# SEMT30002 Scientific Computing and Optimisation

## Week 1 Demos: Finite differences and Euler's method

## Matthew Hennessy

---

## Finite differences

In the first demo, we look at how to numerically approximate the derivative of the function $f(x) = e^{2x}$ using the forwards differencing formula given by

$$f'(a) = \frac{f(a + \Delta x) - f(a)}{\Delta x} + O(\Delta x).$$

We start by importing some functions and packages:

In [1]:
```python
from math import exp
import numpy as np
import matplotlib.pyplot as plt
```

Now we define a Python function that implements the first-order accurate forwards difference approximation

In [2]:
```python
def forward(f, a, dx):
    """
    Implements the first-order forwards difference formula
    """
    return (f(a+dx) - f(a)) / dx
```

We also define a Python function for the mathematical function $f(x) = e^{2x}$.

In [3]:
```python
def f(x):
    """
    Implements the exponential function given by
    f(x) = exp(2 * x) using NumPy functions
    """
    return np.exp(2 * x)
```

Now we use the forwards difference formula to approximate $f'(1)$ using 20 different values of $\Delta x$ that are logarithmically spaced between $10^{-4}$ and $10^{-1}$.

In [4]:
```python
# The point where we want to evaluate the derivative
a = 1

# A NumPy array with values of \Delta x
dx = np.logspace(-4, -1, 20)

# Approximate the derivative using forwards differencing
dfdx_f = forward(f, a, dx)
```

Now we calculate the exact value of the derivative, which is given by $f'(1) = 2e$. Using the exact value of the derivative, we can calculate the truncation error.

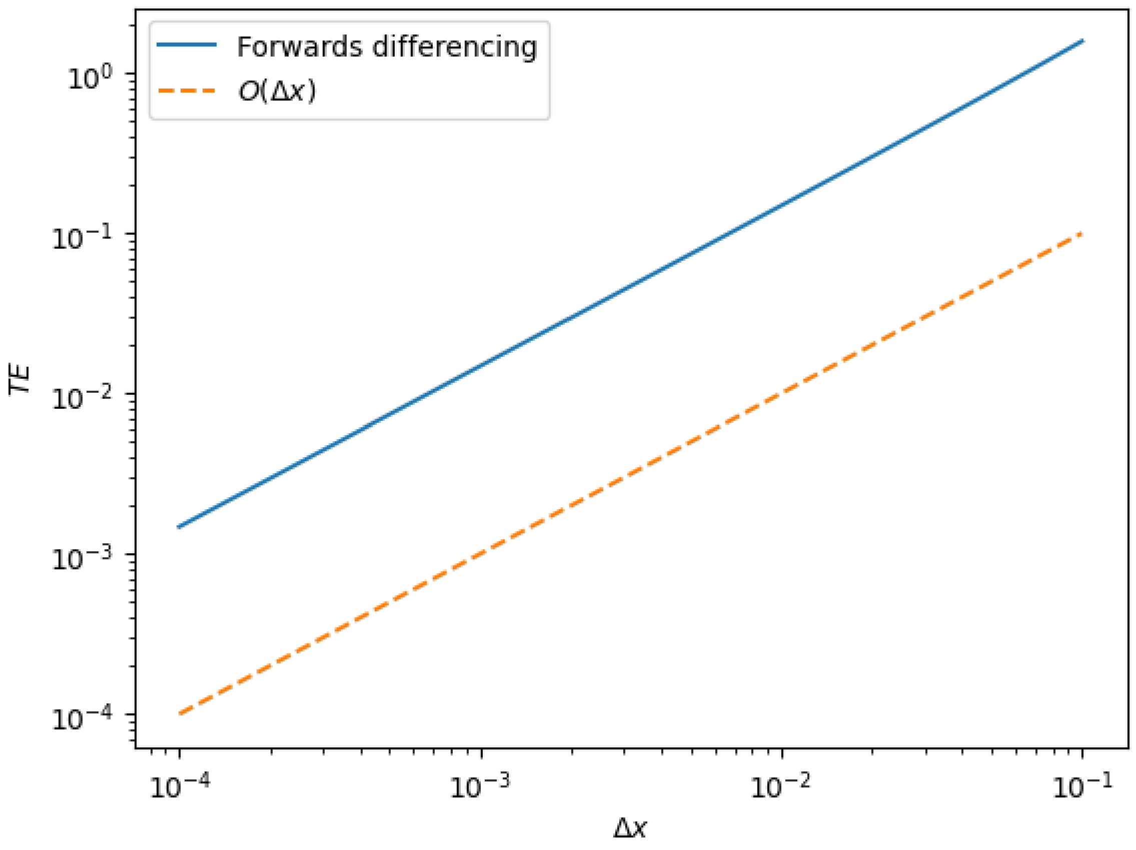
In [5]:
```python
# Compute the exact value of the derivative
dfdx_exact = 2 * exp(2 * a)

# Compute the (absolute value) of the truncation error
TE_f = np.abs(dfdx_exact - dfdx_f)
```

