RECAP AND SOLUTION PREVIOUS EXERCISE

EXPLICIT EULER METHOD FOR INITIAL VALUE PROBLEM

Given the ODE

$$\frac{dy(t)}{dt} = g(t, y(t))$$

The ODE can be solved as follows. Rewrite

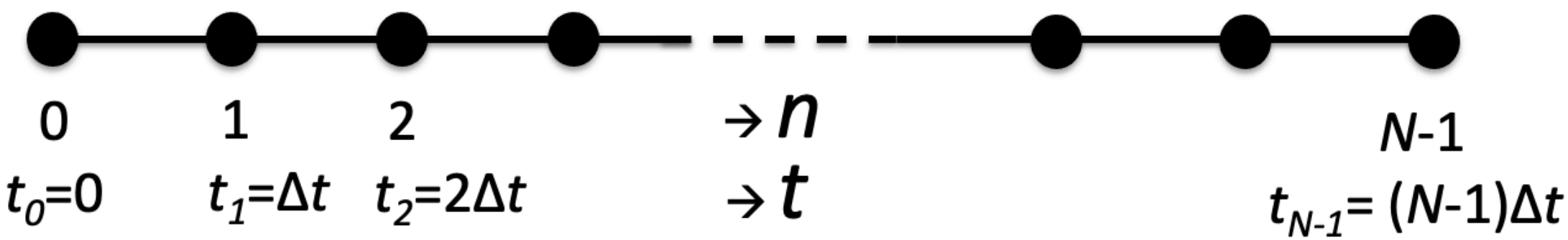
$$\frac{y(t + \Delta t) - y(t)}{\Delta t} \approx g(t, y(t))$$

as

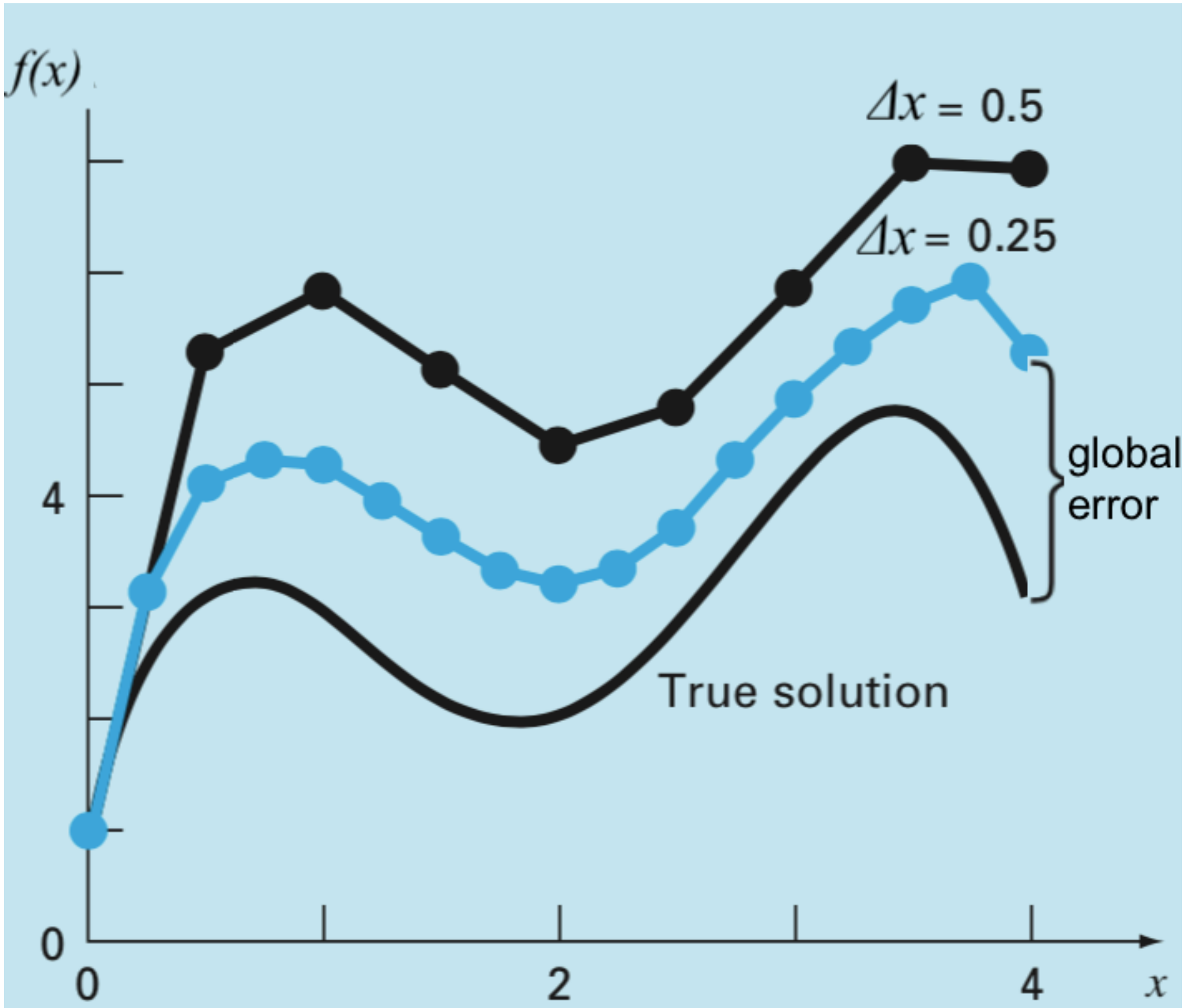
$$\boxed{y(t + \Delta t)_{\text{approx}} \approx y(t) + \Delta t \cdot g(t, y(t))}$$

