# Discretizing first-order ODEs by finite difference methods

**Hans Petter Langtangen**

Aug 23, 2012

**WARNING: ULTRA-PRELIMINARY VERSION!**

## Contents

Finite difference methods for partial differential equations (PDEs) employ a range of concepts and tools that can be introduced and illustrated in the context of simple ordinary differential equation (ODE) examples. By first working with ODEs, we keep the mathematical problems to be solved as simple as possible (but no simpler), thereby allowing full focus on understanding the concepts and tools that will be reused and futher extended when addressing finite difference methods for time-dependent PDEs. The forthcoming treatment of ODEs is therefore solely dominated by reasoning and methods that directly carry over to numerical methods for PDEs.

We study two model problems: an ODE for a decaying phenomena, which will be relevant for PDEs of diffusive nature, and an ODE for oscillating phenomena, which will be relevant for PDEs of wave nature. Both problems are linear with known analytical solutions such that we can easily assess the quality of various numerical methods and analyze their behavior.

# 1 Finite difference methods for an ODE

The purpose of this module is to explain finite difference methods in detail for a simple ordinary differential equation (ODE). Emphasis is put on the reasoning when discretizing the problem, various ways of programming the methods, how to verify that the implementation is correct, experimental investigations of the numerical behavior of the methods, and theoretical analysis of the methods to explain the observations.

## 1.1 A decay problem

Our model problem is perhaps the simplest ODE:

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

Here, $a > 0$ is a constant and $u'(t)$ means differentiation with respect to time $t$. This type of equation arises in a number of widely different phenomena where some quantity $u$ undergoes exponential reduction. Examples include radioactive decay, population decay, investment decay, cooling of an object, pressure decay in the atmosphere, and retarded motion in fluids (for some of these models, $a$ can be negative as well). Studying numerical solution methods for this simple ODE gives imporant insight that can be reused for diffusion PDEs.

The analytical solution of the ODE is found by the method of separation of variables, resulting in

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

for any arbitrary constant $C$. 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 stated as $u(0) = I$. That is, we

know the value $I$ of $u$ when the process starts at $t = 0$. The exact solution is then $u(t) = I\exp(-at)$.

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

## 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. 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 a finite number of $N + 1$ points

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

The collection of points $t_0, t_1, \ldots, t_N$ 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 then often denoted by $\Delta t$, in this case $t_n = n\Delta t$.

We seek the solution $u$ at the mesh points: $u(t_n)$, $n = 1, 2, \ldots, N$ (note that $u^0$ is already known as $I$). A notational short-form for $u(t_n)$, which will be used extensively, is $u^n$. More precisely, we let $u^n$ be the *numerical approximation* to the exact solution at $t = t_n$, $u(t_n)$. When we need to clearly distinguish the numerical and the 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 $n = 0, 1, \ldots, N = 7$ as well as $u_{\mathrm{e}}(t)$ as the dashed line.

Since finite difference methods produce solutions at the mesh points only, it is an open question what the solution is between the mesh points. One can

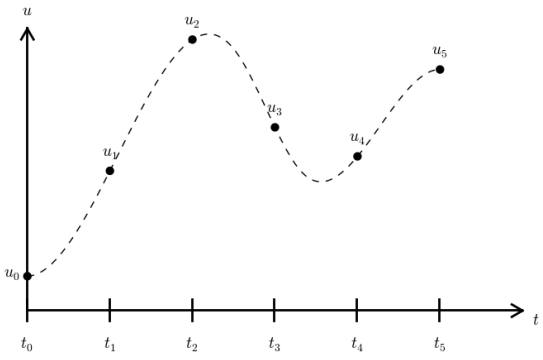


Figure 1: Time mesh with discrete solution values.

use methods for interpolation to compute the value of $u$ between mesh points. The simplest (and most widely used) interpolation method is to assume that $u$ varies linearly between the mesh points, see Figure 2. Given $u^n$ and $u^{n+1}$, the value of $u$ at some $t \in [t_n, t_{n+1}]$ is by linear interpolation

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

**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 $t_1, t_2, \ldots, t_N$ are a natural choice of points. The original ODE is then reduced to the following $N$ equations:

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

**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 one-sided difference approximation (see Figure 3),

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

Inserting this approximation in (4) results in

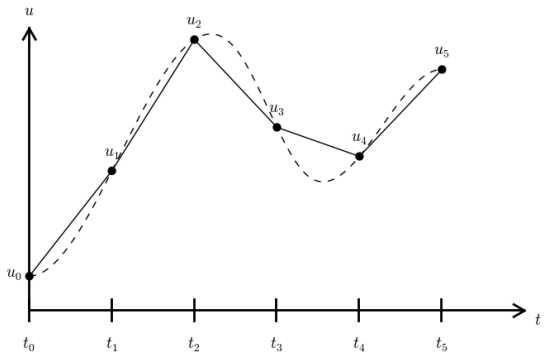


Figure 2: Linear interpolation between the discrete solution values (dashed curve is exact solution).

