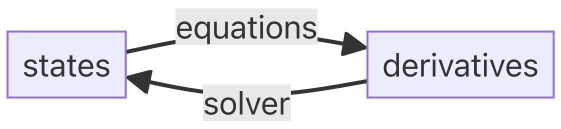
How does an ODE solver work - Simplified example using Forward Euler

Numerical dynamic simulation consists of determining the trajectory of variables over time. This is typically done as follow:

- the state variables are known,
- all other variables are computed based on these - including the state derivatives, and
- the state derivatives are integrated by the solver, to find the value of the state variables at the next time step.



Therefore, understanding what a solver does is key to understanding numerical simulation.

However, solvers is not a simple topic and there are books dedicated to the topic, e.g. Continuous System Simulation from François E. Cellier. Solvers come with many challenges, such as stability and accuracy.

From the many Ordinary Differential Equation (ODE) solvers, the simplest one is the Forward Euler method. It is also very easily unstable - its stability region is very narrow. However, its simplicity makes it very convenient to explain the concept of solvers. Therefore, the Forward Euler method is used here.

Importing relevant python modules

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

Forward Euler method

Many integration methods rely on the **Taylor expansion**, that can be expressed as:

$$f(t) = f(t_0) + f'(t_0)(t-t_0) + \frac{f''(t_0)}{2!}(t-t_0)^2 + \frac{f'''(t_0)}{3!}(t-t_0)^3 + \cdots = \sum_{n=0}^{\infty} \frac{f^{(n)}(t_0)}{n!}(t-t_0)^n$$

Rewritten for the purpose of integration, it becomes:

$$y_{n+1} = y_n + \dot{y}_n.h + \ddot{y}_n.\frac{h^2}{2!} + \dddot{y}_n.\frac{h^3}{3!} + \ldots$$

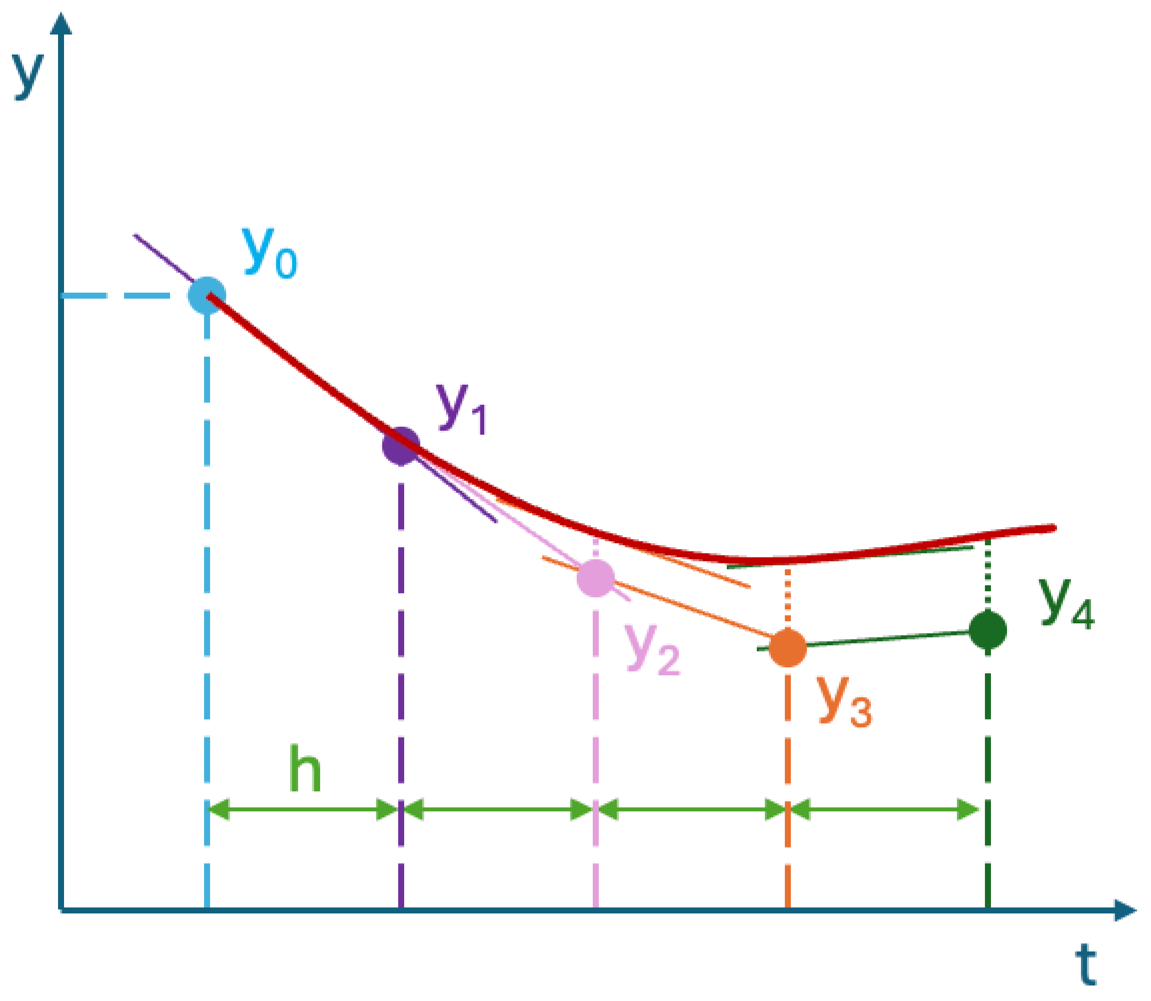
Where:

- y_n is the value of a state variable at the current time
- y_{n+1} is the value of the same state at the next time step
- h is the step size, between the two points in time
- The dot notation is used to represent the derivatives of state variable.

The Forward Euler method consists in truncating the Taylor expansion after the first derivative:

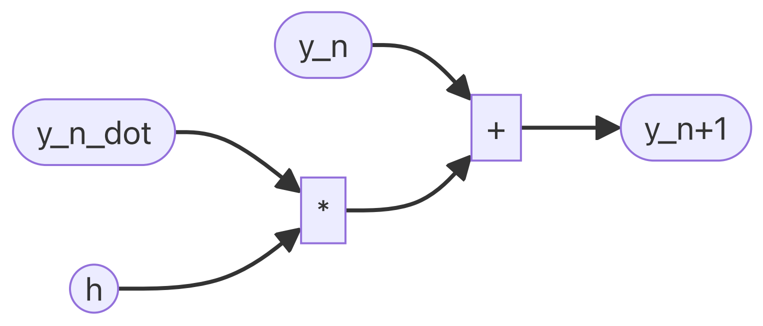
$$y_{n+1} = y_n + \dot{y}_n.h$$

What does this mean? It means that, at a given point, the derivative is computed and is held for the period of the time step.



And this is quite convenient to use as, typically, the numerical model of a dynamic system provides the relationship between the state variables and their derivatives. Accordingly, knowing our model and the values of the states at one point in time, it is possible to compute the values of the states at the next point in time as follow:

Forward Euler integration method



For this purpose, we decided to wrap the Forward Euler method into a python function. Below is a very simple implementation of this method - for one variable only and without any "safety guard" (such as error handling of case like "is the step size positive?").

In [2]:
```python
def forward_Euler(y_n, y_n_dot, step_size):
    """
        Takes values of the state y_n and its derivative y_n_dot on a given time step and the step size, and
        returns the value of the state at the next time step y_n1.
    """
    y_n1 = y_n + y_n_dot*step_size # y_n1 stands for y_{n+1}, which cannot be used as variable name
    return y_n1
```

Using Forward Euler integration method with a simple model

Now the integration method is available, let's try it on a simple model. Consider the following single Ordinary Differential Equation:

$$\dot{y}_n = 2 * \pi * cos(2 * \pi * t)$$

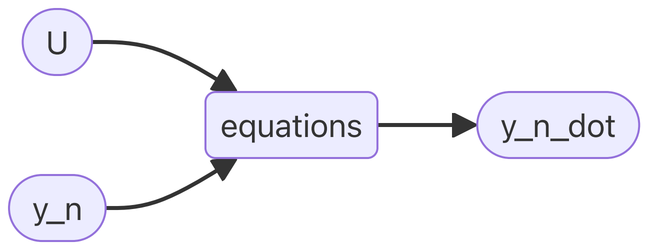
with

$$y_0 = 0$$

This is a simple ODE that can be analytically solved. Indeed, the solution is $y_n = sin(2 * \pi * t)$. However, for the sake of the example, this ODE is numerically solved and the result is compared with the analytical solution.

