Analysis of exponential decay models

Hans Petter Langtangen[1,2]

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

Oct 10, 2015

Contents

1 Experimental investigations **3**
1.1 Discouraging numerical solutions 3
1.2 Detailed experiments 4

2 Stability **8**
2.1 Exact numerical solution 8
2.2 Stability properties derived from the amplification factor 9

3 Accuracy **10**
3.1 Visual comparison of amplification factors 10
3.2 Series expansion of amplification factors 12
3.3 The ratio of numerical and exact amplification factors 14
3.4 The global error at a point 14
3.5 Integrated error 15
3.6 Truncation error 16
3.7 Consistency, stability, and convergence 18

4 Exercises **19**

5 Various types of errors in a differential equation model **20**
5.1 Model errors 20
5.2 Data errors 23
5.3 Discretization errors 24
5.4 Rounding errors 26
5.5 Discussion of the size of various errors 27

References **27**

Index **28**

List of Problems

Problem 1 Visualize the accuracy of finite differences p. 19
Problem 2 Explore the θ-rule for exponential ... p. 20

We address the ODE for exponential decay,

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

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

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

for the numerical solution u^{n+1}, which approximates the exact solution u_{e} at time point t_{n+1}. For constant mesh spacing, which we assume here, $t_{n+1} = (n+1)\Delta t$.

The example programs associated with this chapter are found in the directory `src/analysis`.

1 Experimental investigations

We first perform a series of numerical explorations to see how the methods behave as we change the parameters I, a, and Δt in the problem.

1.1 Discouraging numerical solutions

Choosing $I = 1$, $a = 2$, and running experiments with $\theta = 1, 0.5, 0$ for $\Delta t = 1.25, 0.75, 0.5, 0.1$, gives the results in Figures 1, 2, and 3.

The characteristics of the displayed curves can be summarized as follows:

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

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

Key questions.

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

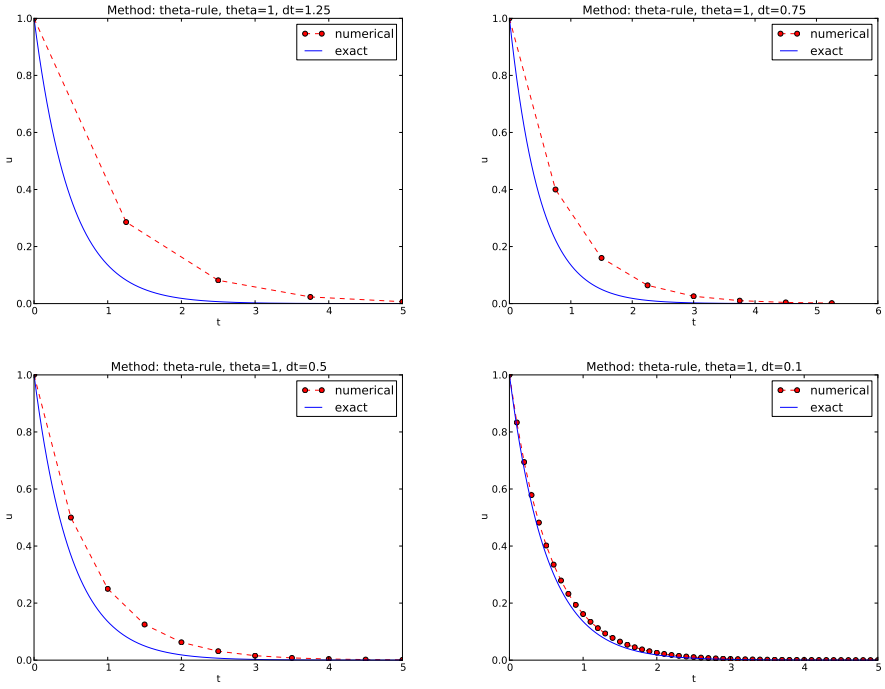

Figure 1: Backward Euler.

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

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

1.2 Detailed experiments

To address the first question above, we may set up an experiment where we loop over values of I, a, and Δt in our chosen model problem. For each experiment, we flag the solution as oscillatory if

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

for some value of n. This seems like a reasonable choice, since we expect u^n to decay with n, but oscillations will make u increase over a time step. Doing

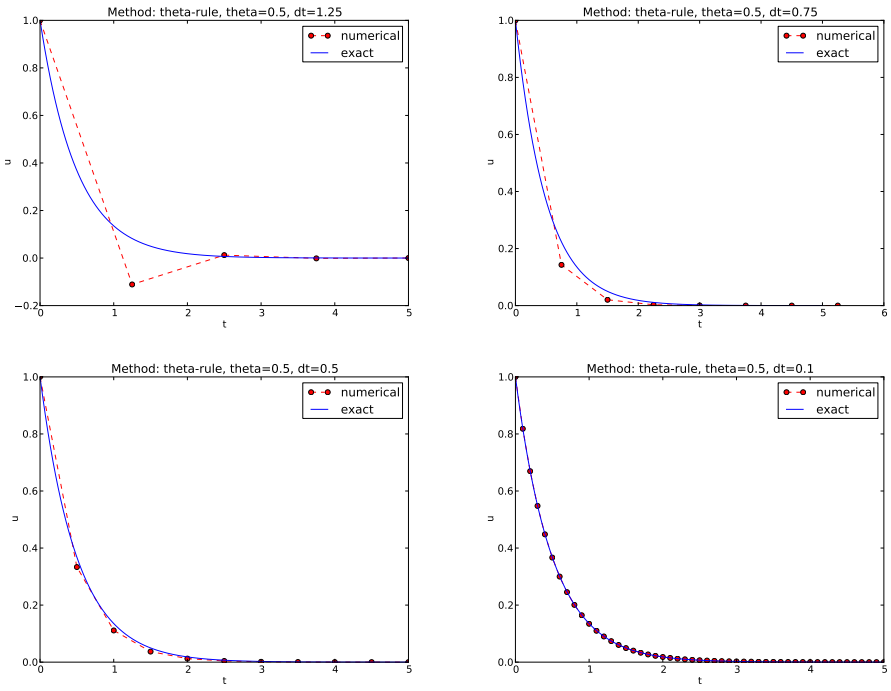

Figure 2: Crank-Nicolson.

