# Lecture 1

## Contents



In lecture 1 we will consider a simple mathematical model for exponential decay. Important topics of the first lecture are

- The finite difference (FD) method
- Forward/Backward Euler methods
- The Crank-Nicolson method
- Stability of numerical schemes
- Implementation of the FD methods using recursive solvers
- Verification of the implementations
- Error norms
- Convergence rates

This first lecture may be easier to follow as the slides that were presented on the first day.

# Finite difference models for exponential decay

A model for exponential decay is

$$\frac{du}{dt} = -au, \quad u(0) = I, \quad t \in [0, T], \tag{1}$$

where $a > 0$ is a constant and $u(t)$ is the solution. For this course it is not very important what $u(t)$ represents, but it could be any scalar like temperature or money. Something that decays exponentially in time.

We want to solve Eq. (1) using a finite difference numerical method. This may seem strange since the exact solution to (1) is trivially obtained as

Skip to main content

$$u(t) = I \exp(-at). \tag{2}$$

However, this exact solution will serve us well for validation of the finite difference schemes. Especially for the computation of convergence rates.

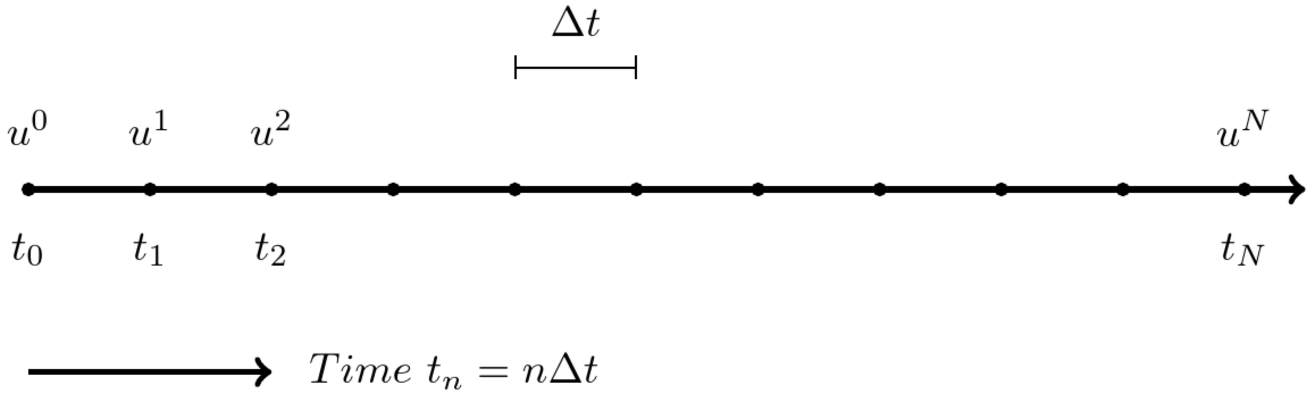
# The Finite difference method

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)

