + a\Delta t}u^n \qquad \text{Backward Euler} \tag{22}$$

$$u^{n+1} = \frac{1 - \frac{1}{2}a\Delta t}{1 + \frac{1}{2}a\Delta t}u^n \qquad \text{Crank-Nicolson} \tag{23}$$

$$u^{n+1} = \frac{1 - (1 - \theta)a\Delta t}{1 + \theta a\Delta t}u^n \qquad \text{The } \theta - \text{rule} \tag{24}$$

It is not accidental that we focus on presenting the Forward Euler, Backward Euler, and Crank-Nicolson schemes. They complement each other with their different pros and cons, thus providing a useful collection of solution methods for many differential equation problems. The unifying notation of the θ-rule makes it convenient to work with all three methods through just one formula. This is particularly advantageous in computer implementations since one avoids if-else tests with formulas that have repetitive elements.

1.7 Mathematical derivation of finite difference formulas

The finite difference formulas for approximating the first derivative of a function have so far been somewhat justified through graphical illustrations in Figures 3, 4, and 5. The task is to approximate the derivative at a point of a curve using only two function values. By drawing a straight line through the points, we have some approximation to the tangent of the curve and use the slope of this line as an approximation to the derivative. The slope can be computed by inspecting the figures.

However, we can alternatively derive the finite difference formulas by pure mathematics. The key tool for this approach is Taylor series, or more precisely, approximation of functions by lower-order Taylor polynomials. Given a function $f(x)$ that is sufficiently smooth (i.e., $f(x)$ has "enough derivatives"), a Taylor

polynomial of degree m can be used to approximate the value of the function $f(x)$ if we know the values of f and its first m derivatives at some other point $x = a$. The formula for the Taylor polynomial reads

$$f(x) \approx f(a) + f'(a)(x-a) + \frac{1}{2}f''(a)(x-a)^2 + \frac{1}{6}f'''(a)(x-a)^3 + \cdots + \frac{1}{m!}\frac{df^{(m)}}{dx^m}(a)(x-a)^m\,. \tag{25}$$

For a function of time, $f(t)$, related to a mesh with spacing Δt, we often need the Taylor polynomial approximation at $f(t_n \pm \Delta t)$ given f and its derivatives at $t = t_n$. Replacing x by $t_n + \Delta t$ and a by t_n gives

$$f(t_n + \Delta t) \approx f(t_n) + f'(t_n)\Delta t + \frac{1}{2}f''(t_n)\Delta t^2 + \frac{1}{6}f'''(t_n)\Delta t^3 + \cdots + \frac{1}{m!}\frac{df^{(m)}}{dx^m}(t_n)\Delta t^m\,. \tag{26}$$

The forward difference. We can use (26) to find an approximation for $f'(t_n)$ simply by solving with respect to this quantity:

$$f'(t_n) \approx \frac{f(t_n+\Delta t) - f(t_n)}{\Delta t} - \frac{1}{2}f''(t_n)\Delta t - \frac{1}{6}f'''(t_n)\Delta t^2 + \cdots - \frac{1}{m!}\frac{df^{(m)}}{dx^m}(t_n)\Delta t^{m-1}\,. \tag{27}$$

By letting $m \to \infty$, this formula is exact, but that is not so much of practical value. A more interesting observation is that all the power terms in Δt vanish as $\Delta t \to 0$, i.e., the formula

$$f'(t_n) \approx \frac{f(t_n+\Delta t) - f(t_n)}{\Delta t} \tag{28}$$

is exact in the limit $\Delta t \to 0$.

The interesting feature of (27) is that we have a measure of the error in the formula (28): the error is given by the extra terms on the right-hand side of (27). We assume that Δt is a small quantity ($\Delta t \ll 1$). Then $\Delta t^2 \ll \Delta t$, $\Delta t^3 \ll \Delta t^2$, and so on, which means that the first term is the dominating term. This first term reads $-\frac{1}{2}f''(t_n)\Delta t$ and can be taken as a measure of the error in the Forward Euler formula.

The backward difference. To derive the backward difference, we use the Taylor polynomial approximation at $f(t_n - \Delta t)$:

$$f(t_n - \Delta t) \approx f(t_n) - f'(t_n)\Delta t + \frac{1}{2}f''(t_n)\Delta t^2 - \frac{1}{6}f'''(t_n)\Delta t^3 + \cdots + \frac{1}{m!}\frac{df^{(m)}}{dx^m}(t_n)\Delta t^m \,. \tag{29}$$

Solving with respect to $f'(t_n)$ gives

$$f'(t_n) \approx \frac{f(t_n) - f(t_n - \Delta t)}{\Delta t} + \frac{1}{2}f''(t_n)\Delta t - \frac{1}{6}f'''(t_n)\Delta t^2 + \cdots - \frac{1}{m!}\frac{df^{(m)}}{dx^m}(t_n)\Delta t^{m-1} \,. \tag{30}$$

The term $\frac{1}{2}f''(t_n)\Delta t$ can be taken as a simple measure of the approximation error since it will dominate over the other terms as $\Delta t \to 0$.

The centered difference. The centered difference approximates the derivative at $t_n + \frac{1}{2}\Delta t$. Let us write up the Taylor polynomial approximations to $f(t_n)$ and $f(t_{n+1})$ around $t_n + \frac{1}{2}\Delta t$:

$$f(t_n) \approx f(t_n + \frac{1}{2}\Delta t) - f'(t_n + \frac{1}{2}\Delta t)\frac{1}{2}\Delta t + f''(t_n + \frac{1}{2}\Delta t)(\frac{1}{2}\Delta t)^2 - f'''(t_n + \frac{1}{2}\Delta t)(\frac{1}{2}\Delta t)^3 + \cdots \tag{31}$$

$$f(t_{n+1}) \approx f(t_n + \frac{1}{2}\Delta t) + f'(t_n + \frac{1}{2}\Delta t)\frac{1}{2}\Delta t + f''(t_n + \frac{1}{2}\Delta t)(\frac{1}{2}\Delta t)^2 + f'''(t_n + \frac{1}{2}\Delta t)(\frac{1}{2}\Delta t)^3 + \cdots \tag{32}$$