The ODE is our system model and allows us to compute the derivative $\dot{y}_n$ as a function of the state y_n and time - though for this simple example, only time. Sometimes, the ODE can also be function of the input vector U - not in this case. Note also that the state here is a scalar while it could be a vector of states.

From state to derivative



The below python function returns the derivative $\dot{y}_n$ as a function of the state y_n.

In [3]:
```python
def system_model(y_n, time):
    """
        Takes the state variable and time as input, and
        returns the derivative on the same time step.
    """
    y_n_dot = 2*np.pi*np.cos(2*np.pi*time) # to simplify, there is no dependency on the state for this example
    return y_n_dot
```

To be able to simulate this example, a few more information is needed, namely:

- the initial condition,
- the initial and final time of the simulation, and
- the step size of the integration

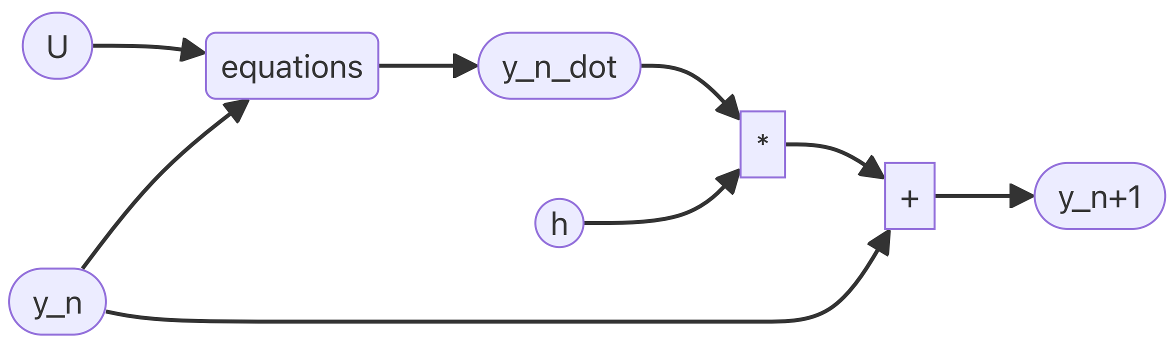
In [4]:
```python
# Define initial condition
y_0 = 1

# Define duration of the simulation and step size
t_init = 0 # seconds
t_final = 1 # seconds
step_sizes = [0.5, 0.1, 0.05, 0.01, 0.005] # List to iterate over and compare
```

Now, it is time to define a simulation function. Again, this is done in its simplest form and no error handling is implemented. (It is for example made sure that the simulation duration $t_{final} - t_{init}$ is a multiple of $step\_size$, resulting in a the last point also being computed.) At each time step, the current time and value of the state are stored in a corresponding arrays. And at the end of the simulation, these are stored in a `results` dictionary as values, where the key is the `step_size`. This is done to later plot the results as a function of the `step_size`.

In [5]:
```python
def run_simulation(t_init, t_final, y_0, step_size, results):
    # Storing results
    time = []
    y = []

    # Initialize variables
    current_time = t_init
    y_n = y_0

    # Time solving
    while current_time <= t_final:
        # Store variables at time step
        time.append(current_time)
        y.append(y_n)
        # Compute derivative
        y_n_dot = system_model(y_n, current_time)
        # Solver to compute next time step value of the state
        y_n = forward_Euler(y_n, y_n_dot, step_size)
        # increase current time by a time step
        current_time += step_size

    # Store results in dictionary
    results[step_size] = (time, y)
```

All the pieces are in place. The complete simulation looks like this (without any input U though):