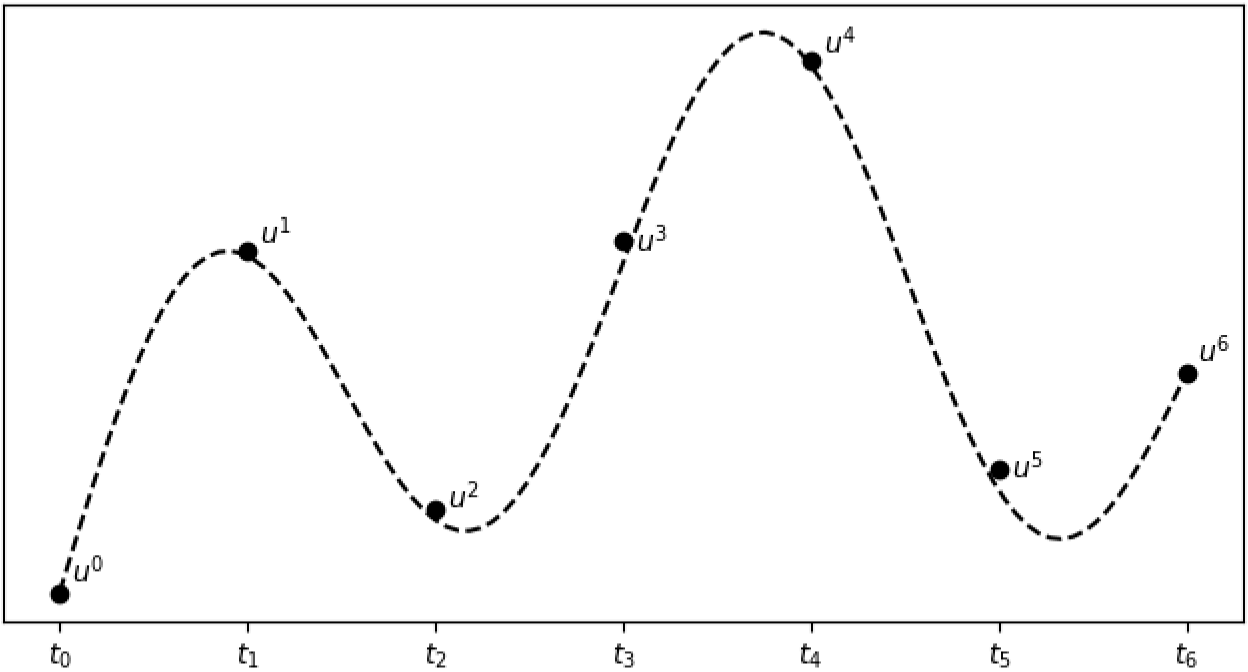
## Step 1 - discretization

The finite difference method divides (in 1D) the line into a mesh and solves equations only for specific locations (nodes) in the mesh. A mesh is created for $t = 0, \Delta t, 2\Delta t, \ldots, N\Delta t$. To this end we use the discrete times $t_n = n\Delta t$ for $n = 0, 1, \ldots N$ and $T = t_N = N\Delta t$. Similarly we use the discrete solutions $u^n = u(t_n)$ for $n = 0, 1, \ldots, N$.




The finite difference solution $\{u^n\}_{n=0}^N$ is a **mesh function** and it is defined only at the mesh points in the domain. For example as shown below. Note that the FD solution is not necessarily equal to the exact solution.

▶ Show code cell source




## Step 2 - fulfilling the equation at discrete time points

The $N+1$ unknowns $\{u^n\}_{n=0}^N$ requires $N+1$ equations. For our problem the initial condition is known and we set $u^0=I$. This leaves $N$ unknowns, or degrees of freedom. In order to find these unknown we can simply demand that

$$u'(t_n) = -au(t_n), \quad \forall\, n = 1, \ldots, N$$

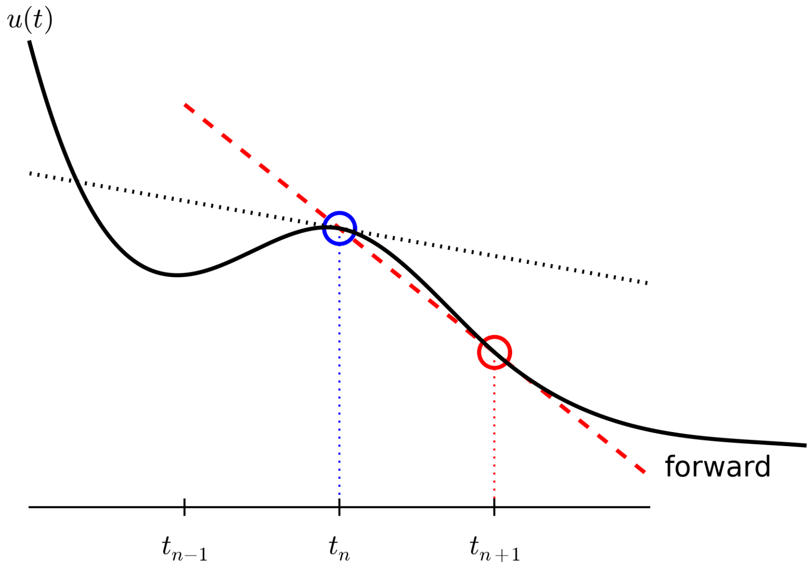
which gives us the $N$ equations that we need.