Subtracting the first from the second gives

$$f(t_{n+1}) - f(t_n) = f'(t_n + \frac{1}{2}\Delta t)\Delta t + 2f'''(t_n + \frac{1}{2}\Delta t)(\frac{1}{2}\Delta t)^3 + \cdots \tag{33}$$

Solving with respect to $f'(t_n + \frac{1}{2}\Delta t)$ results in

$$f'(t_n + \frac{1}{2}\Delta t) \approx \frac{f(t_{n+1}) - f(t_n)}{\Delta t} - \frac{1}{4}f'''(t_n + \frac{1}{2}\Delta t)\Delta t^2 + c\cdots \tag{34}$$

This time the error measure goes like $\frac{1}{4}f'''\Delta t^2$, i.e., it is proportional to Δt^2 and not only Δt, which means that the error goes faster to zero as Δt is reduced. This means that the centered difference formula

$$f'(t_n + \frac{1}{2}\Delta t) \approx \frac{f(t_{n+1}) - f(t_n)}{\Delta t} \tag{35}$$

is more accurate than the forward and backward differences for small Δt.

1.8 Compact operator notation for finite differences

Finite difference formulas can be tedious to write and read, especially for differential equations with many terms and many derivatives. To save space and help the reader spot the nature of the difference approximations, we introduce a compact notation. For a function $u(t)$, a forward difference approximation is denoted by the D_t^+ operator and written as

$$[D_t^+ u]^n = \frac{u^{n+1} - u^n}{\Delta t} \quad \left(\approx \frac{d}{dt} u(t_n) \right) . \tag{36}$$

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

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

$$[D_t u]^n = \frac{u^{n+\frac{1}{2}} - u^{n-\frac{1}{2}}}{\Delta t} \approx \frac{d}{dt} u(t_n), \tag{37}$$

and

$$[D_t^- u]^n = \frac{u^n - u^{n-1}}{\Delta t} \approx \frac{d}{dt} u(t_n) . \tag{38}$$

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

An averaging operator is also convenient to have:

$$[\overline{u}^t]^n = \frac{1}{2}(u^{n-\frac{1}{2}} + u^{n+\frac{1}{2}}) \approx u(t_n) \tag{39}$$

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

With our compact notation, the Backward Euler finite difference approximation to $u' = -au$ can be written as

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

In difference equations we often place the square brackets around the whole equation, to indicate at which mesh point the equation applies, since each term must be approximated at the same point:

$$[D_t^- u = -au]^n . \tag{40}$$

Similarly, the Forward Euler scheme takes the form

$$[D_t^+ u = -au]^n, \tag{41}$$

while the Crank-Nicolson scheme is written as

$$[D_t u = -a\overline{u}^t]^{n+\frac{1}{2}}\,. \tag{42}$$

Question:

By use of (37) and (39), are you able to write out the expressions in (42) to verify that it is indeed the Crank-Nicolson scheme?

The θ-rule can be specified in operator notation by

$$[\bar{D}_t u = -a\overline{u}^{t,\theta}]^{n+\theta}, \,. \tag{43}$$

We define a new time difference

$$[\bar{D}_t u]^{n+\theta} = \frac{u^{n+1} - u^n}{t^{n+1} - t^n}, \tag{44}$$

to be applied at the time point $t_{n+\theta} \approx \theta t_n + (1-\theta)t_{n+1}$. This weighted average gives rise to the *weighted averaging operator*

$$[\overline{u}^{t,\theta}]^{n+\theta} = (1-\theta)u^n + \theta u^{n+1} \approx u(t_{n+\theta}), \tag{45}$$

where $\theta \in [0,1]$ as usual. Note that for $\theta = \frac{1}{2}$ we recover the standard centered difference and the standard arithmetic mean. The idea in (43) is to sample the equation at $t_{n+\theta}$, use a non-symmetric difference at that point $[\bar{D}_t u]^{n+\theta}$, and a weighted (non-symmetric) mean value.

An alternative and perhaps clearer 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 equations, we see immediately which difference approximations that have been used and at which point they apply. Therefore, the compact notation effectively communicates the reasoning behind turning a differential equation into a difference equation.

2 Implementation

We want to make a computer program for solving

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

by finite difference methods. The program should also display the numerical solution as a curve on the screen, preferably together with the exact solution.

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

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

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

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

2.1 Computer language: Python

Any programming language can be used to generate the u^{n+1} values from the formula above. However, in this document we shall mainly make use of Python. There are several good reasons for this choice:

- Python has a very clean, readable syntax (often known as "executable pseudo-code").
- Python code is very similar to MATLAB code (and MATLAB has a particularly widespread use for scientific computing).
- Python is a full-fledged, very powerful programming language.
- Python is similar to C++, but is much simpler to work with and results in more reliable 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), so that existing numerical software can be reused, and thereby easing high computational performance with new implementations.
- Python has extensive support for administrative tasks needed when doing large-scale computational investigations.
- Python has extensive support for graphics (visualization, user interfaces, web applications).

Learning Python is easy. Many newcomers to the language will probably learn enough from the forthcoming examples to perform their own computer experiments. The examples start with simple Python code and gradually make use

of more powerful constructs as we proceed. Unless it is inconvenient for the problem at hand, our Python code is made as close as possible to MATLAB code for easy transition between the two languages.

The coming programming examples assumes familiarity with variables, for loops, lists, arrays, functions, positional arguments, and keyword (named) arguments. A background in basic MATLAB programming is often enough to understand Python examples. Readers who feel the Python examples are too hard to follow will benefit from reading a tutorial, e.g.,

- The Official Python Tutorial
- Python Tutorial on tutorialspoint.com
- Interactive Python tutorial site
- A Beginner's Python Tutorial

The author also has a comprehensive book [3] that teaches scientific programming with Python from the ground up.

2.2 Making a solver function

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

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

An implementation of a numerical algorithm is often referred to as a *solver*. We shall now make a solver for our model problem and realize the solver as a Python function. The function must take the input data I, a, T, Δt, and θ of the problem as arguments and return the solution as arrays `u` and `t` for u^n and t^n, $n = 0, \ldots, N_t$. The solver function used as

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

One can now easily plot `u` versus `t` to visualize the solution.