We now create a log-log plot of the truncation error as a function of $\Delta x$. The use of logarithmic axes will make it easier to determine the rate at which the truncation error decreases with $\Delta x$, since the slope of the curve will indicate the power of $\Delta x$ that the truncation error depends on.

Theory tells us that the truncation error should linearly decrease with $\Delta x$. Therefore, we also plot the curve $TE = \Delta x$. When plotted using log axes, the slopes of the two curves should be the same.

In [6]:
```python
plt.loglog(dx, TE_f, label = 'Forwards differencing')
plt.loglog(dx, dx, '--', label = '$O(\Delta x)$')
plt.xlabel('$\Delta x$')
plt.ylabel('$TE$')
plt.legend()
plt.show()
```



## Forwards vs central differencing

We will now re-calculate the derivatives using the central difference approximation to showcase the increase in accuracy. We start by defining a Python function that implements the central difference formula

In [7]:
```
def central(f, a, dx):
    """
    Implements the second-order central difference formula
    """
    return (f(a+dx) - f(a-dx)) / 2 / dx
```

Now we approximate the derivative and calculate the truncation error

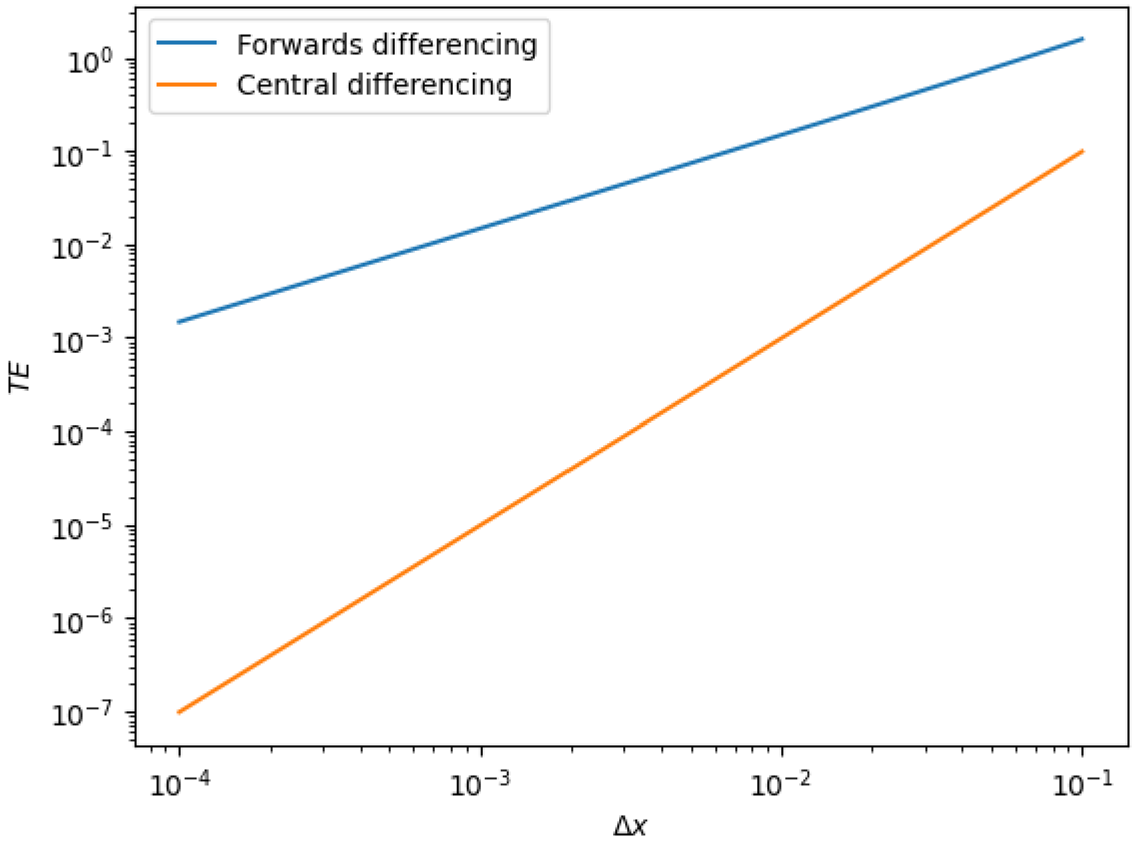
In [8]:
```
# Approximate the derivative using forwards differencing
dfdx_c = central(f, a, dx)

# Calculate the truncation error
TE_c = np.abs(dfdx_exact - dfdx_c)
```

Finally, we plot the truncation errors obtained using the forwards and central difference formulae.

