Simulation Tools
Project 3

Daniel Holm

February 2024

1 The second-order problem

For this project we are considering linear ordinary differential equations of the second order of the type

$$M\ddot{\mathbf{u}} + C\dot{\mathbf{u}} + K\mathbf{u} = \mathbf{f}(t), \tag{1}$$

where $\mathbf{u}$ is the vector of solutions, M, C, K are square matrices with constant coefficients and $\mathbf{f}(t)$ is a vector function of only time. If $\mathbf{u}$ is large it may be costly to transform this into a first-order problem since it doubles the system size. Instead we can use a single-step scheme that deals with the second-order system directly. In particular we will use the Newmark method.

2 Implementing the problem class

The problem class is intended to have the matrices M, C, K and the function $f(t)$ as parameters instead of the usual right-hand-side function `rhs` to define the problem. It also naturally needs the initial conditions $u(0)$ and $\dot{u}(0)$. In order to be compatible with built-in solvers that expect an `rhs` function this was implemented as well. The `rhs` function transforms the problem into a first-order problem the usual way. (See appendix for full code)

3 Implementing the solver classes

The structure of a solver class requires initializing correctly and implementing the `integrate` function. Inspiration was drawn from the BDF-solvers from project 1. In initialization we set up the matrix A as

$$A = \frac{M}{\beta h^2} + \frac{\gamma C}{\beta h} + K. \tag{2}$$

The solver takes implicit or explicit steps depending on if the conditions allow for it. The HHT-α solver was set up in a similar way according to the mathematical formulas.

3.1 Simulating a test problem

For the Newmark and HHT-α simulations a step-size of $h = 10^{-3}$ was used with $\beta = \frac{1}{4}$ and $\gamma = \frac{1}{2}$.

The first problem considered was taken from [1] (section 5.11.2) in order to compare solutions. The matrices were

$$M = \begin{bmatrix} 10 & 0 & 0 \\ 0 & 20 & 0 \\ 0 & 0 & 30 \end{bmatrix}, \quad K = 10^3 \begin{bmatrix} 45 & -20 & -15 \\ -20 & 45 & -25 \\ -15 & -25 & 40 \end{bmatrix}, \quad C = 3 \cdot 10^{-2} K, \tag{3}$$

and

$$\mathbf{f}(t) = \begin{bmatrix} 0 \\ 0 \\ g(t) \end{bmatrix}, \quad g(t) = \begin{cases} 50 \sin \frac{\pi}{0.3} t, & t < 0.3 \\ 0, & t \geq 0.3 \end{cases}, \tag{4}$$

and all initial conditions set to 0. The implemented Newmark solver seems to handle it well as seen in Figure 1. The HHT-α solver showed no noticeable differences either.