This is called the **explicit Euler method**, aka the **forward Euler method**. This allows us to approximate the value of $y(t + \Delta t)$ by only having information on y at the position t.

Since the initial value is given, we can construct the whole solution at the discrete time points, by doing it step-by-step:



$t_0 \to t_1 = t_0 + \Delta t \to t_2 = t_0 + 2\Delta t \to \ldots \to t_{\text{max}} = t_{N-1} = t_0 + (N-1)\Delta t.$


f(x)
Δx = 0.5
Δx = 0.25
global
error
True solution
4
0
0
2
4
x

ALGORITHM IMPLICIT EULER METHOD FOR IVP

Given the ODE

$$\frac{dy(t)}{dt} = g(t, y(t))$$

The explicit method is

$$\frac{y(t+\Delta t) - y(t)}{\Delta t} \approx g(t, y(t))$$

The **implicit Euler method** (also known as **backward Euler method**) is

$$\frac{y(t+\Delta t) - y(t)}{\Delta t} \approx g(t+\Delta t, y(t+\Delta t)) \qquad (*)$$

Notice that now $y(t+\Delta t)$ also occurs at the right hand side of eq. $(*)$. This means that the equation is not any more an explicit equation, but an implicit one.

Rewriting gives

$$\boxed{y(t+\Delta t) \approx y(t) + \Delta t \cdot g(t+\Delta t, y(t+\Delta t))}$$

In order to determine y at the next time step, $y(t+\Delta t)$ (LHS), we already have to know $y(t+\Delta t)$.

If g is linear with y, then it is easy to transform the implicit relation into a explicit one.

If g is non-linear, then it is a bit more tricky. It is an equation containing $y(t+\Delta t)$ and could be solved using a root-finder such as Newton's method.

SOLUTIONS

The answers to the conceptual questions can be found in the lecture notes.

SOLUTION PYTHON CODE EXPLICIT EULER

See Blackboard for `.py` file

```
import numpy as np
import matplotlib.pyplot as plt

t0=0.0        # start time
tmax=1.0      # end time
dt=0.01
y0=1.0
Nint=int(round((tmax-t0)/dt))
t=np.zeros(Nint+1)
y=np.zeros(Nint+1)   # number of points is number of integration steps + 1
# Initial condition
t[0]=t0
y[0]=y0

# ODE: y(t)'=g(t, y(t))
def g(t, y):
    b=1.0
    a=22.0
    return b*t-a*y

# integrate the ODT
for n in range(Nint):
    t[n+1]=t[n]+dt
    y[n+1]=y[n]+ dt * g(t[n], y[n])  # explicit Euler

# Output end point
print("y(t=tmax)=",y[Nint])
```