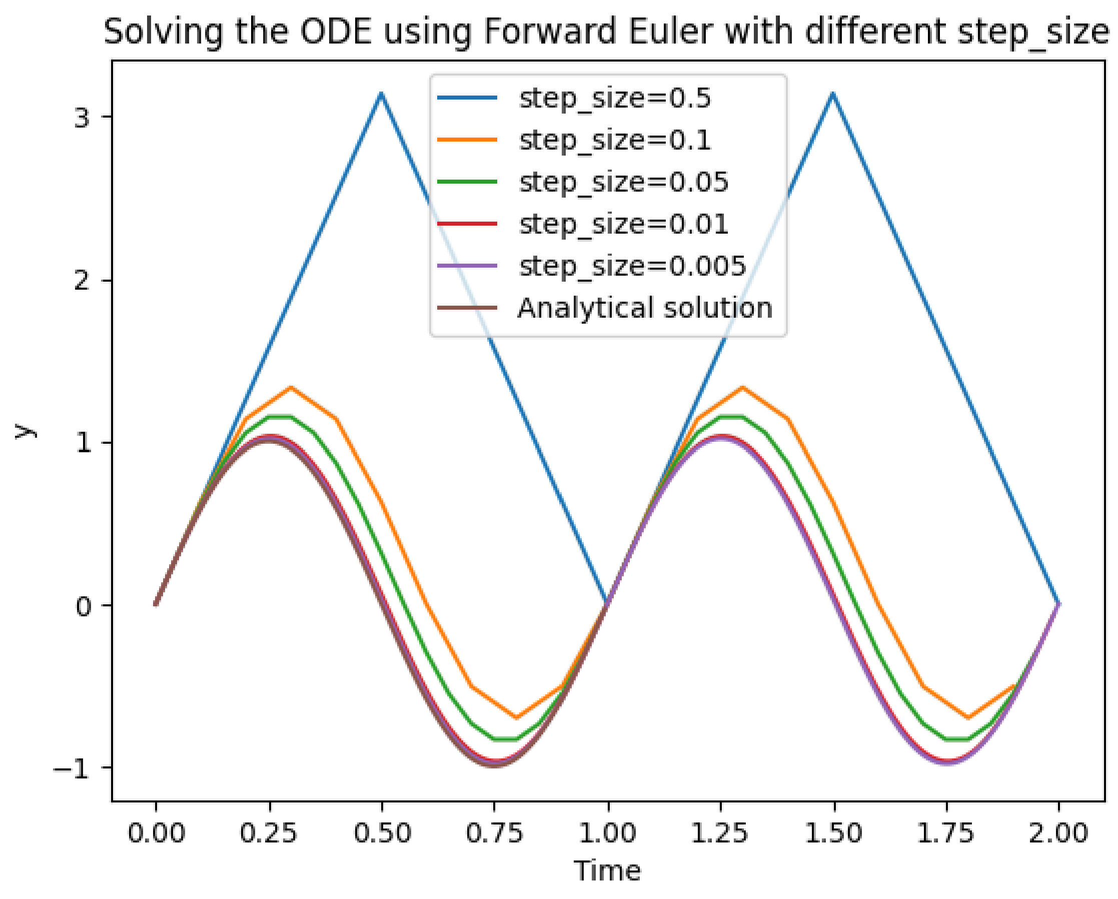
Graphical reprensentation of the Forward Euler integration method



To demostrate the importance of the `step_size` on the numerical integration, the simulation will be ran 5 times with different values of `step_size`, ranging from 0.5 seconds to 5 miliseconds.

The analytical solution is also added to the plot as a comparison.