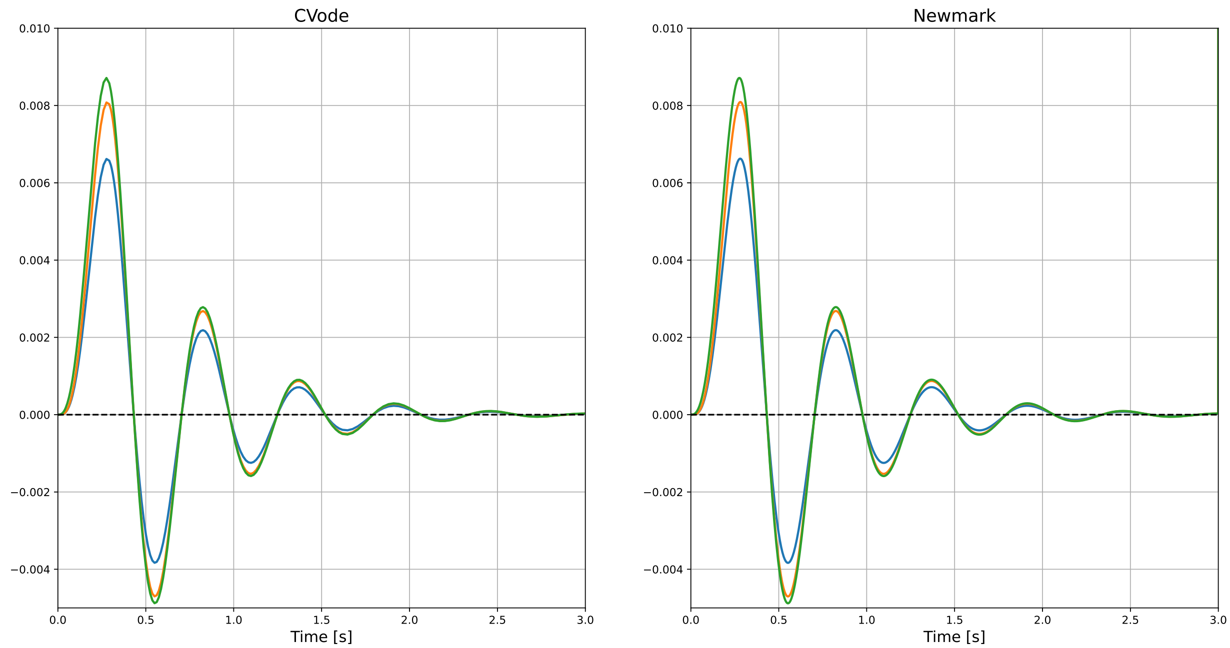


Figure 1

Trying with smaller step-sizes makes it go unstable however. Which is strange since the condition for unconditional stability appears to be met.

4 The elastic pendulum

The elastic pendulum is not a linear equation system so the explicit Newmark method needs to be modified. The equation system can be written in second-order form as

$$\begin{aligned} \ddot{x} + x\lambda(x, y) &= 0 \\ \ddot{y} + y\lambda(x, y) &= -1 \end{aligned} \tag{5}$$

where $\lambda(x, y) = k\frac{\sqrt{x^2+y^2}-1}{\sqrt{x^2+y^2}}$ for some spring constant k. This corresponds to $M = I, C = \mathbf{0}$. The modified scheme must reflect the non-linearity present as we can not simply multiply with some constant matrix K. For $\mathbf{u} = [x, y]$, the modified scheme is written

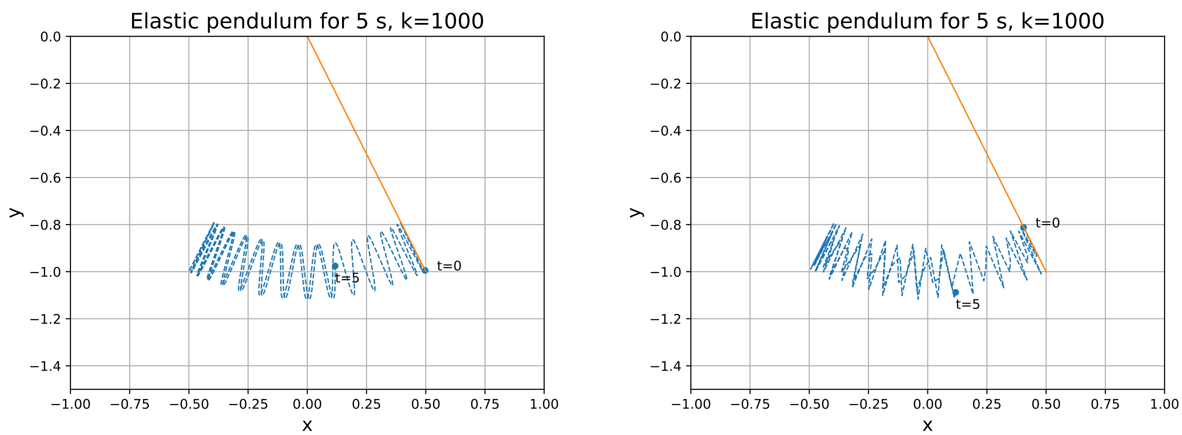
$$\mathbf{u}_{n+1} = \mathbf{u}_n + \dot{\mathbf{u}}_n h + \ddot{\mathbf{u}}_n \frac{h^2}{2} \tag{6}$$

$$\dot{\mathbf{u}}_{n+1} = \dot{\mathbf{u}}_n + \ddot{\mathbf{u}}_n \frac{h}{2} + \ddot{\mathbf{u}}_{n+1} \frac{h}{2} \tag{7}$$

$$\ddot{\mathbf{u}}_{n+1} = (\mathbf{f}_{n+1} - \lambda(x_{n+1}, y_{n+1})\mathbf{u}_{n+1}) \tag{8}$$

where $\mathbf{f}_n = [0, -1]$. Note that $M = M^{-1} = I$ and that this is the special case of $\gamma = \frac{1}{2}$. This also means we cannot use the newly written Newmark class and have to do a custom implementation.