DERIVATION FOR IMPLICIT EULER

$$\frac{y(t+\Delta t)-y(t)}{\Delta t} \approx g(t+\Delta t, y(t+\Delta t))$$

with $g(t,y(t)) = bt - ay(t)$, and hence $g(t+\Delta t, y(t+\Delta t)) = b \cdot (t+\Delta t) - a \cdot y(t+\Delta t)$

Hence

$$\frac{y(t+\Delta t)-y(t)}{\Delta t} \approx b \cdot (t+\Delta t) - a \cdot y(t+\Delta t)$$

The equation can be rewritten by isolating $y(t+\Delta t)$ to the LHS. The following steps show how to do this:

$$
\begin{aligned}
y(t+\Delta t) &\approx y(t) + \Delta t \cdot (b \cdot (t+\Delta t) - a \cdot y(t+\Delta t)) \\
y(t+\Delta t)(1+a\cdot\Delta t) &\approx y(t) + \Delta t \cdot b \cdot (t+\Delta t) \\
y(t+\Delta t) &\approx \frac{y(t) + \Delta t \cdot b \cdot (t+\Delta t)}{(1+a\cdot\Delta t)}
\end{aligned}
$$

In a programming language we could implement this as

```
ynext=(ynow+dt*b*(t+dt))/(1.0+a*dt)
```

or

```
y[n+1]=(y[n]+dt*b*t[n+1])/(1.0+a*dt)
```

WRONG DERIVATION

Note that for the explicit Euler method we had

```
y[n+1]=y[n]+dt*g(t[n], y[n])
```

It is tempting to use a similar construct for the implicit Euler

```
y[n+1]=y[n]+dt*g(t[n+1], y[n+1])  # WRONG
```

This does not work, because we do not know yet `y[n+1]` and hence `y[n+1]` *can not appear on the right-hand side of the assignment*.

We have to do it explicitly, as is done in the following

```
y[n+1]=(y[n]+dt*b*t[n+1])/(1.0+a*dt)   # Correct
```

and such an equation needs to be derived for each differential equation sepeerately. Hence implicit methods require a bit more work.

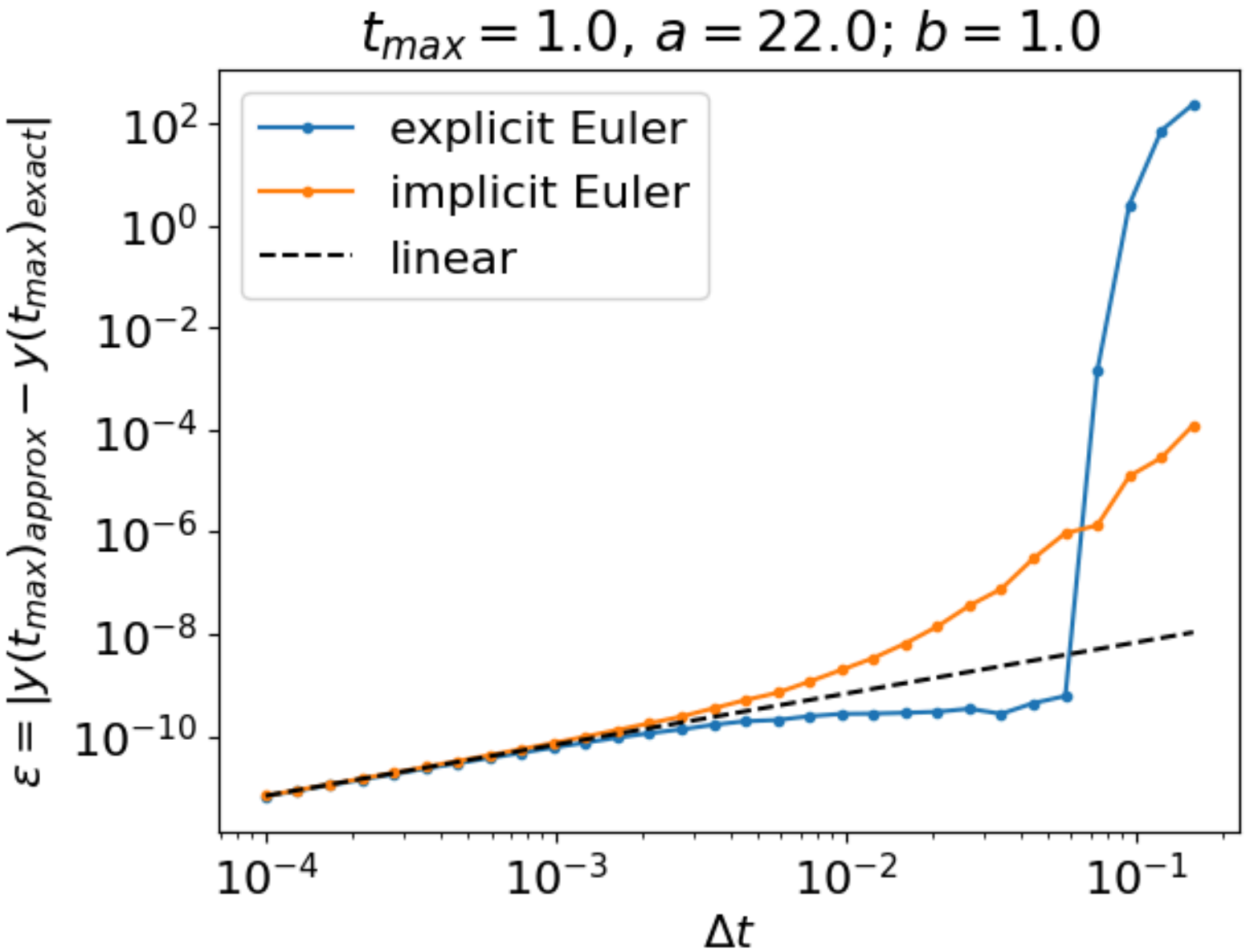
SOLUTION EXERCISE SESSION 1 REGARDING ERROR

