Simulation Tools
Project 1

Daniel Holm

January 2024

The elastic pendulum

The differential equations for an elastic pendulum can be written as

$$\begin{aligned}\dot{y}_1 &= y_3\\ \dot{y}_2 &= y_4\\ \dot{y}_3 &= -y_1\lambda(y_1, y_2)\\ \dot{y}_4 &= -y_2\lambda(y_1, y_2) - 1,\\ \mathbf{y}(0) &= \mathbf{y}_0\end{aligned} \tag{1}$$

where $\lambda(y_1, y_2) = k\frac{\sqrt{y_1^2+y_2^2}-1}{\sqrt{y_1^2+y_2^2}}$ for some spring constant k. The variables y_1, y_2 describe the position of the pendulum end in cartesian coordinates, i.e., $(y_1, y_2) = (x, y)$. The pendulum, or spring, is connected to the origin. When in a typical swinging motion the vertical coordinate, $y_2 = y$, will therefore be negative. The hypotenuse seen in λ, $\sqrt{y_1^2 + y_2^2}$, describes the length of the spring. A length greater than one means the spring is stretched and a length less than 1 means the pendulum is compressed. The variables y_3, y_4 are the velocities of the pendulum in cartesian coordinates, i.e., $(y_3, y_4) = (\dot{x}, \dot{y})$. The variables and states will be used interchangeably. Let $\mathbf{y} = (x, y, \dot{x}, \dot{y})$ for future reference.

Task 1

The problem was simulated for 5 seconds using the built-in solver `CVODE`. The initial condition was set to $\mathbf{y} = (0.5, -1, 0, 0)$. This means the pendulum starts at rest and slightly stretched out to the side (since $\sqrt{x^2 + y^2} > 1$).

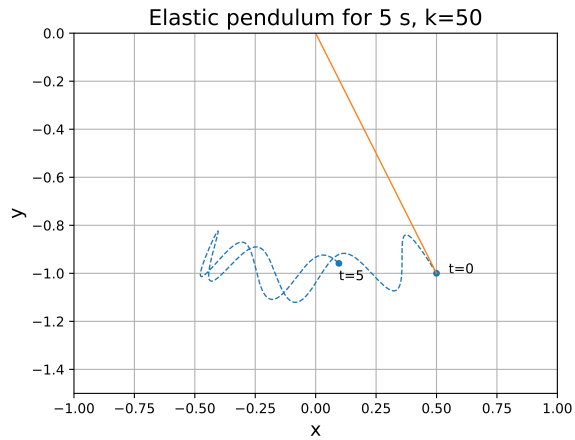


Figure 1: Dashed blue line is the trajectory over time. Orange line represents the pendulum at start.

It appears to be working. The pendulum is clearly swinging side to side while simultaneously bouncing.

Task 2

From the Wikipedia article on Backward differentiation formulas[1] we get

$$y_{n+1} - \frac{18}{11}y_n + \frac{9}{11}y_{n-1} - \frac{2}{11}y_{n-2} = \frac{6}{11}hf(t_{n+1}, y_{n+1}), \tag{2}$$

$$y_{n+1} - \frac{48}{25}y_n + \frac{36}{25}y_{n-1} - \frac{16}{25}y_{n-2} + \frac{3}{25}y_{n-3} = \frac{12}{25}hf(t_{n+1}, y_{n+1}), \tag{3}$$

where h is the step size and $f(t, y)$ is the right-hand-side function from (1). Both schemes were implemented using the provided class BDF_2 as a template. Using the equations (2) and (3), we form the functions

$$F_3(y_{n+1}) = 11y_{n+1} - 18y_n + 9y_{n-1} - 2y_{n-2} - 6hf(t_{n+1}, y_{n+1}), \tag{4}$$

$$F_4(y_{n+1}) = 25y_{n+1} - 48y_n + 36y_{n-1} - 16y_{n-2} + 3y_{n-3} - 12hf(t_{n+1}, y_{n+1}), \tag{5}$$

where the desired y_{n+1} is found by solving $F_3 = 0$ or $F_4 = 0$ using the Newton-Raphson method. In the code implementation this was done using the command `fsolve`.

Task 3

All simulations started with the initial conditions $\mathbf{y} = (0.5, -1, 0, 0)$. The class BDF_4 was used to test out different k.