some initial experimentation with varying I, a, and Δt, quickly reveals that oscillations are independent of I, but they do depend on a and Δt. We can therefore limit the investigation to vary a and Δt. Based on this observation, we introduce a two-dimensional function $B(a, \Delta t)$ which is 1 if oscillations occur and 0 otherwise. We can visualize B as a contour plot (lines for which $B = \text{const}$). The contour $B = 0.5$ corresponds to the borderline between oscillatory regions with $B = 1$ and monotone regions with $B = 0$ in the $a, \Delta t$ plane.

The B function is defined at discrete a and Δt values. Say we have given P values for a, $a_0, \ldots, a_{P-1}$, and Q values for Δt, $\Delta t_0, \ldots, \Delta t_{Q-1}$. These a_i and Δt_j values, $i = 0, \ldots, P-1$, $j = 0, \ldots, Q-1$, form a rectangular mesh of $P \times Q$ points in the plane spanned by a and Δt. At each point $(a_i, \Delta t_j)$, we associate the corresponding value $B(a_i, \Delta t_j)$, denoted B_{ij}. The B_{ij} values are naturally stored in a two-dimensional array. We can thereafter create a plot of the contour line $B_{ij} = 0.5$ dividing the oscillatory and monotone regions. The file `decay_osc_regions.py` given below (`osc_regions` stands for "oscillatory regions") contains all nuts and bolts to produce the $B = 0.5$ line in Figures 4 and 5. The oscillatory region is above this line.

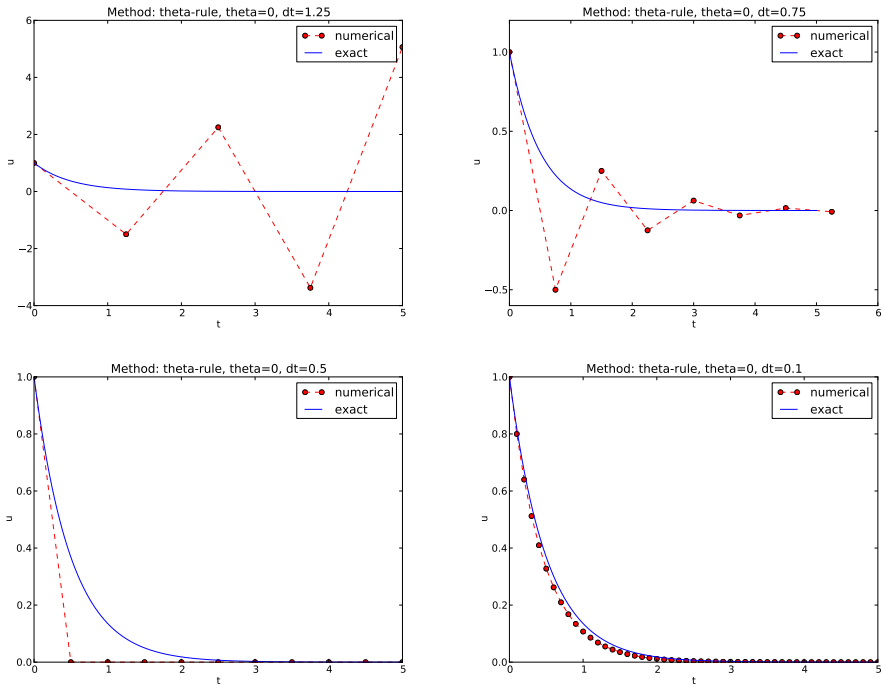

Figure 3: Forward Euler.

