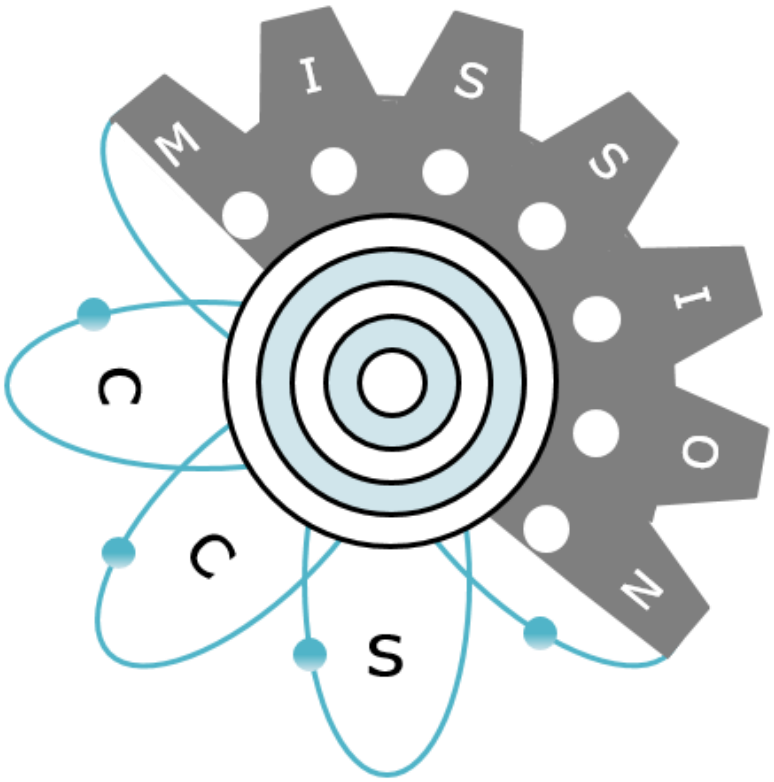


# Numerical Modelling using Python

**Harvey Thompson**

# 1. Initial value problems

The first part of the module will introduce you to the most popular numerical methods used to solve initial value problems (IVP). IVPs consist of a differential equation that describes how some quantity changes over time (see examples below) and a given initial value.

## 1.1 Examples

### 1.1.1 Carbon dating

Carbon dating relies on the radioactive decay of 14N, a radioactive carbon isotope. The method was invented by Willard F. Libby in 1949 and in 1960 he was awarded a Nobel prize for his work. The decay (i.e. *the rate of change with time* of the number of atoms) is proportional to the number of $C_{14}$ atoms in the sample

$$\dot{C_{14}} = -\lambda\, C_{14} \quad (1.1)$$

Throughout these notes, a dot over a quantity means a derivative with respect to time, that is

$$\dot{C_{14}} = \frac{\partial C_{14}}{\partial t} \quad (1.2)$$

and

$$\ddot{C_{14}} = \frac{\partial^2 C_{14}}{\partial t^2} \quad (1.3)$$

In return, one $N_{14}$ atom is generated out of each $C_{14}$ atom which decays. Therefore,

$$\dot{N_{14}} = \lambda\, C_{14} \quad (1.4)$$

The decay constant λ is related to the half-life τ via

$$\tau = \frac{log_e(2)}{\lambda} \quad (1.5)$$

that is the shorter the half-life, the greater the decay constant. The half-life of $C_{14}$ is τ =5730 years so that λ ≈ 0.000121. Equation (1.1) is probably the most simple example of an ordinary differential equation (ordinary means it only contains derivatives with respect to one variable, time t in this example) and we can simply guess its solution

$$C_{14}(t) = C_{14}(0)e^{-\lambda t} \quad (1.6)$$

**Remark 1**

*The model can obviously lead to fractions of atoms being present. How does this limit the applicability of the decay model?*

Note that we need an *initial value* $C_{14}(0)$ to find the solution at time t.

**Carbon dating**

Carbon dating relies on the fact in living plants and animals, $C_{14}$ atoms are constantly replenished, keeping the ratio between radioactive $C_{14}$ and non-radioactive $C_{12}$ roughly constant. After a plant or animal dies, this process of replenishment stops and decaying $C_{14}$ atoms are no longer replaced, slowly changing the ratio over time. We denote the ratio between $C_{14}$ and $C_{12}$ atoms as r(t) and set t=0 as the time the plant/animal died - at this time, the ratio is equal to the atmospheric ratio which is roughly

$$r(0) = \frac{C_{14}(0)}{C_{12}(0)} \approx \frac{1.5}{10^{12}} \qquad (1.7)$$

Then, while $C_{12}$ does not change, $C_{14}$ decreases according to Equation (1.6) so that

$$r(t) = \frac{C_{14}(t)}{C_{12}(t)} = \frac{C_{14}(0)}{C_{12}(0)} e^{-\lambda t} \qquad (1.8)$$

So after we determine the ratio r(t) in e.g. an archaeological sample, we can compute its age t. Let us say we find a ratio of

$$r(t) = \frac{C_{14}(t)}{C_{12}(t)} = 0.5 \times 10^{-12} \qquad (1.9)$$

then we can solve for the age t by

$$t = -\frac{1}{\lambda} log_e \left(\frac{r(t)}{r(0)}\right) \approx \frac{-1}{0.000121} log_e \left(\frac{1}{3}\right) \approx 9047 \text{ years} \qquad (1.10)$$

**Self-study**. Write a simple python function that takes a measured ratio r(t) of $C_{14}$ to $C_{12}$ and returns the time t.

**1.1.2 Chemical batch reactor**

Consider a chemical reactor where two species A and B react to form a species C.
The reaction is

$$A + B \rightarrow nC \qquad (1.11)$$

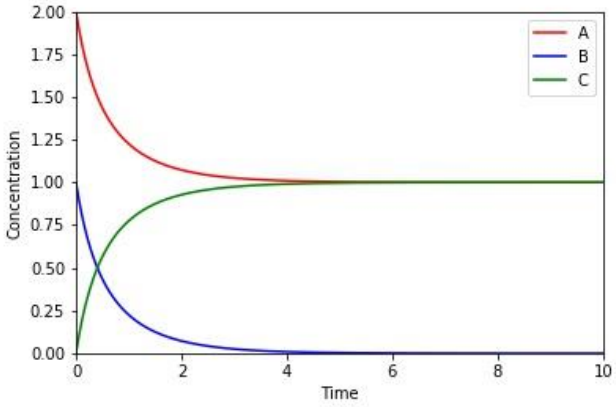


Figure 1.1: Solution of (1.12) with Python's odeint solver in **simple_reaction.py**.

with a reaction rate constant k. In terms of concentrations $C_A$, $C_B$ and $C_C$ this reaction can be modelled as

$$\dot{C_A} = -kC_AC_B \quad (1.12a)$$
$$\dot{C_B} = -kC_AC_B \quad (1.12b)$$
$$\dot{C_C} = kC_AC_B \quad (1.12c)$$

Note that $\dot{C_A} + \dot{C_C} = 0$ and $\dot{C_B} + \dot{C_C} = 0$ which means that

$$C_A(t) + C_C(t) = C_A(0) + C_C(0), C_B(t) + C_C(t) = C_B(0) + C_C(0) \quad (1.13)$$

corresponding to the fact that the total number of atoms stays the same -- mass is conserved.

**Remark 2**
Obviously, concentrations have to remain positive to make any sense. Not all numerical methods guarantee this, however.

**Self-Study**
Figure 1.1 shows a solution of (1.12) for $C_A(0)$=2, $C_B$=1.0 and $C_C$=0.0 computed with Python's **odeint** solver in the program **simple_reaction.py**. This code is given below:

**simple_reaction.py**

```
"""
Solves an initial value problem modelling a simple chemical reaction
 A + B --> C, with rate constant k

"""
import numpy as np
from scipy.integrate import odeint

# define the rhs function, f
def f(u,t,rate_constant):
    return [-rate_constant*u[0]*u[1],-rate_constant*u[0]*u[1],rate_constant*u[0]*u[1]]

rate_constant = 1.0
tend = 10.0
```

```
A_0 = 2.0
B_0 = 1.0
C_0 = 0.0

u0 = [A_0,B_0,C_0]
t = np.linspace(0,tend,1000)
u = odeint(f,u0,t,args=(rate_constant,))

A = u[:,0]
B = u[:,1]
C = u[:,2]

# plot out results
import matplotlib.pyplot as plt
plt.plot(t,A,'r')
plt.plot(t,B,'b')
plt.plot(t,C,'g')
plt.xlim([0,10])
plt.ylim([0,2])
plt.xlabel('Time')
plt.ylabel('Concentration')
plt.legend(['A','B','C'])
plt.savefig('simple_reaction.jpg')
```

Does the solution make sense to you? Modify it to plot $C_A(t)+C_C(t)$ and $C_B(t)+C_C(t)$ over time. What would you expect to see and is this what happens?

### 1.1.3 Non-Hookean mass-spring system

We consider a mass-spring system where a bob of mass m is attached to a spring with a spring constant k. However, instead of linear Hooke's law, we use Duffing's model where the restoring force is

$$F_{resto} = -k(x(t) + \beta x^3(t)) \qquad (1.14)$$

where x(t) is the displacement of the bob from its equilibrium and β some model parameter. Friction is assumed to be linear and modelled via

$$F_{friction} = -b\dot{x}(t) \qquad (1.15)$$

Allowing for an external force $F_{ext}(t)$, Newton's second law then reads

$$m\ddot{x}(t) = -k\big(x(t) + \beta x^3(t)\big) - b\dot{x}(t) + F_{ext}(t) \qquad (1.16)$$

We can introduce velocity $v = \dot{x}(t)$ as a variable to eliminate the second order derivative

$$\dot{x}(t) = v(t);\ \dot{v}(t) = -\frac{k}{m}\big(x(t) + \beta x^3(t)\big) - \frac{b}{m}v(t) + \frac{1}{m}F_{ext}(t) \qquad (1.17a, 1.17b)$$

For β=0 and $F_{ext}$=0, we recover the equations for unforced linear harmonic motion

$$\dot{x}(t) = v(t);\ \dot{v}(t) = -\frac{k}{m}x(t) - \frac{b}{m}v(t) \qquad (1.18a, 1.18b)$$

or, eliminating v(t) again,

$$m\ddot{x}(t) + b\dot{x}(t) + kx(t) = 0 \qquad (1.19)$$

For the case with b=0, we can derive the exact solution by making an educated guess[1]. Let

$$x(t) = Asin(\alpha t) + Bcos(\alpha t) \qquad (1.20)$$

for some parameters A, B and α. First, compute

$$\dot{x}(t) = v(t) = A\alpha \cos(\alpha t) - B\alpha \sin(\alpha t) \qquad (1.21)$$

and then

$$\dot{v}(t) = -A\alpha^2 \sin(\alpha t) - B\alpha^2 \cos(\alpha t) = -\frac{k}{m}x(t) = -\frac{k}{m}A \sin(\alpha t) - \frac{k}{m} B \cos(\alpha t) \qquad (1.22)$$

From that we can conclude that

$$\alpha^2 = \frac{k}{m} \text{ or } \alpha = \sqrt{\frac{k}{m}} \qquad (1.23)$$

We still need to fix parameters A and B. Let $x_0$ and $v_0$ be the initial values given at t=0. Then,

$$x(0) = B = x_0 \qquad (1.24a)$$

$$v(0) = A\alpha = v_0 \qquad (1.24b)$$

so that $B=x_0$ and $A=v_0/\alpha$ fixes the parameters. Figure 1.2 shows the exact and numerical solution with **odeint** to the linearised problem (1.18) and the numerical solution to the nonlinear pendulum (1.17). This is obtained by running the following program, **springmass.py:**

```
"""
springmass.py
"""
import numpy as np
from scipy.integrate import odeint

# define the rhs function, f
def f(u,t,k,m):
    return [u[1],-(k/m)*u[0]]

# define the rhs function, fnonlinear
def f_nonlinear(u,t,k,m,beta):
    return [u[1],-(k/m)*(u[0] + beta*(u[0]**3))]

T = 10.0    # final time until which we compute
N = 100     # number of time steps
taxis = np.linspace(0,T,N+1)
k = 5.0
beta = 0.1
m = 1.0

# compute frequency alpha for linear (Hookean) solution
alpha = np.sqrt(k/m)

# initial values for position and velocity
x0 = 1.0
v0 = 0.0
```

[1] The case b>0 allows a similar derivation using the complex exponential function.

```
u0 = [x0,v0]

# compute parameter A and B of exact linear pendulum
B = x0
A = v0/alpha

# compute exact solution for linear pendulum
xexact = A*np.sin(alpha*taxis) + B*np.cos(alpha*taxis)

# use odeint to solve
t = np.linspace(0,T,N+1)
u = odeint(f,u0,t,args=(k,m,))
u_nonlinear = odeint(f_nonlinear,u0,t,args=(k,m,beta,))

# plot out results
import matplotlib.pyplot as plt
plt.plot(t,u[:,0],'bo')
plt.plot(t,xexact,'k-')
plt.plot(t,u_nonlinear[:,0],'rx-')
plt.xlim([0,10])
plt.ylim([-1.1,1.1])
plt.xlabel('Time')
plt.ylabel(r'$\theta$')
plt.legend(['odeint','Exact linear','odeint nonlinear'])
plt.savefig('springmass.jpg')
```

**Question.**

Figure 1.2 below shows a noticeable difference between the linear and nonlinear system. Does that make the linear model wrong? Changing which parameters would improve agreement between linear and nonlinear model? Run **springmass.py** to explore this.

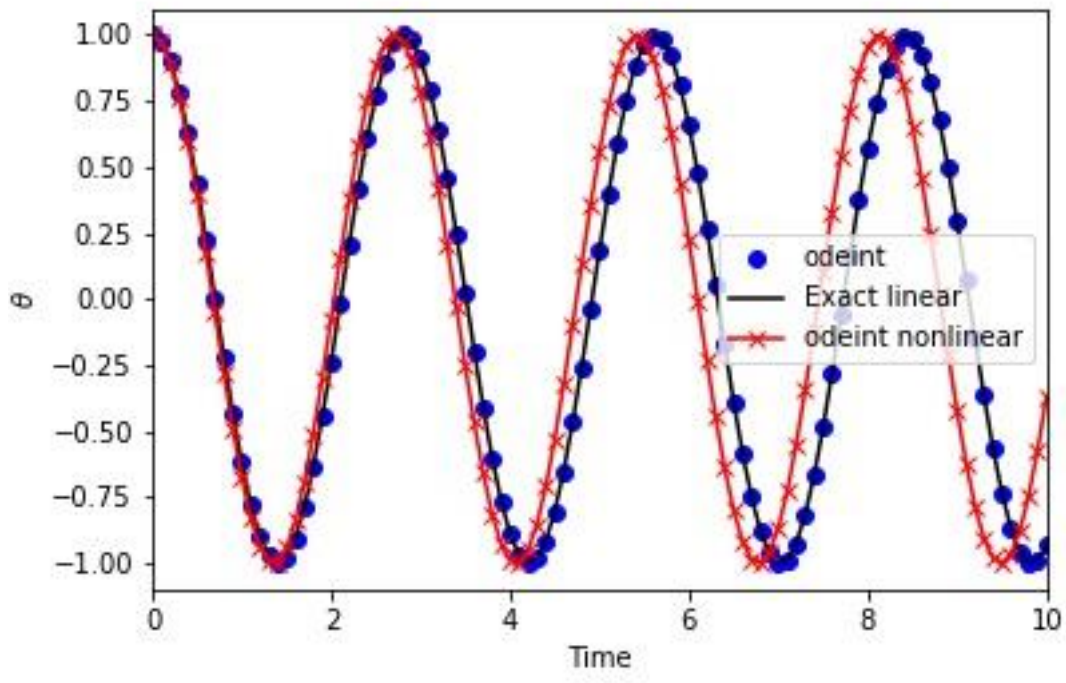


Figure 1.2 Exact (black) and numerical (blue) solution of the Hookean spring-mass system as well as numerical solution of the nonlinear Non-Hookean system (red), obtained using **springmass.py.**

## General initial value problems: notation

We can introduce a general compact way to write down initial value problems

$$\dot{\mathbf{u}}(\mathrm{t}) = \mathbf{f}(\mathbf{u}(\mathrm{t}), \mathrm{t}) \qquad (1.25)$$

where **u**(t) is a time-dependent vector containing all components of the system while **f** is what is referred to as the *right-hand-side function*. For Equation (1.1) we would have

$$\boldsymbol{u}(t) = (C_{14}(t)) \qquad (1.26)$$

and

$$\mathbf{f}\big(\mathbf{u}(\mathbf{t})\big) = \mathbf{f}\big(C_{14}(t)\big) = -\lambda C_{14}(t) \qquad (1.27)$$

Note that in this case **u** is not really a vector because it has only a single component. This changes for the chemical reaction given by Equation (1.11). Here,

$$\mathbf{u}(\mathrm{t}) = \begin{pmatrix} C_A(t) \\ C_B(t) \\ C_C(t) \end{pmatrix} \qquad (1.28)$$

while **f** is a function that takes a vector **u** as argument and returns the vector

$$\mathbf{f}\big(\mathbf{u}(\mathrm{t})\big) = \mathbf{f}\begin{pmatrix} C_A(t) \\ C_B(t) \\ C_C(t) \end{pmatrix} = \begin{pmatrix} -k\, C_A(t) C_B(t) \\ -k\, C_A(t) C_B(t) \\ k\, C_A(t) C_B(t) \end{pmatrix} \qquad (1.29)$$

Finally, for the mass-spring system given by Equation (1.17), we have

$$\mathbf{u}(\mathrm{t}) = \begin{pmatrix} x(t) \\ v(t) \end{pmatrix} \qquad (1.30)$$

and

$$\mathbf{f}\big(\mathbf{u}(\mathrm{t})\big) = \begin{pmatrix} v(t) \\ -\frac{k}{m}x(t) - \frac{k}{m}\beta x^3(t) - \frac{b}{m}v(t) + \frac{1}{m}F_{ext}(t) \end{pmatrix} \qquad (1.31)$$

While this notation may seem abstract at first, it will allow us to write down numerical methods more generally, using the same notation independent of the problem we will apply them to later.