The function `solver` may look as follows in Python:

```
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 function names are similar to what is found in the alternative scientific computing language MATLAB. Here we make use of

- `zeros(Nt+1)` for creating an array of 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`

The `for` loop deserves a comment, especially for newcomers to Python. The construction `range(0, Nt, s)` generates all integers from `0` to `Nt` in steps of `s`, *but not including* `Nt`. Omitting `s` means `s=1`. For example, `range(0, 6, 3)` gives `0` and `3`, while `range(0, 6)` generates the list `[0, 1, 2, 3, 4, 5]`. Our loop implies the following assignments to `u[n+1]`: `u[1]`, `u[2]`, ..., `u[Nt]`, which is what we want since `u` has length `Nt+1`. The first index in Python arrays or lists is *always* `0` and the last is then `len(u)-1` (the length of an array `u` is obtained by `len(u)` or `u.size`).

2.3 Integer division

The shown implementation of the `solver` may face problems and wrong results if `T`, `a`, `dt`, and `theta` are given as integers (see Exercises 3 and 4). The problem is related to *integer division* in Python (as in Fortran, C, C++, and many other computer languages!): `1/2` becomes `0`, while `1.0/2`, `1/2.0`, or `1.0/2.0` all become `0.5`. So, it is enough that at least the nominator or the denominator is a real number (i.e., a `float` object) to ensure a correct mathematical division. Inserting a conversion `dt = float(dt)` guarantees that `dt` is `float`.

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`. 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
```

2.4 Doc strings

Right below the header line in the `solver` function there is a Python 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 does not span 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 by extracting the definition of functions and the contents of doc strings.

It is strongly recommended to equip any function with a doc string, unless the purpose of the function is not obvious. Nevertheless, the forthcoming text deviates from this rule if the function is explained in the text.

2.5 Formatting numbers

Having computed the discrete solution `u`, it is natural to look at the numbers:

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

This compact `print` statement unfortunately gives less readable output because the `t` and `u` values are not aligned in nicely formatted columns. To fix this problem, we recommend to use the *printf format*, supported in most programming languages inherited from C. Another choice is Python's recent *format string syntax*. Both kinds of syntax are illustrated below.

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

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

The percentage signs signify "slots" in the text where the variables listed at the end of the statement are inserted. For each "slot" one must specify a format for

how the variable is going to appear in the string: `f` for float (with 6 decimals), `s` for pure text, `d` for an integer, `g` for a real number written as compactly as possible, `9.3E` for scientific notation with three decimals in a field of width 9 characters (e.g., `-1.351E-2`), or `.2f` for standard decimal notation with two decimals formatted with minimum width. The printf syntax provides a quick way of formatting tabular output of numbers with full control of the layout.

The alternative *format string syntax* looks like

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

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

2.6 Running the program

The function and main program shown above must be placed in a file, say with name `decay_v1.py` (`v1` for 1st version of this program). Make sure you write the code with a suitable text editor (Gedit, Emacs, Vim, Notepad++, or similar). The program is run by executing the file this way:

Terminal

```
Terminal> python decay_v1.py
```

The text `Terminal>` just indicates a prompt in a Unix/Linux or DOS terminal window. After this prompt, which may look different in your terminal window (depending on the terminal application and how it is set up), commands like `python decay_v1.py` can be issued. These commands are interpreted by the operating system.

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

Terminal

```
Terminal> ipython

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

The advantage of running programs in IPython are many, but here we explicitly mention a few of the most useful features:

- previous commands are easily recalled with the up arrow,
- `%pdb` turns on a debugger so that variables can be examined if the program aborts (due to a Python exception),
- output of commands are stored in variables,
- the computing time spent on a set of statements can be measured with the `%timeit` command,
- any operating system command can be executed,
- modules can be loaded automatically and other customizations can be performed when starting IPython

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

Terminal

```
Terminal> python programname
```

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

2.7 Plotting the solution

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

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

It will be illustrative to also plot the exact solution $u_{\mathrm{e}}(t) = Ie^{-at}$ for comparison. We first need to make a Python function for computing the exact solution:

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

It is tempting to just do

```
u_e = u_exact(t, I, a)
plot(t, u, t, u_e)
```

However, this is not exactly what we want: the `plot` function draws straight lines between the discrete points `(t[n], u_e[n])` while $u_{\mathrm{e}}(t)$ varies as an exponential function between the mesh points. The technique for showing the "exact" variation of $u_{\mathrm{e}}(t)$ between the mesh points is to introduce a very fine mesh for $u_{\mathrm{e}}(t)$:

```
t_e = linspace(0, T, 1001)      # fine mesh
u_e = u_exact(t_e, I, a)
```

We can also plot the curves with different colors and styles, e.g.,

```
plot(t_e, u_e, 'b-',          # blue line for u_e
     t,   u,   'r--o')        # red dashes w/circles
```

With more than one curve in the plot we need to associate each curve with a legend. We also want appropriate names on the axes, a title, and a file containing the plot as an image for inclusion in reports. The Matplotlib package (`matplotlib.pyplot`) contains functions for this purpose. The names of the functions are similar to the plotting functions known from MATLAB. A complete function for creating the comparison plot becomes

```
from matplotlib.pyplot import *

def plot_numerical_and_exact(theta, I, a, T, dt):
    """Compare the numerical and exact solution in a plot."""
    u, t = solver(I=I, a=a, T=T, dt=dt, theta=theta)

    t_e = linspace(0, T, 1001)        # fine mesh for u_e
    u_e = u_exact(t_e, I, a)

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

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

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

The complete code is found in the file `decay_v2.py`. The resulting plot is shown in Figure 6. As seen, there is quite some discrepancy between the exact and the numerical solution. Fortunately, the numerical solution approaches the exact one as Δt is reduced.

2.8 Verifying the implementation

It is easy to make mistakes while deriving and implementing numerical algorithms, so we should never believe in the solution before it has been thoroughly verified.

