Algorithms and implementations for exponential decay models

Hans Petter Langtangen[1,2]

Center for Biomedical Computing, Simula Research Laboratory[1]

Department of Informatics, University of Oslo[2]

Aug 21, 2023

Slides selected/modified by Mikael Mortensen

Professor Hans Petter Langtangen (1962-2016)

- 2011-2015 Editor-In-Chief SIAM J of Scientific Computing
- Author of 13 published books on scientific computing
- Professor of Mechanics, University of Oslo 1998
- Developed INF5620 (which became IN5270 and now MAT-MEK4270)
- Memorial page

Myself

- Professor of mechanics (2019-)
- PhD in mathematical modelling of turbulent combustion
- Norwegian Defence Research Establishment (2007-2012)
- Computational Fluid Dynamics
- High Performance Computing

1. MAT-MEK4270 in a nutshell

2. Finite difference methods

3. Implementation

4. Verifying the implementation

MAT-MEK4270 in a nutshell

- Numerical methods for partial differential equations (PDEs)
- How do we solve a PDE in practice and produce numbers?
- How do we trust the answer?
- Approach: *simplify, understand, generalize*
- IN5670 -> IN5270 -> MAT-MEK4270 - Lots of old material

After the course

You see a PDE and can't wait to program a method and visualize a solution! Somebody asks if the solution is right and you can give a convincing answer.

More specific contents: finite difference methods

- Simple ODEs
- Exponential decay $u_t = -au(t)$ in time
- Helmholtz' equation $u_{tt} + \omega^2 u(t) = 0$ (Vibration)
- write your own software from scratch
- understand how the methods work and why they fail

1. Langtangen, Finite Difference Computing with exponential decay - Chapters 1 and 2.
2. Langtangen and Linge, Finite Difference Computing with PDEs - Parts of chapters 1 and 2.

More specific contents: Variational methods (Galerkin)

- Approximating functions with global variational methods
- Approximating functions with finite element methods
- Approximating equations with global variational methods
- Approximating equations with finite element methods

- More advanced PDEs (e.g., $u_{tt} = \nabla^2 u$ in 1D, 2D, 3D)
- perform hand-calculations, write your own software (1D)
- understand how the methods work and why they fail

1. Langtangen and Mardal, Introduction to Numerical Methods for Variational Problems

Philosophy: simplify, understand, generalize

- Start with simplified ODE/PDE problems
- Learn to reason about the discretization
- Learn to implement, verify, and experiment
- Understand the method, program, and results
- Generalize the problem, method, and program

This is the power of applied mathematics!

Required software

- Our software platform: Python (sometimes combined with Cython, Fortran, C, C++)
- Important Python packages: `numpy`, `scipy`, `matplotlib`, `sympy`, `fenics`, `shenfun`, ...
- Anaconda Python
- Jupyter notebooks

Assumed/ideal background

- IN1900: Python programming, solution of ODEs
- Some experience with finite difference methods
- Some analytical and numerical knowledge of PDEs
- Much experience with calculus and linear algebra
- Much experience with programming of mathematical problems
- Experience with mathematical modeling with PDEs (from physics, mechanics, geophysics, or ...)

Start-up example for the course

What if you don't have this ideal background?

- Students come to this course with very different backgrounds
- First task: summarize assumed background knowledge by going through a simple example
- Also in this example:
    - Some fundamental material on software implementation and software testing
    - Material on analyzing numerical methods to understand why they can fail
    - Applications to real-world problems

Start-up example

ODE problem

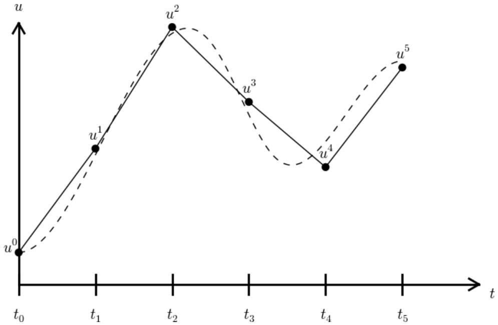
$$u' = -au, \quad u(0) = I,\ t \in (0, T],$$

where $a > 0$ is a constant.