Simulations were done for 5 seconds with the pendulum stretched out slightly to the side. In Figure 2a and Figure 2b we can see simulations for two different step-sizes.



(a) $h = 0.01$

(b) $h = 0.01$

Figure 2: Simulation of elastic pendulum with two different step-sizes h.

With the spring constant at $k = 1000$ it seems as if any step-size $h > 0.06$ makes the solution go wildly unstable. Keeping the step-size at $h = 0.01$ the Newmark method can handle even higher spring constants, see Figure 3

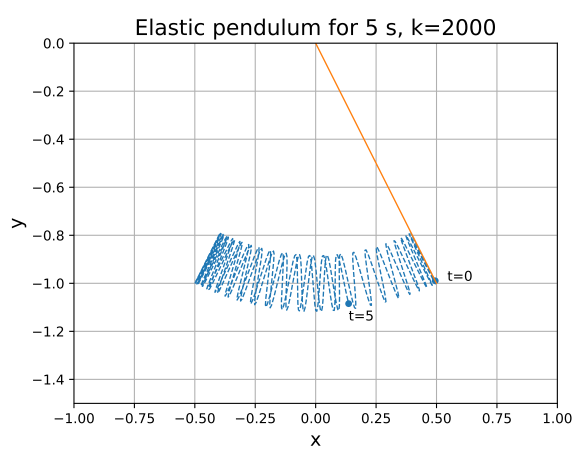


Figure 3: Simulation of elastic pendulum.

5 Simulating the elastodynamic beam problem

By extracting the matrices as in the instructions we can compare built-in methods with the second-order methods. Testing first with both $\eta_M = \eta_K = 0$, giving $C = \mathbf{0}$, i.e., no damping.

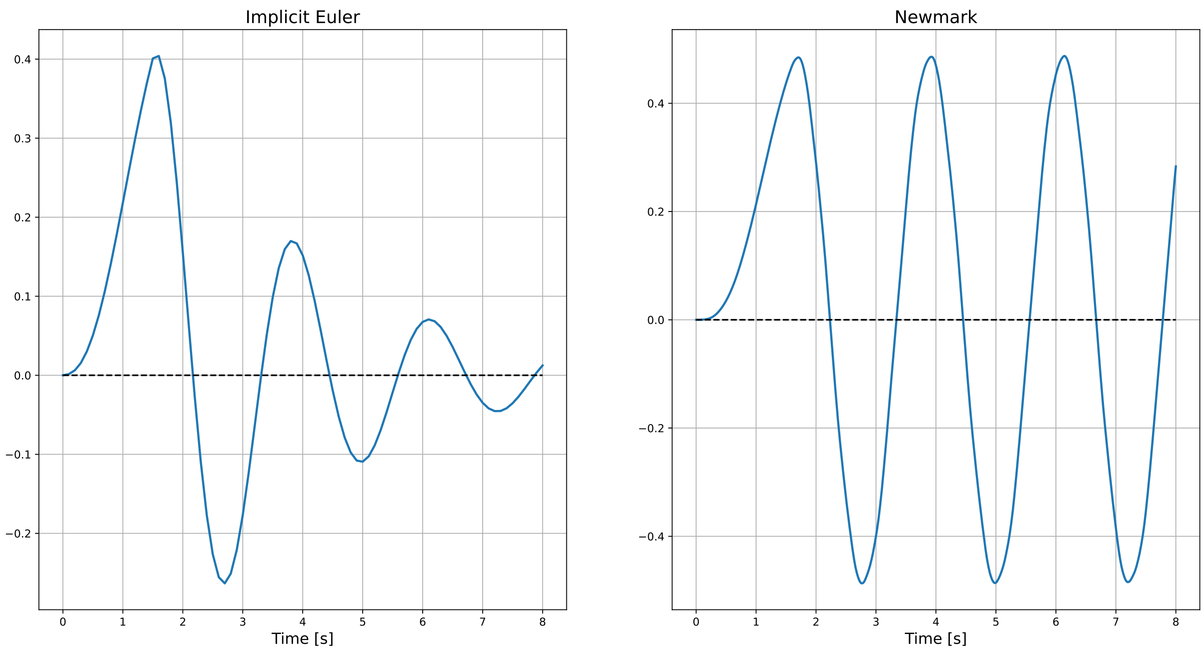


Figure 4: The tip of the elastodynamic beam with no damping.