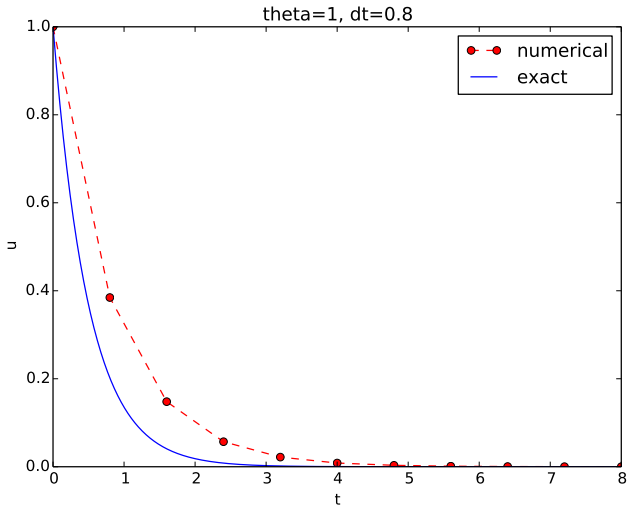


Figure 6: Comparison of numerical and exact solution.

Verification and validation.

The purpose of *verifying* a program is to bring evidence for the property that there are no errors in the implementation. A related term, *validate* (and *validation*), addresses the question if the ODE model is a good representation of the phenomena we want to simulate. To remember the difference between verification and validation, verification is about *solving the equations right*, while validation is about *solving the right equations*. We must always perform a verification before it is meaningful to believe in the computations and perform validation (which compares the program results with physical experiments or observations).

The most obvious idea for verification in our case is to compare the numerical solution with the exact solution, when that exists. This is, however, not a particularly good method. The reason is that there will always be a discrepancy between these two solutions, due to numerical approximations, and we cannot precisely quantify the approximation errors. The open question is therefore whether we have the mathematically correct discrepancy or if we have another, maybe small, discrepancy due to both an approximation error *and* an error in the implementation. It is thus impossible to judge whether the program is correct or not by just looking at the graphs in Figure 6.

To avoid mixing the unavoidable numerical approximation errors and the undesired implementation errors, we should try to make tests where we have

some exact computation of the discrete solution or at least parts of it. Examples will show how this can be done.

Running a few algorithmic steps by hand. The simplest approach to produce a correct non-trivial reference solution for the discrete solution u, is to compute a few steps of the algorithm by hand. Then we can compare the hand calculations with numbers produced by the program.

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

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

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

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

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

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

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

The `test_solver_three_steps` function follows widely used conventions for *unit testing*. By following such conventions we can at a later stage easily execute a big test suite for our software. That is, after a small modification is made to the program, we can by typing just a short command, run through a large number of tests to check that the modifications do not break any computations. The conventions boil down to three rules:

- The test function name must start with `test_` and the function cannot take any arguments.
- The test must end up in a boolean expression that is `True` if the test was passed and `False` if it failed.
- The function must run `assert` on the boolean expression, resulting in program abortion (due to an `AssertionError` exception) if the test failed.

The main program can routinely run the verification test prior to solving the real problem:

```
test_solver_three_steps()
plot_numerical_and_exact(I=1, a=2, T=8, dt=0.8, theta=1)
show()
```

(Rather than calling `test_*()` functions explicitly, one will normally ask a testing framework like nose or pytest to find and run such functions.) The complete program including the verification above is found in the file `decay_v3.py`.

2.9 Computing the numerical error as a mesh function

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

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

$$e^n = u_{\mathrm{e}}(t_n) - u^n, \quad n = 0, 1, \ldots, N_t\,. \tag{46}$$

We may view the mesh function $u^n_{\mathrm{e}} = u_{\mathrm{e}}(t_n)$ as a representation of the continuous function $u_{\mathrm{e}}(t)$ defined for all $t \in [0, T]$. In fact, u^n_{e} is often called the *representative* of u_{e} on the mesh. Then, $e^n = u^n_{\mathrm{e}} - u^n$ is clearly the difference of two mesh functions.

The error mesh function e^n can be computed by

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

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

Array arithmetics.

The last statements

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

demonstrate some standard examples of array arithmetics: `t` is an array of mesh points that we pass to `u_exact`. This function evaluates `-a*t`, which is a scalar times an array, meaning that the scalar is multiplied with each array element. The result is an array, let us call it `tmp1`. Then `exp(tmp1)` means applying the exponential function to each element in `tmp1`, giving an array, say `tmp2`. Finally, `I*tmp2` is computed (scalar times array) and `u_e` refers to this array returned from `u_exact`. The expression `u_e - u`

is the difference between two arrays, resulting in a new array referred to by `e`.

Replacement of array element computations inside a loop by array arithmetics is known as *vectorization*.

2.10 Computing the norm of the error mesh function

Instead of working with the error e^n on the entire mesh, we often want a single number expressing the 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 common norms are

$$||f||_{L^2} = \left(\int_0^T f(t)^2 dt\right)^{1/2}, \tag{47}$$

$$||f||_{L^1} = \int_0^T |f(t)| dt, \tag{48}$$

$$||f||_{L^\infty} = \max_{t\in[0,T]} |f(t)|\,. \tag{49}$$

The L^2 norm (47) ("L-two norm") has nice mathematical properties and is the most popular norm. It is a generalization of the well-known Eucledian norm of vectors to functions. The L^1 norm looks simpler and more intuitive, but has less nice mathematical properties compared to the two other norms, so it is much less used in computations. The L^∞ is also called the max norm or the supremum norm and is widely used. It focuses on a single point with the largest value of $|f|$, while the other norms measure average behavior of the function.

In fact, there is a whole family of norms,

$$||f||_{L^p} = \left(\int_0^T f(t)^p dt\right)^{1/p}, \tag{50}$$

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

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

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

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

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

This is called the discrete L^2 norm and denoted by ℓ^2. If $||f||_{\ell^2}^2$ (i.e., the square of the norm) is used instead of the Trapezoidal integration formula, the error is $\Delta t((f^0)^2 + (f^{N_t})^2)/2$. This means that the weights at the end points of the mesh function are perturbed, but as $\Delta t \to 0$, the error from this perturbation goes to zero. As long as we are consistent and stick to one kind of integration rule for the norm of a mesh function, the details and accuracy of this rule is of no concern.