[1] https://en.wikipedia.org/wiki/Backward_differentiation_formula

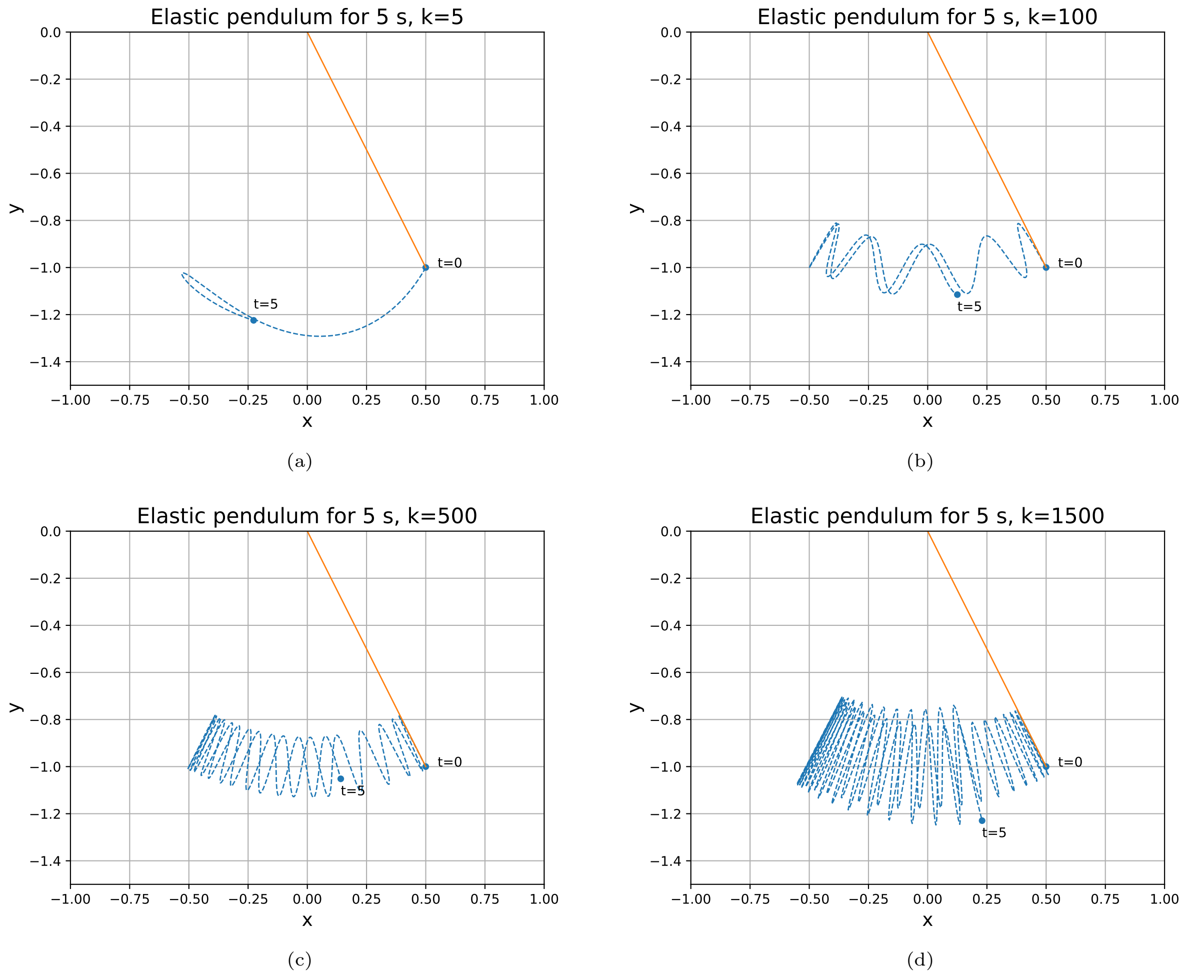
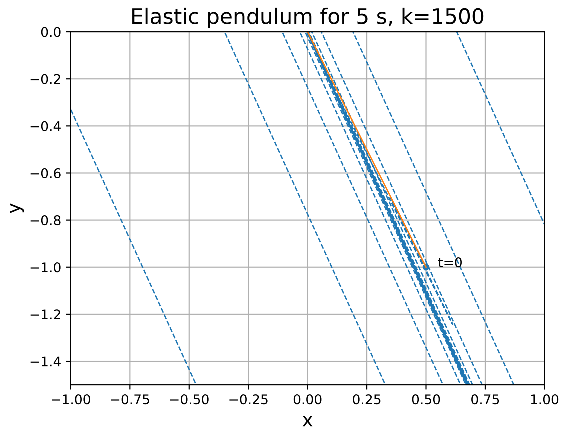