In [9]:
```
plt.loglog(dx, TE_f, label = 'Forwards differencing')
plt.loglog(dx, TE_c, label = 'Central differencing')
plt.xlabel('$\Delta x$')
```

```python
plt.ylabel('$TE$')
plt.legend()
plt.show()
```



The truncation error is much smaller when central differencing is used, indicating this is a much more accurate method. Moreover, it is easy to see that the slope of the orange curve is two, indicating that the truncation error of the central difference formula is $O((\Delta x)^2)$, as expected from theory.

## Is my differentiation formula correct?

Analysing the truncation error can be useful for determining whether a numerical method is working correctly. In the context of numerical differentiation, it is important that the truncation error tends to zero as $\Delta x \to 0$. If this is not observed, then this is a sign that there's a mistake in the approximation formula. However, it is important to remember that round-off error will prevent the truncation error from ever tending to zero as $\Delta x$ decreases.

In this demo, we deliberately introduce an error into the forwards difference formula for the first derivative. We then see how this manifests in the truncation error.

We start by defining an incorrect Python function for the forwards difference formula:

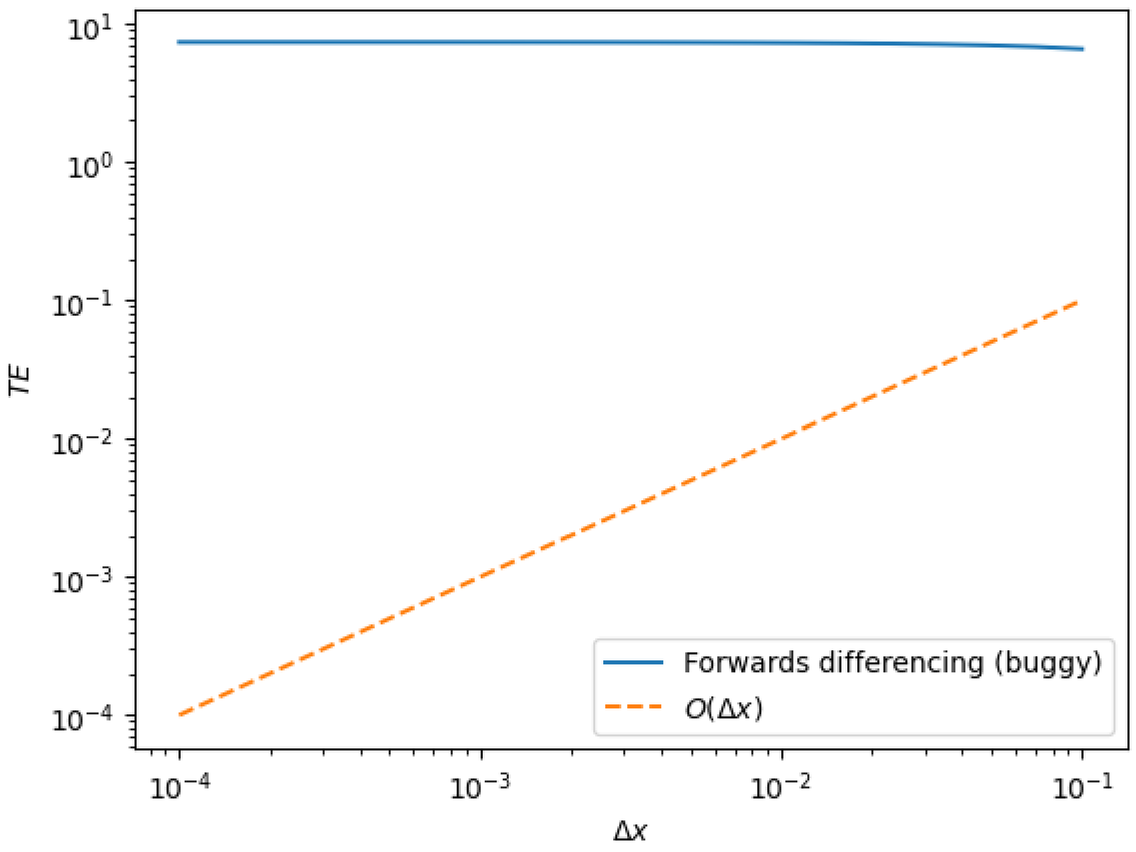
In [10]:
```python
def forward_error(f, a, dx):
    """
    Implements a buggy version of the forwards difference formula
    for the first-order derivative
    """
    return (f(a+dx) - f(a)) / 2 / dx
```