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

This equation is the discrete counterpart to the original ODE problem (1), and often known as a *finite difference scheme*, which yields a straightforward way to compute the solution at the mesh points ($u(t_n)$, $n = 1, 2, \ldots, N$) as shown next.

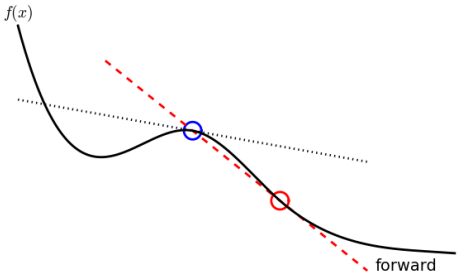


Figure 3: Illustration of a forward difference.

**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 $u$, 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:

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

and so on until we reach $u^N$. In the case $t_{n+1} - t_n$ is a constant, denoted by $\Delta t$, we realize from the above calculations that

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

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.

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

## 1.3 The Backward Euler scheme

There are many 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 the Backward Euler difference. Figure 5 explains the idea.

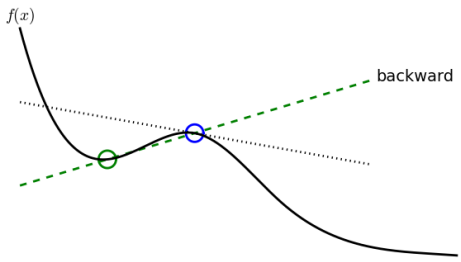


Figure 4: Illustration of a backward difference.

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

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

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

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

## 1.4 The Crank-Nicolson scheme

The finite difference approximations used to derive the schemes (7) and (10) are both one-sided differences, known to be less accurate than central (or midpoint) differences. We shall now 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 the mesh spacing is uniform in time. The approximation reads

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

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 $[t_{n+1}, t_n]$ or at the end points.

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

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

Using (11) in (12) results in

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

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 integer $n$, implying that $u^{n+\frac{1}{2}}$ is not a quantity computed by our method. It must be expressed by the quantities that we actually produce, i.e., $u$ at the mesh points. One possibility is to approximate $u^{n+\frac{1}{2}}$ as an average 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

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

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

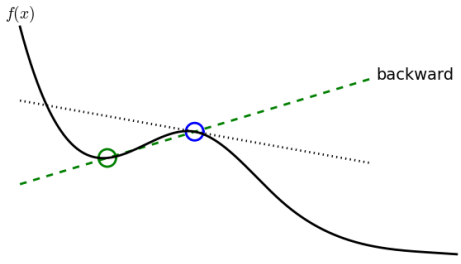


Figure 5: Illustration of a centered difference.

We assume that $u^n$ is already computed so that $u^{n+1}$ is the unknown, which 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 known as the midpoint scheme or the Crank-Nicolson (CN) scheme. We shall use the latter name.

## 1.5 The unifying $\theta$-rule

Let us reconsider the derivation of the Forward Euler, Backward Euler, and Crank-Nicolson schemes. In all the mentioned schemes we replace $u'$ by the fraction

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

and the difference between the methods lies in which point this fraction approximates the derivative; i.e., in 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 average $\frac{1}{2}u^n + \frac{1}{2}u^{n+1}$. If we express $\tilde{t}$ as a weighted average

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

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

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

We can now let 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 (17). The result is

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

This is a generalized scheme for our model problem: $\theta = 0$ gives the Forward Euler scheme, $\theta = 1$ gives the Backward Euler scheme, and $\theta = 1/2$ gives the Crank-Nicolson scheme. In addition, we may 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:

$$u^{n+1} = \frac{1 - (1 - \theta)a(t_{n+1} - t_n)}{1 + \theta a(t_{n+1} - t_n)} \,. \tag{19}$$

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

## 1.6 Constant time step

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

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

$$u^{n+1} = (1 - a\Delta t)u^n \qquad \text{FE} \tag{20}$$

$$u^{n+1} = \frac{1}{1 + a\Delta t}u^n \qquad \text{BE} \tag{21}$$

$$u^{n+1} = \frac{1 - \frac{1}{2}a\Delta t}{1 + \frac{1}{2}a\Delta t}u^n \qquad \text{CN} \tag{22}$$

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

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

## 1.7 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 of the scheme to quickly see the nature of the difference approximations, we introduce a compact notation:

$$[D_t u]^n = \frac{u^{n+\frac{1}{2}} - u^{n-\frac{1}{2}}}{\Delta t} \approx \frac{d}{dt}u(t_n) \tag{24}$$

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

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

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 variable as subscript and a superscript $^-$ for a backward difference

and $^+$ for a forward difference. No superscript implies a central 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.

An averaging operator is also convenient to have:

$$[\overline{u}^t]^n = \frac{1}{2}(u^{n-\frac{1}{2}} + u^{n+\frac{1}{2}}) \approx u(t_n) \tag{27}$$

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 $[\overline{u}^t]^{n+1/2}$.