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

$$||f^n||_{\ell^2} = \left(\Delta t\sum_{n=0}^{N_t}(f^n)^2\right)^{1/2}, \tag{51}$$

$$||f^n||_{\ell^1} = \Delta t\sum_{n=0}^{N_t}|f^n|, \tag{52}$$

$$||f^n||_{\ell^\infty} = \max_{0\le n\le N_t}|f^n|. \tag{53}$$

Note that the L^2, L^1, ℓ^2, and ℓ^1 norms depend on the length of the interval of interest (think of $f = 1$, then the norms are proportional to $\sqrt{T}$ or T). In some applications it is convenient to think of a mesh function as just a vector of function values without any relation to the interval $[0, T]$. Then one can replace Δt by T/N_t and simply drop T (which is just a common scaling factor in the norm, independent of the vector of function values). Moreover, people prefer to divide by the total length of the vector, $N_t + 1$, instead of N_t. This reasoning gives rise to the *vector norms* for a vector $f = (f_0, \ldots, f_N)$:

$$||f||_2 = \left(\frac{1}{N+1}\sum_{n=0}^{N}(f_n)^2\right)^{1/2}, \tag{54}$$

$$||f||_1 = \frac{1}{N+1}\sum_{n=0}^{N}|f_n|, \tag{55}$$

$$||f||_{\ell^\infty} = \max_{0\le n\le N}|f_n|. \tag{56}$$

Here we have used the common vector component notation with subscripts (f_n) and N as length. We will mostly work with mesh functions and use the discrete ℓ^2 norm (51) or the max norm ℓ^∞ (53), but the corresponding vector norms (54)-(56) are also much used in numerical computations, so 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 taken as $||e^n||_{\ell^2}$ and called E:

$$E = \sqrt{\Delta t \sum_{n=0}^{N_t} (e^n)^2} \tag{57}$$

The corresponding Python code, using array arithmetics, reads

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

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

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

```
m = len(u)     # length of u array (alt: u.size)
u_e = zeros(m)
t = 0
for i in range(m):
    u_e[i] = u_exact(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)
```

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

2.11 Experiments with computing and plotting

Let us write down a new function that wraps up the computation and all the plotting statements used for comparing the exact and numerical solutions. This function can be called with various θ and Δt values to see how the error depends on the method and 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 = u_exact(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 = u_exact(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))
        show()
    return E
```

The `figure()` call is key: without it, a new `plot` command will draw the new pair of curves in the same plot window, while we want the different pairs to appear in separate windows and files. Calling `figure()` ensures this.

Instead of including the θ value in the filename to implicitly inform about the applied method, the code utilizes a little Python dictionary that maps each relevant θ value to a corresponding acronym for the method name (FE, BE, or CN):

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

The `explore` function stores the plot in two different image file formats: PNG and PDF. The PNG format is suitable for being included in HTML documents, while the PDF format provides higher quality for LaTeX (i.e., pdfLaTeX) documents. Frequently used viewers for these image files on Unix systems are `gv` (comes with Ghostscript) for the PDF format and `display` (from the ImageMagick software suite) for PNG files:

Terminal

```
Terminal> gv BE_0.5.pdf
Terminal> display BE_0.5.png
```

A main program may run a loop over the three methods (given by their corresponding θ values) and call `explore` to compute errors and make plots:

```
def main(I, a, T, dt_values, theta_values=(0, 0.5, 1)):
    print 'theta   dt       error'  # Column headings in table
    for theta in theta_values:
        for dt in dt_values:
```

```
            E = explore(I, a, T, dt, theta, makeplot=True)
            print '%4.1f %6.2f: %12.3E' % (theta, dt, E)

main(I=1, a=2, T=5, dt_values=[0.4, 0.04])
```

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

Terminal