## Forward and backward Euler method

Most initial value problems will be too complex to be solved by hand. Therefore, we use numerical methods to compute approximate solutions. Consider now the IVP in generic form

$$\dot{\boldsymbol{u}}(\boldsymbol{t}) = \mathbf{f}(\mathbf{u}(\mathrm{t}), \mathrm{t}), \quad \mathbf{u}(0) = \boldsymbol{u_0} \qquad (1.32)$$

Instead of trying to find a continuous function **u**(t) that solves the differential equation at every time t in some interval [0, T], a numerical method will construct a finite number of approximate values at selected points in time: $0 = t_0 < t_1 < t_2 < \ldots < t_N = T$. Here, N is the number of steps we use and, for the sake of simplicity, we will assume that the distance between two points is always the same, that is

$$\Delta t = t_{n+1} - t_n \qquad (1.33)$$

for any n = 0, …, N-1. We will denote approximations at a point in time $t_n$ delivered by a numerical method with a superscript n, that is $\mathbf{u}^n$ is an approximation of the exact solution $\mathbf{u}(t_n)$. Further, we will denote the error made by this approximation as

$$e^n = \|\mathbf{u}^n - \boldsymbol{u}(t_n)\| \qquad (1.34)$$

To compute the error, we need to know the exact solution **u**(t) so this will only be possible when testing our algorithms for simple problems. Finally, we define the global error as the largest value of $e_n$ over all time steps, that is

$$e_{global} = \max_{0 \le n \le N} e^n \qquad (1.35)$$

With this notation, we can now write down the forward or explicit Euler algorithm for the generic IVP (1.32) as

$$\boldsymbol{u}^{n+1} = \boldsymbol{u}^n + \Delta t\, \mathbf{f}(\boldsymbol{u}^n, t_n) \qquad (1.36)$$

Note how starting from the initial value $\mathbf{u_0}$ this rule iteratively generates a series of approximate solutions $\mathbf{u^1}$, $\mathbf{u^2}$,… The method is called *explicit* because computing $\mathbf{u^{n+1}}$ from $\mathbf{u^n}$ requires only evaluating $\mathbf{f}(\mathbf{u^n}, t_n)$ but not solving any linear or nonlinear equations. It also often called the *forward Euler* method. This is because it can be derived by using a (forward) Taylor expansion of the solution **u**(t) around $t=t_n$

$$\boldsymbol{u}(t_{n+1}) = \boldsymbol{u}(t_n) + (t_{n+1} - t_n)\dot{\boldsymbol{u}}(t_n) + \frac{(t_{n+1}-t_n)^2}{2}\ddot{\boldsymbol{u}}(t_n) + \cdots \qquad (1.37)$$

We ignore all terms except the first two on the right hand side and use that $t_{n+1}$-$t_n$ = Δt and that, because **u** solves the differential equation, $\dot{\boldsymbol{u}}(t_n) = \boldsymbol{f}(\boldsymbol{u}(t_n), t_n)$ to get

$$\boldsymbol{u}(t_{n+1}) \approx \boldsymbol{u}(t_n) + \Delta t\, \boldsymbol{f}(\boldsymbol{u}(t_n), t_n) \qquad (1.38)$$

Starting this procedure from $\mathbf{u}(0)=\mathbf{u}_0$ and naming the resulting approximations $\mathbf{u}_n$ results in the forward Euler method (1.36).

The other type of Euler method is backward or implicit Euler

$$\boldsymbol{u}^{n+1} = \boldsymbol{u}^n + \Delta t\, \boldsymbol{f}(\boldsymbol{u}^{n+1}, t_{n+1}). \qquad (1.39)$$

Here, computing $\mathbf{u}^{n+1}$ from $\mathbf{u}^n$ requires the solution of

$$\boldsymbol{u}^{n+1} - \Delta t\, \boldsymbol{f}(\boldsymbol{u}^{n+1}, t_{n+1}) = \boldsymbol{u}^n \qquad (1.40)$$

which, depending on **f**, can be a linear or nonlinear problem. Note that for the radioactive decay equation (1.1) (where $\mathbf{u}=C_{14}$ and $\mathbf{f(u)}= -\lambda C_{14}$) this becomes

$$C_{14}^{n+1} + \Delta t \lambda C_{14}^{n+1} = C_{14}^{n} \qquad (1.41)$$

which can easily be solved to give

$$C_{14}^{n+1} = \frac{C_{14}^{n}}{1+\Delta t \lambda} \qquad (1.42)$$

However, solving Equation (1.40) by hand is only possible for very simple problems. In most cases you will have to use some numerical procedure for that. Python offers the **numpy.linalg** function for linear problems and the **scipy.optimize fsolve** command for nonlinear problems. We will see examples later.

Figure 1.3 shows the approximate solution obtained with forward and backward Euler as well as the exact solution in black, for the case with λ=2 and N=10 time steps. The code used in **carbon_euler.py** given below:

```
"""
carbon_euler.py
"""
import numpy as np
T = 0.5    # final time until which we compute
N = 10     # number of time steps
taxis = np.linspace(0,T,N+1)
dt = T/N    # length of each time step
lam = 2.0    # decay constant
r0 = 1.0    # ratio at t=0 to 1.0

# preallocate two arrays to store all values computed with explicit and
# implicit Euler - this will save a bit of time compared to appending a
# value in each step. The 1 is because the number of components in case of
# the decay equations is one.
rexp = np.zeros(N+1)
rimp = np.zeros(N+1)

# first entry is the initial value r0 for both
rexp[0] = r0
rimp[0] = r0

# forward euler
for i in range(N):
    rexp[i+1] = rexp[i] -dt*lam*rexp[i]

# backward euler
for i in range(N):
    rimp[i+1] = rimp[i]/(1 + dt*lam)

# plot out results
import matplotlib.pyplot as plt
plot1 = plt.figure(1)
plt.plot(taxis,rexp,'ro')
plt.plot(taxis,rimp,'bo')
plt.plot(taxis,r0*np.exp(-lam*taxis),'k-')
plt.xlim([0,0.5])
plt.ylim([0.3,1.0])
```

```
plt.xlabel('Time')
plt.ylabel('C14')
plt.legend(['Explicit Euler','Implicit Euler','Exact solution'])
plt.savefig('carbon_euler.jpg')

# to compute the error, we have to figure out the exact solutions at all
# time points - note that taxis is a row vector whereas rexp, rimp are
# column vectors, so we need to transpose taxis to match
rexact = np.transpose(r0*np.exp(-lam*taxis))

# now compute error of forward and backward Euler at each step
errorexp = np.abs(rexp-rexact)
errorimp = np.abs(rimp-rexact)
plot2 = plt.figure(2)
plt.semilogy(taxis,errorexp,'r')
plt.semilogy(taxis,errorimp,'b')
plt.xlim([0.05,0.5])
plt.xlabel('Time')
plt.ylabel('Error')
plt.legend(['Explicit Euler','Implicit Euler'])
plt.savefig('carbon_euler2.jpg')
```

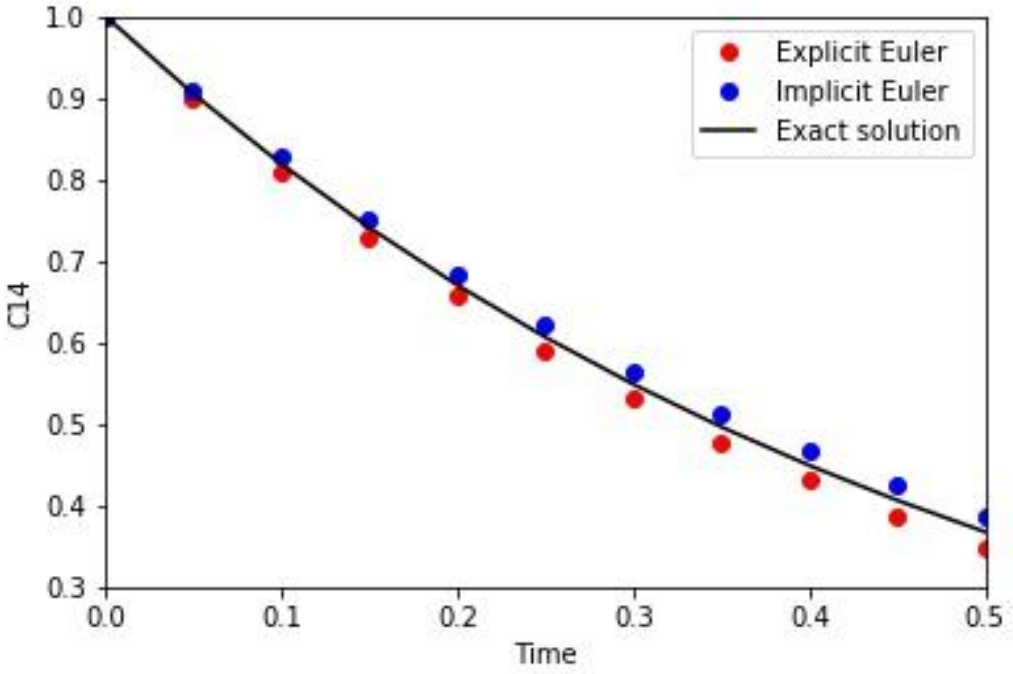


Figure 1.3: Approximations to the exact solution (black) of the radioactive decay equation computed with explicit Euler (red) and implicit Euler (blue) for λ=2 and N=10 time steps, obtained by running **carbon_euler.py**.

Both methods seem to do a reasonably good job in approximating the real solution. This changes drastically if we increase the decay rate λ. Figure 1.4 shows the same experiment but now for a value of λ=50 (note the different scaling of the y-axis). Running **carbon_euler.py** shows that Backward Euler still seems to provide a reasonable approximation, even though it is very difficult to tell given the scaling of the figure.

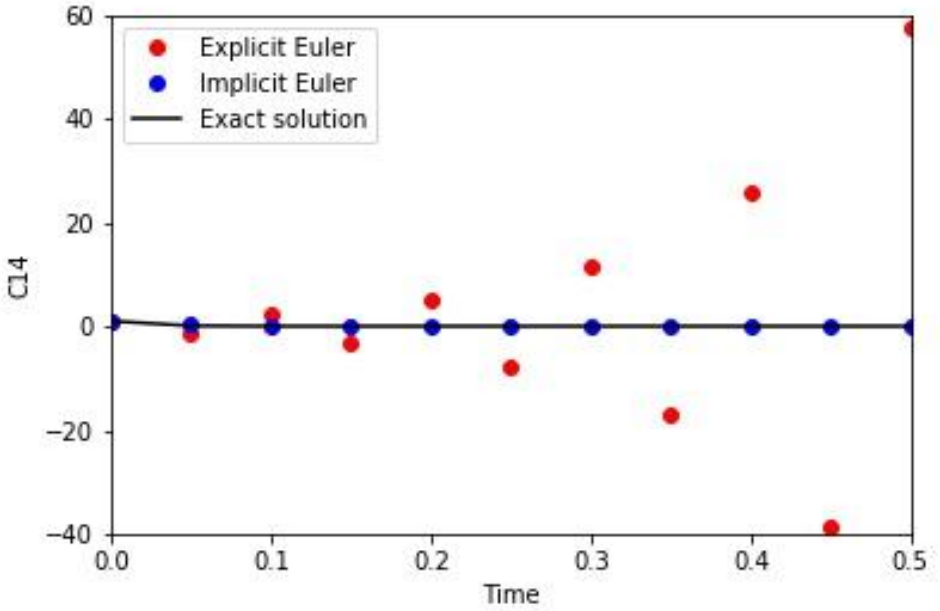


Figure 1.4: Approximations to the exact solution (black) of the radioactive decay equation computed with explicit Euler (red) and implicit Euler (blue) for λ=50 and N=10 time steps.

Forward Euler, however, clearly gives a very wrong result: instead of decaying, the number of $C_{14}$ atoms increases over time. It also oscillates between increasingly large positive and negative values. The provided numerical solution is clearly completely useless.

## Euler methods for the mass-spring system

If we apply forward Euler to (1.17) with m=1, b=0 and no external force we get

$$\begin{pmatrix} x^{n+1} \\ v^{n+1} \end{pmatrix} = \begin{pmatrix} x^n \\ v^n \end{pmatrix} + \Delta t\, \boldsymbol{f}\left(\begin{pmatrix} x^n \\ v^n \end{pmatrix}, t_n\right) = \begin{pmatrix} x^n \\ v^n \end{pmatrix} + \Delta t \begin{pmatrix} v^n \\ -kx^n - k\beta(x^n)^3 \end{pmatrix} \qquad (1.43)$$

or, when writing the components individually,

$$x^{n+1} = x^n + \Delta t\, v^n \qquad (1.44a)$$
$$v^{n+1} = v^n - \Delta t\, (kx^n + k\beta(x^n)^3) \qquad (1.44b)$$

Given values $x^n$, $v^n$ from the previous time step, this is straightforward to compute.

Deriving the backward Euler will be a bit more complicated. First, we can write out the implicit Euler equation (1.40) for the nonlinear mass-spring system

$$\begin{pmatrix} x^{n+1} \\ v^{n+1} \end{pmatrix} = \begin{pmatrix} x^n \\ v^n \end{pmatrix} + \Delta t \begin{pmatrix} v^{n+1} \\ -kx^{n+1} - k\beta(x^{n+1})^3 \end{pmatrix} \qquad (1.45)$$

This is a system of two equations for the two unknowns $x^{n+1}$ and $v^{n+1}$ and because of the $(x^{n+1})^3$ term, this system of equations is nonlinear. For now, we use Python's **fsolve** function to do it

for us. Because **fsolve** can only solve problems of the form **F(u)**=0, we have to rewrite our problem as

$$\mathbf{F}(\boldsymbol{u_{n+1}}) = \boldsymbol{u_{n+1}} - \Delta t\, \mathbf{f}(\boldsymbol{u_{n+1}}, t_{n+1}) - \boldsymbol{u_n} = \begin{pmatrix} x^{n+1} - \Delta t\, v^{n+1} - x^n \\ v^{n+1} + \Delta t\, (kx^{n+1} + k\beta(x^{n+1})^3) - v^n \end{pmatrix} = 0$$

(1.46)

with **u**=($x^{n+1}$,$v^{n+1}$) (be careful, **f** and **F** are different functions, albeit closely related). Figure 1.5 shows the solution for both methods by running the following program: **springmass_euler.py**.

```
"""
springmass_euler.py
Solves the equation for the nonlinear spring-mass system with forward and
backward Euler methods
"""
import numpy as np
from scipy.integrate import odeint
from scipy.optimize import fsolve

# define the rhs function, fnonlinear
def f_nonlinear(u,t,k,m,beta):
    return [u[1],-(k/m)*(u[0] + beta*(u[0]**3))]

# define the function which needs to be solved at each implicit time step
def F(u,t,dt,i,u_init):
    return u - dt*np.array(f_nonlinear(u,(i+1)*dt,k,m,beta)) - u_init

T = 10.0    # final time until which we compute
N = 200     # number of time steps
taxis = np.linspace(0,T,N+1)
dt = T/N
k = 5.0
beta = 0.1
m = 1.0

# compute frequency alpha for linear (Hookean) solution
alpha = np.sqrt(k/m)

# initial values for position and velocity
x0 = 1.0
v0 = 0.0
u0 = [x0,v0]

# allocate vectors to store solution; note that for the pendulum the vector u
# has two components
u_exp = np.zeros([N+1,2])
u_exp[0,:] = u0

# define right hand side function; assume u = [ x, v ] so that u[0]=x,
# u[1]=v. % Note that we allow for an argument t that we do not really need, so that
# we can reuse f later for the odeint function.

# simple Forward Euler first
for i in range(N):
    dudt = np.array(f_nonlinear(u_exp[i,:],(i+1)*dt,k,m,beta))
    u_exp[i+1,:] = u_exp[i,:] + dt*dudt

# solve with odeint
u_nonlinear = odeint(f_nonlinear,u0,taxis,args=(k,m,beta,))

# now Backward Euler
u_imp = np.zeros([N+1,2])
u_imp[0,:] = u0
for i in range(N):
    u_init = u_imp[i,:]
    # val = fsolve(F,u_init,args=((i+1)*dt,dt,i,u_init))
    u_imp[i+1,:] = fsolve(F,u_init,args=((i+1)*dt,dt,i,u_init))

# plot out results
```

```
import matplotlib.pyplot as plt
plt.plot(taxis,u_exp[:,0],'r')
plt.plot(taxis,u_imp[:,0],'b')
plt.plot(taxis,u_nonlinear[:,0],'k-')
plt.xlim([0,10])
plt.ylim([-2.5,2.5])
plt.xlabel('Time')
plt.ylabel('x')
plt.legend(['Explicit','Implicit','odeint'])
plt.savefig('springmass_euler.jpg')
```

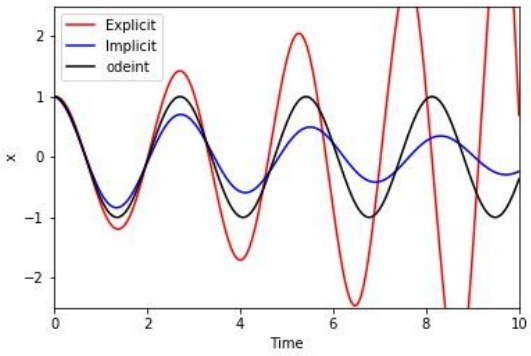


Figure 1.5: Solution of nonlinear pendulum equation with forward (red) and backward (blue) Euler and odeint (black) for reference, obtained using **springmass_euler.py.**

Both are not very satisfactory: implicit Euler causes the amplitude of oscillations to gradually decrease over time until eventually not much oscillation happens at all. In contrast to what should happen, the system has essentially come to rest. Forward Euler does the opposite: the amplitude of the oscillations steadily increases, corresponding to a system that swings faster and faster. Since the example does not include any external forces, this essentially generates kinetic energy out of nothing and is thus a clearly unphysical solution.