Everything we do is motivated by what we need as building blocks for solving Partial Differential Equations (PDEs)!

Finite difference methods

- The finite difference method is the simplest method for solving differential equations
- Satisfies the equations in discrete points, not continuously
- Fast to learn, derive, and implement
- A very useful tool to know, even if you aim at using the finite element or the finite volume method



Topics in the first intro to the finite difference method

Contents

- How to think about finite difference discretization
- Key concepts:
  - mesh
  - mesh function
  - finite difference approximations
- The Forward Euler, Backward Euler, and Crank-Nicolson methods
- Finite difference operator notation
- How to derive an algorithm and implement it in Python
- How to test the implementation

The steps in the finite difference method

Solving a differential equation by a finite difference method consists of four steps:

1. discretizing the domain,
2. fulfilling the equation at discrete time points,
3. replacing derivatives by finite differences,
4. solve the discretized problem. (Often with a recursive algorithm in 1D)

Step 1: Discretizing the domain

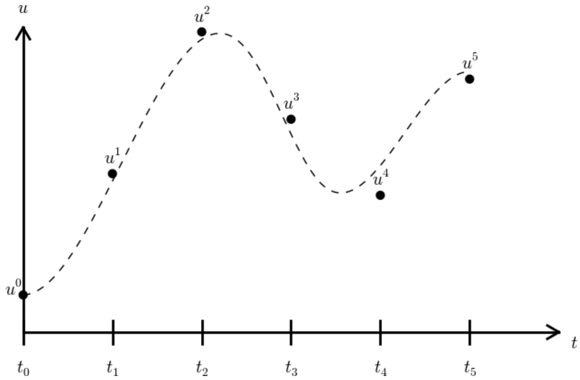
The time domain $[0, T]$ is represented by a *mesh*: a finite number of $N_t + 1$ points

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

- We seek the solution u at the mesh points: $u(t_n)$, $n = 1, 2, \ldots, N_t$.
- Note: u^0 is known as I.
- Notational short-form for the numerical approximation to $u(t_n)$: u^n
- In the differential equation: u is the exact solution
- In the numerical method and implementation: u^n is the numerical approximation, $u_{\mathrm{e}}(t)$ is the exact solution

Step 1: Discretizing the domain

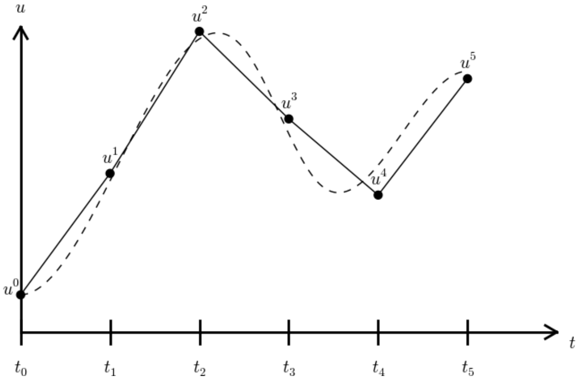
u^n is a mesh function, defined at the mesh points t_n, $n = 0, \ldots, N_t$ only.



What about a mesh function between the mesh points?

Can extend the mesh function to yield values between mesh points by *linear interpolation*:

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



Step 2: Fulfilling the equation at discrete time points

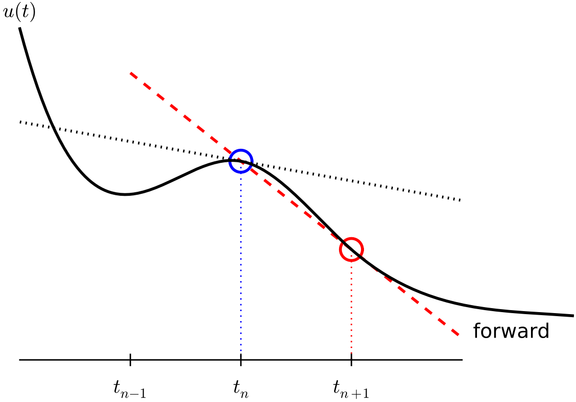
- The ODE holds for all $t \in (0, T]$ (infinite no of points)
- Idea: let the ODE be valid at the mesh points only (finite no of points)