Once again the Newmark solver used a step-size of $h = 10^{-3}$ with $\beta = \frac{1}{4}$ and $\gamma = \frac{1}{2}$. These are the conditions needed for unconditional stability. In Figure 4 we can see the strong numerical damping that the implicit Euler method introduces, whereas the Newmark method appears to correctly give an undamped oscillation. Notice especially the vertical axis ranges in Figure4 The step-size for the `ImplicitEuler` solver was set to $h = 0.1$. Testing with other built-in solvers such as `CVOde`, `ExplicitEuler`, `RungeKutta34` was less successful. They simply take way too long to find a solution. For `RungeKutta34` it reported that the final time could not be reached within the maximum number of steps after several minutes of execution time. The `ExplicitEuler` solver can find a solution but it is clearly unstable.

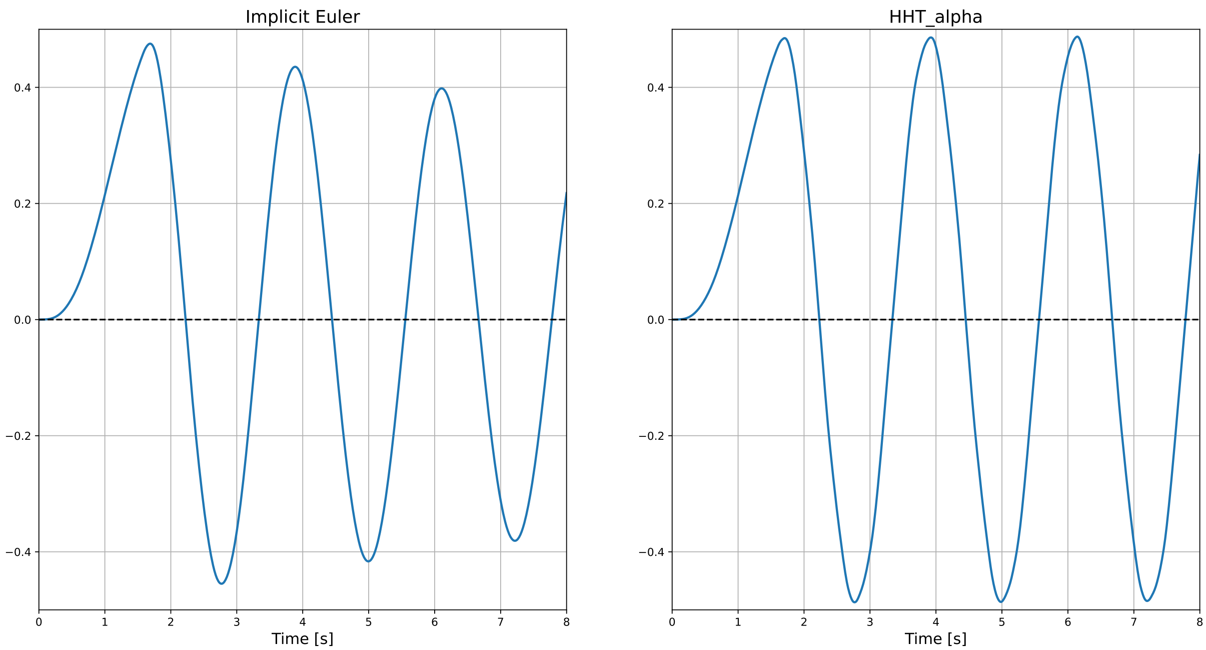


Figure 5: The tip of the elastodynamic beam with no damping.

The implicit Euler solver now used a smaller step-size of $h = 0.01$. The HHT$_\alpha$ method used a step-size of $h = 10^{-3}$ with $\alpha = -\frac{1}{3}$. With a lower step-size the implicit Euler solution comes much closer to the Newmark and HHT$_\alpha$ solutions. The cost however, is a much longer execution time. Perhaps unsurprisingly, the execution time increased nearly tenfold when the step-size was reduced by a factor of 10.

5.1 Introducing damping to the system

Damping was introduced to the system equations by setting $\eta_M = \eta_K = 0.1$. Solver settings were kept the same for Newmark and HHT-α.