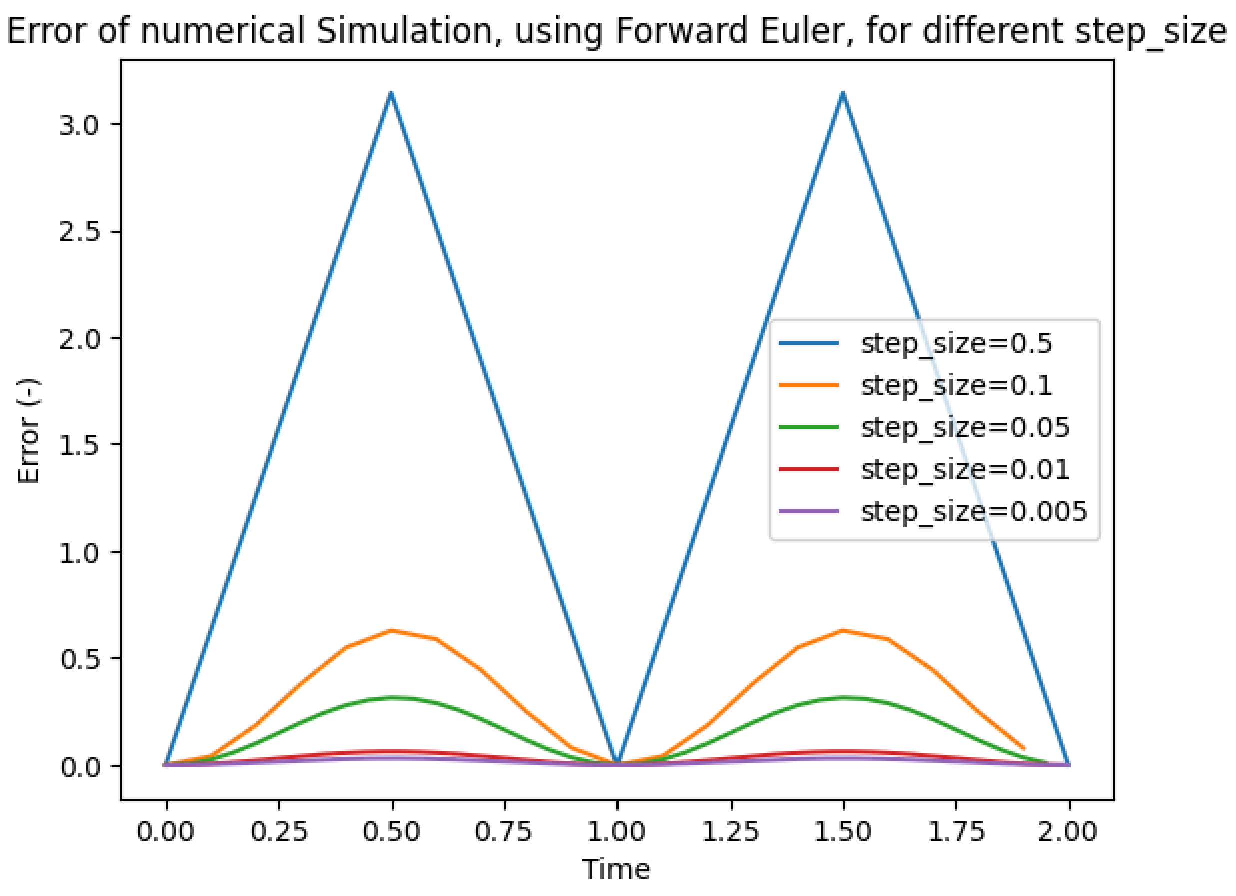
In [6]:
```
# Simulation parameters
t_init = 0
t_final = 2
y_0 = 0

# List of step sizes
step_sizes = [0.5, 0.1, 0.05, 0.01, 0.005]

results = {}
# Run simulations
for step_size in step_sizes:
    run_simulation(t_init, t_final, y_0, step_size, results)

# Plot results
for step_size, (time, y) in results.items():
    plt.plot(time, y, label=f'step_size={step_size}')

# Adding the analytical solution
time = np.arange(0, 1, 0.005)
plt.plot(time, np.sin(2*np.pi*time), label='Analytical solution')

# Show plot
plt.xlabel('Time')
plt.ylabel('y')
plt.legend()
plt.title('Solving the ODE using Forward Euler with different step_size')
plt.show()
```



As it can be seen, the results are very different based on the selected time step (`step_size`). Taking too large steps leads to large errors (compared to the analytical solution.

The following plot illustrate this:

In [7]:
```
# Calculate and plot errors
for step_size, (time, y) in results.items():
    analytical_solution = np.sin(2*np.pi*np.array(time))
    error = np.abs((analytical_solution - np.array(y))) # we do not divide by the reference to avoid division by ze
    plt.plot(time, error, label=f'step_size={step_size}')

# Show error plot
plt.xlabel('Time')
plt.ylabel('Error (-)')
plt.legend()
plt.title('Error of numerical Simulation, using Forward Euler, for different step_size')
plt.show()
```



As it has been shown, reducing the step size can lead to more accuracy. It also leads to more points to be computated - hence a slower simulation.

The time step is here fixed. Many solvers, however, can adapt the size of the time step dynamically during simulation to improve accuracy.

Another way to increase accuracy consists of truncating the Taylor expansion at a higher order.

And as said as the beginning, integration method is a complex topic:

- Some solvers are designed as predictor-corrector, so that they actually first compute a first estimate of the integration and then correct it to a more accurate value. And there are many more ways of doing...
- Each solver is only stable in a defined region and shall not be used outside of this region.
- Many solvers are made for a given type of problem (e.g. the problem stiffness).

Better solver implementations

It has been emphasized several time that this notebook does not pretend to implement the Forward Euler solver in either a complete nor robust manner. Instead, great packages are already available for this. To list only two:

- Sundials
- DifferentialEquations.jl

Acknowledgement

Many thanks to Michael Tiller for providing early feedback to this notebook.