## Step 3: Replacing derivatives by finite differences

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

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

Skip to main content




Inserting the finite difference approximation in

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

gives

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

which is 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$\$

Skip to main content

# Let us apply the scheme by hand

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

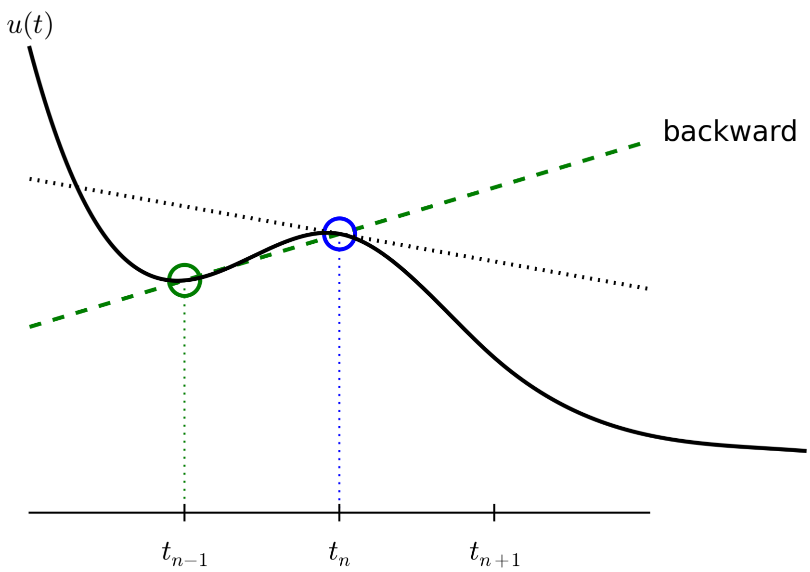
$$
\begin{aligned}
u^0 &= I, \\
u^1 &= I(1 - a\Delta t), \\
u^2 &= I(1 - a\Delta t)^2, \\
&\vdots \\
u^N &= I(1 - a\Delta t)^N
\end{aligned}
$$

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):

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




# The Backward Euler scheme

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

Skip to main content

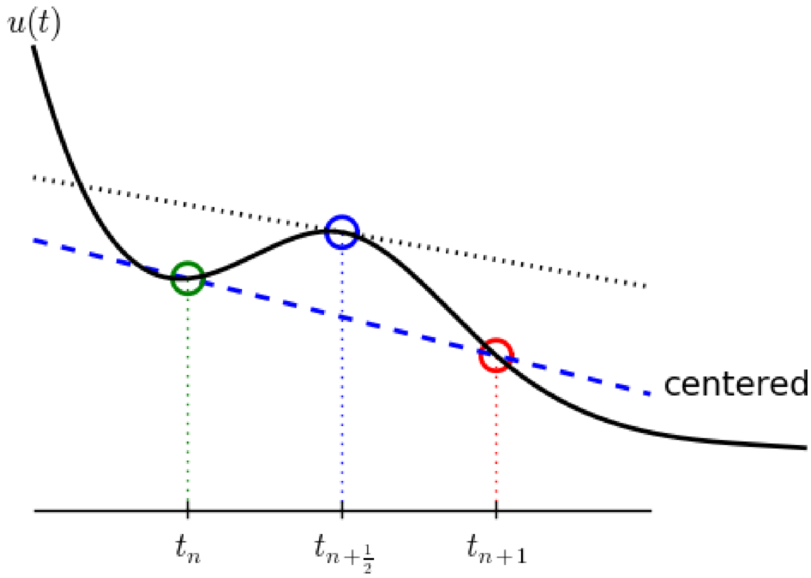
$$\frac{u^n - u^{n-1}}{t_n - t_{n-1}} = -au^n$$

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

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

# 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 $N+1$ points, that is $N$ equations for $n = 0, 1, \ldots N-1$

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

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

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

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

Skip to main content

Solution (linear interpolation):

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

# The Crank-Nicolson scheme; result

Result:

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

Solve wrt to $u^{n+1}$:

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

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

# The unifying $\theta$-rule

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

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

- $\theta = 0$: 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 $u^{n+1}$:

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

Skip to main content

# 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

$$
\begin{aligned}
u^{n+1} &= (1 - a\Delta t)u^n \quad \text{(FE)} \\
u^{n+1} &= \frac{1}{1 + a\Delta t} u^n \quad \text{(BE)} \\
u^{n+1} &= \frac{1 - \frac{1}{2} a\Delta t}{1 + \frac{1}{2} a\Delta t} u^n \quad \text{(CN)} \\
u^{n+1} &= \frac{1 - (1-\theta) a\Delta t}{1 + \theta a\Delta t} u^n \quad (\theta - \text{rule})
\end{aligned}
$$

# 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 $u^n$, $n = 1, 2, \ldots, N$
- Display the numerical and exact solution $u_e(t) = e^{-at}$

Skip to main content

- Compare the numerical and the exact solution in a plot
- Compute the error $u_e(t_n) - u^n$
- If wanted, compute the convergence rate of the numerical scheme

# Algorithm

- Store $u^n$, $n = 0, 1, \ldots, N$ in an array $\boldsymbol{u}$.
- Algorithm:
  - initialize $u^0$
  - for $t = t_n$, $n = 1, 2, \ldots, N$: compute $u^n$ using the $\theta$-rule formula

```
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

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

```
t= 0.000 u=1
t= 0.800 u=0.384615
t= 1.600 u=0.147929
t= 2.400 u=0.0568958
t= 3.200 u=0.021883
t= 4.000 u=0.00841653
t= 4.800 u=0.00323713
t= 5.600 u=0.00124505
t= 6.400 u=0.000478865
t= 7.200 u=0.000184179
t= 8.000 u=7.0838e-05
```

For example, you have three arrays

$$\boldsymbol{u} = (u_i)_{i=0}^N, \boldsymbol{v} = (v_i)_{i=0}^N, \boldsymbol{w} = (w_i)_{i=0}^N$$

Skip to main content

$$w_i = u_i \cdot v_i, \quad \forall\, i = 0, 1, \ldots, N$$

Regular (scalar) implementation:

```
N = 1000
u = np.random.random(N)
v = np.random.random(N)
w = np.zeros(N)

for i in range(N):
    w[i] = u[i] * v[i]
```

Vectorized:

```
w[:] = u * v
```

Numpy is heavily vectorized! So much so that mult, add, div, etc are vectorized by default!

Now lets get rid of the for-loop!

```
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]
```

How? This is difficult because it is a **recursive** update and not regular **elementwise** multiplication. But remember

$$A = (1 - (1-\theta)a\Delta t)/(1 + \theta \Delta t a)$$

$$\begin{aligned}
u^1 &= A u^0, \\
u^2 &= A u^1, \\
&\vdots \\
u^{N_t} &= A u^{N-1}
\end{aligned}$$

Because we have this exact numerical solution we can implement

as follows

because

Skip to main content

$$
u^n = A^n u^0, \quad \text{since} \begin{cases} u^1 &= Au^0, \\ u^2 &= Au^1 = A^2u^0, \\ & \vdots \\ u^{N_t} &= Au^{N-1} = A^N u^0 \end{cases}
$$

To show how cumprod works, just consider the following

```
np.cumprod([1, 2, 2, 2])
```

```
array([1, 2, 4, 8])
```

# Why vectorization?

- Python for-loops are slow!
- Python for-loops usually requires more lines of code.

```
def f0(u, I, theta, a, dt):
    u[0] = I
    u[1:] = (1 - (1-theta)*a*dt)/(1 + theta*dt*a)
    u[:] = np.cumprod(u)
    return u

def f1(u,  I, theta, a, dt):
    u[0] = I
    for n in range(0, len(u)-1):
        u[n+1] = (1 - (1-theta)*a*dt)/(1 + theta*dt*a)*u[n]
    return u

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

assert np.allclose(f0(u.copy(), I, theta, a, dt),
                   f1(u.copy(), I, theta, a, dt))
```

Lets try some timings!