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

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

```
    st.xlabel('a'); st.ylabel('dt')
    st.savefig('osc_region_theta_%s.png' % theta)
    st.savefig('osc_region_theta_%s.pdf' % theta)

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

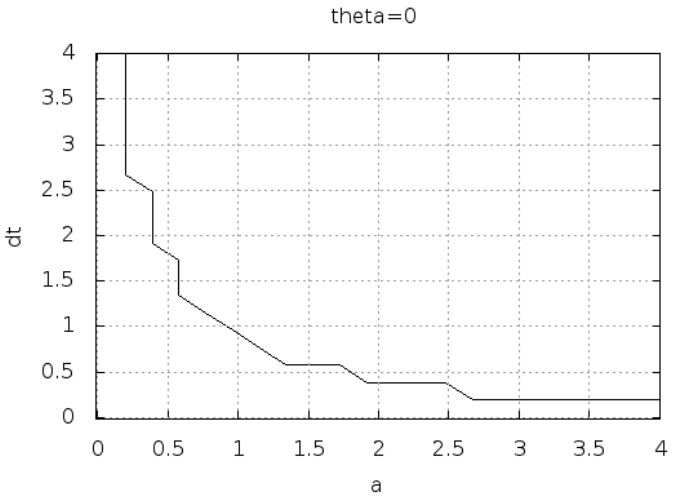

Figure 4: Forward Euler scheme: oscillatory solutions occur for points above the curve.

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

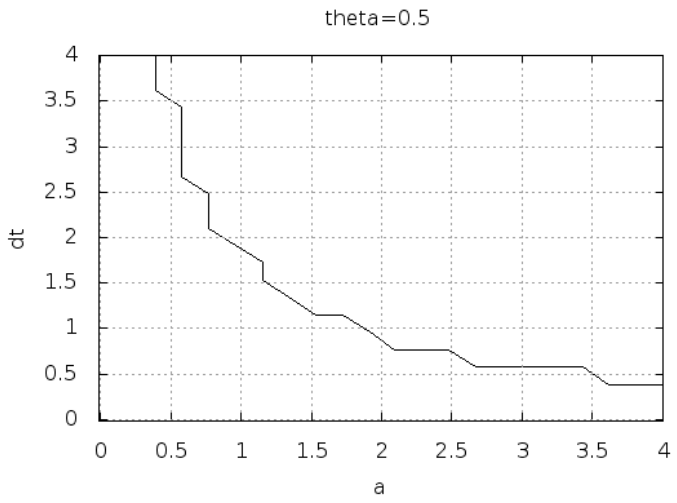

Figure 5: Crank-Nicolson scheme: oscillatory solutions occur for points above the curve.

2 Stability

The goal now is to understand the results in the previous section. To this end, we shall investigate the properties of the mathematical formula for the solution of the equations arising from the finite difference methods.

2.1 Exact numerical solution

Starting with $u^0 = I$, the simple recursion (2) can be applied repeatedly n times, with the result that

$$u^n = IA^n, \quad A = \frac{1-(1-\theta)a\Delta t}{1+\theta a\Delta t}\,. \tag{3}$$

Solving difference equations.

Difference equations where all terms are linear in u^{n+1}, u^n, and maybe u^{n-1}, u^{n-2}, etc., are called *homogeneous, linear* difference equations, and their solutions are generally of the form $u^n = A^n$, where A is a constant to be determined. Inserting this expression in the difference equation and

dividing by A^{n+1} gives a polynomial equation in A. In the present case we get

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

This is a solution technique of wider applicability than repeated use of the recursion (2).

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

Since u^n is a factor A raised to an integer power n, we realize that $A < 0$ will imply $u^n < 0$ for odd n and $u^n > 0$ for even n. That is, the solution oscillates between the mesh points. We have oscillations due to $A < 0$ when

$$(1-\theta)a\Delta t > 1\,. \tag{4}$$

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

$$\Delta t < \frac{1}{(1-\theta)a}\,. \tag{5}$$

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

Looking at the upper left plot in Figure 3, we see that $\Delta t = 1.25$, and remembering that $a = 2$ in these experiments, A can be calculated to be -1.5, so the Forward Euler solution becomes $u^n = (-1.5)^n$ ($I = 1$). This solution oscillates *and* grows. The upper right plot has $a\Delta t = 2 \cdot 0.75 = 1.5$, so $A = -0.5$, and $u^n = (-0.5)^n$ decays but oscillates. The lower left plot is a peculiar case where the Forward Euler scheme produces a solution that is stuck on the t axis. Now we can understand why this is so, because $a\Delta t = 2 \cdot 0.5 = 1$, which gives $A = 0$, and therefore $u^n = 0$ for $n \geq 1$. The decaying oscillations in the Crank-Nicolson scheme in the upper left plot in Figure 2 for $\Delta t = 1.25$ are easily explained by the fact that $A \approx -0.11 < 0$.

2.2 Stability properties derived from the amplification factor

The factor A is called the *amplification factor* since the solution at a new time level is the solution at the previous time level amplified by a factor A. For a decay process, we must obviously have $|A| \leq 1$, which is fulfilled for all Δt if

$\theta \geq 1/2$. Arbitrarily large values of u can be generated when $|A| > 1$ and n is large enough. The numerical solution is in such cases totally irrelevant to an ODE modeling decay processes! To avoid this situation, we must demand $|A| \leq 1$ also for $\theta < 1/2$, which implies

$$\Delta t \leq \frac{2}{(1-2\theta)a}, \tag{6}$$

For example, Δt must not exceed $2/a$ when computing with the Forward Euler scheme.

Stability properties.

We may summarize the stability investigations as follows:

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

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

3 Accuracy

While stability concerns the qualitative properties of the numerical solution, it remains to investigate the quantitative properties to see exactly how large the numerical errors are.

3.1 Visual comparison of amplification factors

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

$$u_{\mathrm{e}}(t_n) = Ie^{-an\Delta t} = I(e^{-a\Delta t})^n\,. \tag{7}$$

From this formula we see that the exact amplification factor is

$$A_{\mathrm{e}} = e^{-a\Delta t}\,. \tag{8}$$

We see from all of our analysis that the exact and numerical amplification factors depend on a and Δt through the dimensionless product $a\Delta t$: whenever there is a Δt in the analysis, there is always an associated a parameter. Therefore, it is convenient to introduce a symbol for this product, $p = a\Delta t$, and view A and A_e as functions of p. Figure 6 shows these functions. The two amplification factors are clearly closest for the Crank-Nicolson method, but that method has the unfortunate oscillatory behavior when $p > 2$.

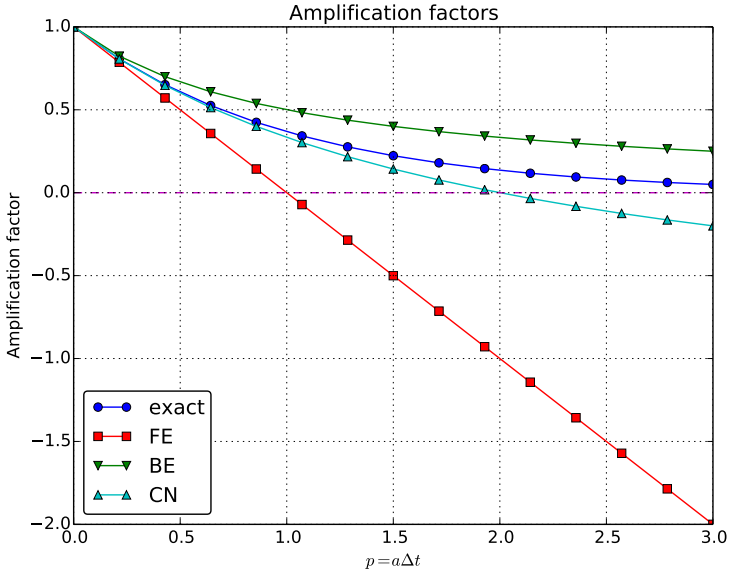

Figure 6: Comparison of amplification factors.

Significance of the $p = a\Delta t$ parameter.

The key parameter for numerical performance of a scheme is in this model problem $p = a\Delta t$. This is a *dimensionless number* (a has dimension 1/s and Δt has dimension s) reflecting how the discretization parameter plays together with a physical parameter in the problem.

One can bring the present model problem on dimensionless form through a process called scaling. The scaled modeled has a modified time $\bar{t} = at$ and modified response $\bar{u} = u/I$ such that the model reads $d\bar{u}/d\bar{t} = -\bar{u}$, $\bar{u}(0) = 1$. Analyzing this model, where there are no physical parameters, we find that $\Delta\bar{t}$ is the key parameter for numerical performance. In the unscaled model, this corresponds to $\Delta\bar{t} = a\Delta t$.

It is common that the numerical performance of methods for solving ordinary and partial differential equations is governed by dimensionless parameters that combine mesh sizes with physical parameters.

3.2 Series expansion of amplification factors

As an alternative to the visual understanding inherent in Figure 6, there is a strong tradition in numerical analysis to establish formulas for approximation errors when the discretization parameter, here Δt, becomes small. In the present case, we let p be our small discretization parameter, and it makes sense to simplify the expressions for A and A_{e} by using Taylor polynomials around $p = 0$. The Taylor polynomials are accurate for small p and greatly simplify the comparison of the analytical expressions since we then can compare polynomials, term by term.

Calculating the Taylor series for A_{e} is easily done by hand, but the three versions of A for $\theta = 0, 1, \frac{1}{2}$ lead to more cumbersome calculations. Nowadays, analytical computations can benefit greatly by symbolic computer algebra software. The Python package `sympy` represents a powerful computer algebra system, not yet as sophisticated as the famous Maple and Mathematica systems, but it is free and very easy to integrate with our numerical computations in Python.

When using `sympy`, it is convenient to enter an interactive Python shell where the results of expressions and statements can be shown immediately. Here is a simple example. We strongly recommend to use `isympy` (or `ipython`) for such interactive sessions.

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

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

Lines with `>>>` represent input lines, whereas without this prompt represent the result of the previous command (note that `isympy` and `ipython` apply other prompts, but in this text we always apply `>>>` for interactive Python computing). Apart from the order of the powers, the computed formula is easily recognized as the beginning of the Taylor series for e^{-p}.

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

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

To work with the factor for the Backward Euler scheme we can substitute the value 1 for `theta`:

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

Similarly, we can substitute `theta` by 1/2 for Crank-Nicolson, preferably using an exact rational representation of 1/2 in `sympy`:

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

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

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

We are now in a position to compare Taylor series:

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