```
Terminal> python decay_plot_mpl.py
theta   dt        error
 0.0   0.40:    2.105E-01
 0.0   0.04:    1.449E-02
 0.5   0.40:    3.362E-02
 0.5   0.04:    1.887E-04
 1.0   0.40:    1.030E-01
 1.0   0.04:    1.382E-02
```

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

Figure 7 demonstrates that the numerical solution produced by the Forward Euler method with $\Delta t = 0.4$ clearly lies below the exact curve, but that the accuracy improves considerably by reducing the time step by a factor of 10.

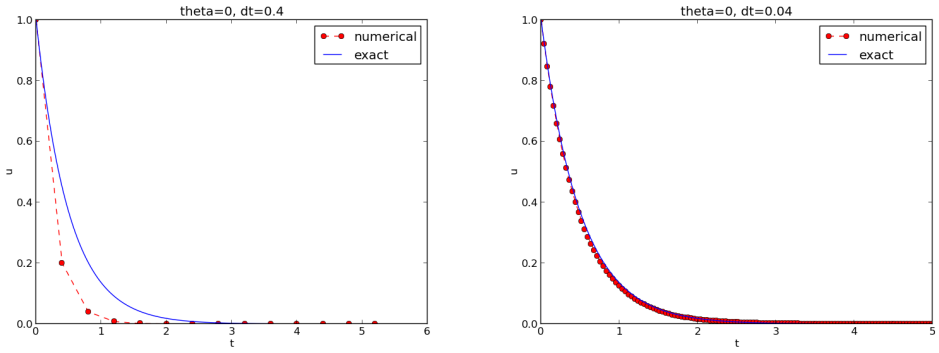


Figure 7: The Forward Euler scheme for two values of the time step.

The behavior of the two other schemes is shown in Figures 8 and 9. Crank-Nicolson is obviously the most accurate scheme from this visual point of view.

Combining plot files. Mounting two PNG files beside each other, as done in Figures 7-9, is easily carried out by the `montage` program from the ImageMagick suite:

Terminal

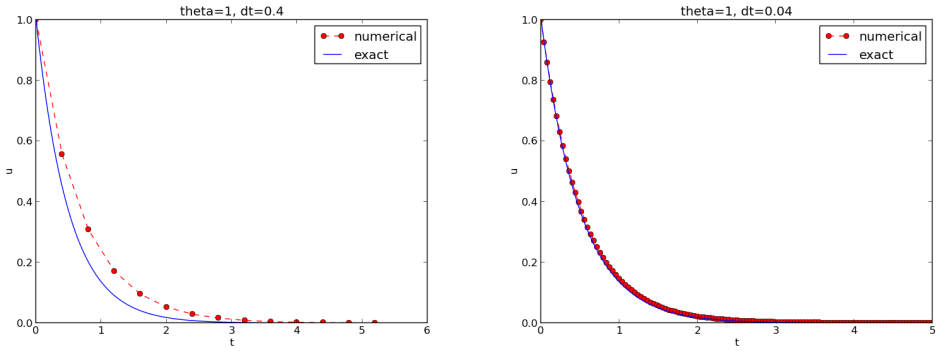


Figure 8: The Backward Euler scheme for two values of the time step.

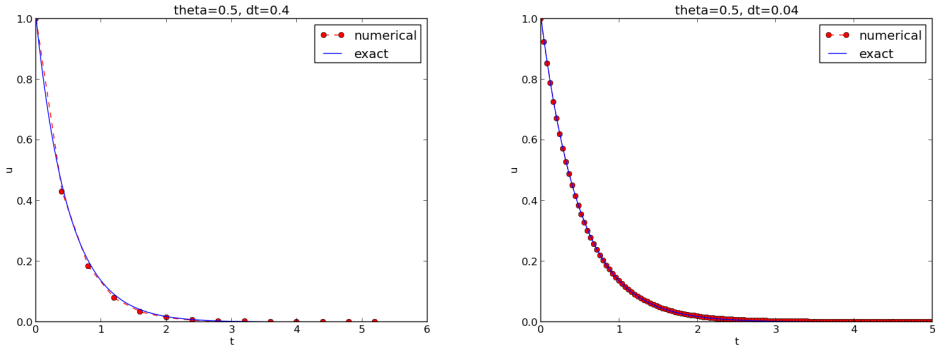


Figure 9: The Crank-Nicolson scheme for two values of the time step.

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

The `-geometry` argument is used to specify the size of the image. Here, we preserve the individual 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 then listed, with the name of the resulting combined image, here `FE1.png` at the end. The `convert -trim` command removes surrounding white areas in the figure (an operation usually known as *cropping* in image manipulation programs).

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

Terminal

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

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

Plotting with SciTools. The SciTools package provides a unified plotting interface, called Easyviz, to many different plotting packages, including Matplotlib, Gnuplot, Grace, MATLAB, VTK, OpenDX, and VisIt. The syntax is very similar to that of Matplotlib and MATLAB. In fact, the plotting commands shown above look the same in SciTool's Easyviz interface, apart from 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 pieces of the Easyviz (`scitools.easyviz`) package, along with some additional numerical functionality.

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

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

The `decay_plot_st.py` file contains such a demo.

By default, Easyviz employs Matplotlib for plotting, but Gnuplot and Grace are viable alternatives:

Terminal

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

The actual tool used for creating plots (called *backend*) and numerous other options can be permanently set in SciTool's configuration file.

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

For more detailed information on syntax and plotting capabilities, we refer to the Matplotlib [1] and SciTools [2] documentation. The hope is that the programming syntax explained so far suffices for understanding the basic plotting functionality and being able to look up the cited technical documentation.

2.12 Memory-saving implementation

The computer memory requirements of our implementations so far consist mainly of the `u` and `t` arrays, both of length $N_t + 1$. Also, for the programs that involve array arithmetics, Python needs memory space for storing temporary arrays. For example, computing `I*exp(-a*t)` requires storing the intermediate result `a*t` before the preceding minus sign can be applied. The resulting array is temporarily stored and provided as input to the `exp` function. Regardless of how we implement simple ODE problems, storage requirements are very modest and put no restrictions on how we choose our data structures and algorithms. Nevertheless, when the presented methods are applied to three-dimensional PDE problems, memory storage requirements suddenly become a challenging issue.

Let us briefly elaborate on how large the storage requirements can quickly be in three-dimensional problems. The PDE counterpart to our model problem $u' = -a$ is a diffusion equation $u_t = a\nabla^2 u$ posed on a space-time domain. The discrete representation of this domain may in 3D be a spatial mesh of M^3 points and a time mesh of N_t points. In many applications, it is quite typical that M is at least 100, or even 1000. Storing all the computed u values, like we have done in the programs so far, would demand storing arrays of size up to $M^3 N_t$. This would give a factor of M^3 larger storage demands compared to what was required by our ODE programs. Each real number in the `u` array requires 8 bytes (b) of storage. With $M = 100$ and $N_t = 1000$, there is a storage demand of $(10^3)^3 \cdot 1000 \cdot 8 = 8$ Gb for the solution array. Fortunately, we can usually get rid of the N_t factor, resulting in 8 Mb of storage. Below we explain how this is done (the technique is almost always applied in implementations of PDE problems).

Let us critically evaluate how much we really need to store in the computer's memory for our implementation of the θ method. To compute a new u^{n+1}, all we need is u^n. This implies that the previous $u^{n-1}, u^{n-2}, \ldots, u^0$ values do not need to be stored, although this is convenient for plotting and data analysis in the program. Instead of the `u` array we can work with two variables for real numbers, `u` and `u_1`, representing u^{n+1} and u^n in the algorithm, respectively. At each time level, we update `u` from `u_1` and then set `u_1 = u`, so that the computed u^{n+1} value becomes the "previous" value u^n at the next time level. The downside is that we cannot plot the solution after the simulation is done since only the last two numbers are available. The remedy is 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`, which is a slight modification of `decay_plot_mpl.py` program.

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

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

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

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