```
%timeit -q -o -n 1000 f0(u, I, theta, a, dt)
```

```
<TimeitResult : 2.08 µs ± 309 ns per loop (mean ± std. dev. of 7 runs, 1,000
```

```
%timeit -q -o -n 1000 f1(u, I, theta, a, dt)
```

Skip to main content

```
<TimeitResult : 2.14 µs ± 33 ns per loop (mean ± std. dev. of 7 runs, 1,000
```

Hmm. Not really what's expected. Why? Because the array `u` is really short! Lets try a longer array

```
print(f"Length of u = {u.shape[0]}")
```

```
Length of u = 11
```

```
dt = dt/10
u, t = solver(I, a, T, dt, theta)
print(f"Length of u = {u.shape[0]}")
```

```
Length of u = 101
```

```
%timeit -q -o -n 100 f0(u, I, theta, a, dt)
```

```
<TimeitResult : 2.94 µs ± 1.24 µs per loop (mean ± std. dev. of 7 runs, 100
```

```
%timeit -q -o -n 100 f1(u, I, theta, a, dt)
```

```
<TimeitResult : 20.3 µs ± 262 ns per loop (mean ± std. dev. of 7 runs, 100 l
```

Even longer array:

```
dt = dt/10
u, t = solver(I, a, T, dt, theta)
print(f"Length of u = {u.shape[0]}")
```

```
Length of u = 1001
```

```
%timeit -q -o -n 100 f0(u, I, theta, a, dt)
```

Skip to main content

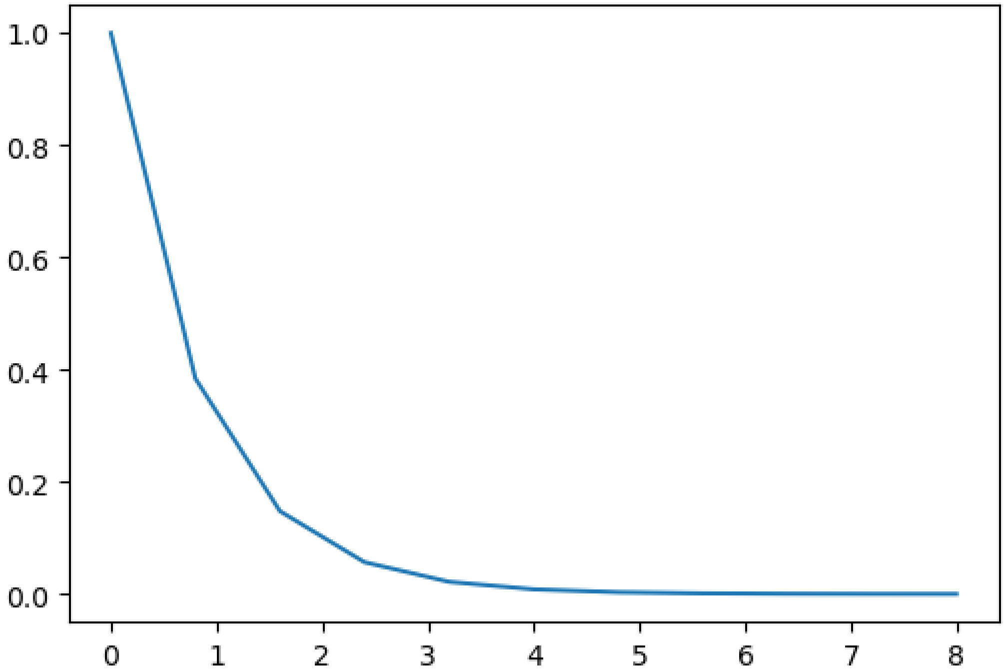
```
%timeit -q -o -n 100 f1(u, I, theta, a, dt)
```