Figure 2: Dashed blue line is the trajectory over time. Orange line represents the pendulum at start. Simulations were done with `BDF_4`.

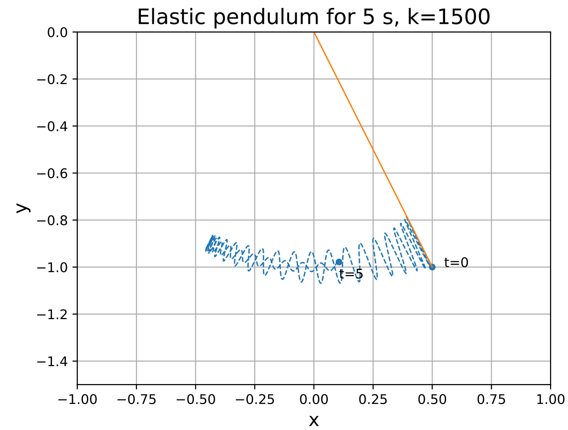
Clearly a higher spring constant increases the frequency at which the spring bounces towards the origin. It also seems to have an effect on the pendulum's angular velocity. In Figure 2a the spring is still on the other side, but in Figure 2b, Figure 2c and Figure 2d the spring has nearly returned to the starting angle. Notice also the increasing bounce amplitude in Figure 2d. This is likely an early sign of the instability in the numerical method.

Examining stability

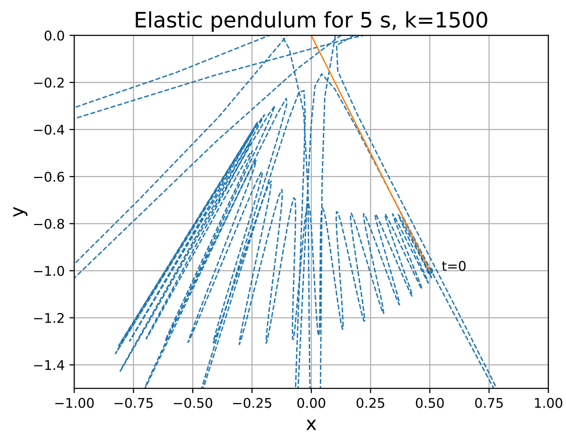
More simulations were performed with $k = 1500$ to further examine stability. This time with different methods.



(a)



(b)

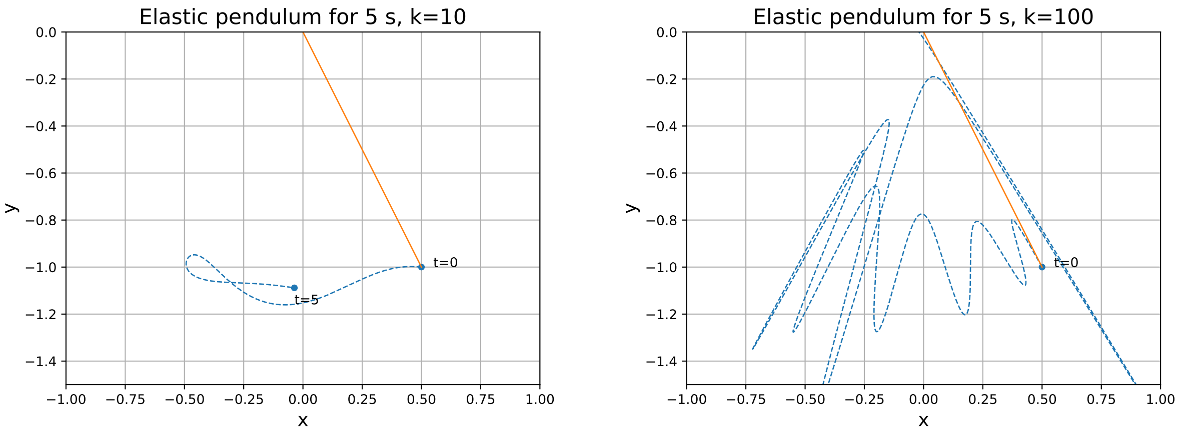


(c)

Figure 3: Dashed blue line is the trajectory over time. Orange line represents the pendulum at start. Simulations were done with explicit Euler, `BDF_2` and `BDF_3` respectively.