From these expressions we see that the error $A - A_{\mathrm{e}} \sim \mathcal{O}(p^2)$ for the Forward and Backward Euler schemes, while $A - A_{\mathrm{e}} \sim \mathcal{O}(p^3)$ for the Crank-Nicolson scheme. The notation $\mathcal{O}(p^m)$ here means a polynomial in p where p^m is the term of lowest-degree, and consequently the term that dominates the expression for $p < 0$. We call this the *leading order term*. As $p \to 0$, the leading order term clearly dominates over the higher-order terms (think of $p = 0.01$: p is a hundred times larger than p^2).

Now, a is a given parameter in the problem, while Δt is what we can vary. Not surprisingly, the error expressions are usually written in terms Δt. We then have

$$A - A_{\mathrm{e}} = \begin{cases} \mathcal{O}(\Delta t^2), & \text{Forward and Backward Euler,} \\ \mathcal{O}(\Delta t^3), & \text{Crank-Nicolson} \end{cases} \tag{9}$$

We say that the Crank-Nicolson scheme has an error in the amplification factor of order Δt^3, while the two other schemes are of order Δt^2 in the same quantity.

What is the significance of the order expression? If we halve Δt, the error in amplification factor at a time level will be reduced by a factor of 4 in the Forward and Backward Euler schemes, and by a factor of 8 in the Crank-Nicolson scheme. That is, as we reduce Δt to obtain more accurate results, the Crank-Nicolson scheme reduces the error more efficiently than the other schemes.

3.3 The ratio of numerical and exact amplification factors

An alternative comparison of the schemes is provided by looking at the ratio A/A_{e}, or the error $1 - A/A_{\mathrm{e}}$ in this ratio:

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

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

3.4 The global error at a point

The error in the amplification factor reflects the error when progressing from time level t_n to t_{n-1} only. That is, we disregard the error already present in the solution at t_{n-1}. The real error at a point, however, depends on the error development over all previous time steps. This error, $e^n = u^n - u_{\mathrm{e}}(t_n)$, is known as the *global error*. We may look at u^n for some n and Taylor expand the mathematical expressions as functions of $p = a\Delta t$ to get a simple expression for the global error (for small p):

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

Note that `sympy` does not sort the polynomial terms in the output, so p^3 appears before p^2 in the output of `BE`.

For a fixed time t, the parameter n in these expressions increases as $p \to 0$ since $t = n\Delta t = \text{const}$ and hence n must increase like Δt^{-1}. With n substituted by $t/\Delta t$ in the leading-order error terms, these become

$$e^n = \frac{1}{2}np^2 = \frac{1}{2}ta^2\Delta t, \qquad \text{Forward Euler} \tag{10}$$

$$e^n = -\frac{1}{2}np^2 = -\frac{1}{2}ta^2\Delta t, \qquad \text{Backward Euler} \tag{11}$$

$$e^n = \frac{1}{12}np^3 = \frac{1}{12}ta^3\Delta t^2, \qquad \text{Crank-Nicolson} \tag{12}$$

The global error is therefore of second order (in Δt) for the Crank-Nicolson scheme and of first order for the other two schemes.

Convergence.

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

3.5 Integrated error

The L^2 norm of the error can be computed by treating e^n as a function of t in `sympy` and performing symbolic integration. For the Forward Euler scheme we have

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

The output reads

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

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

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

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

We have obtained an exact analytical expression for the error at $t = t_n$, but here we use the leading-order error term only since we are mostly interested in how the error behaves as a polynomial in Δt or p, and then the leading order term will dominate. For the Forward Euler scheme, $u_{\mathrm{e}}(t_n) - u^n \approx \frac{1}{2}np^2$, and we have

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

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

$$||e^n||_{\ell^2} = \frac{1}{2}\sqrt{\frac{T^3}{3}} a^2 \Delta t \,. \tag{13}$$

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

$$||e^n||_{\ell^2} = \frac{1}{12}\sqrt{\frac{T^3}{3}} a^3 \Delta t^2 \,. \tag{14}$$

Summary of errors.

Both the global point-wise errors (10)-(12) and their time-integrated versions (13) and (14) show that

- the Crank-Nicolson scheme is of second order in Δt, and
- the Forward Euler and Backward Euler schemes are of first order in Δt.

3.6 Truncation error

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

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

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

which is the short form for

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

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

$$R^n = \frac{u_{\rm e}(t_{n+1}) - u_{\rm e}(t_n)}{\Delta t} + au_{\rm e}(t_n)\,. \tag{15}$$

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

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

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

which may be used to reformulate the fraction in (15) so that

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

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

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

This R^n is the *truncation error*, which for the Forward Euler is seen to be of first order in Δt as $\Delta \to 0$.

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

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

while the Crank-Nicolson scheme gives

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

when $\Delta t \to 0$.

The *order* r of a finite difference scheme is often defined through the leading term Δt^r in the truncation error. The above expressions point out that the Forward and Backward Euler schemes are of first order, while Crank-Nicolson

is of second order. We have looked at other error measures in other sections, like the error in amplification factor and the error $e^n = u_{\mathrm{e}}(t_n) - u^n$, and expressed these error measures in terms of Δt to see the order of the method. Normally, calculating the truncation error is more straightforward than deriving the expressions for other error measures and therefore the easiest way to establish the order of a scheme.

3.7 Consistency, stability, and convergence

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

Consistency means that the error in the difference equation, measured through the truncation error, goes to zero as $\Delta t \to 0$. Since the truncation error tells how well the exact solution fulfills the difference equation, and the exact solution fulfills the differential equation, consistency ensures that the difference equation approaches the differential equation in the limit. The expressions for the truncation errors in the previous section are all proportional to Δt or Δt^2, hence they vanish as $\Delta t \to 0$, and all the schemes are consistent. Lack of consistency implies that we actually solve some other differential equation in the limit $\Delta t \to 0$ than we aim at.

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

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

Convergence is hard to establish theoretically, except in quite simple problems like the present one. Stability and consistency are much easier to calculate. A major breakthrough in the understanding of numerical methods for differential equations came in 1956 when Lax and Richtmeyer established equivalence between convergence on one hand and consistency and stability on the other (the Lax equivalence theorem). In practice it meant that one can first establish that a method is stable and consistent, and then it is automatically convergent (which is much harder to establish). The result holds for linear problems only, and in

the world of nonlinear differential equations the relations between consistency, stability, and convergence are much more complicated.

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

4 Exercises

Problem 1: Visualize the accuracy of finite differences

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

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

and view E as a function of Δt. We expect that $\lim_{\Delta t\to 0} E = 1$, while E may deviate significantly from unity for large Δt. How the error depends on Δt is best visualized in a graph where we use a logarithmic scale for Δt, so we can cover many orders of magnitude of that quantity. Here is a code segment creating an array of 100 intervals, on the logarithmic scale, ranging from 10^{-6} to $10^{-0.5}$ and then plotting E versus $p = a\Delta t$ with logarithmic scale on the p axis:

```
from numpy import logspace, exp
from matplotlib.pyplot import semilogx
p = logspace(-6, -0.5, 101)
y = (1-exp(-p))/p
semilogx(p, y)
```

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

Perform a Taylor series expansions of the error fractions and find the leading order r in the expressions of type $1 + Cp^r + \mathcal{O}(p^{r+1})$, where C is some constant.

Hint. To save manual calculations and learn more about symbolic computing, make functions for the three difference operators and use `sympy` to perform the symbolic differences, differentiation, and Taylor series expansion. To plot a symbolic expression `E` against `p`, convert the expression to a Python function first: `E = sympy.lamdify([p], E)`.
Filename: `decay_plot_fd_error`.

Problem 2: Explore the θ-rule for exponential growth

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

a) Set $a = -1$ and run experiments with $\theta = 0, 0.5, 1$ for various values of Δt to uncover numerical artifacts. Recall that the exact solution is a monotone, growing function when $a < 0$. Oscillations or significantly wrong growth are signs of wrong qualitative behavior.

From the experiments, select four values of Δt that demonstrate the kind of numerical solutions that are characteristic for this model.

b) Write up the amplification factor and plot it for $\theta = 0, 0.5, 1$ together with the exact one for $a\Delta t < 0$. Use the plot to explain the observations made in the experiments.

Hint. Modify the `decay_ampf_plot.py` code (in the `src/analysis` directory). Filename: `exponential_growth`.

5 Various types of errors in a differential equation model

So far we have been concerned with one type of error, namely the discretization error committed by replacing the differential equation problem by a recursive set of difference equations. There are, however, other types of errors that must be considered too. We can classify errors into four groups:

1. model errors
2. data errors
3. discretization errors
4. round-off errors

Below, we shall briefly describe and illustrate these four types of errors.

5.1 Model errors

Any mathematical model like $u' = -au$, $u(0) = I$, is just an approximate description of a real-world phenomenon. Suppose a more accurate model has a as a function of time rather than a constant. Here we take $a(t)$ as a simple linear function: $a + pt$. Obviously, u with $p > 0$ will go faster to zero with time than a constant a.

The solution of

$$u' = (a + pt)u, \quad u(0) = I,$$

can be shown (see below) to be

$$u(t) = Ie^{t\left(-a-\frac{pt}{2}\right)}.$$

With the above u available in a Python function `u_true(t, I, a, p)` and the solution from our model $u' = -au$ available in `u(t, I, a)`, we can make some plots of the two models and the error for some values of p. Figure 7 displays the two curves for $p = 0.01, 0.1, 1$, while Figure 8 shows the difference between the two models as a function of t for the same p values.

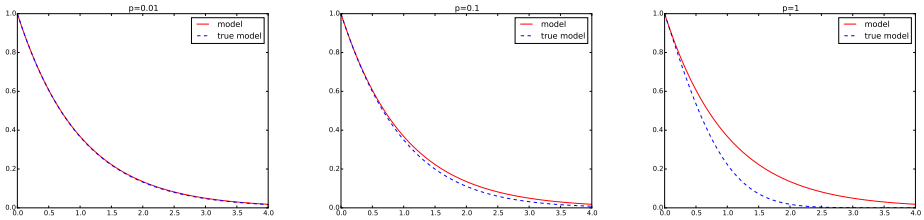

Figure 7: Comparison of two models for three values of p.

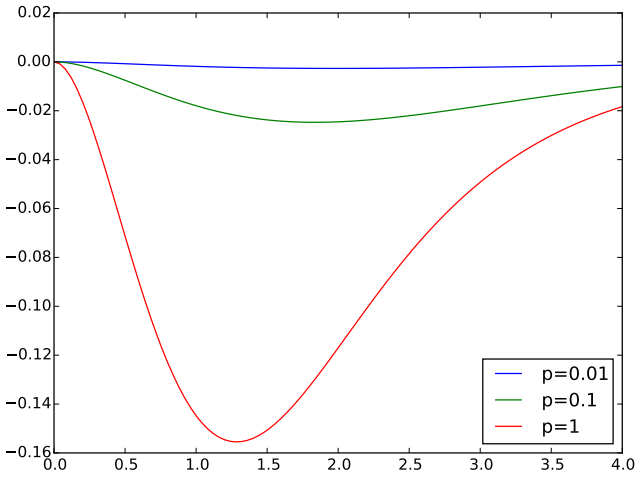

Figure 8: Discrepancy of Comparison of two models for three values of p.

The code that was used to produce the plots looks like