```
<TimeitResult : 213 µs ± 5.21 µs per loop (mean ± std. dev. of 7 runs, 100 l
```

Vectorized code takes the same time! Only overhead costs, not the actual computation.

# Plot the solution

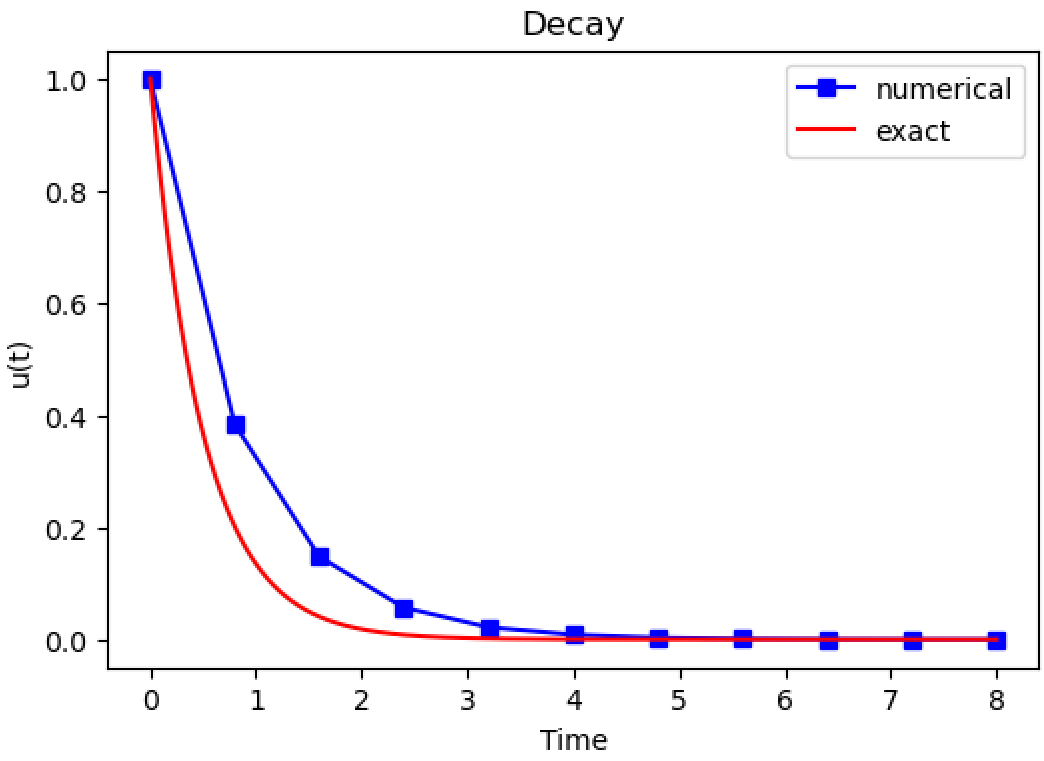
```
import matplotlib.pyplot as plt
I, a, T, dt, theta = 1, 2, 8, 0.8, 1
u, t = solver(I, a, T, dt, theta)
fig = plt.figure(figsize=(6, 4))
ax = fig.gca()
ax.plot(t, u);
```




Add legends, titles, exact solution, etc. Make the plot nice:-)

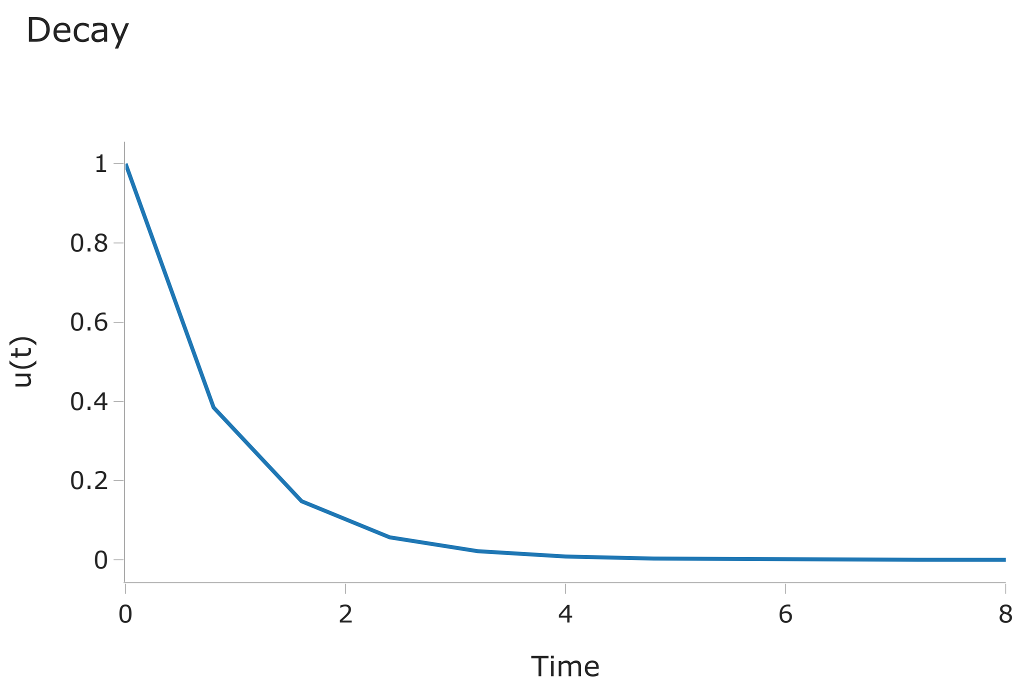
```
u_exact = lambda t, I, a: I*np.exp(-a*t)
I, a, T = 1., 2., 8.
u, t = solver(I=I, a=a, T=T, dt=0.8, theta=1)
te = np.linspace(0, T, 1000)
ue = u_exact(te, I, a)
fig = plt.figure(figsize=(6, 4))
plt.plot(t, u, 'bs-', te, ue, 'r')
plt.title('Decay')
plt.legend(['numerical', 'exact'])
plt.xlabel('Time'), plt.ylabel('u(t)');
```

Skip to main content




# Plotly is a very good alternative