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

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

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 is supposed to be approximated at the same point:

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

The Forward Euler scheme takes the form

$$[D_t^+ u = -au]^n, \tag{29}$$

while the Crank-Nicolson scheme is written as

$$[D_t u = -a\overline{u}^t]^{n+\frac{1}{2}} \,. \tag{30}$$

Just apply (24) and (27) and write out the expressions to see that (30) is indeed the Crank-Nicolson scheme.

The $\theta$-rule can be specified by

$$[\bar{D}_t u = -a\overline{u}^{t,\theta}]^{n+\theta}, \tag{31}$$

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

$$[\bar{D}_t u]^{n+\theta} = \frac{u^{n+1} - u^n}{t^{n+1} - t^n}, \tag{32}$$

$$[\overline{u}^{t,\theta}]^{n+\theta} = (1-\theta)u^n + \theta u^{n+1} \approx u(t_{n+\theta}), \tag{33}$$

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

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

Looking at the various examples above and comparing them with the underlying differential equations, we see immediately which difference approximations

that have been used and at which point they apply. Therefore, the compact notation efficiently communicates the reasoning behind turning a differential equation into a difference equation.

# 2 Implementation

The purpose now is to make a computer program for solving

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

and display the solution on the screen, preferably together with the exact solution. We shall also be concerned with how we can test that the implementation is correct.

All programs referred to in this section are found in the `src/decay` directory.

**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 points of view, it is advantageous to implement the $\theta$-rule

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

since it can generate the three other schemes by various of 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 $\Delta t$, our task is to use the $\theta$-rule to compute $u^1, u^2, \ldots, u^N$, where $t_N = N\Delta t$, and $N$ the closest integer to $T/\Delta t$.

**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 of several reasons:

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

- Python has extensive support for administrative task needed when doing large-scale computational investigations.
- Python has extensive support for graphics (visualization, user interfaces, web applications).
- FEniCS, a very powerful tool for solving PDEs by the finite element method, is most human-efficient to operate from Python.

Learning Python is easy. Many newcomers to the language will probably learn enough from the examples to perform their own computer experiements. The examples start with simple Python code and gradually make use of more powerful constructs as we proceed. As long as it is not 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.

## 2.1 Making a program

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

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

**Function for computing the numerical solution.** The following Python function takes the input data of the problem ($I$, $a$, $T$, $\Delta t$, $\theta$) as arguments and returns two arrays with the solution $u^0, \ldots, u^N$ and the mesh points $t_0, \ldots, t_N$, respectively:

```
from numpy import *

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

    u[0] = I                 # assign initial condition
    for n in range(0, N):    # n=0,1,...,N-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 the alternative scientific computing language MATLAB. Here we make use of

- `zeros(N+1)` for creating an array of a size `N+1` and initializing the elements to zero

- `linspace(0, T, N+1)` for creating an array with `N+1` coordinates uniformly distributed between `0` and `T`

The `for` loop deserves a comment, especially for newcomers to Python. The construction `range(0, N, s)` generates all integers from `0` to `N` in steps of `s`, *but not including* `N`. Omitting `s` means `s=1`. For example, `range(0, 6, 3)` gives `0` and `3`, while `range(0, N)` generates `0`, `1`, ..., `N-1`. In our loop, `n` takes on the values generated by `range(0, N)`, implying the following assignments `u[n+1]`: `u[1]`, `u[2]`, ..., `u[N]`, which is what we want since `u` has length `N+1`. The first index in Python arrays or lists is *always* `0` and the last is then `len(u)-1`.

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

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

**Integer division.** The shown implementation of the `theta_rule` may face problems and wrong results if `T`, `a`, `dt`, and `theta` are given as integers, see Exercises **??** and **??**. The problem is related to *integer division* in Python (as well as in Fortran, C, and C++): `1/2` becomes `0`, while `1.0/2`, `1/2.0`, or `1.0/2.0` all become are `0.5`. It is enough that at least the nominator or the denominator is a real number (i.e., a `float` object) to ensure correct mathematical division. Inserting a conversion `dt = float(dt)` guarantees that `dt` is `float` and avoids problems in Exercise refrefdecay:exer:decay1err.

Another problem with computing $N = T/\Delta t$ is that we should round $N$ to the nearest integer. With `N = int(T/dt)` the `int` operation picks the largest integer smaller than `T/dt`. Correct rounding is obtained by

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

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

```
from numpy import *

def theta_rule(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
    N = int(round(T/dt))     # no of time intervals
    T = N*dt                 # adjust T to fit time step dt
    u = zeros(N+1)           # array of u[n] values
    t = linspace(0, T, N+1)  # time mesh

    u[0] = I                 # assign initial condition
    for n in range(0, N):    # n=0,1,...,N-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 `theta_rule` 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 do not span more than one line, but with triple double quoted strings the text may span several lines:

```
def theta_rule(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 extract the definition of functions and the contents of doc strings and then produce nicely formatted documentation.

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

**Formatting of 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]
```

The convenient `print` statement gives unfortunately quite ugly 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 most programming languages with inheritage from C, or Python's recent *format string syntax*.

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: `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 with 12 (e.g., `-1.351E-2`), or `.2f` for a standard decimal notation, here 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 corresponding 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.