## Euler methods for linear spring-mass system.

Let us see how both Euler methods behave for the slightly simpler linear system following Hooke's law where β=0. For forward Euler, we can use almost the same code as for the nonlinear pendulum, we only need to slightly change the definition of the function **f**. For implicit Euler, we can theoretically do the same and rely again on the **fsolve** function, but that turns out to be very inefficient. Instead, notice that for β=0 we can write **f** as

$$\boldsymbol{f}\begin{pmatrix} x \\ v \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ -k & -b \end{pmatrix}\begin{pmatrix} x \\ v \end{pmatrix} = \begin{pmatrix} v \\ -kx - bv \end{pmatrix} \qquad (1.47)$$

That means we can write **f** as a matrix-vector multiplication. We denote the matrix as

$$\boldsymbol{A} = \begin{pmatrix} 0 & 1 \\ -\mathrm{k} & -\mathrm{b} \end{pmatrix} \qquad (1.48)$$

With that, the equation (1.40) we need to solve for backward Euler becomes

$$\boldsymbol{u_{n+1}} - \Delta\mathrm{t}\,\mathbf{A}\,\boldsymbol{u_{n+1}} = \boldsymbol{u_n} \qquad (1.49)$$

or

$$(\mathrm{I} - \Delta \mathrm{t}\, \mathbf{A})\boldsymbol{u_{n+1}} = \boldsymbol{u_n} \qquad (1.50)$$

with I being the identity matrix. We can obtain $\mathbf{u_{n+1}}$ by solving

$$\boldsymbol{u_{n+1}} = (\mathrm{I} - \Delta \mathrm{t}\, \mathbf{A})^{-1}\boldsymbol{u_n} \qquad (1.51)$$

For a general matrix **M**, using the **numpy.linalg** function in Python will return the solution of the linear system of equations

$$\boldsymbol{Mx} = \boldsymbol{b} \qquad (1.52)$$

Figure 1.6 shows the resulting approximations obtained by running the following program, **springmass_linear_euler.py**:

```
"""
springmass_linear_euler.py
Solves the equation for the linear spring-mass system with forward and
backward Euler methods
"""
import numpy as np
from scipy.integrate import odeint

# define the rhs function for linear spring mass case
def f(u,t,A):
    return np.matmul(A,u)

T = 10.0    # final time until which we compute
N = 200     # number of time steps
taxis = np.linspace(0,T,N+1)
dt = T/N
k = 5.0
b = 0.0
beta = 0.1
m = 1.0

# define the matrix A
A = np.array([[0, 1], [-k, -b]])

# initial values for position and velocity
x0 = 1.0
v0 = 0.0
u0 = [x0,v0]

# allocate vectors to store solution; note that for the pendulum the vector u
# has two components
u_exp = np.zeros([N+1,2])
u_exp[0,:] = u0

# define right hand side function; assume u = [ x, v ] so that u[0]=x,
# u[1]=v. % Note that we allow for an argument t that we do not really need, so that
# we can reuse f later for the odeint function.

# simple Forward Euler first
for i in range(N):
    u_exp[i+1,:] = u_exp[i,:] + dt*f(u_exp[i,:],(i+1)*dt,A)

# now Backward Euler
u_imp = np.zeros([N+1,2])
u_imp[0,:] = u0

M = np.identity(2) - dt*A
for i in range(N):
    u_imp[i+1,:] = np.linalg.solve(M,u_imp[i,:])
```

```
# solve with odeint
u_odeint = odeint(f,u0,taxis,args=(A,))

# plot out results
import matplotlib.pyplot as plt
plt.plot(taxis,u_exp[:,0],'r')
plt.plot(taxis,u_imp[:,0],'b')
plt.plot(taxis,u_odeint[:,0],'k-')
plt.xlim([0,10])
plt.ylim([-2.5,2.5])
plt.xlabel('Time')
plt.ylabel('x')
plt.legend(['Explicit','Implicit','odeint'])
plt.savefig('springmass_linear euler.jpg')
```

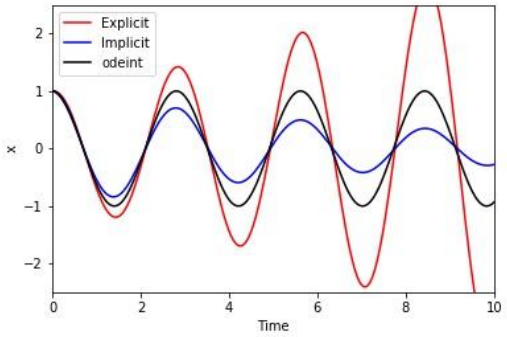


Figure 1.6: Solution of the linearised spring-mass system with explicit (red) and implicit (blue) Euler and the exact solution (black) for comparison, obtained using **springmass_linear_euler.py**

Figure 1.6 shows the resulting approximations. The same issues are present as for the nonlinear pendulum: explicit Euler spuriously increases amplitudes while backward Euler damps them.

## 1.2 Stability

When applied to the Carbon dating equation (1.1), forward Euler becomes

$$C_{14}^{n+1} = C_{14}^{n} - \Delta t\lambda\, C_{14}^{n} = \ (1 - \Delta t\lambda)\, C_{14}^{n} \qquad (1.53)$$

If we apply this recursively, for some given starting value $C_0$ we get

$$C_{14}^{n+1} = (1 - \Delta t\lambda)^{n}\, C_0 \qquad (1.54)$$

We can see that if

$$|1 - \Delta t\lambda| > 1 \qquad (1.55)$$

the number of Carbon atoms $C_{14}^{n}$ in the numerical solution does not decay (as the exact solution (1.6) suggests it should) but explodes, that is $C_{14}^{n} \to \infty$ as $n \to \infty$ -- this is clearly very bad. This is an example of a *numerical instability*, that is a case where the real problem is stable but the numerical solution is not. However, we can see that if the time step Δt is small enough such that

$$|1 - \Delta t\lambda| \leq 1 \qquad (1.56)$$

we get,

$$C_{14}^{n} \to 0 \text{ as } n \to \infty \qquad (1.57)$$

correctly mirroring the exact and physical solution. That means we need to choose the time step Δt such that it satisfies the condition

$$\Delta t \leq \frac{2}{|\lambda|} \qquad (1.58)$$

Therefore, explicit Euler is said to be *conditionally stable*, meaning that it is stable but only if the time step is small enough. In contrast, for backward Euler, we have

$$C_{14}^{n+1} = C_{14}^{n} - \Delta t\lambda\, C_{14}^{n+1} \Rightarrow C_{14}^{n+1} = \frac{1}{1+\Delta t\lambda} C_{14}^{n} = \left(\frac{1}{1+\Delta t\lambda}\right)^{n} C_0 \qquad (1.59)$$

Since both Δt and λ are positive, we have 1 + Δt λ > 1.0 and thus

$$0 \leq \left(\frac{1}{1+\Delta t\lambda}\right) < 1 \qquad (1.60)$$

Therefore, independent of what value we use for Δt, backward Euler always guarantees that

$$C_{14}^{n} \to 0 \text{ as } n \to \infty \qquad (1.61)$$

always mirroring the correct asymptotic behaviour. Implicit Euler is said to be *unconditionally stable*. Note, however, that the solution provided will still be terribly inaccurate for very large Δt. Stability only guarantees that the solution does not blow up, it does not say anything about its accuracy.

### Self Study

Use the example codes **carbon_euler.py** (given above) and **carbon_euler_accuracy.py** from the 'Fig1_8' directory which implements forward and backward Euler for the radioactive decay equation. **carbon_euler_accuracy.py** is given below.

```
"""
carbon_euler_accuracy.py
Solves the decay equation with forward and backward Euler for a range of
time steps to analyse how the error decreases as we make dt smaller
"""
import numpy as np

# exact solution
def u_exact(t,r0,lam):
    return r0*np.exp(-lam*t)

# forward euler function
def exp_euler(u0,Tend,nsteps,lam):
    dt = Tend/nsteps
    u = np.zeros(nsteps+1)
    u[0] = u0
    for i in range(nsteps):
        u[i+1] = u[i] - dt*lam*u[i]
    return u
```

```
# backward euler function
def imp_euler(u0,Tend,nsteps,lam):
    dt = Tend/nsteps
    u = np.zeros(nsteps+1)
    u[0] = u0
    for i in range(nsteps):
        u[i+1] = u[i]/(1 + dt*lam)
    return u

# set up problem parameters
T = 1.0   # time up to which we compute
lam = 1.0   # decay constant
r0 = 1.0   # set ratio at t=0
N = [1000,750,500,250,100,75,50,10]

# allocate vectors to store for every run
err_exp = np.zeros(len(N))
err_imp = np.zeros(len(N))
dts = np.zeros(len(N))

# exact solution
# taxis = np.linspace(0,T,N[0])
# out = u_exact(taxis,r0,lam)

for n in range(len(N)):
    taxis = np.linspace(0,T,N[n]+1)
    u_exp = exp_euler(r0,T,N[n],lam)
    u_imp = imp_euler(r0,T,N[n],lam)

    # stor the time step dt for plotting
    dts[n] = taxis[1] - taxis[0]

    # now compute the errors
    err_exp[n] = max(np.abs(u_exp-u_exact(taxis,r0,lam)))
    err_imp[n] = max(np.abs(u_imp-u_exact(taxis,r0,lam)))

# we fit a line log(err) = p*log(N) + C through the data points for
# reasons that will become clear later
# build 6th order fit to build data
p_exp = np.polyfit(np.log(dts), np.log(err_exp),1)
p_imp = np.polyfit(np.log(dts), np.log(err_imp),1)

# plot out results
import matplotlib.pyplot as plt
plot1 = plt.figure(1)
plt.loglog(dts,err_exp,'ro')
plt.loglog(dts,np.exp(np.polyval(p_exp, np.log(dts))),'r')
plt.xlim([dts[0],dts[len(N)-1]])
figtext='Slope p='+str(round(p_exp[0],2))
plt.text(1e-2,1e-3,figtext)
plt.xlabel(r'$\Delta t$')
plt.ylabel('Error')
plt.legend(['Explicit Eulwer','Linear Fit'])
plt.savefig('carbon_euler_accuracy1.jpg')

plot1 = plt.figure(2)
plt.loglog(dts,err_imp,'bo')
plt.loglog(dts,np.exp(np.polyval(p_imp, np.log(dts))),'b')
plt.xlim([dts[0],dts[len(N)-1]])
figtext='Slope p='+str(round(p_imp[0],2))
plt.text(1e-2,1e-3,figtext)
plt.xlabel(r'$\Delta t$')
plt.ylabel('Error')
plt.legend(['Implicit Eulwer','Linear Fit'])
plt.savefig('carbon_euler_accuracy2.jpg')
```

Confirm that the solution of backward Euler always goes to zero, independent of the time step. Confirm also that the value $\Delta t = 2/|\lambda|$ is the threshold at which forward Euler becomes unstable. What happens directly at the threshold?

### Complex coefficients

Even though the interpretation as an equation modelling radioactive decay is no longer valid in that case, it is instructive to look at the equation

$$\dot{y}(t) = iay(t), \;\; y(0) = y_0 \qquad (1.62)$$

This looks like the decay equation (1.1) but with y(t) instead of $C_{14}$ and a complex decay rate ia instead of -λ. It is easy to verify that the solution to this initial value problem is

$$y(t) = y_0 e^{iat} \qquad (1.63)$$

If we apply forward Euler to (1.62), we get

$$y^{n+1} = y^n + \Delta tiay^n = (1 + \Delta tia)y^n \qquad (1.64)$$

From that it follows that

$$|y^{n+1}| = |1 + \Delta tia||y^n| = \sqrt{1^2 + (\Delta ta)^2}|y^n| = \sqrt{1 + \Delta t^2 a^2}|y^n| = \left(\sqrt{1 + \Delta t^2 a^2}\right)^n |y_0| \qquad (1.65)$$

Because $1 + \Delta t^2 a^2 > 1$, we have

$$\left(\sqrt{1 + \Delta t^2 a^2}\,\right)^n \to \infty \text{ as } n \to \infty \qquad (1.66)$$

independent of the time step Δt. This means that forward Euler applied to (1.62) is unconditionally unstable because no matter what time step Δt > 0 we choose, we get a series of discrete approximations $y^n$ with absolute values going toward infinity, that is

$$|y^n| \to \infty \text{ as } n \to \infty \qquad (1.67)$$

This very much resembles what we diagnosed for the energy when using forward Euler for the spring-mass system and we will see that there is a very close relation. Using a similar argument with slightly more complex arithmetic, one can show that the situation is reversed for backward Euler applied to (1.62). Here, we get

$$|y^n| \to 0 \text{ as } n \to \infty \qquad (1.68)$$

for any time step Δt > 0.

### 1.2.1. Stability of forward and backward Euler for the mass-spring system

We will first analyse the stability of the Euler methods for the linear, unforced, undamped mass-spring systems by analysing how both methods deal with energy. First, note that for $b=m=F_{ext}=0$, equation (1.17) simplifies to

$$\dot{x}(t) = v(t); \;\; \dot{v}(t) = -\frac{k}{m}x(t) \qquad (1.69a, 1.69b)$$

The kinetic energy of the system is given by

$$E_{kin}(t) = \frac{1}{2}mv^2(t) \qquad (1.70)$$

and its potential energy by

$$E_{pot}(t) = \frac{1}{2}kx^2(t) \qquad (1.71)$$

Its total energy therefore is

$$E(t) = E_{kin}(t) + E_{pot}(t) = \frac{1}{2}mv^2(t) + \frac{1}{2}kx^2(t) \qquad (1.72)$$

Note that E(t) is conserved in the sense that

$$\dot{E}(t) = mv(t)\dot{v}(t) + kx(t)\dot{x}(t) = mv(t)(-\left(\frac{k}{m}\right)x(t) + kx(t)v(t) = 0 \qquad (1.73)$$

That is, if $x_0$, $v_0$ are the initial position and velocity, we have

$$E(t) = E(0) = \frac{1}{2}mv_0^2 + \frac{1}{2}kx_0^2 \qquad (1.74)$$

at any time t. The advantage of energy is that we can quantify the energy error of a numerical solution without having to have access to the exact solution. Let $x^n$, $v^n$ be the approximations to position and velocity provided by some numerical scheme at time $t_n$, as before. Then, we can define the *discrete* energy as

$$E^n = \frac{1}{2}k(x^n)^2 + \frac{1}{2}m(v^n)^2 \qquad (1.75)$$

Because $x^0 = x_0$ and $v^0 = v_0$ (since a numerical scheme will always start from the provided initial values), we have $E^0$=E(0). We can then compute the energy error

$$e^n_{energy} = \frac{|E^n - E^0|}{E^0} \qquad (1.76)$$

If a numerical scheme were to conserve energy exactly, we would get $e^n_{energy}$=0 for all $0 \leq n \leq N$. Figure 1.7 shows the energy and energy error for forward Euler, backward Euler and Python's **odeint**, obtained by running **oscillator.py, which** is given below.

```
"""
oscillator.py
Solves the oscillator problem using forward and backward Euler methods
"""
import numpy as np
from scipy.integrate import odeint

# define the rhs function for linear oscillator
def f(u,t,b,k,m,F,Omega):
    return [u[1],-(1/m)*(k*u[0] + b*u[1]) + F*np.cos(Omega*t)]

# forward euler function
def forward_euler(f,x0,Tend,nsteps,b,k,m,F,Omega):
    dt = Tend/nsteps
    y = np.zeros([nsteps+1,2])
    y[0,:] = x0
    for i in range(nsteps):
        t = i*dt
        y[i+1,:] = y[i,:] + dt*np.array(f(y[i,:],t,b,k,m,F,Omega))

    return y

# backward euler function
```

```
def backward_euler(f,x0,Tend,nsteps,b,k,m,F,Omega):

    # define the matrix for the oscillator problem in vector form
    # define the matrix A
    dt = Tend/nsteps
    A = np.array([[0, 1], [-k/m, -b/m]])
    M = np.identity(2) - dt*A
    y = np.zeros([nsteps+1,2])
    y[0,:] = x0
    for i in range(nsteps):
        y[i+1,:] = np.linalg.solve(M,y[i,:])

    return y

# physical properties of the oscillator
k = 1.0   # spring constant
b = 0.0   # damping
m = 1.0   # mass
F = 0.0   # amplitude of cosine forcing
Omega = 0.5   # frequency of cosine forcing

# initial values for position and velocity
x0 = 1.0
v0 = 0.0
u0 = [x0,v0]

# final time and number of time steps
Tend = 50.0    # final time until which we compute
nsteps = 1000

taxis = np.linspace(0,Tend,nsteps+1)
dt = Tend/nsteps

# initial values for position and velocity
x0 = 1.0
v0 = 0.0
u0 = [x0,v0]

# solve with odeint
y_odeint = odeint(f,u0,taxis,args=(b,k,m,F,Omega,))

# solve with backward euler
y_ie = backward_euler(f,u0,Tend,nsteps,b,k,m,F,Omega)

# solve with forward euler
y_ee = forward_euler(f,u0,Tend,nsteps,b,k,m,F,Omega)

# plot out positions for all solutions
import matplotlib.pyplot as plt
plot1 = plt.figure(1)
plt.plot(taxis,y_ee[:,0],'r')
plt.plot(taxis,y_ie[:,0],'b')
plt.plot(taxis,y_odeint[:,0],'k-')
plt.xlim([0,50])
plt.ylim([-4,4])
plt.xlabel('Time')
plt.ylabel('x')
plt.legend(['Explicit','Implicit','odeint'])
plt.savefig('oscillator1.jpg')

# compute the total, potential and kinetic energy over time
E_pot = 0.5*y_odeint[:,0]**2
E_kin = 0.5*y_odeint[:,1]**2
E_tot = E_pot + E_kin
plot2 = plt.figure(2)
plt.plot(taxis,E_pot,'r')
plt.plot(taxis,E_kin,'b')
plt.plot(taxis,E_tot,'k-')
plt.xlim([0,50])
plt.ylim([0,0.6])
plt.legend(['Potential','Kinetic','Energy'])

# plot out the energy over time
E_ie = 0.5*k*y_ie[:,0]**2 + 0.5*m*y_ie[:,1]**2
E_ee = 0.5*k*y_ee[:,0]**2 + 0.5*m*y_ee[:,1]**2
```

```
plot3 = plt.figure(3)
plt.plot(taxis,E_pot+E_kin,'k-')
plt.plot(taxis,E_ie,'b')
plt.plot(taxis,E_ee,'r-')
plt.xlim([0,50])
plt.ylim([0,7])
plt.legend(['odeint','Implicit Euler','Forward Euler'])
```