```
import plotly.express as px
pfig = px.line(x=t, y=u, labels={'x': 'Time', 'y': 'u(t)'},
               width=600, height=400, title='Decay',
               template="simple_white")
pfig.show()
```




Skip to main content

# 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

## Comparison with exact numerical solution

Repeated use of the $\theta$-rule gives exact numerical solution:

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

Exact solution on the other hand:

$$u(t) = \exp(-at), \quad u(t_n) = \exp(-at_n)$$

## Making a test based on an exact numerical solution

The exact discrete solution is

$$u^n = IA^n$$

Test if your solver gives

$$\max_n |u^n - IA^n| < \epsilon \sim 10^{-15}$$

for a few precalculated steps.

## Run a few numerical steps by hand

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

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

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

## The test based on exact numerical solution

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

    Nt = 3  # number of time steps
    u, t = solver(I=I, a=a, T=Nt*dt, dt=dt, theta=theta)
    tol = 1E-14  # tolerance for comparing floats
    diff = abs(u - u_by_hand).max()
    success = diff < tol
    assert success, diff

test_solver_three_steps(solver)
```

# Quantifying the error

## Computing the norm of the error

- $e^n = u^n - u_e(t_n)$ is a **mesh function**
- Usually we want one number for the error
- Use a norm of $e^n$

Norms of a function $f(t)$:

$$
\begin{aligned}
||f||_{L^2} &= \left(\int_0^T f(t)^2 dt\right)^{1/2} \\
||f||_{L^1} &= \int_0^T |f(t)| dt \\
||f||_{L^\infty} &= \max_{t\in[0,T]} |f(t)|
\end{aligned}
$$

## 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:

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

Common simplification yields the $\ell^2$ norm of a mesh function:

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

## Norms - notice!

- The *continuous* norms use capital $L^2, L^1, L^\infty$
- The *discrete* norm uses lowercase $\ell^2, \ell^1, \ell^\infty$

## Implementation of the error norm

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

Python with vectorization:

Skip to main content

```python
u_exact = lambda t, I, a: I*np.exp(-a*t)
I, a, T, dt, theta = 1., 2., 8., 0.8, 1
u, t = solver(I, a, T, dt, theta)
en = u_exact(t, I, a) - u
E = np.sqrt(dt*np.sum(en**2))
print(f'Errornorm = {E}')
```

```
Errornorm = 0.1953976935916231
```

Previous
Lecture Notes MATMEK-4270

Next
Lecture 2