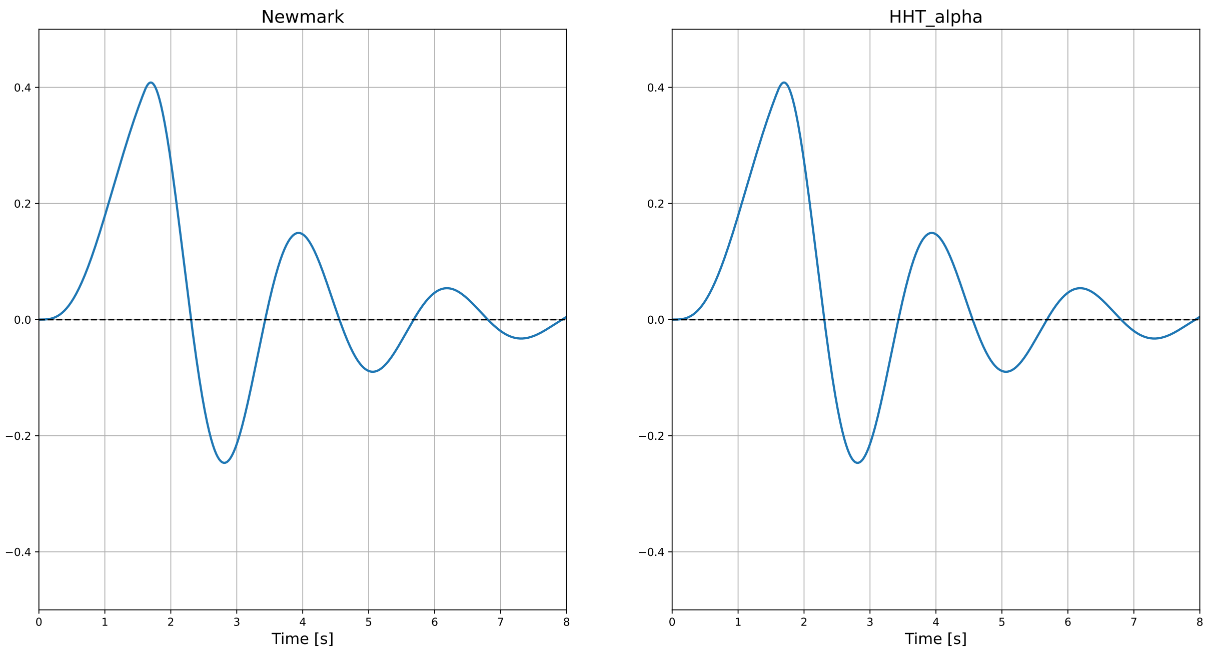


Figure 6: The tip of the elastodynamic beam with damping.

The same conditions were simulated with implicit Euler, step-size once again at $h = 0.01$.

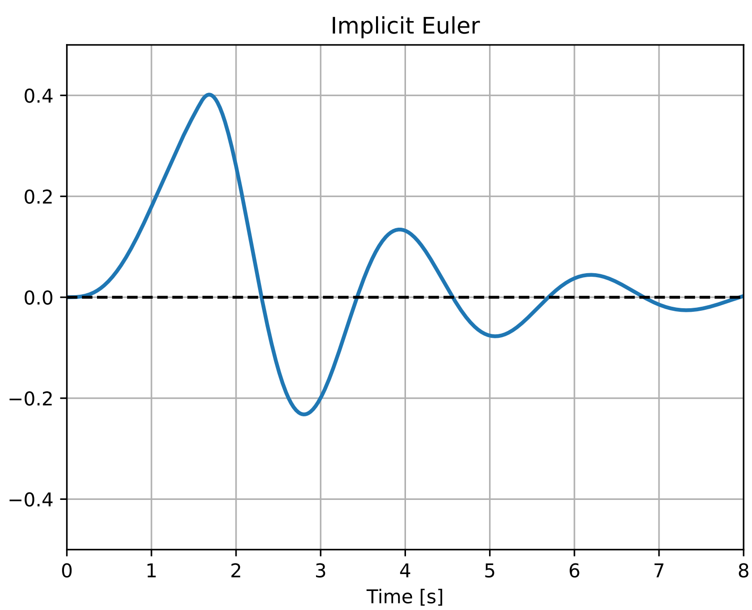


Figure 7: The tip of the elastodynamic beam with damping.