Explicit Euler

- $\Delta t = 0.01 : y(1) = 0.04338843$
- $\Delta t = 0.1 : y(1) = 6.2479177$

Implicit Euler

- $\Delta t = 0.01 : y(1) = 0.04338843$
- $\Delta t = 0.1 : y(1) = 0.04339733$


$t_{max} = 1.0,\ a = 22.0;\ b = 1.0$
explicit Euler
implicit Euler
linear
$\varepsilon \equiv |y(t_{max})_{approx} - y(t_{max})_{exact}|$
10^{2}
10^{0}
10^{-2}
10^{-4}
10^{-6}
10^{-8}
10^{-10}
10^{-4}
10^{-3}
10^{-2}
10^{-1}
Δt

USING NUMBERS IN PROGRAMMING LANGUAGES

Always use integers for quantities you can count: number of integration steps, etc.

Always use floating point numbers for quantities you cannot count.

A 32 bit floating point number is accurate up to around 7 decimal digits (2^{-23}), while a 64 bit floating point number is accurate up to around 16 decimal digits (2^{-52}). Therefore a 64 bit float is advised.

In Python

```
a=3
```

implies that `a` is an integer (of infinite precision), while

```
a=3.0
```

implies that `a` is a 64 bit float. So one has to be aware of the way of entering numbers.

CONVERTING FLOAT TO INTEGERS AND VICE VERSA

To convert a float, say 3.0, to an integer use

- To round to the nearest number in Python use `int(round(...))`
- To round towards zero in Python use `int(...)`

To convert an int, say 2, to a float, use

- Python: `float(2)`, or just multiply by `1.0`

In Python, the resulting type can be checked using `type`. E.g., `type(int(3.0))` gives `<class 'int'>`.

TYPICAL PROGRAMMING MISTAKES

- Using `int` instead of `float` or vice versa
- If round is not used, then `int(tmax/dt)` can be off-by-one! Why? Because tmax/dt could be either slightly higher than an integer or slightly lower. Example: $1/0.01 \approx 100.00001$ or 99.99999. If it is slightly lower it will be rounded down towards the integer below it. This is also a common mistake. Correct way in Python is `int(round(tmax/dt))`.