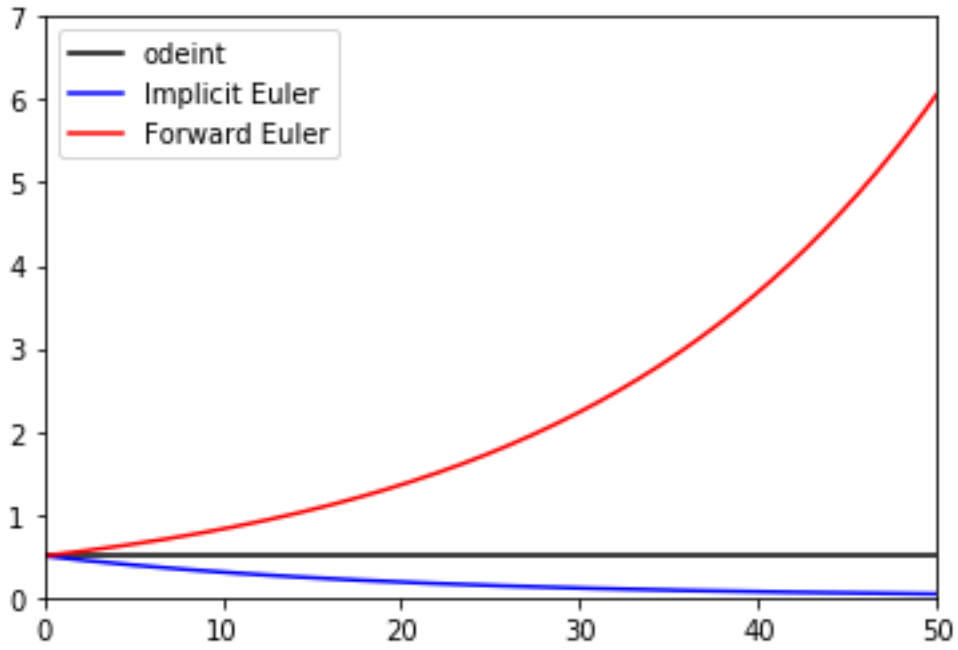


Figure 1.7: Energy for the linear mass-spring system for forward Euler, backward Euler and Python's odeint solver, obtained using oscillator.py.

In line with the results we saw previously, forward Euler does indeed artificially generate energy out of nothing. Its energy error grows rapidly and without bound. In contrast, backward Euler continuously loses energy. Note that its energy error remains bounded because eventually the numerical solution will approach the resting state with zero energy, at which point the energy error

$$e^n_{energy} = \frac{|0-E^0|}{E^0} = 1 \qquad (1.77)$$

We can back up this observation with mathematical analysis. Forward Euler applied to system (1.69) reads, in components,

$$x^{n+1} = x^n + \Delta t\, v^n$$
$$v^{n+1} = v^n - \frac{\Delta t k}{m}\, x^n$$

Using definition (1.75) of the discrete energy, we can compute

$$\begin{aligned} 2E^{n+1} &= m(v^{n+1})^2 + k(x^{n+1})^2 \\ &= m\left(v^n - \Delta t(\tfrac{k}{m})x^n\right)^2 + k(x^n + \Delta t v^n)^2 \\ &= m(v^n)^2 - 2k\Delta t v^n x^n + \tfrac{(\Delta t k x^n)^2}{m} + k(x^n)^2 + 2k\Delta t x^n v^n + k(\Delta t v^n)^2 \\ &= 2E^n + (\Delta t)^2\left(\tfrac{(kx^n)^2}{m} + k(v^n)^2\right) > 2E^n \end{aligned}$$

That is, instead of conserving energy from time step to time step, forward Euler *adds* (because $C^n>0$) a small amount of energy in every time step. While we recover energy conservation in the limit $\Delta t \to 0$, for any finite time step $\Delta t > 0$, energy will always increase over time. When applied to the mass-spring system, forward Euler is *unconditionally unstable*.

To analyse backward Euler, we need to use the matrix representation

$$\left[\begin{pmatrix}1 & 0\\ 0 & 1\end{pmatrix} - \Delta t\begin{pmatrix}0 & 1\\ -1 & 0\end{pmatrix}\right]\begin{pmatrix}x^{n+1}\\ v^{n+1}\end{pmatrix} = \begin{pmatrix}x^n\\ v^n\end{pmatrix} \qquad (1.78)$$

where we set k = m = 1 for simplicity. The matrix in the system has the inverse

$$\begin{pmatrix}1 & -\Delta t\\ \Delta t & 1\end{pmatrix}^{-1} = \frac{1}{1+(\Delta t)^2}\begin{pmatrix}1 & \Delta t\\ -\Delta t & 1\end{pmatrix} \qquad (1.79)$$

so that

$$\begin{pmatrix}x^{n+1}\\ v^{n+1}\end{pmatrix} = \frac{1}{1+(\Delta t)^2}\begin{pmatrix}1 & \Delta t\\ -\Delta t & 1\end{pmatrix}\begin{pmatrix}x^n\\ v^n\end{pmatrix} \qquad (1.80)$$

Written in components, this becomes

$$x^{n+1} = \frac{1}{1+(\Delta t)^2}(x^n + \Delta t v^n) \qquad (1.81)$$
$$v^{n+1} = \frac{1}{1+(\Delta t)^2}(v^n - \Delta t x^n) \qquad (1.82)$$

We can now compute the discrete energy again.

$$\begin{aligned}2E^{n+1} &= (x^{n+1})^2 + (v^{n+1})^2\\ &= \frac{1}{(1+(\Delta t)^2)^2}[(x^n + \Delta t v^n)^2 + (v^n - \Delta t x^n)^2]\\ &= \frac{1}{(1+(\Delta t)^2)^2}\left[(x^n)^2 + 2\Delta t\, x^n v^n + (\Delta t v^n)^2 + (v^n)^2 - 2\Delta t\, x^n v^n + (\Delta t x^n)^2\right]\\ &= \frac{1}{(1+(\Delta t)^2)^2}[2E^n + (\Delta t)^2 E^n]\\ &= \frac{2}{1+(\Delta t)^2}E^n\end{aligned}$$

Because in every numerical simulation we have $\Delta t > 0$, we have $1/(1+(\Delta t)^2) < 1$ so that for every step

$$E^{n+1} < E^n \qquad (1.83)$$

As seen in the numerical experiment, backward Euler continuously loses energy. While we get $E^{n+1}=E^n$ in the limit $\Delta t \to 0$, in every real simulation with a finite time step, backward Euler does not conserve energy. Therefore, as for the decay equation, backward Euler is *unconditionally stable* when applied to the mass-spring system. As seen, however, it is in a sense too stable: the loss of energy means that it will only provide useful numerical approximations over a relatively short time window.

### 1.2.2 Stability and eigenvalues

We can link the instability of forward Euler for the mass-spring system to the instability for the imaginary test problem. As we saw above, we can write the linear mass-spring system as

$$\dot{\boldsymbol{u}}(\mathrm{t}) = \mathrm{A}\mathbf{u}(\mathrm{t}) \qquad (1.84)$$

with the matrix A defined in (1.48). By finding its eigenvalues and eigenvectors, we can compute the eigendecomposition

$$A = Q\Sigma Q^{-1} \qquad (1.85)$$

with

$$Q = \begin{pmatrix} i\sqrt{m/k} & -i\sqrt{m/k} \\ 1 & 1 \end{pmatrix} \qquad (1.86)$$

and

$$\Sigma = \begin{pmatrix} -i\sqrt{k/m} & 0 \\ 0 & i\sqrt{k/m} \end{pmatrix} \qquad (1.87)$$

Using this, the linear mass-spring system becomes

$$Q^{-1}\dot{\boldsymbol{u}}(\mathrm{t}) = \Sigma Q^{-1}\mathbf{u}(\mathrm{t}) \qquad (1.88)$$

after multiplication with $Q^{-1}$. Define the transformed solution in eigencoordinates as

$$\mathbf{z}(\mathrm{t}) = Q^{-1}\mathbf{v}(\mathrm{t}) \qquad (1.89)$$

It satisfies the differential equation

$$\dot{\mathbf{z}}(\mathrm{t}) = \Sigma\mathbf{z}(\mathrm{t}) \qquad (1.90)$$

or, written in components $\mathbf{z}(t)=(z_1(t),z_2(t))$,

$$\dot{z_1} = -i\sqrt{\frac{k}{m}}z_1(t) \qquad (1.91a)$$

$$\dot{z_2} = i\sqrt{\frac{k}{m}}z_2(t) \qquad (1.91b)$$

We can make two observations: first, in eigencoordinates the equations for both components are decoupled. $z_1(t)$ evolves independently of $z_2(t)$ and vice versa. For both components, the resulting equations have the form of the complex scalar test equation (1.62) for which we know forward Euler to be unstable. Therefore, we can explain the instability of forward Euler for the linear mass-spring system by the fact that the matrix A has imaginary eigenvalues which gives rise to equations (1.91) in eigencoordinates for which we know forward Euler to be unstable.

## 1.3 Accuracy

Stability helps us to see if a method fails catastrophically, like forward Euler for the oscillator. However, even a stable method does not necessarily guarantee accurate results. Therefore, we will now investigate how the global error, e, of a method decreases as we make the time step smaller.

Of course, any reasonable numerical method should retrieve the correct solution in the limit Δt->0. Because very small time steps will translate into high computational cost, using an arbitrarily small step is not an efficient strategy.

### 1.3.1 Error and order of accuracy

The *order of accuracy p* of a time stepping method describes how the error is reduced as the time step Δt is made smaller. Here, *error* denotes how far from the exact solution the approximate solution provided by the method is. For example, for the radioactive decay problem, the exact solution was

$$C(t) = C_0 e^{\lambda t} \qquad (1.92)$$

If $C_1 \approx C(\Delta t)$, $C_2 \approx C(2\Delta t)$ etc. be approximations of the exact solution produced e.g. by a forward Euler method. The error at the end of the simulation would then be

$$e_{global}(\Delta t) = \max_{n=1,\ldots,N} |C_{14}^n - C_{14}(t_n)| \qquad (1.93)$$

compare for (1.35). Note that we make the dependence of $e_{global}$ on Δt explicit now, because we will study systemically how it changes with time step length. Figure 1.8 shows the error from explicit Euler (left) and implicit Euler (right) when applied to the decay equation with N time steps, that is a time step Δt=1/N. Note that both the x- and y-axis are scaled logarithmically. Figure 1.8 is generated by running **carbon_euler_accuracy.py**, given below.

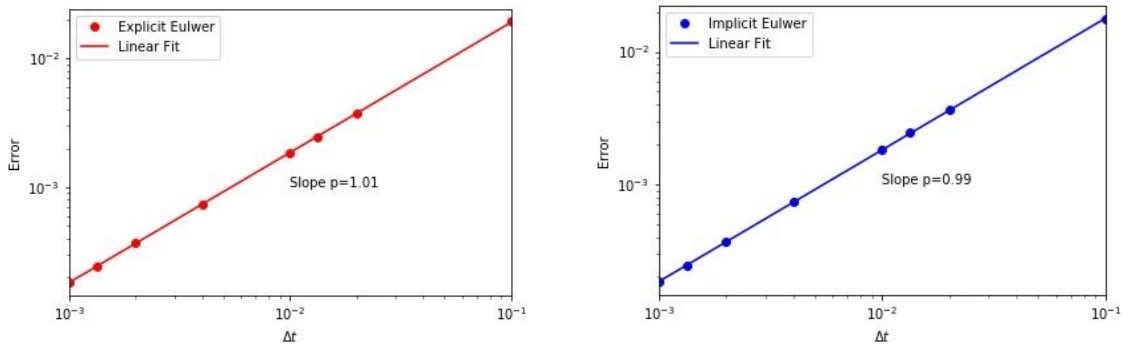


Figure 1.8: Error of forward Euler (left) and backward Euler (right) depending on the number N of time steps, obtained using carbon_euler_accuracy.py.

```
"""
carbon_euler_accuracy.py
Solves the decay equation with forward and backward Euler for a range of
time steps to analyse how the error decreases as we make dt smaller
"""
import numpy as np

# exact solution
def u_exact(t,r0,lam):
    return r0*np.exp(-lam*t)

# forward euler function
def exp_euler(u0,Tend,nsteps,lam):
    dt = Tend/nsteps
    u = np.zeros(nsteps+1)
    u[0] = u0
    for i in range(nsteps):
        u[i+1] = u[i] - dt*lam*u[i]
    return u
```

```
# backward euler function
def imp_euler(u0,Tend,nsteps,lam):
    dt = Tend/nsteps
    u = np.zeros(nsteps+1)
    u[0] = u0
    for i in range(nsteps):
        u[i+1] = u[i]/(1 + dt*lam)
    return u

# set up problem parameters
T = 1.0   # time up to which we compute
lam = 1.0   # decay constant
r0 = 1.0   # set ratio at t=0
N = [1000,750,500,250,100,75,50,10]

# allocate vectors to store for every run
err_exp = np.zeros(len(N))
err_imp = np.zeros(len(N))
dts = np.zeros(len(N))

# exact solution
# taxis = np.linspace(0,T,N[0])
# out = u_exact(taxis,r0,lam)

for n in range(len(N)):
    taxis = np.linspace(0,T,N[n]+1)
    u_exp = exp_euler(r0,T,N[n],lam)
    u_imp = imp_euler(r0,T,N[n],lam)

    # stor the time step dt for plotting
    dts[n] = taxis[1] - taxis[0]

    # now compute the errors
    err_exp[n] = max(np.abs(u_exp-u_exact(taxis,r0,lam)))
    err_imp[n] = max(np.abs(u_imp-u_exact(taxis,r0,lam)))

# we fit a line log(err) = p*log(N) + C through the data points for
# reasons that will become clear later
# build 6th order fit to build data
p_exp = np.polyfit(np.log(dts), np.log(err_exp),1)
p_imp = np.polyfit(np.log(dts), np.log(err_imp),1)

# plot out results
import matplotlib.pyplot as plt
plot1 = plt.figure(1)
plt.loglog(dts,err_exp,'ro')
plt.loglog(dts,np.exp(np.polyval(p_exp, np.log(dts))),'r')
plt.xlim([dts[0],dts[len(N)-1]])
figtext='Slope p='+str(round(p_exp[0],2))
plt.text(1e-2,1e-3,figtext)
plt.xlabel(r'$\Delta t$')
plt.ylabel('Error')
plt.legend(['Explicit Eulwer','Linear Fit'])
plt.savefig('carbon_euler_accuracy1.jpg')

plot1 = plt.figure(2)
plt.loglog(dts,err_imp,'bo')
plt.loglog(dts,np.exp(np.polyval(p_imp, np.log(dts))),'b')
plt.xlim([dts[0],dts[len(N)-1]])
figtext='Slope p='+str(round(p_imp[0],2))
plt.text(1e-2,1e-3,figtext)
plt.xlabel(r'$\Delta t$')
plt.ylabel('Error')
plt.legend(['Implicit Eulwer','Linear Fit'])
plt.savefig('carbon_euler_accuracy2.jpg')
```

**Self-study**

Use **carbon_euler_accuracy.py** and redo the plots without the logarithmic axes scaling. Why is this not a particularly useful way to visualise the error?

For comparison, a straight line

$$log_e\left(e_{global}(\Delta t)\right) = p\ log_e(\Delta t) + a \qquad (1.94)$$

is fitted through the data, using Python's numpy **polyfit** function. Clearly, the obtained data points line up excellently with the fit which has a slope of approximately p=1. Applying the exponential function on both sides yields

$$e^{log_e(e_{global}(\Delta t))} = \left(e^{log_e(\Delta t)}\right)^p e^a \qquad (1.95)$$

or

$$e_{global}(\Delta t) = C\ (\Delta t)^p \qquad (1.96)$$

with $C = e^a > 0$.