References

[1] George Lindfield and John Penny. "Chapter 5 - Solution of Differential Equations". In: *Numerical Methods (Fourth Edition)*. Ed. by George Lindfield and John Penny. Fourth Edition. Academic Press, 2019, pp. 239–299. ISBN: 978-0-12-812256-3. DOI: https://doi.org/10.1016/B978-0-12-812256-3.00014-2. URL: https://www.sciencedirect.com/science/article/pii/B9780128122563000142.

Code

The code was divided into one file `classes.py`, which contained the written solvers, and other files for the specific tasks. Some of the code is omitted, for example plot commands.

classes.py

```
from assimulo.explicit_ode import Explicit_ODE
from assimulo.problem import Explicit_Problem
from assimulo.ode import *
from numpy import hstack
from scipy.linalg import solve

import matplotlib.pyplot as mpl

class Explicit_Problem_2nd(Explicit_Problem):
        def __init__(self, M, C, K, f, u0, up0, t0):
                self.u0 = u0
                self.up0 = up0
                self.t0 = t0

                self.M = M
                self.C = C
                self.K = K
                self.f = f
                Explicit_Problem.__init__(self, self.rhs, hstack((self.u0, self.up0)), t0)

        def rhs(self, t, y):
                u = y[0:len(y)//2]
                up = y[len(y)//2:len(y)]

                y1dot = up
                y2dot = solve(self.M, -self.K@u - self.C@up + self.f(t))

                return hstack((y1dot, y2dot))

class Explicit_ODE_2nd(Explicit_ODE):
        def __init__(self, problem):
                Explicit_ODE.__init__(self, problem)
                self.M = problem.M
                self.C = problem.C
                self.K = problem.K
                self.f = problem.f
                self.u0 = problem.u0
                self.up0 = problem.up0
                self.t0 = problem.t0

```

```python
class Newmark(Explicit_ODE_2nd):
        gamma = 1/2
        Beta = 1/4
        h = 1e-3

        def __init__(self, problem):
                Explicit_ODE_2nd.__init__(self, problem)
                self.up = self.up0

                self.A = self.M / (self.Beta*self.h**2) + self.gamma*self.C / (self.Beta*self.h) + self.K

        def integrate(self, t0, u0, up0, tf, opts):
                h = min(self.h, abs(tf-t0))
                upp0 = solve(self.M, self.f(0) - self.K@u0)
                if self.C.any() != 0 and self.Beta != 0:
                        upp0 -= solve(self.M, self.C@up0)

                tres = []
                ures = []

                t, u, up, upp = t0, u0, up0, upp0

                while t < tf:
                        if self.C.all() == 0 and self.Beta == 0:
                                t, u, up, upp = self.explicit_step(t, u, up, upp, h)
                        else:
                                t, u, up, upp = self.implicit_step(t, u, up, upp, h)

                        tres.append(t)
                        ures.append(u.copy())

                        h = min(self.h, abs(tf-t))

                return ID_PY_OK, tres, ures

        def simulate(self, tf):
                flag, t, u = self.integrate(self.t0, self.u0, self.up0, tf, opts=None)
                return t, u

        def explicit_step(self, t, u, up, upp, h):
                u_next = u + up*h + upp*h**2/2
                upp_next = solve(self.M, self.f(t) - self.K@u)
                up_next = up + upp*h*(1-self.gamma) + self.gamma*upp_next*h

                return t+h, u_next, up_next, upp_next

        def implicit_step(self, t, u, up, upp, h):
                bh = self.Beta*h
                bh2 = self.Beta*h**2
                inv2bmo = 1/(2*self.Beta) - 1
                omgb = 1 - self.gamma/self.Beta
                omg2b = 1 - self.gamma/(2*self.Beta)

                t_next = t+h

                Bn = self.f(t_next) + self.M @ (u/bh2 + up/bh + upp*inv2bmo) + self.C @ (self.gamma*u/bh - up*omgb - h*upp*omg2
```

```
        u_next = solve(self.A, Bn)
        up_next = self.gamma*(u_next - u)/bh + up*omgb + h*upp*omg2b
        upp_next = (u_next - u)/bh2 - up/bh - upp*inv2bmo

        return t_next, u_next, up_next, upp_next

class HHT_alpha(Explicit_ODE_2nd):
    alpha = -1/3
    Beta = (1-alpha)**2/4
    gamma = 1/2 - alpha
    h = 1e-3

    def __init__(self, problem):
        Explicit_ODE_2nd.__init__(self, problem)

        self.up = self.up0

        self.A = self.M / (self.Beta*self.h**2) + self.gamma*self.C / (self.Beta*self.h) + (1+self.alpha)*self.K

    def step(self, t, u, up, upp, h):
        bh = self.Beta*h
        bh2 = self.Beta*h**2
        inv2bmo = 1/(2*self.Beta) - 1
        omgb = 1 - self.gamma/self.Beta
        omg2b = 1 - self.gamma/(2*self.Beta)

        t_next = t+h

        Bn = self.f(t_next) + self.M @ (u/bh2 + up/bh + upp*inv2bmo) + self.C @ (self.gamma*u/bh - up*omgb - h*upp*omg2

        u_next = solve(self.A, Bn)
        up_next = self.gamma*(u_next - u)/bh + up*omgb + h*upp*omg2b
        upp_next = (u_next - u)/bh2 - up/bh - upp*inv2bmo

        return t_next, u_next, up_next, upp_next

    def integrate(self, t, u, up, tf, opts):
        h = min(self.h, abs(tf-t))
        upp = solve(self.M, self.f(0) - self.K@u - self.C@up)

        tres = []
        ures = []

        while t < tf:
            t, u, up, upp = self.step(t, u, up, upp, h)

            tres.append(t)
            ures.append(u.copy())

            h = min(self.h, abs(tf-t))

        return ID_PY_OK, tres, ures

    def simulate(self, tf):
        flag, t, u = self.integrate(self.t0, self.u0, self.up0, tf, opts=None)
```

```
                return t, u
```