$$u'(t_n) = -au(t_n), \quad n = 1, \ldots, N_t \tag{2}$$

Step 3: Replacing derivatives by finite differences

Now it is time for the *finite difference* approximations of derivatives:

$$u'(t_n) \approx \frac{u^{n+1} - u^n}{t_{n+1} - t_n} \qquad (3)$$



Step 3: Replacing derivatives by finite differences

Inserting the finite difference approximation in

$$u'(t_n) = -au(t_n)$$

gives

$$\frac{u^{n+1} - u^n}{t_{n+1} - t_n} = -au^n, \quad n = 0, 1, \ldots, N_t - 1 \tag{4}$$

(Known as *discrete equation*, or *discrete problem*, or *finite difference method/scheme*)

Step 4: Formulating a recursive algorithm

How can we actually compute the u^n values?

- given $u^0 = I$
- compute u^1 from u^0
- compute u^2 from u^1
- compute u^3 from u^2 (and so forth)

In general: we have u^n and seek u^{n+1}

The Forward Euler scheme

Solve wrt u^{n+1} to get the computational formula:

$$u^{n+1} = u^n - a(t_{n+1} - t_n)u^n \tag{5}$$

Let us apply the scheme by hand

Assume constant time spacing: $\Delta t = t_{n+1} - t_n = \text{const}$ such that $u^{n+1} = u^n(1 - a\Delta t)$

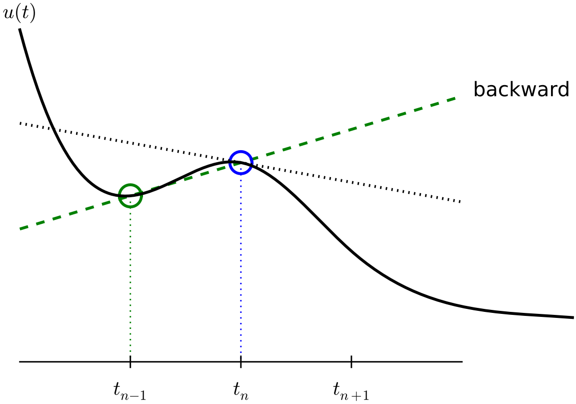
$$
\begin{aligned}
u^0 &= I,\\
u^1 &= I(1 - a\Delta t),\\
u^2 &= I(1 - a\Delta t)^2,\\
&\vdots\\
u^{N_t} &= I(1 - a\Delta t)^{N_t}
\end{aligned}
$$

Ooops - we can find the numerical solution by hand (in this simple example)! No need for a computer (yet)...

A backward difference

Here is another finite difference approximation to the derivative (backward difference):

$$u'(t_n) \approx \frac{u^n - u^{n-1}}{t_n - t_{n-1}} \tag{6}$$



The Backward Euler scheme

Inserting the finite difference approximation in $u'(t_n) = -au(t_n)$ yields the Backward Euler (BE) scheme:

$$\frac{u^n - u^{n-1}}{t_n - t_{n-1}} = -au^n \qquad (7)$$

Solve with respect to the unknown u^{n+1}:

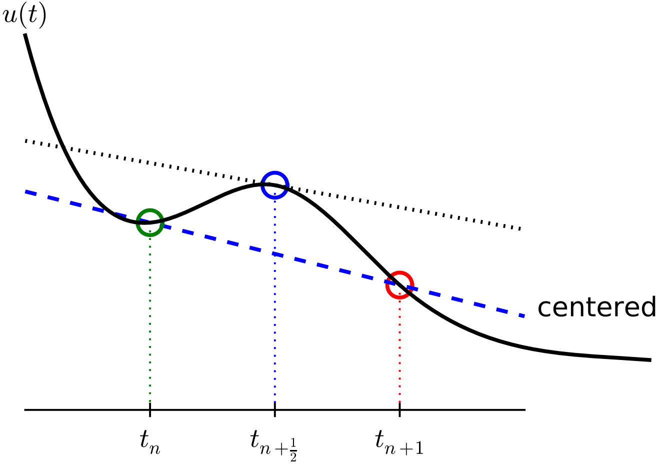
$$u^{n+1} = \frac{1}{1 + a(t_{n+1} - t_n)} u^n \qquad (8)$$

Notice

We use u^{n+1} as unknown, so above we rename $u^n \longrightarrow u^{n+1}$ and $u^{n-1} \longrightarrow u^n$.