Both explicit Euler and `BDF_3` showcase unstable behavior, seen in Figure 3a and Figure 3c respectively. The `BDF_2` solver (seen in Figure 3b) appears to have another interesting property. The bounce amplitude is decreasing instead of increasing. The `BDF_2` solver uses fixed point iteration instead of Newton-Raphson. This is likely the cause of this.

The explicit Euler method appears to be especially bad for this problem. Further examination was done with lower values of k.



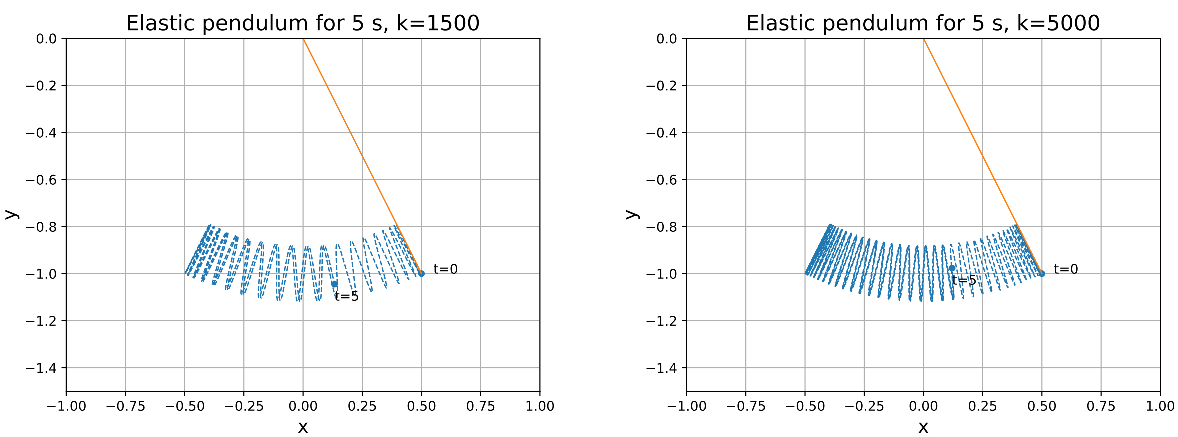
(a) (b)

Figure 4: Dashed blue line is the trajectory over time. Orange line represents the pendulum at start. Simulations were done with explicit Euler.

From Figure 4a and Figure 4b we can see that normal, expected behavior requires quite small values of k compared to the other methods.

Task 4

The above simulations were repeated using the built-in solver `CVODE`. This solver can handle much higher values of k without going unstable. See Figure 5a and Figure 5b.



(a) (b)

Figure 5: Dashed blue line is the trajectory over time. Orange line represents the pendulum at start. Simulations were done with `CVODE`.

Without stating any optional parameters the solver used BDF with a maximal order of 5 and the Newton-Raphson method for the non-linear solver. Both absolute and relative tolerance was 10^{-6}.

Examining tolerances and maximal order

It seems that reducing maximal order has little effect. Even setting it to 1, i.e. simple implicit Euler method, leads to nice graphs for a large k. It must be noted however that the `CVODE` solver takes a much larger amount of steps, specifically 166722 steps for the simulation in Figure 6. All other simulations have been done with a much larger step size, leading to only hundreds of steps or even less.