TYPES OF NUMERICAL ERRORS OCCURING WITH A NUMERICAL METHOD

We distinguish two types of errors

1. *Truncation error* of the Taylor series for the method (Euler method: truncated after first order)
- Local error (error after one integration step)
- Global error (error after integrating till the end of the interval) 2. *Rounding error*

The smaller the error, the more accurate the result.

TRUNCATION ERROR EULER METHOD (DISCUSSED BEFORE)

In the Euler method we approximate $f(x + \Delta x)$ by

$$f(x + \Delta x)_{\text{approx}} \approx f(x) + \Delta x \cdot g(x, f(x))$$

this gives the error:

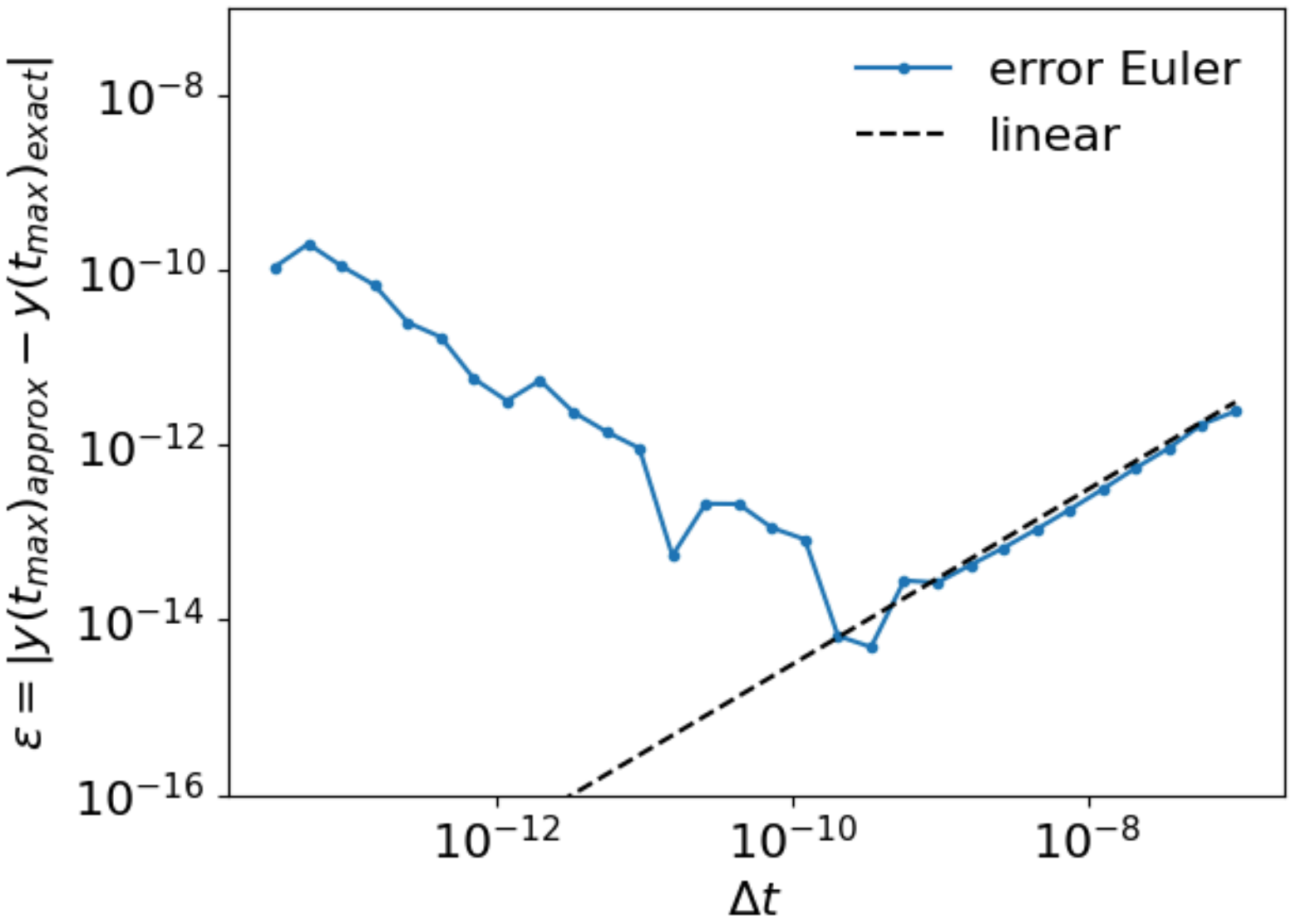
$$f(x + \Delta x)_{\text{approx}} - f(x + \Delta x)_{\text{exact}} = \Delta x^2 f''(x)/2! + O(\Delta x^3)$$

where we have used $g(x, f(x)) = f'(x)$.