### Remark 3.

Using **carbon_euler_accuracy.py** to produce a plot like the ones in Figure 1.8 to verify that a method gives the expected order of accuracy is one good way to validate that a method has been correctly implemented.

This motivates the following definition.

**Definition 1**

A numerical method is said to be of order p if its error is proportional to $(\Delta t)^p$, that is

$$e_{global}(\Delta t) \leq C(\Delta t)^p \qquad (1.97)$$

for some positive number C>0 that does not depend on Δt. From our observation above, we can conclude that both Euler methods are first order accurate, that is they satisfy (1.97) with p=1. Note that for a method of order p, reducing the time step by half results in

$$e_{global}\left(\frac{\Delta t}{2}\right) \approx C\,\frac{(\Delta t)^p}{2^p} \approx \frac{1}{2^p}\,e_{global}(\Delta t) \qquad (1.98)$$

Therefore, for a first order method with p=1, halving the time step will also reduce the error by half. For a second order method with p=2, halving Δt will reduce the error by a factor of 4. For p=3, it will reduce the error by a factor of 8 and so on.

A simple method with order better than one is Heun's method. It relies on a forward Euler predictor step

$$\widetilde{\boldsymbol{u}}^{\mathrm{n}+1} = \boldsymbol{u}^{\boldsymbol{n}} + \Delta \mathrm{t}\, \boldsymbol{f}(\boldsymbol{u}^{\boldsymbol{n}}, \mathrm{t_n}) \qquad (1.99)$$

but then follows that up with a second step

$$\boldsymbol{u}^{\boldsymbol{n+1}} = \boldsymbol{u}^{\boldsymbol{n}} + \frac{\Delta \mathrm{t}}{2}\left(\boldsymbol{f}(\boldsymbol{u}^{\boldsymbol{n}}, \mathrm{t_n}) + \boldsymbol{f}(\,\widetilde{\boldsymbol{u}}^{\mathrm{n}+1}, \mathrm{t_{n+1}})\right) \qquad (1.100)$$

In terms of computational cost, Heun's method is roughly twice as expensive as a forward Euler step, because it has to evaluate the right hand side function **f** twice whereas Euler has to evaluate it only

once. However, because the error decays much faster as Δt becomes small because of the higher order, Heun's method can be more efficient than forward Euler.

Using **carbon_euler_heun_workprecision.py**, given below, Figure 1.9 shows the error achieved by forward Euler and Heun's method depending on the total number of evaluations of **f** required.

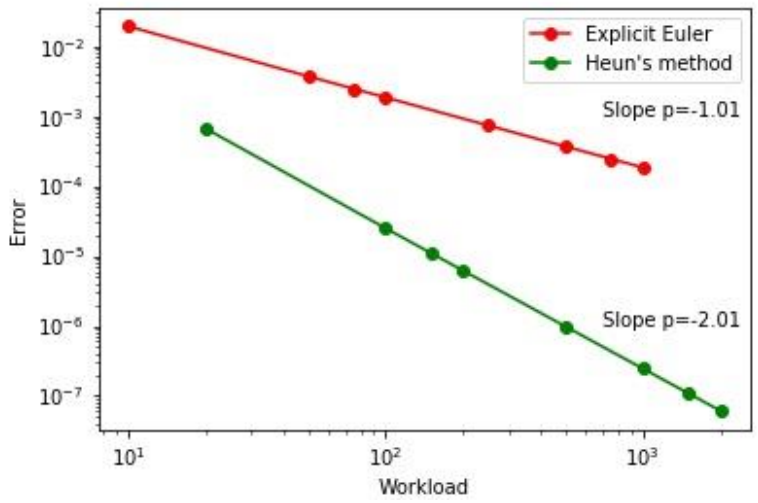


Figure 1.9: Error versus number of evaluations of the right hand side function **f** for forward Euler and Heun's method, obtained using carbon_euler_heun_workprecision.py.

Note that the x-axis now shows total computational work measured in how often the method has to evaluate the right hand side function **f.**

**carbon_euler_heun_workprecision.py**

```
"""
carbon_euler_heun_workprecision.py
Solves the decay equation with forward and backward Euler for a range of
time steps to analyse how the error decreases as we make dt smaller
"""
import numpy as np

##############################################################################
# Functions
##############################################################################
# exact solution
def u_exact(t,r0,lam):
    return r0*np.exp(-lam*t)

# forward euler function
def exp_euler(u0,Tend,nsteps,lam):
    dt = Tend/nsteps
    u = np.zeros(nsteps+1)
    u[0] = u0
    for i in range(nsteps):
        u[i+1] = u[i] - dt*lam*u[i]
    return u

# backward euler function
def imp_euler(u0,Tend,nsteps,lam):
    dt = Tend/nsteps
    u = np.zeros(nsteps+1)
    u[0] = u0
    for i in range(nsteps):
        u[i+1] = u[i]/(1 + dt*lam)
    return u
```

```
# function definition of time derivative
def f(u,lam):
    return -lam*u

# heun's method
def heun(u0,Tend,nsteps,lam):
    dt = Tend/nsteps
    u = np.zeros(nsteps+1)
    u[0] = u0
    for i in range(nsteps):
        utemp = u[i] + dt*f(u[i],lam)
        u[i+1] = u[i] + 0.5*dt*(f(u[i],lam) + f(utemp,lam))

    return u

#############################################################################

# set up problem parameters
T = 1.0   # time up to which we compute
lam = 1.0   # decay constant
r0 = 1.0   # set ratio at t=0
N = [1000,750,500,250,100,75,50,10]

# allocate vectors to store for every run
err_exp = np.zeros(len(N))
err_heun = np.zeros(len(N))

# Instead of dt, we now store the workload, measured in the number of times
# that a method has to evaluate the right hand side function
workload_exp = np.zeros(len(N))
workload_heun = np.zeros(len(N))

for n in range(len(N)):
    taxis = np.linspace(0,T,N[n]+1)   # add +1 to account for t=0
    u_exp = exp_euler(r0,T,N[n],lam)
    u_heun = heun(r0,T,N[n],lam)

    # stor the time step dt for plotting
    workload_exp[n] = N[n]
    workload_heun[n] = 2*N[n]

    # now compute the errors
    err_exp[n] = max(np.abs(u_exp-u_exact(taxis,r0,lam)))
    err_heun[n] = max(np.abs(u_heun-u_exact(taxis,r0,lam)))

# Find slope of lines for Euler and Heun's method
p_exp = np.polyfit(np.log(workload_exp), np.log(err_exp),1)
p_heun = np.polyfit(np.log(workload_heun), np.log(err_heun),1)

# plot out results
import matplotlib.pyplot as plt
plot1 = plt.figure(1)
plt.loglog(workload_exp,err_exp,'ro-')
plt.loglog(workload_heun,err_heun,'go-')
txt_exp='Slope p='+str(round(p_exp[0],2))
plt.text(7e2,1e-3,txt_exp)
txt_heun='Slope p='+str(round(p_heun[0],2))
plt.text(7e2,1e-6,txt_heun)
plt.xlabel('Workload')
plt.ylabel('Error')
plt.legend(['Explicit Euler','Heun\'s method'])
plt.savefig('carbon_euler_precision.jpg')
```

However, because workload grows as time step Δt shrinks, the lines are now sloping in the opposite direction compared to Fig 1.8 and values for p are now negative. The first important observation is that Heun's method does indeed provide a line with a slope of p=2, in contrast to forward Euler with p=1. This confirms that it is indeed second order accurate. Furthermore, Heun's method is clearly more efficient than forward Euler. If, say, a precision of $e_{global}(\Delta t) < 10^{-3}$ were required, Heun's method can deliver this with 20 evaluations of **f** whereas forward Euler would need around 200, that is ten times more. This motivates the search for methods that have a higher order of accuracy.

### 1.4 Modelling Homogeneous Chemical Reactions in CO2 pipeline corrosion

Dissolved $CO_2$ in pipeline systems leads to corrosive attack of the pipeline due to formation of Carbonic Acid and associated reactions.

**NOTE:** The reaction rate constants below are not all expressed in terms of the SI units, so when these equations are finally solved, these have been converted for concentrations written in terms of the SI units, mol/m$^3$. In the code example, all reaction rate constants are taken from the paper: *S. Nordsveen, S. Nesic, R. Nyborg, A. Stangeland. A mechanistic model for carbon dioxide corrosion of mild steel in the presence of protective iron carbonate films – part 1: theory and validation, Corrosion, vol. 59(5), 2003, 443-456, table 2, p 447.*

The reactions are summarised by the following steps.

Step 1: Dissolution of $CO_2$

The dissolution of $CO_2$ in water is represented by the reaction:

$$CO_2(g) \Leftrightarrow CO_2\ (aq) \qquad (1.101)$$

The partial pressure of $CO_2$ enables the initial concentration of $CO_2$ (aq) to be calculated via:

$$K_{sol} = \frac{c_{CO_2}}{p_{CO_2}} \qquad (1.102)$$

where $c_{CO_2}$ is the concentration of dissolved $CO_2$ in moles/litre and $p_{CO2}$ is the partial pressure of $CO_2$ in bar**.** There are numerous possible expressions that can be used for $K_{sol}$ but here the following expression is used for $K_{sol}$ with units moles/(litre. bar):

$$K_{sol} = \frac{14.5}{1.00258}\ 10^{exp} \text{ where } exp = -(2.27 + 0.00565T_f - 8.06 \times 10^{-6}T_f^2 + 0.075I) \qquad (1.103)$$

where $T_f$ is the temperature in degrees Fahrenheit and I is the ionic strength in moles/litre.

The ionic strength, I, of the electric field in a solution, is equal to the sum of the molarities of each type of ion present multiplied by the square of the charges.

$$I = \frac{1}{2}\sum_{i=1}^{n} c_i z_i^2$$

where $c_i$ = molar concentration of ion i (mol/litre) and $z_i$ = charge number of ion i.

Example: To calculate the ionic strength of 0.05M $Na_2SO_4$ and 0.02M KCl solution:

$$I = \frac{1}{2}\begin{pmatrix} [Na_2SO_4]\ \times (\#Na\ ions) \times (charge\ of\ Na\ ion)^2 + \\ [Na_2SO_4]\ \times (\#SO_4\ ions) \times (charge\ of\ SO_4\ ion)^2 \\ +\ [KCl]\ \times (\#K\ ions) \times (charge\ of\ K\ ion)^2 + \\ [KCl]\ \times (\#Cl\ ions) \times (charge\ of\ Cl\ ion)^2 \end{pmatrix}$$

$$I = \frac{1}{2}\begin{pmatrix} 0.05\ \times 2 \times 1^2 + \\ 0.05\ \times 1 \times (-2)^2 \\ +\ 0.02\ \times 1 \times 1^2 + \\ 0.02\ \times 1 \times (-1)^2 \end{pmatrix} = 0.17M$$

***In the code example solved here I=0.1711 and $T_f$=68°F.***

Combining the above equations yields an estimate for the initial concentration of $CO_2$, $c_{H_2CO_3}$, in SI units:

$$c_{CO_2} = 1000\, K_{sol} p_{CO_2} \quad (1.104)$$

Step 2: Carbonic Acid Hydration

The carbonic acid hydration reaction is given by:

$$CO_2 + H_2O \Leftrightarrow H_2CO_3 \quad (1.105)$$

The rate of production of carbonic acid, $H_2CO_3$, is given by the following differential equation:

$$R_{H_2CO_3} = \frac{d}{dt}\left(c_{H_2CO_3}\right) = K_{f,hy} c_{CO_2} c_{H_2O} - K_{b,hy}\, c_{H_2CO_3} \quad (1.106)$$

In terms of the forward reaction rate $K_{f,hy}$ and backward reaction rate, $K_{b,hy}$. In practice, carbonic acid never represents more than 1% of the total dissolved $CO_2$ as $K_{b,hy} >> K_{f,hy}$. Here, we assume that

$$K_{hy} = \frac{K_{f,hy}}{K_{b,hy}} = 0.00258 \quad (1.107)$$

is independent of temperature. The forward reaction rate constant, $K_{f,hy}$, is given by

$$K_{f,hy} = \frac{14.5}{1.00258}\, 10^{exp} \text{ where } exp = (329.85 - 110.541 log_{10}(T_K) - \left(\frac{17265.4}{T_K}\right)) \quad (1.108)$$

Combining the latter two equations yields the following estimate of the initial value of $c_{H_2CO_3}$ (in SI units) is given by:

$$c_{H_2CO_3} = 1000\, K_{hy} K_{sol} p_{CO_2} \quad (1.109)$$

***In the code example solved here $T_K$=293.15K.***

Step 3: Carbonic Acid Dissociation

The carbonic acid dissociation reaction is given by:

$$H_2CO_3 \Leftrightarrow H^+ + HCO_3^- \quad (1.110)$$

In terms of the forward reaction rate $K_{f,ca}$, and the backward reaction rate $K_{b,ca}$ the rate of production of $H_2CO_3$ is given by:

$$R_{H_2CO_3} = \frac{d}{dt}\left(c_{H_2CO_3}\right) = -(K_{f,ca}\, c_{H_2CO_3} - K_{b,ca} c_{H^+} c_{HCO_3^-}) \quad (1.111)$$

Here we use the expressions:

$$K_{f,ca} = 10^{exp}\ s^{-1}, \text{ where } exp = 5.71 + 0.0526 T_C - 2.94 \times 10^{-4} T_C^2 + 7.91 \times 10^{-7} T_C^3 \quad (1.112)$$

in terms of the temperature in degrees Celsius, Tc. The steady-state value is taken to be

$$K_{ca} = \frac{K_{f,ca}}{K_{b,ca}} = 387.6 \times 10^{-exp} \text{ molar} \quad (1.113)$$

where $exp = (6.41 - 1.594 \times 10^{-3} T_f + 8.52 \times 10^{-6} T_f^2 - 3.07 \times 10^{-5} p - 0.4772 I^{0.5} - 0.118 I)$ in terms of pressure p (in psi), ionic strength I in molar and $T_f$, the temperature in degrees Fahrenheit.

***In the code example solved here $T_C$=20°C, p=14.9 psi, I=0.1711 and $T_f$=68°F.***

Step 4: Bicarbonate Ion Dissociation

This reaction is given by:

$$HCO_3^- \Leftrightarrow H^+ + CO_3^{2-} \qquad (1.114)$$

and leads to the differential equation

$$R_{HCO_3^-} = \frac{d}{dt}\left(c_{HCO_3^-}\right) = -(K_{f,bi}\, c_{HCO_3^-} - K_{b,bi} c_{H^+} c_{CO_3^{2-}}) \qquad (1.115)$$

in terms of the forward reaction rate $K_{f,bi}$, and backward reaction rate $K_{b,bi}$. Here,

$$K_{bi} = \frac{K_{f,bi}}{K_{b,bi}} = 10^{exp} \text{ molar} \qquad (1.116)$$

where

$$\exp=-(10.61-4.97\text{x}10^{-3}T_f+1.331\text{x}10^{-5}T_f^2-2.624\text{x}10^{-5}p-1.166I^{0.5}+0.3466I). \qquad (1.117)$$

The forward reaction rate: $K_{f,bi}=10^9\ s^{-1}$ is assumed to be independent of temperature.

***In the code example solved here p=14.9 psi, I=0.1711 and $T_f$=68°F.***

Step 5: Water Dissociation

The water dissociation reaction is given by:

$$H_2O \Leftrightarrow H^+ + OH^- \qquad (1.118)$$

The rate of production of $H_2O$ is given by

$$R_{H_2O} = \frac{d}{dt}\left(c_{H_2O}\right) = -(K_{f,wa} c_{H_2O} - K_{b,wa} c_{H^+} c_{OH^-}) \qquad (1.119)$$

In terms of the forward reaction rate constant, $K_{f,wa}$, and the backward reaction rate constant $K_{b,wa}$. The rate of production of $H^+$ ions, $R_{H^+}$, and $OH^-$ ions, $R_{OH^-}$, from this reaction are given by

$$R_{H^+} = R_{OH^-} = -R_{H_2O}. \qquad (1.120)$$

$K_{wa} = 10^{-exp}$ molar$^2$, where $exp$ =(29.3868-0.0737549$T_K$+7.47881x10$^{-5}$ $T_K^2$)

in terms of the absolute temperature in Kelvin, $T_K$.

The backward reaction rate constant, $K_{b,wa}$ is independent of temperature and takes the value $K_{b,wa}=7.85\text{x}10^{10}M^{-1}s^{-1}$. This enables the forward reaction rate constant, $K_{f,wa}$ to be determined from

$$K_{f,wa} = K_{wa} \times K_{b,wa} \qquad (1.121)$$

***In the code example solved here $T_K$=293.15K.***

Step 6: Charge Balance

The final relationship is obtained by the assumption of charge balance throughout the bulk liquid hence

$$C_{H^+} = C_{HCO_3^-} + 2C_{CO_3^{2-}} + C_{OH^-} \qquad (1.122)$$

The final set of equations are obtained by adding up all the reaction rates across all the reaction equations.