As before, we calculate the derivative and the truncation error. The results are then plotted

In [11]:
```
# calc the derivative
dfdx_num = forward_error(f, a, dx)

# calc the truncation error
TE = np.abs(dfdx_exact - dfdx_num)

# plot
plt.loglog(dx, TE, label = 'Forwards differencing (buggy)')
plt.loglog(dx, dx, '--', label = '$O(\Delta x)$')
plt.xlabel('$\Delta x$')
plt.ylabel('$TE$')
plt.legend()
plt.show()
```



We clearly see that the truncation error is not decreasing with $\Delta x$, so there must be a bug in the code!

## Euler's method and vectorisation

We now turn to Euler's method; the simplest numerical method for solving ODE initial value problems (IVPs). This is a straightforward method to implement using `for` loops, but this can lead to slow implementations. Hence, it is better to use vectorised operations when possible.

This demo will showcase the benefits of vectorisation through Euler's method.

Consider the system of $M$ linear ODEs given by

$$\frac{d\boldsymbol{u}}{dt} = -\mathbf{A}\boldsymbol{u}$$

where $\boldsymbol{u} = (u_1, u_2, \dots, u_M)^T$ and $\mathbf{A}$ is an $M \times M$ random matrix with entries between 0 and $10^{-2}$. The initial condition is $\boldsymbol{u}(0) = 1$.

We will use two versions of Euler's method to solve this problem over the time range $0 \le t \le 1$ using $N_t = 100$ time steps. We'll assume the system has $M = 200$ ODEs. Let's define some Python variables for this problem

In [2]:
```
"""
Parameters for time stepping
"""

# Max time
t_max = 1

# Number of time steps
Nt = 100

# Calculation of time step size \Delta t
dt = t_max / Nt

# Calculation of time points (including t = 0)
t = dt * np.arange(Nt + 1)

"""
Parameters for the ODE system
"""
# Size of the ODE system
M = 200

# Generate the random matrix
A = 1e-2 * np.random.random((M, M))
```

We will also pre-allocate an array to store the solution at each time step to improve the performance of the solver. This array will have size $M \times (N_t + 1)$, where the $N_t + 1$ arises from storing the initial condition and the solution computed at $N_t$ Euler steps. This shaping of the array implies that columns of the array will correspond to different time points and rows will correspond to different solution components.

We initially set the values in the array to zero. The correct values will be placed in the array as they are computed.

In [3]:
```
"""
Pre-allocation
"""
u = np.zeros((M, Nt+1))
```

## Slow implementation without vectorisation

We now write the right-hand side ODE system using native Python operations that do not use vectorised operations. That is, we use nested for loops to compute the matrix-vector product $-\boldsymbol{A}\boldsymbol{u}$.

In [10]:
```
def f(u, t, A):
    """
    Computes the RHS of the ODE system; in this case,
    f(u, t) = -A * u
    """

    # Get dimension of ODE system
    M = len(u)

    # Pre-allocate an array to store the RHS of the ODE
    rhs = np.zeros(M)

    # Compute matrix-vector product and store in the RHS array
    for i in range(M):
        for j in range(M):
            rhs[i] -= A[i,j] * u[j]

    # Return the RHS array
    return rhs
```

Now we set the initial condition. We do this by overwriting the first column of the solution array `u` that was pre-allocated

In [11]:
```
"""
Setting the initial condition
"""
for i in range(M):
    u[i,0] = 1
```

Having defined all this, we can now define a Python function that implements Euler's method. Recall, that Euler's method is defined through the iteration formula as

$$\boldsymbol{u}^{n+1} = \boldsymbol{u}^n + \Delta t \boldsymbol{f}(\boldsymbol{u}^n, t^n)$$