ROUND-OFF ERRORS

The **round-off errors** arise because of *a finite representation of a floating point number* in a programming language. Round-off errors do not only occur for rounding numbers to their integer value, but also due to rounding a number to a limited fractional value. An example is rounding π to 3.1416. Then $\pi - 3.1416 \approx -7.346 \times 10^{-6}$.

ROUND-OFF ERROR ILLUSTRATED BY THE EXPLICIT EULER METHOD


error Euler
linear
10^{-8}
10^{-10}
10^{-12}
10^{-14}
10^{-16}
$\varepsilon = |y(t_{max})_{approx} - y(t_{max})_{exact}|$
10^{-12}
10^{-10}
10^{-8}
Δt

From $\Delta t \lesssim 10^{-9}$ the error does not decrease any more with smaller time-step. The reason: limited accuracy of the 64 bit floating point number is reached. For $\Delta t \approx 10^{-10}$ the error in the calculation $\epsilon \approx 10^{-13}$.

The error is much larger than the threshold for a `np.float64`, $\approx 10^{-16}$. This is caused by the *accumulation* of round-off errors due to the many additions of order N_{int}. Since N_{int} increases with smaller Δt, the *error will even increase* with smaller Δt.

However, this is only the case for very small Δt and in this module the main focus will be on truncation errors.

HIGHER ORDER INTEGRATION METHODS

Let us shorten the notation a bit: $t = t_i = i\Delta t$, and hence $y(t) = y(t_i) = y_i$. For the initial value problem ODE

$$\dot{y}(t) = g(t, y(t))$$

the (explicit) forward Euler method is then

$$y_{i+1} \approx y_i + \Delta t \cdot g(t_i, y_i)$$

and the (implicit) backward Euler method

$$y_{i+1} \approx y_i + \Delta t \cdot g(t_{i+1}, y_{i+1})$$

These are both first-order methods, since the global error is $O(\Delta t)$.

RUNGE KUTTA METHODS

Now we will consider *explicit* methods, but of higher order, of the form

$$\boxed{y_{i+1} \approx y_i + \Delta t \cdot \phi(t_i, y_i, \Delta t)} \qquad (*)$$

where ϕ is called the *increment function*.

Note that if $\phi(t_i, y_i, \Delta t) = g(t_i, y_i)$ we have again the simple Euler method.

For the higher order methods the increment function ϕ is not simply equal to g. **Runge-Kutta** methods can all be cast in the form of eq. $(*)$, with the simplest one being the forward Euler method.