**Total Reaction Rate of $CO_2$**

$CO_{2\ only}$ reacts via carbon dioxide hydration, hence

$$R_{CO_2} = -R_{H_2CO_3} = \frac{d}{dt}\left(c_{CO_2}\right) = \left(K_{b,hy}\, c_{H_2CO_3} - K_{f,hy} c_{CO_2} c_{H_2O}\right) \qquad (1.123)$$

**Total Reaction Rate of $H_2CO_3$**

$H_2CO_3$ reacts via carbon dioxide hydration and carbonic acid dissociation. Hence,

$$R_{H_2CO_3} = \frac{dc_{H_2CO_3}}{dt} = -\left(K_{b,hy}\, c_{H_2CO_3} - K_{f,hy} c_{CO_2} c_{H_2O}\right) - \left(K_{f,ca}\, c_{H_2CO_3} - K_{b,ca} c_{H^+} c_{HCO_3^-}\right) \qquad (1.124)$$

**Total Reaction Rate of $\mathbf{HCO_3^-}$ ions**

$HCO_3^-$ ions are created by carbonic acid dissociation and bicarbonate ion dissociation. Hence,

$$R_{HCO_3^-} = \frac{dc_{HCO_3^-}}{dt} = \left(K_{f,ca}\, c_{H_2CO_3} - K_{b,ca} c_{H^+} c_{HCO_3^-}\right) - \left(K_{f,bi}\, c_{HCO_3^-} - K_{b,bi} c_{H^+} c_{CO_3^{2-}}\right) \qquad (1.125)$$

**Total Reaction Rate of $\mathbf{CO_3^{2-}}$ ions**

$CO_3^{2-}$ ions are only created by bicarbonate ion dissociation. Hence,

$$R_{CO_3^{2-}} = \frac{dc_{CO_3^{2-}}}{dt} = \left(K_{f,bi}\, c_{HCO_3^-} - K_{b,bi} c_{H^+} c_{CO_3^{2-}}\right) \qquad (1.126)$$

**Total Reaction Rate of $OH^-$ ions**

$OH^-$ ions are only created by water dissociation. Hence,

$$R_{OH^-} = \frac{dc_{OH^-}}{dt} = -R_{H_2O} = K_{f,wa} c_{H_2O} - K_{b,wa} c_{H^+} c_{OH^-} \qquad (1.127)$$

**Total Reaction Rate of $H^+$ ions**

$H^+$ ions are created by carbonic acid dissociation, bicarbonate ion dissociation and water dissociation. Hence,

$$R_{H^+} = \frac{dc_{H^+}}{dt} = \left(K_{f,ca}\, c_{H_2CO_3} - K_{b,ca} c_{H^+} c_{HCO_3^-}\right) + \left(K_{f,bi}\, c_{HCO_3^-} - K_{b,bi} c_{H^+} c_{CO_3^{2-}}\right) + \left(K_{f,wa} c_{H_2O} - K_{b,wa} c_{H^+} c_{OH^-}\right) \qquad (1.128)$$

**Numerical Solution of the Chemical Equations**

Introducing the chemical concentrations in SI units (mol/m$^3$):

$$x_1 = c_{CO_2}, x_2 = c_{H_2CO_3}, x_3 = c_{HCO_3^-}, x_4 = c_{CO_3^{2-}}, x_5 = c_{OH^-}, x_6 = c_{H^+}$$

and

$$c_1 = K_{b,hy}, c_2 = -K_{f,hy}, c_3 = K_{f,ca}, c_4 = \frac{-K_{b,ca}}{1000}, c_5 = -K_{f,bi}, c_6 = \frac{K_{b,bi}}{1000}, c_7 = K_{f,bi},$$

$$c_8 = \frac{-K_{b,bi}}{1000}, c_9 = 1000\, K_{f,wa}, c_{10} = \frac{-K_{b,wa}}{1000}$$

gives the following system of coupled first order odes for the concentrations in SI units:

$$\frac{dx_1}{dt} = c_1 x_2 + c_2 x_1 \quad (1.129)$$

$$\frac{dx_2}{dt} = -c_1 x_2 - c_2 x_1 - c_3 x_2 - c_4 x_3 x_6 \quad (1.130)$$

$$\frac{dx_3}{dt} = c_3 x_2 + c_4 x_3 x_6 + c_5 x_3 + c_6 x_4 x_6 \quad (1.131)$$

$$\frac{dx_4}{dt} = c_7 x_3 + c_8 x_4 x_6 \quad (1.132)$$

$$\frac{dx_5}{dt} = c_9 + c_{10} x_5 x_6 \quad (1.133)$$

$$x_6 = x_6 + 2x_4 + x_5 \quad (1.134)$$

These should be integrated subject to the initial conditions, at t=0:

$$x_1 = 1000\, K_{sol} p_{CO_2}, x_2 = 1000\, K_{hy} K_{sol} p_{CO_2}, x_3 = x_4 = x_5 = x_6 = 0 \quad (1.135)$$

The coupled system of odes (1.129)-(1.134) are solved subject to the initial conditions (1.135) using the Python programs in the main, driver program, *v1homogeneous_chemistry.py* using the functions contained in *chemical_functions.py*. These are available under the *Fig1_10* directory.

**v1homogeneous_chemistry.py**

```
# Program to calculate homogeneous chemistry based on theory developed by Nesic
# and co-workers. Theory being used from following references:
# Nordsveen et al. Corrosion 59(5), 443-456.

import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint
from chemical_functions import set_chemistry, dxdt_5_pHvariable

# define chemistry data dictionary
chemparam = set_chemistry()
cFe = 0.0   # Fe2+ concentration = 0 in bulk solution

# define arrays for solution of chemical equations
ndim = 5  # number of independent chemical reactions

# create constants for use in the implicit solution scheme
c1 = chemparam['c1']
c2 = chemparam['c2']
c3 = chemparam['c3']
```

```
c4 = chemparam['c4']
c5 = chemparam['c5']
c6 = chemparam['c6']
c7 = chemparam['c7']
c8 = chemparam['c8']
c9 = chemparam['c9']
c10 = chemparam['c10']

def f(y, t):

        x1 = y[0]   # concentration of cO2 (mol/m3)
        x2 = y[1]   # concentration of H2CO3 (mol/m3)
        x3 = y[2]   # concentration of HCO3- (mol/m3)
        x4 = y[3]   # concentration of CO32- (mol/m3)
        x5 = y[4]   # concentration of OH- (mol/m3)

        [f0,f1,f2,f3,f4]= dxdt_5_pHvariable(x1,x2,x3,x4,x5,chemparam,0.0)
        return [f0, f1, f2, f3, f4]

# initialise the concentrations CO2, H2CO3, HCO3-, CO32- and COH-. in practice the
# solutions are not found to be sensitive to these specific values
xb0 = 10**-4
cCO2_init = chemparam['cCO2_init']
y0 = [cCO2_init, xb0, xb0, xb0, xb0]

# t  = np.linspace(0,1)   # time grid
t  = np.linspace(0,1.0,101)   # time grid

# solve the time dependent chemical equations using PYTHON functions
soln = odeint(f, y0, t)
cCO2 = soln[:, 0]   # concentration of CO2 at ith time step
cH2CO3 = soln[:, 1]   # concentration of H2CO3 at ith time step
cHCO3 = soln[:, 2]   # concentration of HCO3- at ith time step
cCO3 = soln[:, 3]   # concentration of CO32- at ith time step
cOH = soln[:, 4]   # concentration of OH-- at ith time step
# calculate concentration of H+ by  charge balance equation
cH = np.zeros(len(cOH),dtype=float)
for j in range(len(cOH)):
```

```
    cH[j] = cHCO3[j] + 2*cCO3[j] + cOH[j]

# set steady state concentrations in bulk
cCO2_steady = cCO2[-1]
cH2CO3_steady = cH2CO3[-1]
cHCO3_steady = cHCO3[-1]
cCO3_steady = cCO3[-1]
cOH_steady = cOH[-1]
cH_steady = cH[-1]
pH_steady = -np.log10(cH_steady/1000)
print("cCO2 steady = {0:10.5e}".format(cCO2_steady))
print("cH2CO3 steady = {0:10.5e}".format(cH2CO3_steady))
print("CHCO3 steady = {0:10.5e}".format(cHCO3_steady))
print("cCO3 steady = {0:10.5e}".format(cCO3_steady))
print("cOH steady = {0:10.5e}".format(cOH_steady))
print("cH steady = {0:10.5e}".format(cH_steady))
print("pH steady = {0:10.5e}".format(pH_steady))

# plot out chemical concentrations
plt.ion()
fig=plt.figure()
plt.semilogy(t,cCO2,'k-',markersize=5,label='CO2')
plt.semilogy(t,cH2CO3,'b-',markersize=5,label='H2CO3')
plt.semilogy(t,cHCO3,'r-',markersize=5,label='HC03-')
plt.semilogy(t,cCO3,'g-',markersize=5,label='C032-')
plt.semilogy(t,cOH,'m-',markersize=5,label='OH-')
plt.semilogy(t,cH,'y-',markersize=5,label='H+')

plt.legend(loc='best')
plt.xlabel('time (secs)',style='italic')
plt.ylabel('Species concentrations (mol/m3)',style='italic')
```

### chemical_functions.py

```
# functions used to set us equations for homoegeneous chemical reactions
import numpy as np
def dxdt_5_pHvariable(x1,x2,x3,x4,x5,chemparam,cFe):

    # set parameters from dictionary
```

```
    c1 = chemparam['c1']
    c2 = chemparam['c2']
    c3 = chemparam['c3']
    c4 = chemparam['c4']
    c5 = chemparam['c5']
    c6 = chemparam['c6']
    c7 = chemparam['c7']
    c8 = chemparam['c8']
    c9 = chemparam['c9']
    c10 = chemparam['c10']

    x6 = x3 + 2*x4 + x5 - 2*cFe  # cH
    # rate of change of CO2
    dx1dt = c1*x2 + c2*x1
    # rate of change of H2CO3
    dx2dt = -dx1dt -c3*x2 -c4*x3*x6
    # rate of change of HCO3-
    dx3dt = c3*x2 + c4*x3*x6 + c5*x3 + c6*x4*x6
    # rate of change of CO32-
    dx4dt = c7*x3+c8*x4*x6
    # rate of change of OH-
    dx5dt = c9 + c10*x5*x6

    return [dx1dt,dx2dt,dx3dt,dx4dt,dx5dt]

# function to set chemical parameters needed in solutions of chemical equations
def set_chemistry():

    # pH not specified but determined from chemical reaction equations
    # set ionic strength
    I=0.1711   # value for 1% NaCl
    Tc=20
    pCO2=1   # 1 bar

    # total pressure P = PH2O + PCO2
    # PH2O calculated using steam tables at Tc. E.g. 25oC, PH2O = 0.0313 Atm
    P=14.8   # total pressure expressed in p.s.i.
    # PH=4     # specify pH and hence H+ concentration
```

```
T=Tc+273.15   # Kelvin
Tf=Tc*9/5 +32   # Fahrenheit
Tk=Tc+273.15   # Kelvin

# Calculate steady state chemistry in bulk solution
#*************************************************
#CO2 aqueous. Ksol taken from Nordsveen et al (2003), Table 2, p 447.
Ksol= (14.5/1.00258)*10**-(2.27+0.00565*Tf-8.06*(10**-6)*(Tf**2)
+0.075*I)

#dissociation of water. Reaction constants taken from Nordsveen et al (2003),
# table 2, p 447.
Kwa= 10**-(29.3868-0.0737549*Tk+7.47881*(10**-5)*Tk**2)
Kbwa= 7.85* 10**10 *(Tc/Tc)
Kfwa=Kwa*Kbwa

#hydration of H2CO3. Reaction constants taken from Nordsveen et al (2003),
# table 2, p 447.
Khy=2.58*(10**-3) *(Tc/Tc)
Kfhy=10**(329.85-110.541*np.log10(Tk)-(17265.4/Tk))
Kbhy=Kfhy/Khy

#dissociation of H2CO3. Reaction constants taken from Nordsveen et al (2003),
# table 2, p 447.
Kca=387.6*10**-(6.41-1.594*(10**(-3))*Tf
+(8.52* 10**(-6) *Tf**(2))-3.07*10**(-5)*P-0.4772*(I**0.5)+0.118*I)
Kfca= 10**(5.71+0.0526*Tc-2.94*10**-4 *Tc**2 +7.91*10**-7*Tc**3)
Kbca= (Kfca/Kca)

#dissociation of HCO3-(matched depends on variables). Reaction constants
# taken from Nordsveen et al (2003), table 2, p 447.
Kbi = 10**-(10.61-4.97*(10**-3)*Tf+1.331*(10**-5)*(Tf**2)-2.624*(10**-5)*P-
1.166*(I**0.5)+0.3466*I)
Kfbi=10**9 *(Tc/Tc)
Kbbi= Kfbi/Kbi

# calculate steady state concentrations of CO2, H2CO3 in mol/m3
```

```
cCO2_init = pCO2*Ksol*1000
cH2CO3_init = pCO2*Ksol*Khy*1000

# create constants for use in the implicit solution scheme
c1 = Kbhy
c2 = -Kfhy
c3 = Kfca
c4 = -Kbca*0.001
c5 = -Kfbi
c6 = Kbbi*0.001
c7 = Kfbi
c8 = -Kbbi*0.001
c9 = Kfwa*1000
c10 = -Kbwa*0.001

# set chemistry dictionary values
chemparam = dict([('I',I),('Tc',Tc),('pcO2',pCO2),('P',P),('T',T),('Tf',Tf),
                  ('Tk',Tk),('cCO2_init',cCO2_init),('cH2CO3_init',cH2CO3_init),
                  ('c1',c1),('c2',c2),('c3',c3),('c4',c4),('c5',c5),('c6',c6),
                  ('c7',c7),('c8',c8),('c9',c9),('c10',c10)])
return chemparam
```

This results in the following figure, showing the evolution of each of the chemical species towards a steady state.

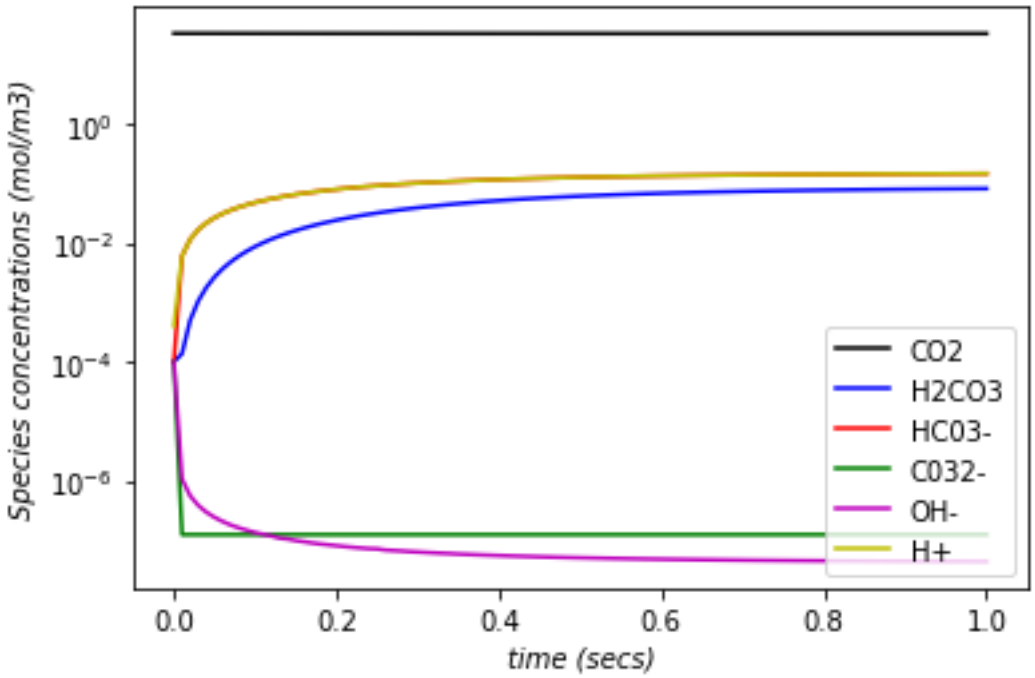


Figure 1.10: Evolution of the chemical species during the homogeneous chemical reactions during $CO_2$ corrosion, obtained using v1homogeneous_chemistry.py and chemical_functions.py.

## 1.5 Modelling of Flow-Induced Vibration (FIV)

### 1.5.1 Introduction

Pipelines conveying fluids play a significant role in modern industry, including chemical processing, power generation and the transportation of commodities such as oil and gas. These are safety critical since system failures can lead to the spillage of fluids which are detrimental to human health and the environment. They are also of enormous economic significance too, so there is clearly a need to identify and address the root causes of failure in pipeline systems. Flow-induced vibration fatigue and failure is one of the most common causes of failure in pipeline systems, accounting for more than 15% of all pipeline failures in Western Europe, Mpofu (2023).

In many practical applications, the operators aim to operate at the highest possible flow velocities to maximise production and profits, however these are associated with more severe vibrations and therefore higher pipe failure rates, Paidoussis (2008). Hence it is very important to be able to identify the safe and optimal operating ranges for a piping system. Studies show that vibration of pipelines results from interactions between a fluid and a structural component, and are thereby influenced by the physical and structural properties of both the fluid and the pipe. The support conditions for the pipes are also found to be very important too. The most stable case is the *clamped-clamped* condition where the ends of the pipes are rigidly attached to physical supports, whereas the least stable support method is the *simply supported-simply supported* condition, where the pipe simply lies on top of a support structure but is not rigidly attached to it. The precise form of support method has a large influence on key aspects of pipeline stability, such as the critical flow velocity above which the pipeline becomes unstable and is therefore prone to large FIV and failure.

This section aims to give a brief introduction to mathematical and computational methods that can be used to analyse FIV and covers key areas of model development including model validation and verification.

### 1.5.2. Mathematical Modelling

A general equation of motion for the vibration of a pipe which is supported for 0≤x≤L is given by

$$EI\frac{\partial^4 y}{\partial x^4} + m_f V^2 \frac{\partial^2 y}{\partial x^2} + 2m_f V \frac{\partial^2 y}{\partial x \partial t} + \left(m_f + m_p\right)\frac{\partial^2 y}{\partial t^2} = 0 \quad (1.136)$$

where y(x,t) is the lateral displacement of the pipe at position x and time t, E is the Young's modulus of the pipe material (Pa), I is the moment of area of the pipe ($m^4$), $m_p$ is the mass of the pipe per unit length (kg/m), $m_f$ is the mass of the fluid per unit length (kg/m), V is the mean velocity of the fluid flow inside the pipe (m/s).