```
def read_file(filename='sol.dat'):
    infile = open(filename, 'r')
    u = [];  t = []
    for line in infile:
        words = line.split()
        if len(words) != 2:
            print 'Found more than two numbers on a line!', words
            sys.exit(1)  # abort
        t.append(float(words[0]))
        u.append(float(words[1]))
    return np.array(t), np.array(u)
```

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

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

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

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

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

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

3 Exercises

Exercise 1: Define a mesh function and visualize it

a) Write a function `mesh_function(f, t)` that returns an array with mesh point values $f(t_0), \ldots, f(t_{N_t})$, where `f` is a Python function implementing a mathematical function `f(t)` and $t_0, \ldots, t_{N_t}$ are mesh points stored in the array `t`. Use a loop over the mesh points and compute one mesh function value at the time.

Solution. The function may look like

```
def mesh_function(f, t):
    u = np.zeros(len(t))  # or t.size
    for i in range(len(t)):
        u[i] = f(t[i])
    return u
```

b) Use `mesh_function` to compute the mesh function corresponding to

$$f(t) = \begin{cases} e^{-t}, & 0 \leq t \leq 3, \\ e^{-3t}, & 3 < t \leq 4 \end{cases}$$

Choose a mesh $t_n = n\Delta t$ with $\Delta t = 0.1$. Plot the mesh function.

Solution. An appropriate function is

```
def demo():
    def f(t):
        if t <= 3:
            return np.exp(-t)
        else:
            return np.exp(-3*t)

    # Compute mesh and mesh function
    t = np.linspace(0, 4, 41)
    u = mesh_function(f, t)

    # Plot
    import matplotlib.pyplot as plt
```

```
    plt.plot(t, u)
    plt.xlabel('t')
    plt.ylabel('mesh function')
    plt.savefig('tmp.png'); plt.savefig('tmp.pdf')
    plt.show()
```

Filename: `mesh_function`.

Remarks. In Section 2.9 we show how easy it is to compute a mesh function by array arithmetics (or array computing). Using this technique, one could simply implement `mesh_function(f,t)` as `return f(t)`. However, `f(t)` will not work if there are if tests involving `t` inside `f` as is the case in b). Typically, `if t < 3` must have `t < 3` as a boolean expression, but if `t` is array, `t < 3`, is an *array of boolean values*, which is not legal as a boolean expression in an if test. Computing one element at a time as suggested in a) is a way of out of this problem.

We also remark that the function in b) is the solution of $u' = -au$, $u(0) = 1$, for $t \in [0, 4]$, where $a = 1$ for $t \in [0, 3]$ and $a = 3$ for $t \in [3, 4]$.

Problem 2: Differentiate a function

Given a mesh function u^n as an array `u` with u^n values at mesh points $t_n = n\Delta t$, the discrete derivative can be based on centered differences:

$$d^n = [D_{2t}u]^n = \frac{u^{n+1} - u^{n-1}}{2\Delta t}, \quad n = 1, \ldots, N_t - 1\,. \tag{58}$$

At the end points we use forward and backward differences:

$$d^0 = [D_t^+ u]^n = \frac{u^1 - u^0}{\Delta t},$$

and

$$d^{N_t} = [D_t^- u]^n = \frac{u^{N_t} - u^{N_t-1}}{\Delta t}\,.$$

a) Write a function `differentiate(u, dt)` that returns the discrete derivative d^n of the mesh function u^n. The parameter `dt` reflects the mesh spacing Δt. Write a corresponding test function `test_differentiate()` for verifying the implementation.

Hint. The three differentiation formulas are exact for quadratic polynomials. Use this property to verify the program.

Solution. The functions can be written as

```
import numpy as np

def differentiate(u, dt):
    dudt = np.zeros(len(u))
    for i in range(1, len(dudt)-1, 1):
        dudt[i] = (u[i+1] - u[i-1])/(2*dt)
    i = 0
    dudt[i] = (u[i+1] - u[i])/dt
    i = len(dudt)-1
    dudt[i] = (u[i] - u[i-1])/dt
    return dudt

def test_differentiate():
    """Test differentiate with a linear u."""
    # Expect exact results
    t = np.linspace(0, 4, 9)
    u = 2*t + 7
    dudt = differentiate(u, dt=t[1]-t[0])
    diff = abs(dudt - 2).max()
    tol = 1E-15
    assert diff < tol
```

b) A standard implementation of the formula (58) is to have a loop over i. For large N_t, such loop may run slowly in Python. A technique for speeding up the computations, called vectorization or array computing, replaces the loop by array operations. To see how this can be done in the present mathematical problem, we define two arrays

$$u^+ = (u^2, u^3, \ldots, u^{N_t}), u^- = (u^0, u^1, \ldots, u^{N_t-2}) .$$

The formula (58) can now be expressed as

$$(d^1, d^2, \ldots, d^{N_t-1}) = \frac{1}{2\Delta t}(u^+ - u^-) .$$

The corresponding Python code reads

```
d[1:-1] = (u[2:] - u[0:-2])/(2*dt)
# or
d[1:N_t] = (u[2:N_t+1] - u[0:N_t-1])/(2*dt)
```

Recall that an array slice `u[1:-1]` contains the elements in `u` starting with index 1 and going all indices up to, but not including, the last one (`-1`).

Use the ideas above to implement a vectorized version of the `differentiate` function without loops. Make a corresponding test function that compares the result with that of `differentiate`.

Solution. Appropriate functions are

```
def differentiate_vec(u, dt):
    dudt = np.zeros(len(u))
    dudt[1:-1] = (u[2:] - u[0:-2])/(2*dt)
    dudt[0] = (u[1] - u[0])/dt
```

```
    dudt[-1] = (u[-1] - u[-2])/dt
    return dudt

def test_differentiate_vec():
    """Test differentiate_vec by comparing with differentiate."""
    t = np.linspace(0, 4, 9)
    u = 2*np.sin(t) + 7
    dudt_expected = differentiate(u, dt=t[1]-t[0])
    dudt_computed = differentiate_vec(u, dt=t[1]-t[0])
    diff = abs(dudt_expected - dudt_computed).max()
    tol = 1E-15
    assert diff < tol
```

Filename: `differentiate`.

Problem 3: Experiment with divisions

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