**Running the program.** The function and main program shown above must be placed in a file, say with name `decay1.py`. 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 decay1.py
```

The text `Terminal>` just signifies the prompt that one has in a Unix/Linux or DOS terminal window.

We strongly recommend to run Python programs within the IPython shell. First start IPython by typing

Terminal

```
Terminal> ipython
```

in a terminal window. The program is run by the command

```
In [1]: run decay1.py
```

The advantage of running programs in IPython are many: previous commands are easily recalled, `%pdb` turns on debugging so that variables can be examined if the program aborts due to an exception, output of commands are stored in variables, programs and statements can be profiled, any operating system command can be executed, modules can be loaded automatically and other customizations can be performed when starting IPython – to mention some of the most useful featuers.

Although running programs in IPython is strongly recommended, most execution examples in the forthcoming text simply use a minimal text like `Terminal> python programname`.

## 2.2 Verifying the implementation

Since it is easy to make mistakes while implementating numerical algorithms, we should believe in the printed $u$ values before we have a done a more thorough test. Many will think of comparing the computed solution with the exact solution can prove that the implementation is correct, but there is a difference between the computed solution and the exact formula $Ie^{-at}$. We do not know what this difference is. The challenging question is whether we have the mathematically correct difference or if we have another, maybe small, difference that is due to both an approximation error and an error in the implementation.

The purpose of *verifying* a program is to bring evidence for the fact that there are no errors in the implementation. To avoid mixing necessary approximation errors and undesired implementation errors, we should try to make tests where we have some exact computation of the discrete solution or parts of it.

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

With the present test example we could use a calculator and compute $u^1$, $u^2$, and $u^3$. However, the chosen values of $I$ and $\theta$ given in the execution example above are not good: 0 and 1 can easily simplify formulas too much for test purposes. For example, with $\theta = 1$ the nominator if the formula for $u^n$ will be the same for all $a$ and $\Delta t$ values. We therefore choose more "arbitrary" values, say $\theta = 0.8$ and $I = 0.1$. Hand calculations with the aid of a calculator gives

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

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

Instead of doing this by hand, we may write the formulas in Python and let Python do the calculations:

```
def verify_three_steps():
    # Three manual steps
    theta = 0.8; a = 2; I = 0.1; dt = 0.8
    factor = (1 - (1-theta)*a*dt)/(1 + theta*dt*a)
    u1 = factor*I
    u2 = factor*u1
    u3 = factor*u2

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

    print u[1:]  # u[1], u[2], ...
    print u1, u2, u3
```

Visual inspection of the printed output is tedious and errorprone. What we really want to test is whether the difference between the "hand calculations" `u1`, `u2`, and `u3` and the general numerical solution in the `u` array is within the machine precision. The last two print statements in the above `verify_three_steps` function should therefore be replaced by an automated comparison:

```
def verify_three_steps():
    ...

    tol = 1E-15  # tolerance for comparing floats
    difference = abs(u1-u[1]) + abs(u2-u[2]) + abs(u3-u[3])
    success = difference <= tol
    return success
```

We also put the main program, where a complete numerical simulation of a case is carried out, in a separate function, `main`, since this makes it easier to run the verify function and real simulations as desired. The `main` function is then

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

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

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

Since the verification test is always done, future errors introduced accidentally in the program have a good chance of being detected. It is extremely important that verification test can always be easily executed. There are test frameworks and corresponding programming rules that allow us to request running through a suite of test cases, but in this very early stage of Python programming we just implement and run the verification in our own code so that every detail is visible and understood.

The complete program including the `verify_three_steps*` functions is found in the file `decay2.py`. A problem with the `verify_three_steps` function is that one often ends up copying expressions from the implementation that is to be tested. Buggy expressions are then tested against the same buggy expressions. A much safer approach is described next.

**Finding an exact discrete solution.** Sometimes it is possible to find an *exact discrete solution*. That is, we have a formula for $u^n$ that fulfilles the discrete

finite difference equations. The implementation can then be verified against the exact discrete solution. This is usually the best technique for verification.

Define

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

Manual computations with the $\theta$-rule results in

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

We have then established the exact discrete solution as

$$u^n = IA^n thinspace. \tag{34}$$

Note that $n$ has different meaning on the left- and right-hand side of this equation. On the left, $n$ is a superscript reflecting a counter of mesh points, while on the right, $n$ is a power reflecting exponentiation.

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

```
def verify_exact_discrete_solution():

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

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

Note that one can define a function inside another function (but such a function is invisible outside the function in which it is defined). The complete program is found in the file `decay3.py`.

## 2.3 Computing the numerical error

Now that we have evidence for a correct implementation, we are in a position to compare the computed $u^n$ values in the `u` array with the exact $u$ values at the mesh points. The purpose is not to check that the program is correct, but to study the error in the numerical solution.