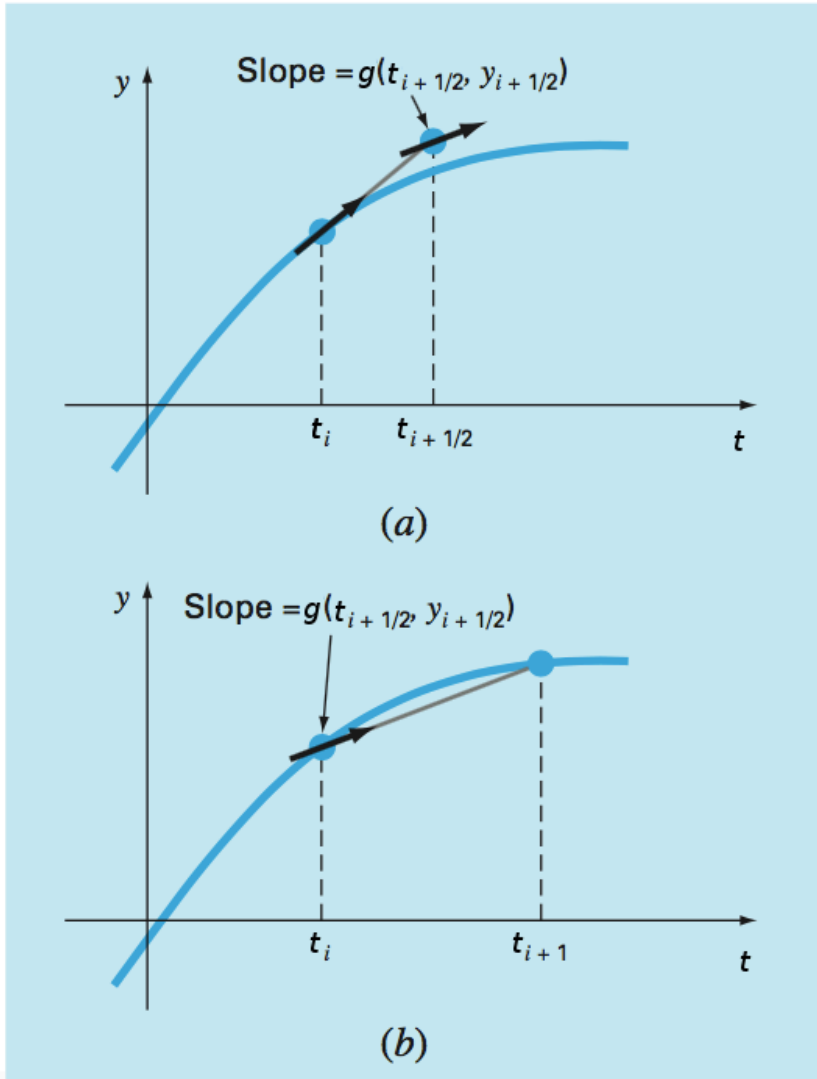
MIDPOINT METHOD

The **second order Runge Kutta midpoint method** for the ODE $\dot{y}(t) = g(t, y(t))$ is

$$\boxed{y_{i+1} \approx y_i + \Delta t \cdot g(t_i + \Delta t/2, y_i + g(t_i, y_i)\Delta t/2)}$$

Now the function g is evaluated at around half the time step in between t_i and t_{i+1}, where this half point is approximated by the derivative *at* t_i

One can show that this method is second order in the global truncation error (as opposed to Euler, which is first order)!



RALSTON'S METHOD

Instead of halfway, we can evaluate it at 2/3th of the interval. One can show that then the local truncation error has a minimum bound, and the resulting method is called **Ralston's method**

$$\boxed{y_{i+1} \approx y_i + \Delta t \cdot \left(\frac{1}{4} g(t_i, y_i) + \frac{3}{4} g(t_i + \frac{2}{3}\Delta t, y_i + \frac{2}{3} g(t_i, y_i)\Delta t) \right)}$$

IMPLEMENTATION RALSTON

In practise it is easier to introduce the intermediate variables k_1 and k_2

$$k_1 \equiv g(t_i, y_i)$$
$$k_2 \equiv g(t_i + \frac{2}{3}\Delta t, y_i + \frac{2}{3}\Delta t \cdot k_1)$$