```
>>> dt = 3
>>> T = 8
>>> Nt = T/dt
>>> Nt
2
>>> theta = 1; a = 1
>>> (1 - (1-theta)*a*dt)/(1 + theta*dt*a)
0
```

Solution. We add some more investigations of the types and values involved in the computations:

```
>>> dt = 3
>>> T = 8
>>> Nt = T/dt
>>> Nt
2
>>> type(Nt)
<type 'int'>
>>> from numpy import linspace
>>> theta = 1; a = 1
>>> (1 - (1-theta)*a*dt)/(1 + theta*dt*a)
0
>>> (1 - (1-theta)*a*dt)
1
>>> (1 + theta*dt*a)
2
```

From this we realize that the unexpected results are caused by integer division: `int` object divided by `int` object. For example, `Nt` is an integer, not a real as we want, because 8/3 according integer division is 2, not the 2.66666 approximation to $\frac{8}{3}$.
Filename: `pyproblems`.

Problem 4: Experiment with wrong computations

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

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

The output becomes

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

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

Solution. A code for investigating this problem may look as follows.

```
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
        factor1 = (1 - (1-theta)*a*dt)
        factor2 = (1 + theta*dt*a)
        factor3 = factor1/factor2
        print factor1, type(factor1), factor2, type(factor2),
        print factor3, type(factor3)
        u[n+1] = (1 - (1-theta)*a*dt)/(1 + theta*dt*a)*u[n]
    return u, t

from numpy import *
u, t = solver(I=1, a=1, T=7, dt=2, theta=1)

# Write out a table of t and u values:
for i in range(len(t)):
    print 't=%6.3f u=%g' % (t[i], u[i])
    # or print 't={t:6.3f} u={u:g}'.format(t=t[i], u=u[i])
```

Running this code shows

Terminal

```
Terminal> python decay_v1_err.py
1 <type 'int'> 3 <type 'int'> 0 <type 'int'>
1 <type 'int'> 3 <type 'int'> 0 <type 'int'>
1 <type 'int'> 3 <type 'int'> 0 <type 'int'>
t= 0.000 u=1
t= 2.000 u=0
t= 4.000 u=0
t= 6.000 u=0
```

We realize that we have integer divided by integer in the numerical formula, i.e., the result of $1/3$ is 0 because of integer division. This fraction is multiplied by `u[n]`, but the result remains zero.
Filename: `decay_v1_err`.

Problem 5: Plot the error function

Solve the problem $u' = -au$, $u(0) = I$, using the Forward Euler, Backward Euler, and Crank-Nicolson schemes. For each scheme, plot the error mesh function $e^n = u_{\mathrm{e}}(t_n) - u^n$ for $\Delta t = 0.1, 0.05, 0.025$, where u_{e} is the exact solution of the ODE and u^n is the numerical solution at mesh point t_n.

Hint. Modify the `decay_plot_mpl.py` code.

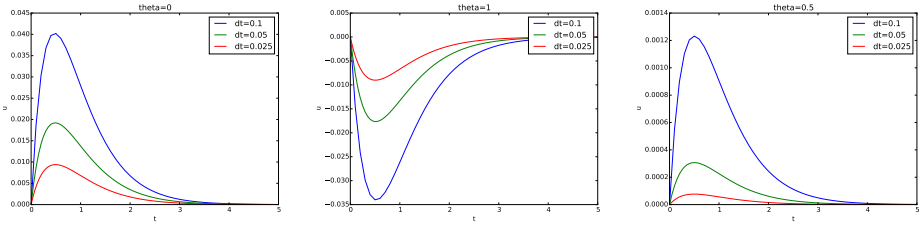
Solution. Looking at the `decay_plot_mpl.py` file, we realize that we only need to change the `explore` method such that the error and not the solutions are plotted. We also need to run a loop over Δt values and get all the corresponding curves in the same plot. Quite some editing is necessary. Also the `main` function needs some edits.

The `solver` and `exact_solution` functions are not altered. The `explore` and `main` functions as well as the call to `main` are edited to:

```
def explore(I, a, T, dt_values, theta=0.5):
    """
    Run cases with the solver, compute error measure,
    and plot the error.
    """
    figure() # create new plot
    for dt in dt_values:
        print 'dt', dt
        u, t = solver(I, a, T, dt, theta)
        # Numerical solution
        u_e = u_exact(t, I, a)
        e = u_e - u
        plot(t, e)
        hold('on')
    legend(['dt=%g' % dt for dt in dt_values])
    xlabel('t')
    ylabel('u')
    title('theta=%g' % theta)
    theta2name = {0: 'FE', 1: 'BE', 0.5: 'CN'}
    savefig('%s_%g.png' % (theta2name[theta], dt))
    savefig('%s_%g.pdf' % (theta2name[theta], dt))

def main(I, a, T, dt_values, theta_values=(0, 0.5, 1)):
    for theta in theta_values:
        explore(I, a, T, dt_values, theta)

dt = 0.1
dt_values = [dt, dt/4, dt/8]
main(I=1, a=2, T=5, dt_values=dt_values)
show()
```



Filename: `decay_plot_error`.

Problem 6: Change formatting of numbers and debug

The `decay_memsave.py` program writes the time values and solution values to a file which looks like

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

Modify the file output such that it looks like

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

If you have just modified the formatting of numbers in the file, running the modified program

Terminal

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

leads to printing of the message `Bug in the implementation!` in the terminal window. Why?

Solution. The new formatting is obtained by replacing the format `%.16E` for `t` by `%.3f` and the format for `u` must be `%.5f`.

With only 5 decimals in the file, the `test_solver_minmem` function compares truncated elements `u`, accurate only to 10^{-5} with the exact discrete solution and applies a far too small `tol` value. `tol` must be `1E-4`.
Filename: `decay_memsave_v2`.

References

[1] J. D. Hunter, D. Dale, E. Firing, and M. Droettboom. Matplotlib documentation, 2012. http://matplotlib.org/users/.

[2] H. P. Langtangen. SciTools documentation. http://hplgit.github.io/scitools/doc/web/index.html.

[3] H. P. Langtangen. *A Primer on Scientific Programming with Python*. Texts in Computational Science and Engineering. Springer, fourth edition, 2014.

Index