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

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

A natural way to compare the exact and discrete solutions is to calculate their difference at the mesh points:

$$e_n = u_{\mathrm{e}}(t_n) - u_n, \quad n = 0, 1, \ldots, Nthinspace. \tag{35}$$

These numbers are conveniently computed by

```
u, t = theta_rule(I, a, T, dt, theta)  # Numerical solution
u_e = exact_solution(t, I, a)
e = u_e - u
```

The last two statements make use of array arithmetics: `t` is an array of mesh points that we pass to `exact_solution`. 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 `tmp`, resulting an array, say `tmp2`. Finally, `I*tmp2` is computed (scalar times array) and `u_e` refers to this array returned from `exact_solution`. The expression `u_e - u` is the difference between two arrays, resulting in a new array referred to by `e`.

The array `e` is the current problem's disctete *error function*. Very often we want to work with just one number reflecting the size of the error. A common choice is to integrate $e_n^2$ over the mesh and take the square root. Assuming the exact and discrete solution to vary linearly between the mesh points, the integral is given exactly by the Trapezoidal rule:

$$\hat{E}^2 = \Delta t \left( \frac{1}{2} e_0^2 + \frac{1}{2} e_N^2 + \sum_{n=1}^{N-1} e_n^2 \right)$$

A common approximation of this expression, for convenience, is

$$\hat{E}^2 \approx E^2 = \Delta t \sum_{n=0}^{N} e_n^2$$

The error in this approximation is not much of a concern: it means that the error measure is not exactly the Trapezoidal rule of an integral, but a slightly different measure. We could equally well have chosen other error messages, but the choice is not important as long as we use the same error measure consistently when investigating the error.

The error measure $\hat{E}$ or $E$ is referred to as the $L_2$ norm of the discrete error function. The formula for $E$ will be frequently used:

$$E = \sqrt{\sum_{n=0}^{N} e_n^2} \tag{36}$$

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.

Instead of doing array computing 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] = 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)
```

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

## 2.4 Plotting solutions

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

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

It will be illustrative to also plot $u_{\mathrm{e}}(t)$ for comparison. Doing a `plot(t, u_e)` 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 $u_{\mathrm{e}}(t)$:

```
# Plot the error using a very fine mesh
t_e = linspace(0, T, 1001)
u_e = exact_solution(t_e, I, a)
plot(t,   u,   'r-')            # red  line for u
plot(t_e, u_e, 'b-')            # blue line for u_e
```

With more than one curve in the plot we need to associate each curve with a legend. We also want appropriate names on the axis, 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 plot session then becomes

```
from matplotlib.pyplot import *

figure()                          # create new plot
t_e = linspace(0, T, 1001)        # very fine mesh for
u_e = exact_solution(t_e, I, a)
plot(t,   u,   'ro')              # red circles for u
plot(t_e, u_e, 'b-')              # blue line for u_e
legend(['numerical', 'exact'])
xlabel('t')
ylabel('u')
title('Method: theta-rule, theta=%g, dt=%g' % (theta, dt))
savefig('%s_%g.png' % (theta, dt))
show()
```

Note that `savefig` here creates a PNG file whose name reflects the values of $\theta$ and $\Delta t$ so that we can easily distinguish files from different runs with $\theta$ and $\Delta t$.

A bit more sophisticated and easy-to-read filename can be generated by mapping the $\theta$ value to acronyms for the three common schemes: FE (Forward Euler, $\theta = 0$), BE (Backward Euler, $\theta = 1$), CN (Crank-Nicolson, $\theta = 0.5$). A Python dictionary is ideal for such a mapping from numbers to strings:

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

Let us wrap up the computation of the error measure and all the plotting statements in a function `explore`. This function can be called for various $\theta$ 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 theta_rule, compute error measure,
    and plot the numerical and exact solutions (if makeplot=True).
    """
    u, t = theta_rule(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)        # very fine mesh for u_e
        u_e = exact_solution(t_e, I, a)
        plot(t,   u,   'ro')              # red circles for u
        plot(t_e, u_e, 'b-')              # blue line for u_e
        legend(['numerical', 'exact'])
        xlabel('t')
        ylabel('u')
        title('Method: theta-rule, theta=%g, dt=%g' % (theta, dt))
        theta2name = {0: 'FE', 1: 'BE', 0.5: 'CN'}
        savefig('%s_%g.png' % (theta2name[theta], dt))
        show()
    return E

I = 1
a = 2
T = 5
for theta in 0, 0.5, 1:
```

```
    for dt in 0.4, 0.04:
        E = explore(I, a, T, dt, theta, makeplot=True)
        print '%3.1f %6.2f: %12.3E' % (theta, dt, E)
```

The `figure()` call is key here, without it a new `plot` command will just earse the last plot instead of creating a new plot in a separate window. The complete code resides in the file `decay4.py`. Running this program results in