In [12]:
```
def euler():
    """
    Slow implementation of Euler's method.  The lack of arguments
    to this function means global variables are used, which is
    not good programming practice, but it keeps the demo simple
    """

    # Loop over time steps
    for n in range(Nt):

        # Compute the RHS using the solution at the current time step
        f_array = f(u[:, n], t[n], A)

        # Loop over solution components.  Notice here the values
        # in the pre-allocated array are being overwritten
        for i in range(M):
            u[i, n+1] = u[i, n] + dt * f_array[i]
```

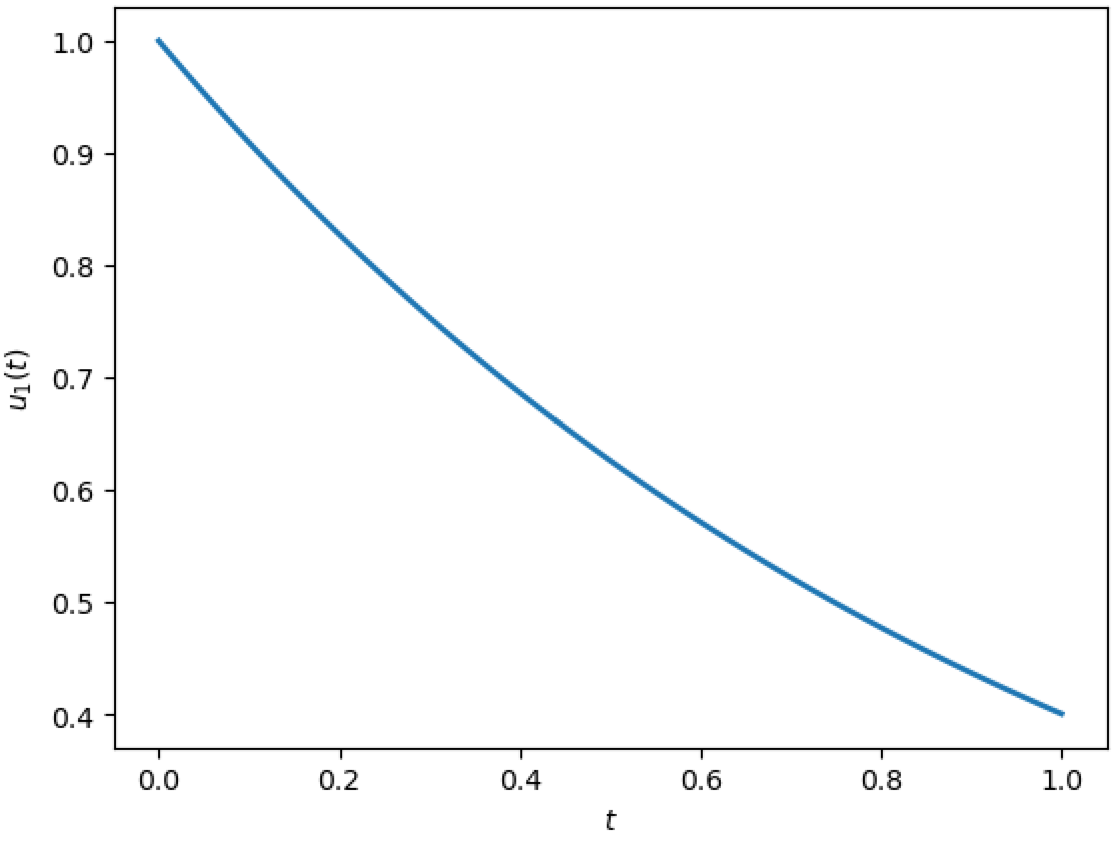
We now run the code three times and time each run:

In [13]:
```
%timeit -r 3 -n 1 euler()
```

```
1.64 s ± 53.4 ms per loop (mean ± std. dev. of 3 runs, 1 loop each)
```

And we can plot the first component of the solution

In [14]:
```
plt.plot(t, u[0,:], lw = 2)
plt.xlabel('$t$')
plt.ylabel('$u_1(t)$')
plt.show()
```



## Fast (vectorised) implementation

We can speed-up Euler's method using vectorised operations. One place this can be done is in the function that evaluates the right-hand side of the ODE. Rather than using nested for loops to compute the matrix-vector product, vectorised NumPy operations can be used:

```
In [4]: def f_vectorised(u, t, A):
            """
            Computes the RHS of the ODE system; in this case,
            f(u, t) = -A * u using vectorised operations.
            Recall that matrix-vector multiplication in NumPy
            is done using @ and not *
            """

            return -A @ u
```