A centered difference

Centered differences are better approximations than forward or backward differences.



The Crank-Nicolson scheme; ideas

Idea 1: let the ODE hold at $t_{n+\frac{1}{2}}$. With $N_t + 1$ points, that is N_t equations for $n = 0, 1, \ldots N_t - 1$

$$u'(t_{n+\frac{1}{2}}) = -au(t_{n+\frac{1}{2}})$$

Idea 2: approximate $u'(t_{n+\frac{1}{2}})$ by a centered difference

$$u'(t_{n+\frac{1}{2}}) \approx \frac{u^{n+1} - u^n}{t_{n+1} - t_n} \tag{9}$$

Problem: $u(t_{n+\frac{1}{2}})$ is not defined, only $u^n = u(t_n)$ and $u^{n+1} = u(t_{n+1})$

Solution:

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

The Crank-Nicolson scheme; result

Result:

$$\frac{u^{n+1} - u^n}{t_{n+1} - t_n} = -a\frac{1}{2}(u^n + u^{n+1}) \tag{10}$$

Solve wrt to u^{n+1}:

$$u^{n+1} = \frac{1 - \frac{1}{2}a(t_{n+1} - t_n)}{1 + \frac{1}{2}a(t_{n+1} - t_n)}u^n \tag{11}$$

This is a Crank-Nicolson (CN) scheme or a midpoint or centered scheme.

The unifying θ-rule

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

$$\frac{u^{n+1} - u^n}{t_{n+1} - t_n} = -a(\theta u^{n+1} + (1 - \theta)u^n) \tag{12}$$

- $\theta = 0$: Forward Euler
- $\theta = 1$: Backward Euler
- $\theta = 1/2$: Crank-Nicolson
- We may alternatively choose any $\theta \in [0, 1]$.

u^n is known, solve for u^{n+1}:

$$u^{n+1} = \frac{1 - (1 - \theta)a(t_{n+1} - t_n)}{1 + \theta a(t_{n+1} - t_n)} u^n \tag{13}$$

Constant time step

Very common assumption (not important, but exclusively used for simplicity hereafter): constant time step $t_{n+1} - t_n \equiv \Delta t$

Summary of schemes for constant time step

$$u^{n+1} = (1 - a\Delta t)u^n \qquad \text{Forward Euler} \tag{14}$$

$$u^{n+1} = \frac{1}{1 + a\Delta t}u^n \qquad \text{Backward Euler} \tag{15}$$

$$u^{n+1} = \frac{1 - \frac{1}{2}a\Delta t}{1 + \frac{1}{2}a\Delta t}u^n \qquad \text{Crank-Nicolson} \tag{16}$$

$$u^{n+1} = \frac{1 - (1 - \theta)a\Delta t}{1 + \theta a\Delta t}u^n \qquad \text{The } \theta - \text{rule} \tag{17}$$

Compact operator notation for finite differences

- Finite difference formulas can be tedious to write and read/understand
- Handy tool: finite difference operator notation
- Advantage: communicates the nature of the difference in a compact way

$$[D_t^- u = -au]^n \tag{18}$$

Specific notation for difference operators

Forward difference:

$$[D_t^+ u]^n = \frac{u^{n+1} - u^n}{\Delta t} \approx \frac{d}{dt} u(t_n) \tag{19}$$

Centered difference (around t_n):

$$[D_t u]^n = \frac{u^{n+\frac{1}{2}} - u^{n-\frac{1}{2}}}{\Delta t} \approx \frac{d}{dt} u(t_n), \tag{20}$$