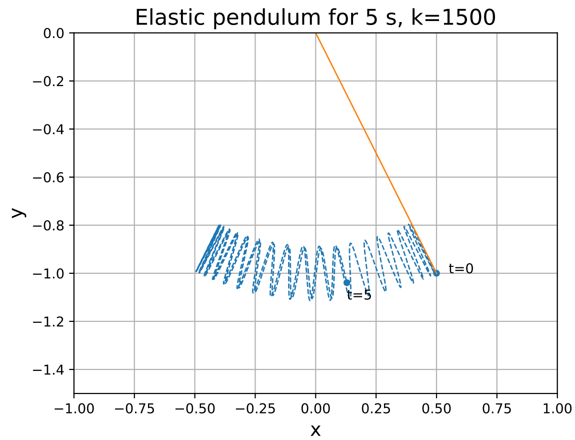
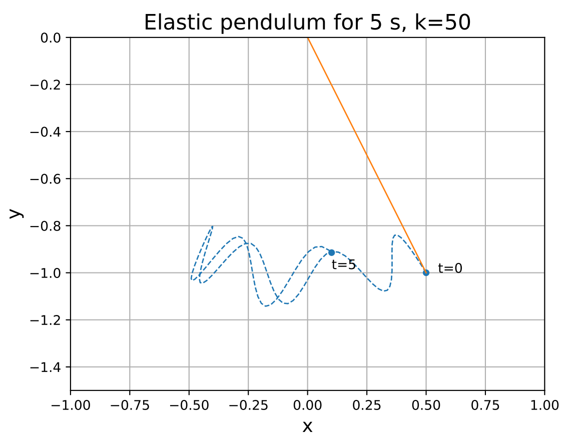
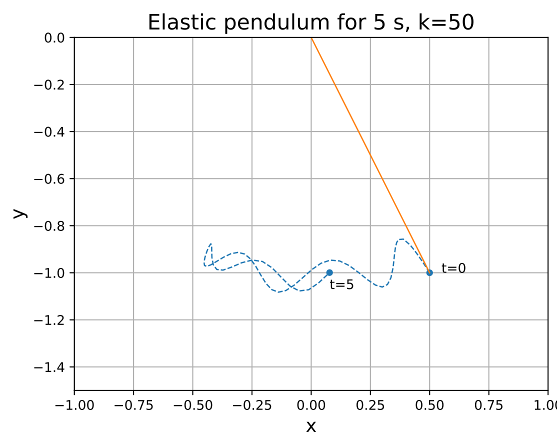


Figure 6: Dashed blue line is the trajectory over time. Orange line represents the pendulum at start. Simulation was done with `CVODE`.

The most reliable way to make the solver produce bad results is to reduce the tolerances. Going to back a non-specified maximal order led to the results in Figure 7a and Figure 7b.



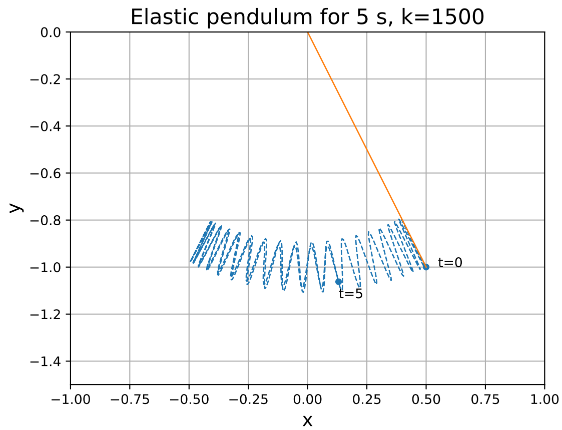
(a)



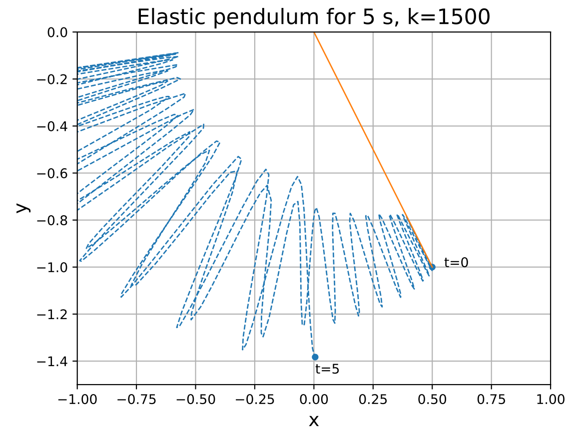
(b)

Figure 7: Dashed blue line is the trajectory over time. Orange line represents the pendulum at start. Simulations were done with `CVODE`. Both tolerances were set to 10^{-3} and 10^{-2} respectively.