The first term $EI\frac{\partial^4 y}{\partial x^4}$ represents the flexural restoring force. The second term, $m_f V^2 \frac{\partial^2 y}{\partial x^2}$, centrifugal force due to flow in the curved pipe, the third, $2m_f V \frac{\partial^2 y}{\partial x \partial t}$, the Coriolis force arising from the relative motion of the pipe and the fluid and the final term, $\left(m_f + m_p\right)\frac{\partial^2 y}{\partial t^2}$, the inertial force of the pipe and fluid system.

This mathematical model is based on the assumptions that:

1. The pipe behaves like a perfectly elastic beam so that the Young's modulus is constant
2. The pipe is slender which implies that the amplitude of vibration is small compared to the length
3. The fluid flow is fully developed
4. The fluid is incompressible

Although the fluid is not idealised as inviscid, the equation does not have any term with a viscosity coefficient. It has been shown by Paidoussis (2008) that fluid viscosity does not have a significant effect on the motion.

A further key simplifying assumption is that the Coriolis force term $2m_f V \frac{\partial^2 y}{\partial x \partial t}$ can be neglected in comparison with the other three terms in the equation of motion. This can be justified since many previous studies have neglected the Coriolis term from the equation of motion, e.g. Udoetek (2018) and Yi-min et al (2010), and have found that this leads to an error typically less than 3% in the prediction of natural frequencies that are needed to predict the onset of FIV instabilities.

The next section will consider the *simply supported-simply supported* condition.

**In your assignment, you will develop the equivalent mathematical model and Python programs for *clamped-clamped* conditions.**

### 1.5.2. Mathematical Modelling of the Simply Supported-Simply Supported Case

The boundary conditions are obtained from the nature of the end supports. The case considered here is with a simply supported-simply supported pipe as shown in Figure 1.11.

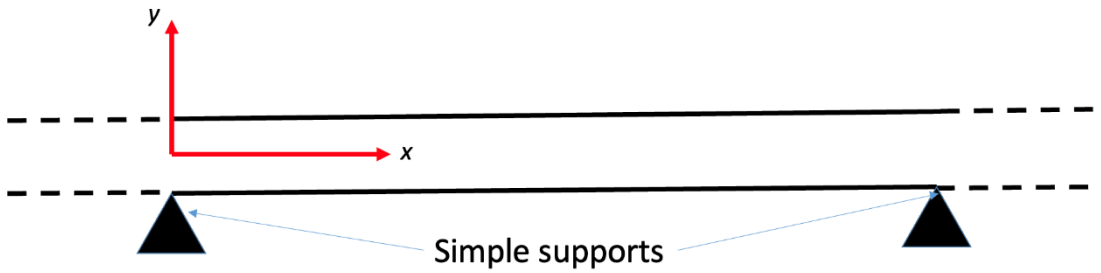


Fig 1.11: A simply supported-simply supported pipe system.

The boundary conditions to be applied in this case are: $y = \frac{\partial^2 y}{\partial x^2} = 0$ at the ends x=0, x=L. Note the second derivative condition indicates that the *bending moment=0* at the simply-supported ends.

Finite Difference Discretisation

The finite difference method was used to represent the continuous pipe system as a discrete system. A uniform discretisation was used for the time and spatial domains, as shown in Figure 1.12. Finite Difference approximations were applied at these discrete nodes.

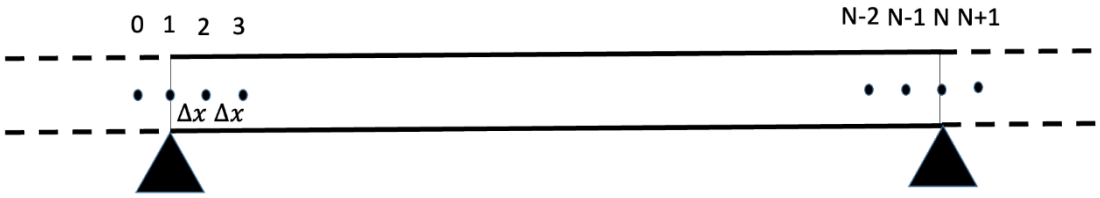


Fig 1.12: Uniform spatial discretisation of a simply-supported pipe system.

Second order discretisations for the second and fourth order derivative terms are used:

$$\frac{d^2y}{dx^2} \approx \frac{y_{i-1} - 2y_i + y_{i+1}}{(\Delta x)^2} \qquad (1.137)$$

$$\frac{d^4y}{dx^4} \approx \frac{y_{i-2} - 4y_{i-1} + 6y_i - 4y_{i+1} + y_{i+2}}{(\Delta x)^4} \qquad (1.138)$$

where $y_i$ represents the value of y at the ith node x=$x_i$ and Δx is the grid spacing. The same second order expression is used for the second order time derivative.

Computation of the Natural Frequency

Since the equation of motion varies in both space and time, the method of separation of variables can be used to obtain an equation in space from which the natural frequency was then computed using the finite difference method. This approach has been used successfully in similar beam models, Rao (2011). The solution $y(x,t)$ is written as

$$y(x,t) = Y(x)T(t) \qquad (1.139)$$

The equation of motion then becomes

$$EI\frac{d^4Y}{dx^4}T + m_f V^2\frac{d^2Y}{dx^2}T + \left(m_f + m_p\right)\frac{d^2T}{dt^2}Y = 0 \qquad (1.140)$$

This can be re-written as

$$\frac{1}{(m_f+m_p)Y}\left(EI\frac{d^4Y}{dx^4}T + m_f V^2\frac{d^2Y}{dx^2}\right) = -\frac{1}{T}\frac{d^2T}{dt^2} = constant = \lambda \qquad (1.141)$$

Hence

$$\frac{EI}{m_{tot}}\frac{d^4Y}{dx^4} + \frac{m_f V^2}{m_{tot}}\frac{d^2Y}{dx^2} - \lambda Y = 0 \qquad (1.142)$$

where $m_{tot} = m_f + m_p$ is the total mass of the pipe and fluid per unit length.

Applying a uniform spatial discretisation leads to:

$$\frac{EI}{m_{tot}}\left(\frac{Y_{i-2}-4Y_{i-1}+6Y_i-4Y_{i+1}+Y_{i+2}}{(\Delta x)^4}\right) + \frac{m_f V^2}{m_{tot}}\left(\frac{Y_{i-1}-2Y_i+Y_{i+1}}{(\Delta x)^2}\right) - \lambda Y_i = 0 \qquad (1.143)$$

Let $a = \frac{EI}{m_{tot}(\Delta x)^4}$ and $b = \frac{m_f V^2}{m_{tot}(\Delta x)^2}$ and simplifying the equation leads to:

$$aY_{i-2} + (b-4a)Y_{i-1} + (6a-2b-\lambda)Y_i + (b-4a)Y_{i+1} + aY_{i+2} = 0 \qquad (1.144)$$

Putting $C_1 = a, C_2 = b-4a, C_3 = 6a-2b, C_4 = b-4a, C_5 = a$ this equation can be rewritten as

$$C_1Y_{i-2} + C_2Y_{i-1} + (C_3-\lambda)Y_i + C_4Y_{i+1} + C_5Y_{i+2} = 0 \qquad (1.145)$$

This was then applied to the internal nodes i=2 to i=N-1, giving the following linear equations:

For i=2: $$C_1Y_0 + C_2Y_1 + (C_3-\lambda)Y_2 + C_4Y_3 + C_5Y_4 = 0 \qquad (1.146)$$

For i=3: $$C_1Y_1 + C_2Y_2 + (C_3-\lambda)Y_3 + C_4Y_4 + C_5Y_5 = 0 \qquad (1.147)$$

For i=4 to i= N-3:

$$C_1Y_{i-2} + C_2Y_{i-1} + (C_3-\lambda)Y_i + C_4Y_{i+1} + C_5Y_{i+2} = 0 \qquad (1.148)$$

For i=N-2

$$C_1Y_{N-4} + C_2Y_{N-3} + (C_3-\lambda)Y_{N-2} + C_4Y_{N-1} + C_5Y_N = 0 \qquad (1.149)$$

For i=N-1

$$C_1Y_{N-3} + C_2Y_{N-2} + (C_3-\lambda)Y_{N-1} + C_4Y_N + C_5Y_{N+1} = 0 \qquad (1.150)$$

We can now simplify these using the boundary conditions, which are: $Y_1 = Y_N = 0$. The second condition is that the bending moment is zero at both simply-supported ends, leading to the second derivative boundary condition that $\frac{d^2Y}{dx^2} = 0$ at the clamped ends leads to:

$$\frac{Y_2-2Y_1+Y_0}{2\Delta x} = 0 \Rightarrow Y_2 = -Y_0 \qquad (1.151)$$

$$\frac{Y_{N+1}-2Y_N+Y_{N-1}}{2\Delta x} = 0 \Rightarrow Y_{N+1} = -Y_{N-1} \qquad (1.152)$$

The equations and boundary conditions can then be expressed in the following matrix form:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| $C_3$-$C_1$-λ | $C_4$ | $C_5$ | 0 | … | … | 0 | $Y_2$ | | 0 |

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| $C_2$ | $C_3$-λ | $C_4$ | $C_5$ | 0 | … | … | $Y_3$ | | 0 |
| $C_1$ | $C_2$ | $C_3$-λ | $C_4$ | $C_5$ | 0 | … | $Y_4$ | | 0 |
| 0 | … | … | … | … | … | … | $Y_5$ | | 0 |
| … | … | … | … | … | … | … | … | = | 0 |
| … | … | … | … | … | … | … | … | | 0 |
| … | 0 | $C_1$ | $C_2$ | $C_3$-λ | $C_4$ | $C_5$ | $Y_{N-3}$ | | 0 |
| … | … | 0 | $C_1$ | $C_2$ | $C_3$-λ | $C_4$ | $Y_{N-2}$ | | 0 |
| 0 | … | … | 0 | $C_1$ | $C_2$ | $C_3$-$C_5$-λ | $Y_{N-1}$ | | 0 |

This is a sparse pentadiagonal matrix of size (N-2)x(N-2) which can be expressed in the form M – $\lambda I$ where I is the identify matrix and M is an (N-2)x(N-2) matrix given by:

| | | | | | | |
|---|---|---|---|---|---|---|
| $C_3$-$C_1$ | $C_4$ | $C_5$ | 0 | … | … | 0 |
| $C_2$ | $C_3$ | $C_4$ | $C_5$ | 0 | … | … |
| $C_1$ | $C_2$ | $C_3$ | $C_4$ | $C_5$ | 0 | … |
| 0 | … | … | … | … | … | … |
| … | … | … | … | … | … | … |
| … | … | … | … | … | … | … |
| … | 0 | $C_1$ | $C_2$ | $C_3$ | $C_4$ | $C_5$ |
| … | … | 0 | $C_1$ | $C_2$ | $C_3$ | $C_4$ |
| 0 | … | … | 0 | $C_1$ | $C_2$ | $C_3$-$C_5$ |

It can be observed that the constant $\lambda$ is the eigenvalue which is equal to $\omega^2$, where $\omega$ is the natural frequency. The natural frequencies are found by solving the eigenvalue problem $[M - \lambda I] = 0$ so that

$$\omega_i = \sqrt{\lambda_i} \qquad (1.153)$$

**NOTE: The critical velocity at which the pipe loses stability is computed by the condition that one of natural frequencies becomes zero.**

A Python program was developed to determine the 1st and 2nd natural frequencies.

Verification and Validation of the Finite Difference Solver

The effect of grid density on the computed natural frequency for an experimental case due to Dodds & Runyan (1965), also reported in Dangal & Ghimire (2019), of an aluminium pipe with L=3.048m, E=68.9 GPa. I=8.73x10$^{-9}$ (kgm$^2$), $m_f$=0.38 kg/m, $m_{tot}$=0.715 kg/m, V=13.10 m/s. The grid convergence study in Figure 1.13 showed the effect of the number of nodes on the calculated natural frequency. This is obtained by running the Python program *gridconvergence_FIV_natural_frequency_fdm.py* in the *Fig1_13* directory.

**gridconvergence_FIV_natural_frequency_fdm.py**

```
"""
 FIV_natural_frequency_fdm.py
 Carries of grid convergence studies for the  natural frequencies in a
 FIV system with simply supported-simply supported ends
"""
```

```
import numpy as np

#############################################################################
def gridconv_natural_frequencies(E,L,I,mtot,mf,V,Nx):

    #############################################################################
    # discretisation of pipe
    deltax = L/(Nx-1)    # spatial grid spacing

    #############################################################################
    # defining the constants for the set of linear finite difference euqations
    a = (E*I)/(mtot*(deltax)**4)
    b = (mf*(V**2))/(mtot*(deltax)**2)
    C1 = a
    C2 = b-4*a
    C3 = 6*a-2*b
    C4 = C2
    C5 = C1

    # creation of FDM matrix M
    n = Nx-2
    M = np.zeros([n,n])
    M[0][0]=C3-C1
    M[0][1]=C4
    M[0][2]=C5

    M[1][0]=C2
    M[1][1]=C3
    M[1][2]=C4
    M[1][3]=C5

    for k in range(2,n-2):
        M[k][k-2]=C1
        M[k][k-1]=C2
        M[k][k]=C3
        M[k][k+1]=C4
        M[k][k+2]=C5
```

```
    M[n-2][n-4]=C1
    M[n-2][n-3]=C2
    M[n-2][n-2]=C3
    M[n-2][n-1]=C4
    M[n-1][n-3]=C1
    M[n-1][n-2]=C2
    M[n-1][n-1]=C3-C5

    # calculate natural frequencies from the real parts of the first two eigenvalues of M
    evals, evecs = np.linalg.eig(M)
    omega_natural = np.sqrt(evals)
    omega_1 = omega_natural[n-1].real
    omega_2 = omega_natural[n-2].real

    return omega_1,omega_2

# problem parameters from Dangal & Ghimire
steel_pipe = 1; aluminium_pipe = 2; CPVC_pipe = 3
material = 2
V = 13.1
if (material == steel_pipe):
    # Steel pipe
    E = 207e9       # Young's modules of pipe (Pa)
    L = 3.048       # pipe clamp spacing (m)
    I = 8.73e-9     # moment of area of pipe (m^4)
    mtot = 1.386    # total mass of pipe and fluid per unit length (kg/m)
    mf = 0.38       # mass of fluid per unit length (kg/m)
elif (material == aluminium_pipe):
    # Aluminium pipe
    E = 68.9e9       # Young's modules of pipe (Pa)
    L = 3.048       # pipe clamp spacing (m)
    I = 8.73e-9     # moment of area of pipe (m^4)
    mtot = 0.715    # total mass of pipe and fluid per unit length (kg/m)
    mf = 0.38       # mass of fluid per unit length (kg/m)
elif (material == CPVC_pipe):
    # CPVC pipe
    E = 2.9e9       # Young's modules of pipe (Pa)
    L = 3.048       # pipe clamp spacing (m)
```

```
    I = 8.73e-9     # moment of area of pipe (m^4)
    mtot = 0.574    # total mass of pipe and fluid per unit length (kg/m)
    mf = 0.38       # mass of fluid per unit length (kg/m)

##########################################################################
# discretisation of pipe
Nx = [10,20,30,40,50,60,70]

# allocate vectors to store errors for every run
omega_1 = np.zeros(len(Nx))

for n in range(len(Nx)):
    om_1,om_2 = gridconv_natural_frequencies(E,L,I,mtot,mf,V,Nx[n])
    omega_1[n] = om_1

print('Fine grid natural frequency= {0:6.3f}'.format(omega_1[len(Nx)-1]))

# plot out results
import matplotlib.pyplot as plt
plot1 = plt.figure(1)
plt.plot(Nx,omega_1,'b-o')
plt.xlabel('Number of nodes along the pipe')
plt.ylabel('Natural frequency (rad/s)')
plt.text(20,29.05,'fine grid frequency={0:6.2f}'.format(omega_1[len(Nx)-1]))
plt.title('Natural Frequencies of a simply-supported aluminium pipe, V=13.10m/s')
plt.savefig('al_gridconvergence_FIV_natural_frequency_fdm.jpg')

# Calculation of the order of accuracy of the method
Nxcoarse = 40; Nxmedium = 80; Nxfine = 160; r = 0.5;
omega_coarse,om_2 = gridconv_natural_frequencies(E,L,I,mtot,mf,V,Nxcoarse)
omega_medium,om_2 = gridconv_natural_frequencies(E,L,I,mtot,mf,V,Nxmedium)
omega_fine,om_2 = gridconv_natural_frequencies(E,L,I,mtot,mf,V,Nxfine)
p = np.log((omega_fine-omega_medium)/(omega_medium-omega_coarse))/np.log(r)
print('Estimate of order of convergence, p = {0:6.3f}'.format(p))
```

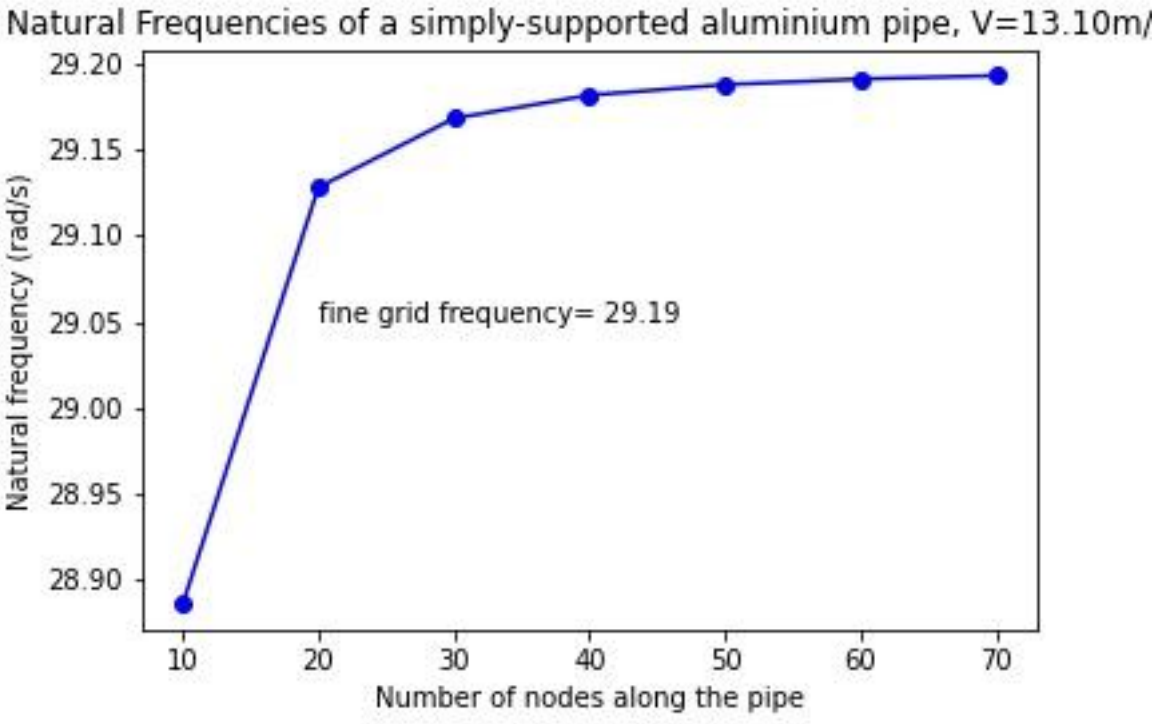


Figure 1.13: Effect of grid density on the natural frequency of a simply supported-simply supported Aluminium pipe considered by Dodds & Runyan (1965).