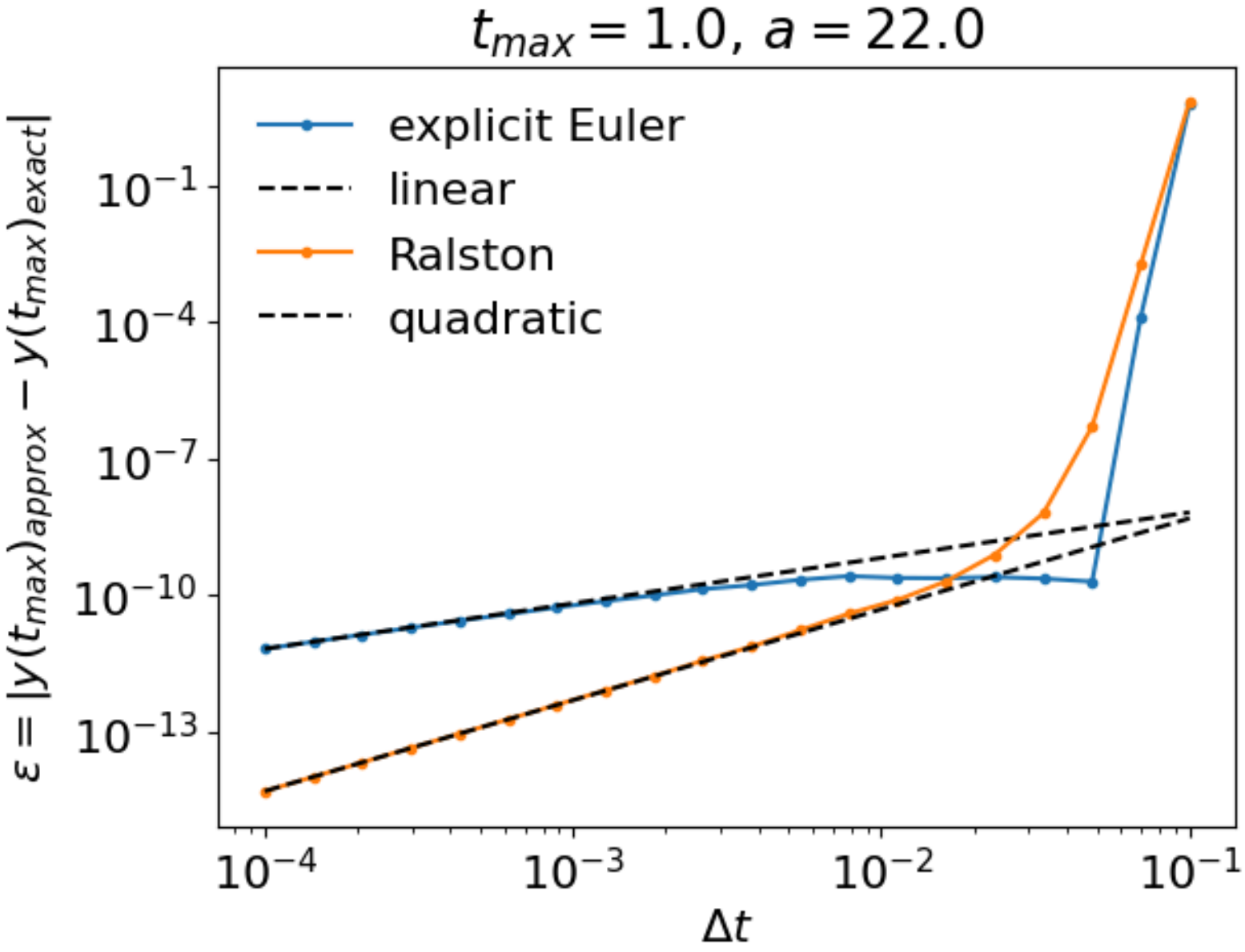
and write the algorithm as

$$y_{i+1} \approx y_i + \Delta t \cdot \left(\frac{1}{4} k_1 + \frac{3}{4} k_2 \right)$$

This also typically leads to less mistakes when implementing in a programming language.

COMPARISON METHODS

The following numerical integration confirms that Ralston's method is second order: (global) error decreases as $O(\Delta t^2)$. It is more accurate than Euler at small enough Δt (here $\Delta t \lesssim 0.02$).


$t_{max} = 1.0,\ a = 22.0$
explicit Euler
linear
Ralston
quadratic
$\varepsilon \equiv |y(t_{max})_{approx} - y(t_{max})_{exact}|$
10^{-1}
10^{-4}
10^{-7}
10^{-10}
10^{-13}
10^{-4}
10^{-3}
10^{-2}
10^{-1}
Δt

EXERCISES SESSION 2

1. Assume you solve a differential equation, $y'(t) = g(t, y(t))$, using the forward Euler method using a step size of $\Delta t = 0.01$. Assume the (global) error is exactly as in the linear regime, and the error in $y(t_{\max})$ is 0.04. What would the error be if $\Delta t = 0.005$?

What woul the global error be for $\Delta t = 0.005$ if Ralston is used instead (assuming an error of 0.03 for $\Delta t = 0.01$)?

2. The remaining exercises are on Blackboard, assessment section.