Extension to a variable coefficient; Forward and Backward Euler

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

The Forward Euler scheme:
\[ \frac{u^{n+1} - u^n}{\Delta t} = -a(t_n)u^n \]  

The Backward Euler scheme:
\[ \frac{u^n - u^{n-1}}{\Delta t} = -a(t_n)u^n \]  

Extension to a variable coefficient; Crank-Nicolson

Evaluating at \( t_n + \frac{1}{2} \) and using an average for \( u \):
\[ \frac{u^{n+1} - u^n}{\Delta t} = -\frac{1}{2}a(t_n)u^n + \frac{1}{2}a(t_{n+1})u^{n+1} \]  

Using an average for \( a \) and \( u \):
\[ \frac{u^{n+1} - u^n}{\Delta t} = -\frac{1}{2}a(t_n)u^n + \frac{1}{2}a(t_{n+1})u^{n+1} \]  

Extension to a variable coefficient; \( \theta \)-rule

The \( \theta \)-rule unifies the three mentioned schemes:
\[ u_{n+1} - u_n \Delta t = -a ((1 - \theta)t_n + \theta t_{n+1})((1 - \theta)u_n + \theta u_{n+1}) \]  
or,
\[ u_{n+1} - u_n \Delta t = -(1 - \theta)a(t_n)u^n - \theta a(t_{n+1})u^{n+1} \]  

Extension to a variable coefficient; operator notation

\[
\begin{align*}
[D]u &= -a_D u^0, \\
[D]_--u &= -a_{-} u^0, \\
[D]_+u &= -a_+ \frac{u^{n+1}}{2}, \\
[D]_-u &= -a_- \frac{u^{n+1}}{2} \\
\end{align*}
\]  

Extension to a source term

\[ u'(t) = -a(t)u(t) + b(t), \quad t \in [0, T], \quad u(0) = I \]  

\[
\begin{align*}
[D]u &= -a_D u^0, \\
[D]_--u &= -a_{-} u^0, \\
[D]_+u &= -a_+ b^{n+1} + \frac{b^0}{2}, \\
[D]_-u &= -a_- b^{n+1} + \frac{b^0}{2} \\
\end{align*}
\]
Implementation of the generalized model problem

\[ u^{n+1} = \left(1 - \Delta t(1 - \theta) a^n \right) u^n + \Delta t(\theta b^n + (1 - \theta) b^n) \right) \left(1 + \Delta t(\theta a^n) \right) \]

Implementation where \( a(t) \) and \( b(t) \) are given as Python functions (see file decay_vc.py):

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

Implementations of variable coefficients; functions

Plain functions:

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

def b(t):
    return 1
```

Implementations of variable coefficients; classes

Better implementation: class with the parameters \( a_0, tp, \) and \( k \) as attributes and a special method \( \_\_call\_\_ \) for evaluating \( a(t) \):

```python
class A:
    def __init__(self, a0=1, k=2):
        self.a0, self.k = a0, k
    def __call__(self, t):
        return self.a0 if t < self.tp else self.k*self.a0
```

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

Implementations of variable coefficients; lambda function

Quick writing: a one-liner

```python
lambda t: a_0 if t < tp else k*a_0
```

In general,

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

is equivalent to

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

One can use lambda functions directly it calls:

```python
u, t = solver(1, lambda t: 1, lambda t: 1, T, dt, theta)
```

for a problem \( u' = -u + 1, u(0) = 1 \).

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

Verification via trivial solutions

Start debugging of a new code with trying a problem where \( u = \text{const} \neq 0 \).

Choose \( u = C \) (a constant). Choose any \( a(t) \) and set \( b = a(t)C \) and \( I = C \).

*All* numerical methods will reproduce \( u = \text{const} \neq 0 \) exactly (machine precision).

Often \( u = C \) eases debugging.

If the example any error in the formula for \( u^{n+1} \) make \( u \neq C \)!
Estimating the convergence rate r

Perform 1: Numerical experiments: \((\Delta t_i, E_i)\), \(i = 0, \ldots, m - 1\). Two methods for finding \(r\) and \(C\):

- Take the logarithm of (13), \( E = r \Delta t + C \), and fit a straight line to the data pairs \((\Delta t_i, E_i)\), \(i = 0, \ldots, m - 1\).
- Consider two consecutive experiments, \((\Delta t_i, E_i)\) and \((\Delta t_{i+1}, E_{i+1})\). Dividing the equation \(E_{i+1} = C \Delta t_{i+1} \) by \(E_i = C \Delta t_i \) and solving for \(r\) yields

\[
\log(E_{i+1}/E_i) / \log(\Delta t_{i+1}/\Delta t_i) = r
\]

for \(i = 1, \ldots, m - 1\).

Method 2 is best.
We embed the code in a real test function.