Backward difference:

$$[D_t^- u]^n = \frac{u^n - u^{n-1}}{\Delta t} \approx \frac{d}{dt} u(t_n) \tag{21}$$

The Backward Euler scheme with operator notation

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

Common to put the whole equation inside square brackets:

$$[D_t^- u = -au]^n \tag{22}$$

The Forward Euler scheme with operator notation

$$[D_t^+ u = -au]^n \tag{23}$$

The Crank-Nicolson scheme with operator notation

Introduce an averaging operator:

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

The Crank-Nicolson scheme can then be written as

$$[D_t u = -a\overline{u}^t]^{n+\frac{1}{2}} \tag{25}$$

Test: use the definitions and write out the above formula to see that it really is the Crank-Nicolson scheme!

Implementation

Model:

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

Numerical method:

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

for $\theta \in [0, 1]$. Note

- $\theta = 0$ gives Forward Euler
- $\theta = 1$ gives Backward Euler
- $\theta = 1/2$ gives Crank-Nicolson

Requirements of a program

- Compute the numerical solution u^n, $n = 1, 2, \ldots, N_t$
- Display the numerical and exact solution $u_{\mathrm{e}}(t) = e^{-at}$
- Bring evidence to a correct implementation (*verification*)
- Compare the numerical and the exact solution in a plot
- Compute the error $u_{\mathrm{e}}(t_n) - u^n$

Tools to learn

- Basic Python programming
- Array computing with `numpy`
- Plotting with `matplotlib.pyplot`
- File writing and reading

Why implement in Python?

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

Why implement in Python?

- 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.

Algorithm

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

Translation to Python function

```
import numpy as np

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

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

Note about the for loop: `range(0, Nt, s)` generates all integers from `0` to `Nt` in steps of `s` (default 1), *but not including* `Nt` (!).

Sample call:

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

Integer division

Python applies integer division: `1/2` is 0, while `1./2` or `1.0/2` or `1/2.` or `1/2.0` or `1.0/2.0` all give 0.5.

A safer `solver` function (`dt = float(dt)` - guarantee float):

```python
import numpy as np

def solver(I, a, T, dt, theta):
    """Solve u'=-a*u, u(0)=I, for t in (0,T] with steps of dt."""
    dt = float(dt)            # avoid integer division
    Nt = int(round(T/dt))     # no of time intervals
    T = Nt*dt                 # adjust T to fit time step dt
    u = np.zeros(Nt+1)           # array of u[n] values
    t = np.linspace(0, T, Nt+1)  # time mesh
    u[0] = I                  # assign initial condition
    for n in range(0, Nt):    # n=0,1,...,Nt-1
        u[n+1] = (1 - (1-theta)*a*dt)/(1 + theta*dt*a)*u[n]
    return u, t
```

Doc strings

- First string after the function heading
- Used for documenting the function
- Automatic documentation tools can make fancy manuals for you
- Can be used for automatic testing

```
def solver(I, a, T, dt, theta):
    """
    Solve

        u'(t) = -a*u(t),

    with initial condition u(0)=I, for t in the time interval
    (0,T]. The time interval is divided into time steps of
    length dt.

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

Formatting of numbers

Can control formatting of reals and integers through the *printf* format:

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

Or the alternative *format string syntax*:

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

Or even better through the alternative *f-string syntax*:

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

Running the program

How to run the program `decay_v1.py`.

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

Can also run it as "normal" Unix programs: `./decay_v1.py` if the first line is

```
`#!/usr/bin/env python`
```

Then

```
Terminal> chmod a+rx decay_v1.py
Terminal> ./decay_v1.py
```

Plotting the solution

Basic syntax:

```
import matplotlib.pyplot as plt

plt.plot(t, u)
plt.show()
```

Can (and should!) add labels on axes, title, legends.

Comparing with the exact solution

Python function for the exact solution $u_{\rm e}(t) = Ie^{-at}$:

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

Quick plotting:

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

Problem: $u_{\rm e}(t)$ applies the same mesh as u^n and looks as a piecewise linear function.