Terminal

```
Terminal> python decay4.py
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 $\Delta t$ by a factor of 10 increases the accuracy for all three methods ($\theta$ values). 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 $\theta = 0$ with $\Delta t = 0.4$ result in the least accurate solutions.

Figure 6 demonstrates that the numerical solution for $\Delta t = 0.4$ clearly lies below the exact curve, but that the accuracy improves considerably by using 1/10 of this time step.

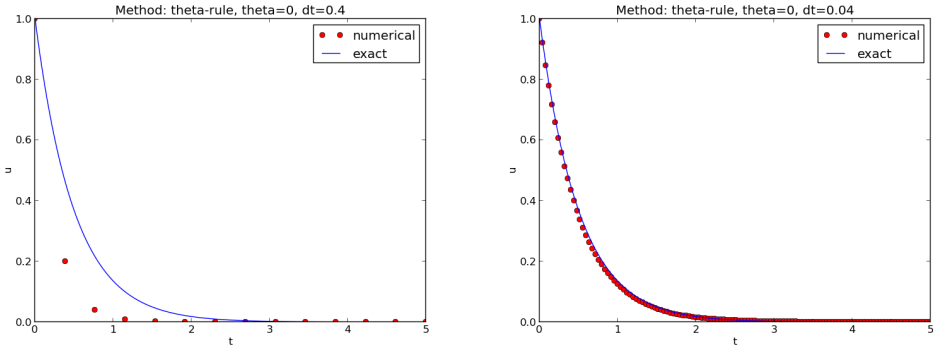


Figure 6: The Forward Euler scheme for two values of $\Delta t$.

Mounting two PNG files, as done in the figure, is easily done by the `montage` program from the ImageMagick suite:

Terminal

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

The `-geometry` argument is used to specify the size of the image, and 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 behavior of the two other schemes are shown in Figures 7 and 8. Crank-Nicolson is obviously the most accurate scheme from a visual point of view.

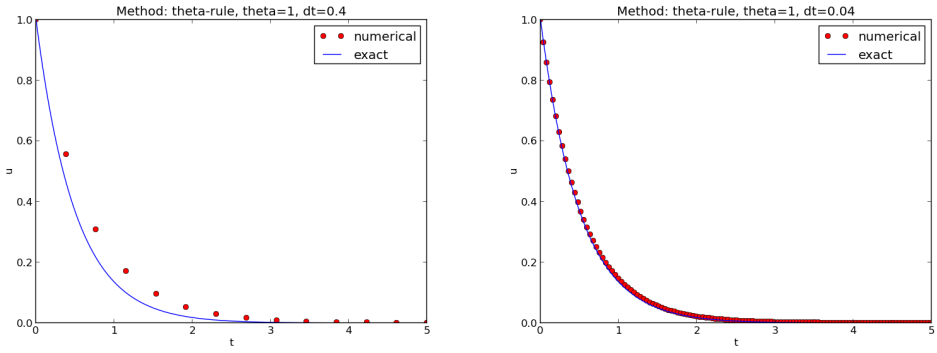


Figure 7: The Backward Euler scheme for two values of $\Delta t$.

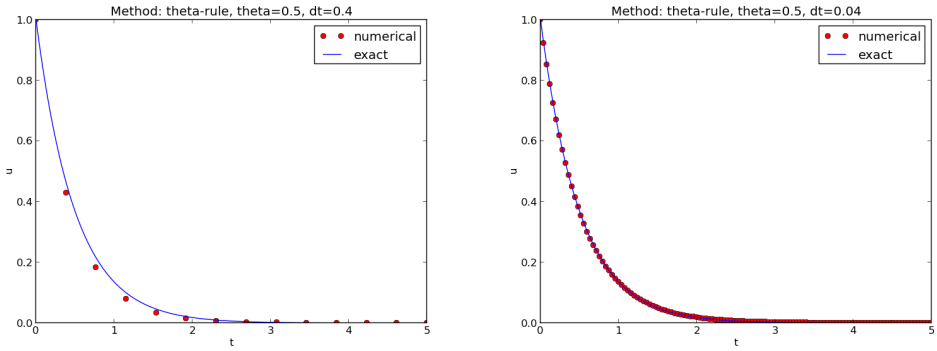


Figure 8: The Crank-Nicolson scheme for two values of $\Delta t$.

## 2.5 Plotting with SciTools

The SciTools package provides a plotting interface, called Easyviz (`scitools.easyviz`), to many different plotting packages, including Matplotlib. 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 import also performs a `from numpy import *` (and more) so that array functionality becomes available as well.

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

```
plot(t,   u,   'ro',             # red circles for u
     t_e, u_e, 'b-',             # blue line for u_e
     legend=['numerical', 'exact'],
     xlabel='t',
     ylabel='u',
     title='Method: theta-rule, theta=%g, dt=%g' % (theta, dt),
     savefig='%s_%g.png' % (theta2name[theta], dt),
     show=True)
```

The `decay5.py` file contains such a demo.

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

Terminal