When the tolerances were set to 10^{-3} it looks like the bounce amplitude is slightly increasing but for the larger tolerance the bounce amplitude is decreasing. The tests were repeated for $k = 1500$.



(a)



(b)

Figure 8: Dashed blue line is the trajectory over time. Orange line represents the pendulum at start. Simulations were done with `CVODE`. Both tolerances were set to 10^{-3} and 10^{-2} respectively.

This appears to have the opposite results. The smaller tolerance dampens the bounce amplitude while the larger tolerance increases the bounce amplitude severely.

Code

The code was divided into a few different files. The file `classes.py` contains the written solvers, and the other files were for the specific tasks. The code here is technically incomplete as it does not include the commands for creating the plots as well as some of the code given to us.

classes.py

```
from assimulo.explicit_ode import Explicit_ODE, Explicit_ODE_Exception
import numpy as np
import scipy.linalg as SL
from scipy.optimize import fsolve

class BDF_2(Explicit_ODE):
        """
        BDF-2   (Example of how to set-up own integrators for Assimulo)
        """
        tol = 1.e-8
        maxit = 100
        maxsteps = 1000

        alpha = [3./2., -2., 1./2]

        def __init__(self, problem):
                Explicit_ODE.__init__(self, problem) #Calls the base class

                #Solver options
                self.options["h"] = 0.01

                #Statistics
                self.statistics["nsteps"] = 0
```

```
        self.statistics["nfcns"] = 0

    def _set_h(self,h):
            self.options["h"] = float(h)

    def _get_h(self):
        return self.options["h"]

    h=property(_get_h,_set_h)

    def integrate(self, t, y, tf, opts):
        """
        _integrates (t,y) values until t > tf
        """
        h = self.options["h"]
        h = min(h, abs(tf-t))

        #Lists for storing the result
        tres = []
        yres = []

        for i in range(self.maxsteps):
            if t >= tf:
                break
            self.statistics["nsteps"] += 1

            if i==0:  # initial step
                t_np1,y_np1 = self.step_EE(t,y, h)
            else:
                t_np1, y_np1 = self.step_BDF2([t,t_nm1], [y,y_nm1], h)
            t,t_nm1=t_np1,t
            y,y_nm1=y_np1,y

            tres.append(t)
            yres.append(y.copy())

            h=min(self.h,np.abs(tf-t))
        else:
            raise Explicit_ODE_Exception('Final time not reached within maximum number of steps')

        return ID_PY_OK, tres, yres

    def step_EE(self, t, y, h):
        """
        This calculates the next step in the integration with explicit Euler.
        """
        self.statistics["nfcns"] += 1

        f = self.problem.rhs
        return t + h, y + h*f(t, y)

    def step_BDF2(self, T, Y, h):
        """
        BDF-2 with Fixed Point Iteration and Zero order predictor

        alpha_0*y_np1+alpha_1*y_n+alpha_2*y_nm1=h f(t_np1,y_np1)
```

```
                alpha=[3/2,-2,1/2]
                """

                f = self.problem.rhs

                t_n, t_nm1 = T
                y_n, y_nm1 = Y
                # predictor
                t_np1 = t_n+h
                y_np1_i = y_n   # zero order predictor
                # corrector with fixed point iteration
                for i in range(self.maxit):
                        self.statistics["nfcns"] += 1

                        y_np1_ip1 = -(self.alpha[1]*y_n + self.alpha[2]*y_nm1) + h*f(t_np1,y_np1_i) / self.alpha[0]
                        if SL.norm(y_np1_ip1 - y_np1_i) < self.tol:
                                return t_np1, y_np1_ip1
                        y_np1_i = y_np1_ip1
                else:
                        raise Explicit_ODE_Exception('Corrector could not converge within % iterations'%i)

        def print_statistics(self, verbose=NORMAL):
        ...

class BDF_3(Explicit_ODE):
        """
        BDF-3
        """
        tol = 1.e-8
        maxit = 1000
        maxsteps = 1000
        alpha = [11./6., -3., 1.5, -1./3.]

        def __init__(self, problem):
                Explicit_ODE.__init__(self, problem) #Calls the base class

                #Solver options
                self.options["h"] = 0.01

                #Statistics
                self.statistics["nsteps"] = 0
                self.statistics["nfcns"] = 0

        def _set_h(self,h):
                        self.options["h"] = float(h)

        def _get_h(self):
                return self.options["h"]

        h=property(_get_h,_set_h)

        def integrate(self, t, y, tf, opts):
                """
                _integrates (t,y) values until t > tf
                """
                h = self.options["h"]
```