It can be seen that the solutions have converged for Nx>50.

The computed fine grid natural frequency of 29.19 rad/s compares with the experimental value of Dodds & Runyan (1965) of 26.10 rad/s: an error of 11.8%. The other experimental cases considered by Dodds & Runyan (1965) are for:

- V=23.485 m/s, the predicted value of 25.25 rad/s compares with the experimental value of 24.11 rad/s – a 4.7% error
- V=29.722 m/s, the predicted value of 21.22 rad/s compares with the average experimental value of 19.93 rad/s – a 6.5% error.

These can be summarised in the following table, which also includes the numerical predictions of Dangal & Ghimire (2019) who used a Finite Element method. Table 1 shows that the numerical method is generally reasonably accurate and compares well with previous relevant studies.

| Flow velocity m/s | Experiment (Dodds & Runyan (1965) rad/s | Finite Element (Dangal & Ghimire (2019) rad/s | Finite Difference model rad/s |
|---|---|---|---|
| 13.10 | 26.10 | 29.00 | 29.19 |
| 23.485 | 24.11 | 24.73 | 25.25 |
| 29.722 | 19.93 | 20.47 | 21.11 |

Table 1: Validation of effect of flow velocity on natural frequency for a simply supported-simply supported pipe.

Order of Accuracy of the Numerical Method

The order of accuracy of the numerical method can be estimated as follows. If $\omega_{fine}$, $\omega_{medium}$ and $\omega_{coarse}$ refer to the natural frequencies calculated using three different grid levels with the same node spacing reduction rate, r, such that the respective grid spacings are $\Delta x_{fine} = r^2 \Delta x_{coarse}$, $\Delta x_{medium} = r\,\Delta x_{coarse}$, then if $\omega_{actual}$ is the actual value of the natural frequency then

$$\frac{\omega_{fine} - \omega_{medium}}{\omega_{medium} - \omega_{coarse}} = \frac{(\omega_{fine} - \omega_{actual}) - (\omega_{medium} - \omega_{actual})}{(\omega_{medium} - \omega_{actual}) - (\omega_{coarse} - \omega_{actual})}$$
$$= \frac{C(r^2\Delta x_{coarse})^p - C(r\Delta x_{coarse})^p}{C(r\Delta x_{coarse})^p - C(\Delta x_{coarse})^p} = \frac{r^{2p} - r^p}{r^p - 1} = r^p$$

Hence, taking logarithms of both sides, gives

$$p = \frac{ln\left(\frac{\omega_{fine} - \omega_{medium}}{\omega_{medium} - \omega_{coarse}}\right)}{\ln r}$$

Comparing the errors using 40, 80 and 160 nodes for the coarse, medium and fine grids, gives an estimate of p=2.04 => indicating it is second order accurate, as expected since these are based on second order finite difference approximations.

Critical Velocities

As the velocity of the fluid in the pipe increases, the natural frequency decreases. The *critical velocity for instability* is the velocity for which the natural frequency first becomes equal to zero. The analysis for predicting the natural frequency described above can then be used to predict the critical velocity.

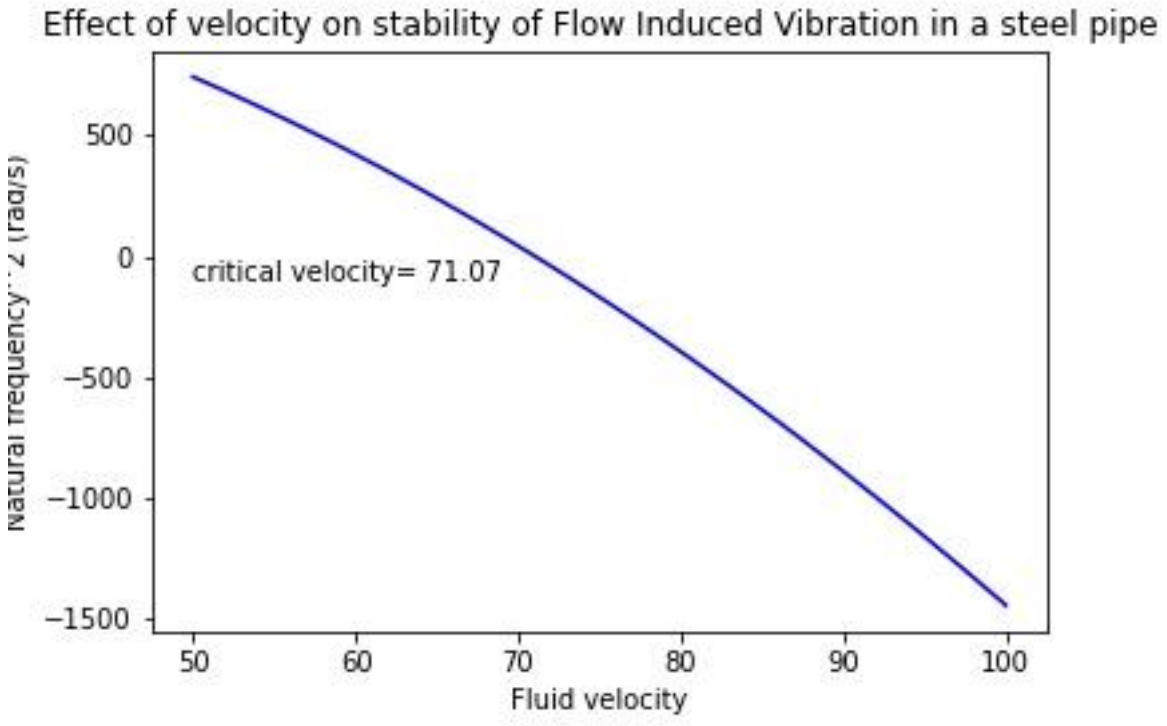


Figure 1.14: Effect of flow velocity on the natural frequency of a simply supported-simply supported steel pipe considered by Dangal & Ghimire (2019).

The following table compares the predictions of critical velocities for the simply-supported simply-supported case, using the present Finite Difference model with Nx=60 nodes (implemented in the Python program *criticalvelocity_FIV_fdm.py* on the *Fig1_14* directory) against the Finite Element predictions of Dangal & Ghimire (2019).

| Critical velocity m/s | Finite Element (Dangal & Ghimire (2019) | Finite Difference model |
|---|---|---|
| Steel pipe | 70.72 | 71.07 |
| Aluminium pipe | 41.25 | 41.00 |
| CPVC pipe | 8.46 | 8.41 |

Table 2: Comparison of the predictions of critical velocities for simply supported-simply supported steel, aluminium and CPVC pipes, with Nx=60 against the FE predictions of Dangal & Ghimire (2019).

Again, the agreement between the two methods is very good.

**criticalvelocity_FIV_fdm.py**

```
"""
 FIV_natural_frequency_fdm.py
 Calculates the critical velocity for instability of a
 FIV system with clamped-clamped ends
"""
import numpy as np

#########################################################################
def natural_frequencies(E,L,I,mtot,mf,V,Nx):

    #########################################################################
    # discretisation of pipe
    deltax = L/(Nx-1)    # spatial grid spacing

    #########################################################################
    # defining the constants for the set of linear finite difference euqations
    a = (E*I)/(mtot*(deltax)**4)
    b = (mf*(V**2))/(mtot*(deltax)**2)
    C1 = a
    C2 = b-4*a
    C3 = 6*a-2*b
    C4 = C2
    C5 = C1

    # creation of FDM matrix M
    n = Nx-2
```

```
    M = np.zeros([n,n])
    M[0][0]=C3-C1
    M[0][1]=C4
    M[0][2]=C5

    M[1][0]=C2
    M[1][1]=C3
    M[1][2]=C4
    M[1][3]=C5

    for k in range(2,n-2):
        M[k][k-2]=C1
        M[k][k-1]=C2
        M[k][k]=C3
        M[k][k+1]=C4
        M[k][k+2]=C5

    M[n-2][n-4]=C1
    M[n-2][n-3]=C2
    M[n-2][n-2]=C3
    M[n-2][n-1]=C4
    M[n-1][n-3]=C1
    M[n-1][n-2]=C2
    M[n-1][n-1]=C3-C5

    # calculate natural frequencies from the real parts of the first two eigenvalues of M
    evals, evecs = np.linalg.eig(M)
    omega_squared = evals[n-1]
    omega_natural = np.sqrt(evals)
    omega_1 = omega_natural[n-1].real
    omega_2 = omega_natural[n-2].real

    return omega_1,omega_2,omega_squared

# problem parameters from Dangal & Ghimire
steel_pipe = 1; aluminium_pipe = 2; CPVC_pipe = 3
material = 1;
if (material == steel_pipe):
```

```
    # Steel pipe
    E = 207e9       # Young's modules of pipe (Pa)
    L = 3.048       # pipe clamp spacing (m)
    I = 8.73e-9     # moment of area of pipe (m^4)
    mtot = 1.386    # total mass of pipe and fluid per unit length (kg/m)
    mf = 0.38       # mass of fluid per unit length (kg/m)
elif (material == aluminium_pipe):
    # Aluminium pipe
    E = 68.9e9       # Young's modules of pipe (Pa)
    L = 3.048       # pipe clamp spacing (m)
    I = 8.73e-9     # moment of area of pipe (m^4)
    mtot = 0.715    # total mass of pipe and fluid per unit length (kg/m)
    mf = 0.38       # mass of fluid per unit length (kg/m)
elif (material == CPVC_pipe):
    # CPVC pipe
    E = 2.9e9       # Young's modules of pipe (Pa)
    L = 3.048       # pipe clamp spacing (m)
    I = 8.73e-9     # moment of area of pipe (m^4)
    mtot = 0.574    # total mass of pipe and fluid per unit length (kg/m)
    mf = 0.38       # mass of fluid per unit length (kg/m)

########################################################################
# discretisation of pipe
Nx = 60

# allocate vectors to store errors for every run
Npts = 101
omega_squared = np.zeros(Npts)
V = np.linspace(50,100,Npts)

for n in range(Npts):
    om_1,om_2,om_squared = natural_frequencies(E,L,I,mtot,mf,V[n],Nx)
    omega_squared[n] = om_squared

# find velocity where square of natural frequency becomes negative
om1 = omega_squared[0]
crit_velocity = 0
for n in range(Npts-1):
```

```
    product = omega_squared[n]*omega_squared[n+1]
    if (product < 0):    # use linear interpolation for greater accuracy
        fac1 = np.abs(omega_squared[n])
        fac2 = np.abs(omega_squared[n+1])
        critical_velocity = (fac2*V[n]+fac1*V[n+1])/(fac1+fac2)
        break


# plot out results
import matplotlib.pyplot as plt
plot1 = plt.figure(1)
plt.plot(V,omega_squared,'b-')
plt.xlabel('Fluid velocity')
plt.ylabel('Natural frequency^2 (rad/s)')
plt.title('Effect of velocity on stability of Flow Induced Vibration in a steel pipe')
plt.text(50,-100,'critical velocity={0:6.2f}'.format(critical_velocity))
plt.savefig('steel_velocity_stability_fdm.jpg')
```

**YOUR NUMERICAL MODELLING ASSIGNMENT: THE CLAMPED-CLAMPED CASE**

To test your understanding of the numerical modelling concepts introduced above, and to give you some experience of developing your own Python programs, we would like you to extend the analysis and programs for the Simply Supported-Simply Supported Case to the case where the pipe is clamped at both ends – referred to as *Clamped-Clamped* conditions.

The boundary conditions are obtained from the nature of the end supports. The *clamped-clamped* case considered here is with fixed ends as shown in Figure 1.15.

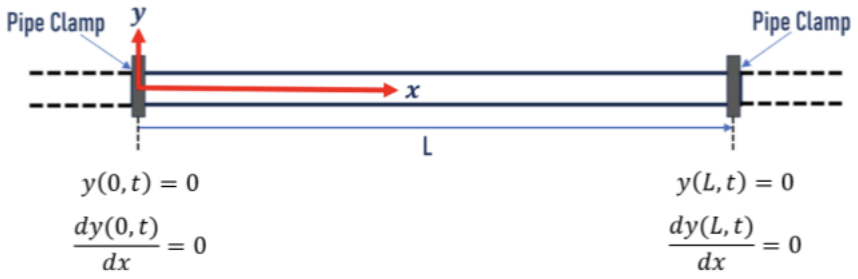


Fig 1.15: Pipe with clamped ends showing the boundary conditions at the clamps, Mpofu (2023).

Finite Difference Discretisation

The finite difference method was used to represent the continuous pipe system as a discrete system. Once again a uniform discretisation was used for the spatial domain, as shown in Figure 1.16. Finite Difference approximations were applied at these discrete nodes as described in the simply supported-simply supported case described above.

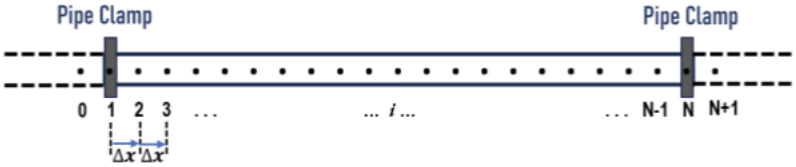


Fig 1.16: Uniform spatial discretisation of the clamped pipe system, Mpofu (2023).

The analysis proceeds as for the simply supported-simply supported case but with different boundary conditions. Using the boundary conditions, which are: $Y_1 = Y_N = 0$. The derivative boundary condition that $\frac{dY}{dx} = 0$ at the clamped ends leads to:

$$\frac{Y_2 - Y_0}{2\Delta x} = 0 => Y_2 = Y_0$$

$$\frac{Y_{N+1} - Y_{N-1}}{2\Delta x} = 0 => Y_{N+1} = Y_{N-1}$$

**For the clamped-clamped case, your tasks are to:**

- Derive the matrix form of the finite difference equations
- Write a Python program that plots out the natural frequency for between 10 and 70 nodes along the pipe
- Write a Python program that shows the effect of flow velocity on the natural frequency of the clamped-clamped steep pipe considered by Dangal & Ghimire (2019)
- Write a Python program that determines the critical velocity for the steel, Aluminimu and CPVC pipe considered by Dangal & Ghimire (2019)

**References**

Dangal, M, Ghimire, S.K. Modeling and Analysis of Flow Induced Vibration in Pipes using Finite Element Approach, *Proceedings of IOE Graduate Conference*, 6, 725-32, 2019.

Dodds, H.L., Runyan, H.L. Effect of high velocity fluid flow on the bending vibrations and static divergence of a simply-supported pipe, *NASA*, 1965.

Mpofu, B. Computational Modelling and Analysis of Flow Induced Vibrations in Piping, MSc project report, School of Mechanical Engineering, University of Leeds, 2023.

Paidoussis, M.P., The canonical problem of the fluid-conveying pipe and radiation of the knowledge gained to other dynamics problems across Applied Mechanics. *Journal of Sound and Vibration*, 310(3), 462-92, 2008.

Rao, SS *Mechanical Vibrations*, 5th edition, Prentice Hall, 2011.

Udoetek, E.S. Internal fluid flow induced vibration of pipes. Journal of Mechanical Design and Vibration, 6(1), 1-8, 2018.

Yi-min, H., Yong-shou, L., Bao-hui, L., Yan-jiang, L., Zhu-feng, Y. Natural frequency analysis of fluid conveying pipeline with different boundary conditions, *Nuclear Engineering and Design*, 240(3), 461-7, 2010.

Zhang, Y.L., Gorman, D.G., Reese, J.M. Analysis of the Vibration of Pipes Conveying Fluid, *Proc. Instn. Mech. Engrs*, 213, 849-860, 1999.