```
from numpy import linspace, exp
from matplotlib.pyplot import \
     plot, show, xlabel, ylabel, legend, savefig, figure, title

def model_errors():
```

```
    p_values = [0.01, 0.1, 1]
    a = 1
    I = 1
    t = linspace(0, 4, 101)
    legends = []
    # Work with figure(1) for the discrepancy and figure(2+i)
    # for plotting the model and the true model for p value no i
    for i, p in enumerate(p_values):
        u = model(t, I, a)
        u_true = true_model(t, I, a, p)
        discrepancy = u_true - u
        figure(1)
        plot(t, discrepancy)
        figure(2+i)
        plot(t, u, 'r-', t, u_true, 'b--')
        legends.append('p=%g' % p)
    figure(1)
    legend(legends, loc='lower right')
    savefig('tmp1.png'); savefig('tmp1.pdf')
    for i, p in enumerate(p_values):
        figure(2+i)
        legend(['model', 'true model'])
        title('p=%g' % p)
        savefig('tmp%d.png' % (2+i)); savefig('tmp%d.pdf' % (2+i))
```

To derive the analytical solution of the model $u' = -(a + pt)u$, $u(0) = I$, we can use SymPy and the code below. This is somewhat advanced SymPy use for a newbie, but serves to illustrate the possibilities to solve differential equations by symbolic software.

```
def derive_true_solution():
    import sympy as sym
    u = sym.symbols('u', cls=sym.Function)  # function u(t)
    t, a, p, I = sym.symbols('t a p I', real=True)

    def ode(u, t, a, p):
        """Define ODE: u' = (a + p*t)*u. Return residual."""
        return sym.diff(u, t) + (a + p*t)*u

    eq = ode(u(t), t, a, p)
    s = sym.dsolve(eq)
    # s is sym.Eq object u(t) == expression, we want u = expression,
    # so grab the right-hand side of the equality (Eq obj.)
    u = s.rhs
    print u
    # u contains C1, replace it with a symbol we can fit to
    # the initial condition
    C1 = sym.symbols('C1', real=True)
    u = u.subs('C1', C1)
    print u
    # Initial condition equation
    eq = u.subs(t, 0) - I
    s = sym.solve(eq, C1)  # solve eq wrt C1
    print s
    # s is a list s[0] = ...
    # Replace C1 in u by the solution
    u = u.subs(C1, s[0])
    print 'u:', u
    print sym.latex(u)  # latex formula for reports