```python
def test_convergence_rates():
    # Create a manufactured solution
    # define u_exact(t), a(t), b(t)
    # Create a manufactured solution
    # Turn sympy expressions into Python function
    b = sym.lambdify([t], b, modules='numpy')
    a = sym.lambdify([t], a, modules='numpy')
    u_exact = sym.lambdify([t], u_exact, modules='numpy')
    # Turn sympy expressions into Python function
    E_values = []
    for dt in dt_values:
        E_values.append(E)
        E = sqrt(dt*sum(e**2))
    E = u_e - u
    u_e = u_exact(t)
    u, t = solver(I=I, a=a, b=b, T=6, dt=dt, theta=theta)
    expected_rate = 2 if theta == 0.5 else 1
    tol = 0.1
    diff = abs(expected_rate - r[-1])
    assert diff < tol
```

The Backward Euler method gives a system of algebraic equations

The Backward Euler scheme:

\[
\begin{align*}
u^{n+1} &= u^n + \Delta t (a u^n + b v^n) \\
v^{n+1} &= v^n + \Delta t (c u^n + d v^n)
\end{align*}
\]

which is a \(2 \times 2\) linear system:

\[
\begin{align*}
(1 - \Delta t a) u^{n+1} + b v^{n+1} &= u^n \\
c u^{n+1} + (1 - \Delta t d) v^{n+1} &= v^n
\end{align*}
\]

Crank-Nicolson also gives a \(2 \times 2\) linear system.
Generic form

The standard form for ODEs:
\[ u' = f(u, t), \ u(0) = I \] (23)

\( u \) and \( f \): scalar or vector.

Vectors in case of ODE systems:
\[ u(t) = (u_0(t), u_1(t), \ldots, u_m(t)) \]
\[ f(u, t) = (f_0(u), f_1(u), \ldots, f_m(u)) \]
\[ f^0(u_0, u_1, \ldots, u_m), \ldots, f^{m-1}(u_0(t), u_1(t), \ldots, u_m(t)) \]

The θ-rule

\[ u^{n+1} - u^n = \frac{\theta f(u^{n+1}, t_{n+1}) + (1 - \theta)f(u^n, t_n)}{\Delta t} \] (24)

Bringing the unknown \( u^{n+1} \) to the right-hand side and the known \( u^n \) to the left-hand side gives
\[ u^{n+1} - \Delta t f(u^n, t_n) = u^n + \Delta t (1 - \theta)f(u^n, t_n) \] (25)

This is a nonlinear equation in \( u^{n+1} \) (unless \( f \) is linear in \( u \))!

The Leapfrog scheme

Idea:
\[ u'(t_n) \approx \frac{u^{n+1} - u^{n-1}}{2\Delta t} = [D^2 t u]_n \] (26)

Scheme:
\[ [D^2 t u = f(u, t)]_n \]

or written out,
\[ u^{n+1} = u^n - 2\Delta t f(u^n, t_n) \] (27)

- Some other scheme must be used as starter (%u^n%).
- Explicit scheme - a nonlinear \( f \) (in \( u \)) is usually hard.
- Downside: Leapfrog is always unstable after some time.

The filtered Leapfrog scheme

After computing \( u^{n+1} \), stabilize Leapfrog by
\[ u^n \leftarrow u^n + \gamma(u^{n+1} - 2u^n + u^{n+1}) \] (28)

2nd-order Runge-Kutta scheme

Forward-Euler + approximate Crank-Nicolson:
\[ u^n = u^n + \Delta t f(u^n, t_n) \]
\[ u^{n+1} = u^n + \Delta t \left( f(u^n, t_n) + f(u^n + \Delta t f(u^n, t_n)) \right) \] (30)

\[ u^{n+1} - u^n = \frac{\theta f(u^{n+1}, t_{n+1}) + (1 - \theta)f(u^n, t_n)}{\Delta t} \] (24)

Bringing the unknown \( u^{n+1} \) to the right-hand side and the known \( u^n \) to the left-hand side gives
\[ u^{n+1} - \Delta t f(u^n, t_n) = u^n + \Delta t (1 - \theta)f(u^n, t_n) \] (25)

This is a nonlinear equation in \( u^{n+1} \) (unless \( f \) is linear in \( u \))!
The most famous and widely used ODE method
4 evaluations of $f$ per time step
Its derivation is a very good illustration of numerical thinking!

$$u^{n+1} = u^n + \frac{1}{2} \Delta t \left( 3f(u^n, t_n) - f(u^{n-1}, t_{n-1}) \right)$$  (31)

The Odespy software
Odespy features simple Python implementations of the most fundamental schemes as well as Python interfaces to several famous packages for solving ODEs: ODEPACK, Vode, rkc, RKF45, Radau5, as well as the ODE solvers in Scipy, SymPy, and odeib.

Typical usage:
# Define right-hand side of ODE
def f(u, t):
    return -a*u
import odespy
import numpy as np
# Set parameters and time mesh
I = 1; a = 2; T = 6; dt = 1.0
Nt = int(round(T/dt))
t_mesh = np.linspace(0, T, Nt+1)
# Use a 4th-order Runge-Kutta method
solver = odespy.RK4(f)
solver.set_initial_condition(I)
u, t = solver.solve(t_mesh)

# + lots of plot code...

Example: Runge-Kutta methods
solvers = [odespy.RK2(f),
          odespy.RK3(f),
          odespy.RK4(f),
          odespy.BackwardEuler(f, nonlinear_solver='Newton')]
for solver in solvers:
solver.set_initial_condition(I)
u, t = solver.solve(t)
# + lots of plot code...

Plots from the experiments
The 4-th order Runge-Kutta method (RK4) is the method of choice!
Example: Adaptive Runge-Kutta methods

- Adaptive methods find "optimal" locations of the mesh points to ensure that the error is less than a given tolerance.
- Disadvantage: approximate error estimation, not always optimal locations of points.
- "Industry standard ODE solver": Dormand-Prince 4/5-th order Runge-Kutta (MATLAB's famous ode45).