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

        #Lists for storing the result
        tres = []
        yres = []

        t_nm2, t_nm1 = 0., h
        y_nm2 = y

        for i in range(self.maxsteps+1):
            if t >= tf:
                break
            self.statistics["nsteps"] += 1

            if i == 0:  # initial steps
                t_np1, y_np1 = self.step_EE(t, y, h)
                t = t_np1
                y = y_np1
                y_nm1 = y
            elif i == 1:
                t_np1, y_np1 = self.step_EE(t, y, h)
                t = t_np1
                y = y_np1
            else:
                t_np1, y_np1 = self.step_BDF3([t, t_nm1, t_nm2], [y, y_nm1, y_nm2], h)
                t, t_nm1, t_nm2 = t_np1, t, t_nm1
                y, y_nm1, y_nm2 = y_np1, y, y_nm1

            tres.append(t)
            yres.append(y.copy())

            h=min(self.h, np.abs(tf - t))
        else:
            raise Explicit_ODE_Exception('Final time not reached within maximum number of steps')

        return ID_PY_OK, tres, yres

    def step_EE(self, t, y, h):
        """
        This calculates the next step in the integration with explicit Euler.
        """
        self.statistics["nfcns"] += 1

        f = self.problem.rhs
        return t + h, y + h*f(t, y)

    def step_BDF3(self, T, Y, h):
        """
        BDF-3: Backward differentiation formula
        y_np1 = 1/11 * [18y_n - 9y_nm1 + 2y_nm2 + 6hf(t_np1, y_np1)]

        F(y_np1) = 11/6 * y_np1 - 3y_n + 1.5y_nm1 - 1/3 * y_nm2 - hf(t_np1, y_np1)
        Find F(y_np1) = 0 with Newton-Raphson iteration
        y_np1_ip1 = y_np1_i - J(y_np1_i)^-1 * F(y_np1_i)
        """
        f=self.problem.rhs
```

```
            t_np1 = T[0] + h

            def F(y_np1):
                return self.alpha[0]*y_np1 + self.alpha[1]*Y[0] + self.alpha[2]*Y[1] + self.alpha[3]*Y[2] - h*f(t_np1,

            y_np1, infodict, ier, _ = fsolve(func=F, x0=Y[0], full_output=True, xtol=self.tol, maxfev=self.maxit)
            self.statistics["nfcns"] += infodict.get('nfev')

            if ier == 1:
                return t_np1, y_np1
            else:
                raise Explicit_ODE_Exception('Corrector could not converge within %s iterations' % self.maxit)

    def print_statistics(self, verbose=NORMAL):
       ...

class BDF_4(Explicit_ODE):
    """
    BDF-4
    """
    tol = 1.e-8
    maxit = 100
    maxsteps = 1000
    alpha = [25./12., -4., 3., -4./3., .25]

    def __init__(self, problem):
        Explicit_ODE.__init__(self, problem) #Calls the base class

        #Solver options
        self.options["h"] = 0.01

        #Statistics
        self.statistics["nsteps"] = 0
        self.statistics["nfcns"] = 0

    def _set_h(self,h):
            self.options["h"] = float(h)

    def _get_h(self):
        return self.options["h"]

    h=property(_get_h,_set_h)

    def integrate(self, t, y, tf, opts):
        """
        _integrates (t,y) values until t > tf
        """
        h = self.options["h"]
        h = min(h, abs(tf-t))