    # Consistency check: u must fulfill ODE and initial condition
```

```
    print 'ODE is fulfilled:', sym.simplify(ode(u, t, a, p))
    print 'u(0)-I:', sym.simplify(u.subs(t, 0) - I)

    # Convert u expression to Python numerical function
    # (modules='numpy' allows numpy arrays as arguments,
    # we want this for t)
    u_func = sym.lambdify([t, I, a, p], u, modules='numpy')
    return u_func

true_model = derive_true_solution()
```

5.2 Data errors

By "data" we mean all the input parameters to a model, in our case I and a. The values of these may contain errors, or at least uncertainty. Suppose I and a are measured from some physical experiments. Ideally, we have many samples of I and a and from these we can fit probability distributions. Assume that I turns out to be normally distributed with mean 1 and standard deviation 0.2, while a is uniformly distributed in the interval $[0.5, 1.5]$.

How will the uncertainty in I and a propagate through the model $u = Ie^{-at}$? That is, what is the uncertainty in u at a particular time t? This answer can easily be answered using *Monte Carlo simulation*. It means that we draw a lot of samples from the distributions for I and a. For each combination of I and a sample we compute the corresponding u value for selected values of t. Afterwards, we can for each selected t values make a histogram of all the computed u values to see what the distribution of u values look like. Figure 9 shows the histograms corresponding to $t = 0, 1, 3$. We see that the distribution of u values is much like a symmetric normal distribution at $t = 0$, centered around $u = 1$. At later times, the distribution gets more asymmetric and narrower. It means that the uncertainty decreases with time. From the computed u values we can easily calculate the mean and standard deviation. The table below shows the mean and standard deviation values along with the value if we just use the formula $u = Ie^{-at}$ with the mean values of I and a: $I = 1$ and $a = 1$. As we see, there is some discrepancy between this latter (naive) computation and the mean value produced by Monte Carlo simulation.

time	mean	st.dev.	$u(t; I = 1, a = 1)$
0	1.00	0.200	1.00
1	0.38	0.135	0.37
3	0.07	0.060	0.14

Actually, $u(t; I, a)$ becomes a stochastic variable for each t when I and a are stochastic variables, as they are in the above Monte Carlo simulation. The mean of the stochastic $u(t; I, a)$ is not equal to $u(t; I = 1, a = 1)$ unless u is linear in I and a (here u is nonlinear in a). The accuracy of the Monte Carlo results increases with increasing number of samples.

The computer code required to do the Monte Carlo simulation and produce the plots in Figure 9 is shown below.

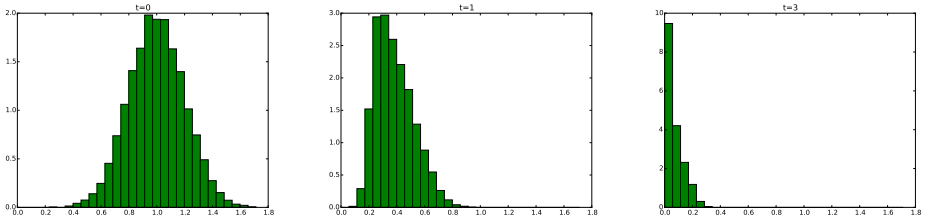

Figure 9: Histogram of solution uncertainty at three time points, due to data errors.

```
def data_errors():
    from numpy import random, mean, std
    from matplotlib.pyplot import hist
    N = 10000
    # Draw random numbers for I and a
    I_values = random.normal(1, 0.2, N)
    a_values = random.uniform(0.5, 1.5, N)
    # Compute corresponding u values for some t values
    t = [0, 1, 3]
    u_values = {}  # samples for various t values
    u_mean = {}
    u_std = {}
    for t_ in t:
        # Compute u samples corresponding to I and a samples
        u_values[t_] = [model(t_, I, a)
                        for I, a in zip(I_values, a_values)]
        u_mean[t_] = mean(u_values[t_])
        u_std[t_] = std(u_values[t_])

        figure()
        dummy1, bins, dummy2 = hist(
            u_values[t_], bins=30, range=(0, I_values.max()),
            normed=True, facecolor='green')
        #plot(bins)
        title('t=%g' % t_)
        savefig('tmp_%g.png' % t_); savefig('tmp_%g.pdf' % t_)
    # Table of mean and standard deviation values
    print 'time   mean   st.dev.'
    for t_ in t:
        print '%3g    %.2f    %.3f' % (t_, u_mean[t_], u_std[t_])
```

5.3 Discretization errors

The errors implied by solving the differential equation problem by the θ-rule has been thoroughly analyzed in the previous sections. Below are some plots of the error versus time for the Forward Euler (FE), Backward Euler (BN), and Crank-Nicolson (CN) schemes for decreasing values of Δt. Since the difference in magnitude between the errors in the CN scheme versus the FE ad BN schemes grows significantly as Δt is reduced (the error goes like Δt^2 for CN versus Δt for FE/BE), we have plotted the logarithm of the absolute value of the numerical error.

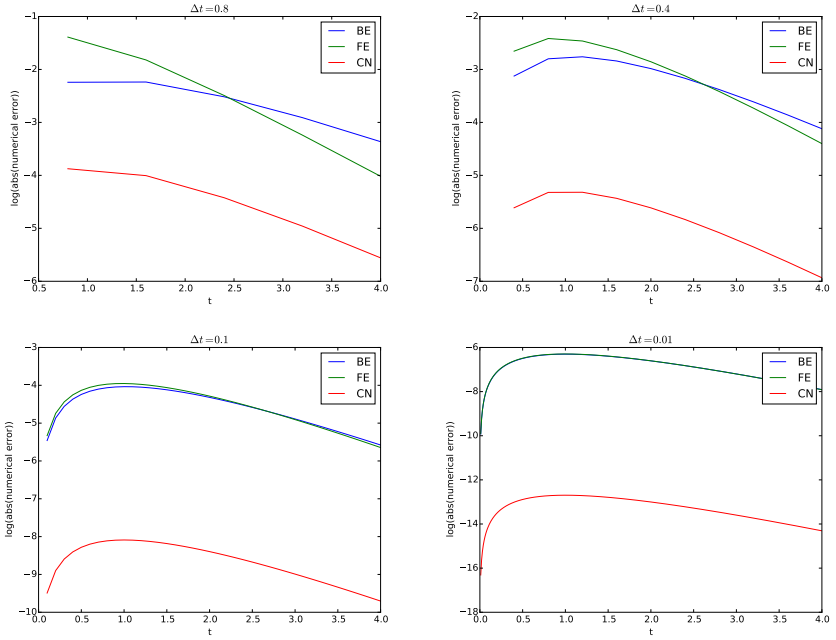

Figure 10: Discretization errors in various schemes for 4 time step values.

The computer code used to generate the plots appear next. It makes use of a `solver` function for computing the numerical solution of $u' = -au$ with the θ-rule.