This system is now solved in an vectorised implementation of Euler's method that I've coded in a Python package called `ode_solvers`. This package uses object-oriented programming to create classes for different types of ODE problems.

```
In [5]: import ode_solvers as ode
```

The code below sets up the time points and the ODE IVP

```
In [6]: # create an object that stores the time points
        grid = ode.TimeGrid(dt, Nt)

        # Set the initial condition
        u_0 = np.ones(M)

        # Create an object for the ODE IVP
        ode_problem = ode.IVP(
            lambda u,t: f_vectorised(u, t, A), # RHS of the ODE
            u_0, # initial condition
            grid # grid of time points
        )
```

Now the IVP is solved using Euler's method and timed

```
In [17]: %timeit -r 5 -n 5 ode_problem.solve(method = 'Euler')
```

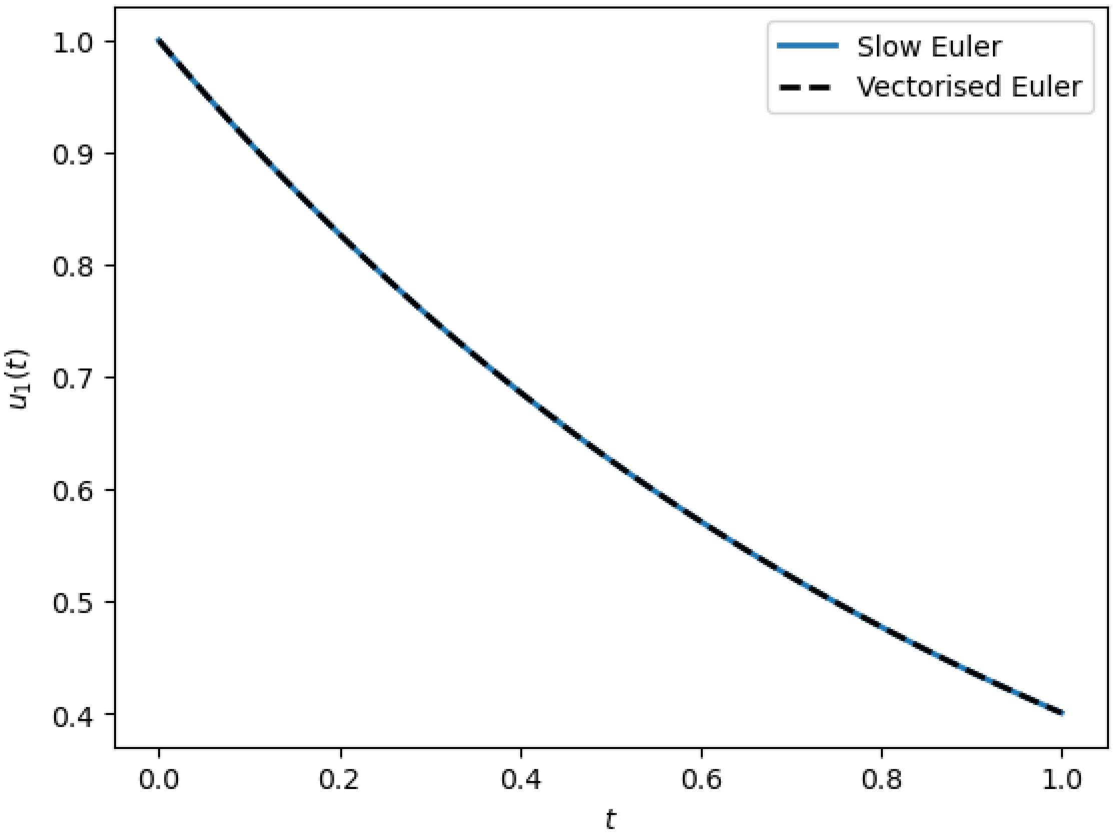
```
6.52 ms ± 1.68 ms per loop (mean ± std. dev. of 5 runs, 5 loops each)
```

We see the vectorised code is much faster (about 100 times faster).

We can also extract the solution and compare it to the previous solution to show the two methods produce the same result

```
In [16]: # Solve
u_vectorised = ode_problem.solve(method = 'Euler')

# Plot
plt.plot(t, u[0,:], label = 'Slow Euler', lw = 2)
plt.plot(grid.t, u[0,:], 'k--', label = 'Vectorised Euler', lw = 2)
plt.xlabel('$t$')
plt.ylabel('$u_1(t)$')
plt.legend()
plt.show()
```


Slow Euler
Vectorised Euler
$u_1(t)$
$t$