        #Lists for storing the result
        tres = []
        yres = []

        t_nm3, t_nm2, t_nm1 = 0., h, 2.*h
```

```
        y_nm3 = y

        for i in range(self.maxsteps+1):
            if t >= tf:
                break
            self.statistics["nsteps"] += 1

            if i == 0:  # initial steps
                t_np1, y_np1 = self.step_EE(t, y, h)
                t = t_np1
                y = y_np1
                y_nm2 = y
            elif i == 1:
                t_np1, y_np1 = self.step_EE(t, y, h)
                t = t_np1
                y = y_np1
                y_nm1 = y
            elif i == 2:
                t_np1, y_np1 = self.step_EE(t, y, h)
                t = t_np1
                y = y_np1
            else:
                t_np1, y_np1 = self.step_BDF4([t, t_nm1, t_nm2, t_nm3], [y, y_nm1, y_nm2, y_nm3], h)
                t, t_nm1, t_nm2, t_nm3 = t_np1, t, t_nm1, t_nm2
                y, y_nm1, y_nm2, y_nm3 = y_np1, y, y_nm1, y_nm2

            tres.append(t)
            yres.append(y.copy())

            h=min(self.h, np.abs(tf - t))
        else:
            raise Explicit_ODE_Exception('Final time not reached within maximum number of steps')

        return ID_PY_OK, tres, yres

    def step_EE(self, t, y, h):
        """
        This calculates the next step in the integration with explicit Euler.
        """
        self.statistics["nfcns"] += 1

        f = self.problem.rhs
        return t + h, y + h*f(t, y)

    def step_BDF4(self, T, Y, h):
        """
        BDF-4: Backward differentiation formula
        y_np1 = 1/25 * [48y_n -36y_nm1 + 16y_nm2 -3y_nm3 + 12hf(t_np1, y_np1)]

        F(y_np1) = alpha*[y_np1, Y] - hf(t_np1, y_np1)
        Find F(y_np1) = 0 with fsolve()
        """
        f=self.problem.rhs

        t_np1 = T[0] + h

```

```
        def F(y_np1):
            return self.alpha[0]*y_np1 + self.alpha[1]*Y[0] + self.alpha[2]*Y[1] + self.alpha[3]*Y[2] + self.alpha[

        y_np1, infodict, ier, _ = fsolve(func=F, x0=Y[0], full_output=True, xtol=self.tol, maxfev=self.maxit)
        self.statistics["nfcns"] += infodict.get('nfev')

        if ier == 1:
            return t_np1, y_np1
        else:
            raise Explicit_ODE_Exception('Corrector could not converge within %s iterations' % self.maxit)

    def print_statistics(self, verbose=NORMAL):
        ...
```

Tasks1,4.py

```
from assimulo.problem import Explicit_Problem
from assimulo.solvers import CVode
import numpy as np
import matplotlib.pyplot as mpl

# Define the rhs function
def rhs(t,y):
    global k
    root = np.sqrt(y[0]**2 + y[1]**2)
    temp = k*(root - 1.)/root

    y1dot = y[2]
    y2dot = y[3]
    y3dot = -y[0]*temp
    y4dot = -y[1]*temp - 1.

    return np.array([y1dot, y2dot, y3dot, y4dot])

# Spring constant, change as you like
k = 1500.

# Initial conditions
y0 = np.array([.5, -1, 0., 0.])
t0 = 0.
tf = 5.

model = Explicit_Problem(rhs, y0, t0)
model.name = 'Elastic Pendulum'

sim = CVode(model)
sim.atol = 1e-2
sim.rtol = 1e-2
# sim.maxord = 3

t, y = sim.simulate(tf)
x = [states[0] for states in y]
y = [states[1] for states in y]

```

```
# plot commands
...
```

Tasks2,3.py

```
from assimulo.problem import Explicit_Problem
from assimulo.solvers import ExplicitEuler
import numpy as np
import matplotlib.pyplot as mpl
from classes import BDF_2, BDF_3, BDF_4

def rhs(t,y):
        global k
        root = np.sqrt(y[0]**2 + y[1]**2)
        temp = k*(root - 1.)/root

        y1dot = y[2]
        y2dot = y[3]
        y3dot = -y[0]*temp
        y4dot = -y[1]*temp - 1.

        return np.array([y1dot, y2dot, y3dot, y4dot])

# Spring constant, change as you like
k = 1500.

# Initial conditions
y0 = np.array([.5, -1, 0., 0.])
t0 = 0.
tf = 5.

model = Explicit_Problem(rhs, y0, t0)
model.name = 'Elastic Pendulum'

sim = BDF_2(model) # Create a BDF solver of choice
EE_sim = ExplicitEuler(model)
t, y = sim.simulate(tf)

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

# plot commands
...
```