```
Terminal> python decay5.py --SCITOOLS_easyviz_backend gnuplot
Terminal> python decay5.py --SCITOOLS_easyviz_backend grace
```

All the plot windows are launched with the 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 plot files. This is particularly handy when making plot files in the PostScript format (`savefig('myplot.eps')`).

The default backend for Easyviz (together with numerous other features) can be set in a configuration file

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

## 2.6 Reading input from the command line

It is good programming practice to let programs read input from the user rather than require the user to edit the source code when changing parameters. Reading input from the command line is a simple and flexible way of interacting with the user. Python stores all the command-line arguments in the list `sys.argv`. There are, in principle, two ways of programming with command-line arguments in Python:

- Decide upon a sequence of parameters on the command line and read their values directly from the `sys.argv[1:]` (`sys.argv[0]` is the program name).
- Introduce default values for parameters, use option-value pairs on the command line to override the default values, and use the `argparse.ArgumentParser` tool to interact with the command line.

Both strategies will be illustrated next.

**Reading a sequence of command-line arguments.** The `decay4.py` program needs the following input data:

1. $I$
2. $a$
3. $T$
4. an option to turn the plot on or off (`makeplot`)
5. a list of $\Delta t$ values

The simplest way of reading this input from the command line is to say that the first four command-line arguments correspond to the first four points in the list above, in that order, and that the rest of the command-line arguments are the $\Delta t$ values. The input given for `makeplot` can be a string among `'on'`, `'off'`, `'True'`, and `'False'`. The code for reading this input can be put in a function:

```
import sys

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

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

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

First note that everything on the command line ends up in a string in the list `sys.argv`. Then note that the value of `makeplot` is determined from a boolean expression, which becomes `True` if the command-line argument is either `'on'` or `'True'`, and `False` otherwise. Also note how easy it is to build the list of $\Delta t$ value: we simply run through the rest of the list, `sys.argv[5:]`, convert each command-line argument to `float`, and collect these `float` objects in a list, using the compact and convenient *list comprehension* syntax in Python.

The loops over $\theta$ and $\Delta t$ values can be coded in a `main` function:

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

The complete program can be found in `decay6.py`.

**Working with an argument parser.** Python's `ArgumentParser` tool in in the `argparse` module makes it easy to create a professional command-line interface to any program. The documentation of 'ArgumentParser' demonstrates its versatile applications, so we shall here just list an example containing the most used features. On the command line we want to specify option value pairs for $I$, $a$, and $T$, e.g., `--a 3.5 --I 2 --T 2`. Including `--makeplot` turns the plot on and excluding this option turns the plot off. The $\Delta t$ values can be given as `--dt 1 0.5 0.25 0.1 0.01`. Each parameter must have a sensible default value so that we specify the option on the command line only when the default value is not suitable.

We introduce a function for defining the mentioned command-line options:

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

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

Terminal

```
Terminal> python decay7.py -h
  ...
  --I I, --initial_condition I
```

```
                        initial condition, u(0)
  ...
```

The structure of this explanation is

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

The `--makeplot` option is a pure flag without any value, implying a true value if the flag is present and otherwise a false value. The `action='store_true'` makes an option such a flag. Finally, the `--dt` option demonstrates how to allow for more than one value (separated by blanks) through the `nargs='+'` keyword argument. After the command line is parsed, we get an object where the values of the options are stored as attributes. The attribute name is specified by the `dist` keyword argument, which for the `--dt` option reads `dt_values`. The code below demonstrates how to read the command line and extract the values for each option:

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

The `main` function remains the same as in the `decay6.py` code based on reading from `sys.argv` directly. A complete program using the demo above of `ArgumentParser` appears in the file `decay7.py`.

## 2.7 Computing convergence rates

We normally expect that the error $E$ in the numerical solution is reduced if the mesh size $\Delta t$. More specifically, many numerical methods obey a power-law relation between $E$ and $\Delta t$, if the latter is sufficiently small:

$$E = C\Delta t^r, \tag{37}$$

where $C$ and $r$ are (usually unknown) constants independent of $\Delta t$. The parameter $r$ is known as the *convergence rate*. For example, if the convergence rate is 2, halving $\Delta t$ reduces the error by a factor of $E$. Diminishing $\Delta t$ then has a greater impact on the error compared with methods that have $r = 1$. For a given value of $r$, we refer to the method as of $r$-th order. First- and second-order methods are most common in scientific computing.

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

1. Take the logarithm of 37, $\ln E = r \ln \Delta t + \ln C$, and fit a straight line to the data points $(\Delta t_i, E_i)$, $i = 0, \ldots, m-1$.

2. Consider two consecutive experiments, $(\Delta t_i, E_i)$ and $(\Delta t_{i-1}, E_{i-1})$. Dividing the equation $E_{i-1} = C\Delta t_{i-1}^r$ by $E_i = C\Delta t_i^r$ and solving for $r$ yields

$$r_{i-1} = \frac{\ln(E_{i-1}/E_i)}{\ln(\Delta t_{i-1}/\Delta t_i)} \tag{38}$$

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

The disadvantage of method 1 is that 37 might not be valid for the coarsest meshes (largest $\Delta t$ values), and fitting a line to all the data points is then misleading. However, we usually have no idea of which $\Delta t$ values to exclude. Method 2 computes convergence rates for pairs of experiments and allows us to see if the sequence $r_i$ converges to some value as $i \to m-1$. The final $r_{m-1}$ can then be taken as the convergence rate. If the coarsest meshes have a differing rate, the corresponding time steps are probably too large for 37 to be valid. (We say that the those time steps are not in the asymptotic range.)