Task23.py

```
from classes import Explicit_Problem_2nd, Explicit_ODE_2nd, Newmark, HHT_alpha
from assimulo.solvers import CVode, RungeKutta4, Dopri5
import numpy as np
import matplotlib.pyplot as mpl

omega = np.pi / .3

def f1(t):
    return np.zeros(2,)

def f2(t):
        return np.array([0, 0, 50*np.sin(omega*t)]) if t < np.pi / omega else np.zeros(3,)

# simple 2x2 system for testing
M1 = np.diag([1, 1])
K1 = np.array([[1, 1], [2, -3]])
C1 = np.array([[2, 0], [-1, 0]])

# 3x3 system from the source
M2 = np.diag([10, 20, 30])
K2 = 1e3 * np.array([[45, -20, -15], [-20, 45, -25], [-15, -25, 40]])
C2 = 3e-2 * K2

# initial conditions, change as you like
u0 = np.zeros(3,)
up0 = np.zeros(3,)

t0 = 0
tf = 3

model = Explicit_Problem_2nd(M2, C2, K2, f2, u0, up0, t0)
cvode = CVode(model)
sim = Newmark(model) # can be Newmark or HHT_alpha
sim.h = 1e-3

t1, u_all = cvode.simulate(tf)
t2, u2 = sim.simulate(tf)
u1 = [states[0:len(states)//2] for states in u_all]

# plot commands
...
```

Task4.py

```
from assimulo.explicit_ode import Explicit_ODE
from assimulo.problem import Explicit_Problem
from assimulo.ode import *
```

```python
import numpy as np

import matplotlib.pyplot as mpl

# spring constant and step-size, change as you like
k = 1000
h = 6e-2

def lambdafunc(x, y):
        hyp = np.hypot(x, y)
        return k*(hyp-1)/hyp

def step(t, u, up, upp, h):
        u_next = u + up * h + upp * h**2/2
        upp_next = np.array([0, -1]) - lambdafunc(u_next[0], u_next[1]) * u_next
        up_next = up + (upp + upp_next)*h/2

        return t+h, u_next, up_next, upp_next

t0 = 0
tf = 5

# initial conditions
u0 = np.array([.5, -1])
up0 = np.zeros(2,)

upp0 = np.array([0, -1]) - lambdafunc(u0[0], u0[1]) * u0

# setting up the arrays
tres = []
ures = []

t, u, up, upp = t0, u0, up0, upp0

while t < tf:
        t, u, up, upp = step(t, u, up, upp, h)

        tres.append(t)
        ures.append(u.copy())

        h = min(h, abs(tf-t))

x = [states[0] for states in ures]
y = [states[1] for states in ures]

# plot commands
...
```