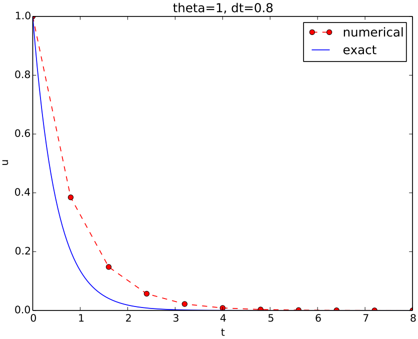
Remedy: Introduce a very fine mesh for $u_{\rm e}$.

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

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

Add legends, axes labels, title, and wrap in a function

```
def plot_numerical_and_exact(theta, I, a, T, dt):
    """Compare the numerical and exact solution in a plot."""
    u, t = solver(I=I, a=a, T=T, dt=dt, theta=theta)
    t_e = np.linspace(0, T, 1001)        # fine mesh for u_e
    u_e = u_exact(t_e, I, a)
    plt.plot(t, u, 'r--o', t_e, u_e, 'b-')
    plt.legend(['numerical', 'exact'])
    plt.xlabel('t'); plt.ylabel('u')
    plt.title('theta=%g, dt=%g' % (theta, dt))
    plt.savefig('plot_%s_%g.png' % (theta, dt))
```



Verifying the implementation

- Verification = bring evidence that the program works
- Find suitable test problems
- Make function for each test problem
- Later: put the verification tests in a professional testing framework

Simplest method: run a few algorithmic steps by hand

Use a calculator ($I = 0.1$, $\theta = 0.8$, $\Delta t = 0.8$):

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

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

See the function `test_solver_three_steps` in `decay_v3.py`.

Comparison with an exact discrete solution

Best verification

Compare computed numerical solution with a closed-form *exact discrete solution* (if possible).

Define

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

Repeated use of the θ-rule:

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

Making a test based on an exact discrete solution

The exact discrete solution is

$$u^n = IA^n \tag{26}$$

Notice

Understand what n in u^n and in A^n means!

Test if

$$\max_n |u^n - u_{\mathrm{e}}(t_n)| < \epsilon \sim 10^{-15}$$

Computing the numerical error as a mesh function

Task: compute the numerical error $e^n = u_{\mathrm{e}}(t_n) - u^n$

Exact solution: $u_{\mathrm{e}}(t) = Ie^{-at}$, implemented as

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

Compute e^n by

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

Array arithmetics - we compute on entire arrays!

- `u_exact(t, I, a)` works with `t` as array
- Must have `exp` from `numpy` (not `math`)
- `e = u_e - u`: array subtraction
- Array arithmetics gives shorter and much faster code

Computing the norm of the error

- e^n is a mesh function
- Usually we want one number for the error
- Use a norm of e^n

Norms of a function $f(t)$:

$$||f||_{L^2} = \left(\int_0^T f(t)^2 dt\right)^{1/2} \tag{27}$$

$$||f||_{L^1} = \int_0^T |f(t)| dt \tag{28}$$

$$||f||_{L^\infty} = \max_{t\in[0,T]} |f(t)| \tag{29}$$

Norms of mesh functions

- Problem: $f^n = f(t_n)$ is a mesh function and hence not defined for all t. How to integrate f^n?
- Idea: Apply a numerical integration rule, using only the mesh points of the mesh function.

The Trapezoidal rule:

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

Common simplification yields the L^2 norm of a mesh function:

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

Norms - notice!

Notice

- The *continuous* norms use capital L^2, L^1, L^∞
- The *discrete* norm uses lowercase ℓ^2

Implementation of the norm of the error

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

Python w/array arithmetics:

```
e = u_exact(t) - u
E = np.sqrt(dt*np.sum(e**2))
```

Comment on array vs scalar computation

Scalar computing of `E = np.sqrt(dt*np.sum(e**2))`:

```
m = len(u)      # length of u array (alt: u.size)
u_e = np.zeros(m)
t = 0
for i in range(m):
    u_e[i] = u_exact(t, a, I)
    t = t + dt
e = np.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 = np.sqrt(dt*s)
```

Scalar computing

takes more code, is less readable and runs much slower

Rule

Compute on entire arrays (when possible)! Vectorization!