It is straightforward to extend the `main` function in the program `decay7.py` with statements for computing $r_0, r_1, \ldots, r_{m-2}$:

```
from math import log

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

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

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

The program is called `decay8.py`.

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

```
[1.331919482274763, 1.1488178494691532, 1.0657737105411782,  ...]
```

Instead, we format each number with 2 decimals, using a list comprehension to turn the list of numbers, `r[method]`, into a list of formatted strings. Then

we join these strings with a space in between to get a sequence of rates on one line in the terminal window. In general, `d.join(list)` joins the strings in the list `list` to one string, with `d` as delimiter between `list[0]`, `list[1]`, etc.

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

Terminal

```
Terminal> python decay8.py --dt 0.5 0.25 0.1 0.05 0.025 0.01
...
Pairwise convergence rates for theta=0:
1.33 1.15 1.07 1.03 1.02

Pairwise convergence rates for theta=0.5:
2.14 2.07 2.03 2.01 2.01

Pairwise convergence rates for theta=1:
0.98 0.99 0.99 1.00 1.00
```

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

Very often, we have some theory that predicts what $r$ is for a numerica method. It can be shown that in case of the $\theta$-rule, $r = 2$ for $\theta = 0.5$ and $r = 1$ otherwise. The computed estimates of $r$ are in very good agreement with these theoretical values. The strong practical application of computing convergence rates is for verification: wrong convergence rates point to errors in the code, and correct convergence rates brings evidence that the implementation is correct. Experience shows bugs in the code easily destroys the expected convergence rate.

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

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

Running the same `decay8.py` command as above gives the expected convergence rates (!). Why? The reason is that we just specified the $\Delta t$ values are relied on default values for other parameters. The default value of $a$ is 1. Forgetting the factor `a` has then no effect. This example shows how importance it is to avoid parameters that are 1 or 0 when verifying implementations. Running the code with $a = 2.1$ and $I = 0.1$ yields

Terminal

```
Terminal> python decay8.py --a 2.1 --I 0.1  \
          --dt 0.5 0.25 0.1 0.05 0.025 0.01
...
Pairwise convergence rates for theta=0:
1.49 1.18 1.07 1.04 1.02

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

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

---

This time we see that the expected convergence rates for the Crank-Nicolson and Backward Euler methods are not obtained, while $r = 1$ for the Forward Euler method. The reason for correct rate in the latter case is that $\theta = 0$ and the wrong `theta*dt` term in the denominator vanishes anyway.

The error

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

manifests itself through wrong rates $r \approx 0$ for all three methods. About the same results arise from an erroneous initial condition, `u[0] = 1`, or wrong loop limits, `range(1,N)`. It seems that in this simple problem, most bugs we can think of are detected by the convergence rate test.

A `verify_convergence_rate` function could compute the dictionary of list via `main` and check if the final rate estimates ($r_{m-2}$) are sufficiently close to the expected ones. A tolerance of 0.1 seems appropriate:

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

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

## 2.8 Memory-saving implementation

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

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 $N$ points. A typical desired value for $M$ is 100 in many applications, and be may

1000. Storing all the computed $u$ values, like we have done in the programs so far, demands storage of some arrays of size $M^3N$, giving a factor of $M^3$ larger storage demands compared to our ODE programs. Each real number in the array for $u$ requires 8 bytes of storage, resulting in a demand for 8 Gb of memory for only one array. Then there are needs for good ideas on how to lower the storage requirements. Fortunately, we can almost always get rid of the $M^3$ factor. Below we explain how this is done, and 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 in our implementation of the $\theta$ method. To compute a new $u^{n+1}$, all we need is $u^n$. This imples that the previous $u^{n-1}, u^{n-2}, \ldots, u^0$ values do not need to be stored in an array, although this is convenient for plotting and data analysis in the program. Instead of the `u` array we can work with two variables from real 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 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 `decay9.py`, which is a merge of the `decay4.py` and `decay7.py` programs, using module prefixes `np` for `numpy` and `plt` for `matplotlib.pyplot`.

The following function implements the algorithm without using arrays and stores the solution in a file:

```
def theta_rule_minmem(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 store on file
    (with name filename) for later plotting.
    """
    dt = float(dt)        # avoid integer division
    N = 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, N+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 on how to perform file writing in Python.

Reading the data in the file into arrays `t` and `u` are done by the function

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

Such a file with numbers in rows and columns is very common, and `numpy` has a function `loadtxt` which loads the data into a two-dimensional array, say `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 `decay4.py` must run `theta_rule_minmem` 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 = theta_rule_minmem(I, a, T, dt, theta, filename)

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

The `decay9.py` also includes command-line options `--I`, `--a`, `--T`, `--dt`, `--theta`, and `--makeplot` for controlling input parameters and *making a single run*. For example,

Terminal

```
Terminal> python decay9.py --T 10 --theta 1 --dt 2
I=1.0, a=1.0, T=10.0, makeplot=True, theta=1.0, dt=2.0
theta=1.0 dt=2 Error=3.136E-01
```