```
def discretization_errors():
    from numpy import log, abs
    I = 1
    a = 1
    T = 4
    t = linspace(0, T, 101)
    schemes = {'FE': 0, 'BE': 1, 'CN': 0.5}  # theta to scheme name
    dt_values = [0.8, 0.4, 0.1, 0.01]
    for dt in dt_values:
        figure()
        legends = []
        for scheme in schemes:
            theta = schemes[scheme]
            u, t = solver(I, a, T, dt, theta)
            u_e = model(t, I, a)
            error = u_e - u
            # Plot log(error), but exclude error[0] since it is 0
            plot(t[1:], log(abs(error[1:])))
            legends.append(scheme)
        xlabel('t');  ylabel('log(abs(numerical error))')
        legend(legends, loc='upper right')
        title(r'$\Delta t=%g$' % dt)
        savefig('tmp_dt%g.png' % dt); savefig('tmp_dt%g.pdf' % dt)
```

5.4 Rounding errors

Real numbers on a computer are represented by floating-point numbers, which means that just a finite number of digits are stored and used. Therefore, the floating-point number is an approximation to the underlying real number. When doing arithmetics with floating-point numbers, there will be small approximation errors, called round-off errors or rounding errors, that may or may not accumulate in comprehensive computations. A typical example is shown below.

```
>>> 1.0/51*51
1.0
>>> 1.0/49*49
0.9999999999999999
```

We see there is not an exact result in the latter case, but a rounding error of 10^{-16}. This is the typical level of a rounding error from an arithmetic operation with 64 bit floating-point numbers (`float` object in Python, often called `double` or double precision in other languages).

What is the effect of using `float` objects and not exact arithmetics? Python has a `Decimal` object in the `decimal` module that allows us to use as many digits in floating-point numbers as we like. We take 1000 digits as the true answer where rounding errors are negligible, and then we run our numerical algorithm (the Crank-Nicolson scheme to be precise) with `Decimal` objects for all real numbers and test the error arising from using 4, 16, 64, and 128 digits.

When computing with numbers around unity in size, we typically get a rounding error of 10^{-d}, where d is the number of digits used. However, if we compute with numbers that are larger, e.g., the u values implied by $I = 1000$ and $a = 100$, the rounding errors increase to about 10^{-d+3}. Below is a table of the the computed maximum rounding error for various number of digits and two different magnitudes of I and a.

digits	$I = 1$, $a = 1$	$I = 1000$, $a = 100$
4	$3.05 \cdot 10^{-4}$	$3.05 \cdot 10^{-1}$
16	$1.71 \cdot 10^{-16}$	$1.58 \cdot 10^{-13}$
64	$2.99 \cdot 10^{-64}$	$2.06 \cdot 10^{-61}$
128	$1.60 \cdot 10^{-128}$	$2.41 \cdot 10^{-125}$

The computer code for doing these experiments need a new version of the `solver` function where we do arithmetics with `Decimal` objects:

```
def solver_decimal(I, a, T, dt, theta):
    """Solve u'=-a*u, u(0)=I, for t in (0,T] with steps of dt."""
    from numpy import zeros, linspace
    from decimal import Decimal as D
    dt = D(dt)
    a = D(a)
    theta = D(theta)
    Nt = int(round(D(T)/dt))
    T = Nt*dt
    u = zeros(Nt+1, dtype=object)  # array of Decimal objects
    t = linspace(0, float(T), Nt+1)
```

```
    u[0] = D(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 function below carries out the experiments. Note that we can set the number of digits as we want through the `decimal.getcontext().prec` variable.

```
def rounding_errors(I=1, a=1, T=4, dt=0.1):
    import decimal
    from numpy import log, array, abs
    digits_values = [4, 16, 64, 128]
    # "Exact" arithmetics is taken as 1000 decimals here
    decimal.getcontext().prec = 1000
    u_e, t = solver_decimal(I=I, a=a, T=T, dt=dt, theta=0.5)
    for digits in digits_values:
        decimal.getcontext().prec = digits  # set no of digits
        u, t = solver_decimal(I=I, a=a, T=T, dt=dt, theta=0.5)
        error = u_e - u
        error = array(error[1:], dtype=float)
        print '%d digits, max abs(error): %.2E' % \
              (digits, abs(error).max())
```

5.5 Discussion of the size of various errors

References

Index

A-stable methods, 10
amplification factor, 9

consistency, 18
convergence, 18

error
    amplification factor, 14
    global, 14

interactive Python, 12
`isympy`, 12

L-stable methods, 10

Monte Carlo simulation, 23

stability, 9, 18
`sympy`, 12