

# 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  $\lambda$ ,  $\sqrt{y_1^2 + y_2^2}$ , 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  $\sqrt{x^2 + y^2} > 1$ ).

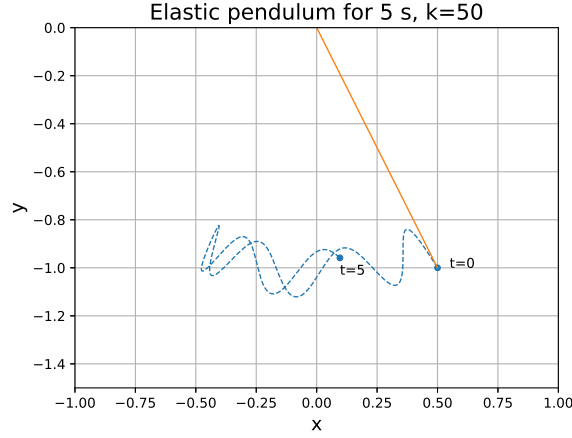


Figure 1: Dashed blue line is the trajectory over time. Orange line represents the pendulum at start.

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<sup>1</sup> we get

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

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

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

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

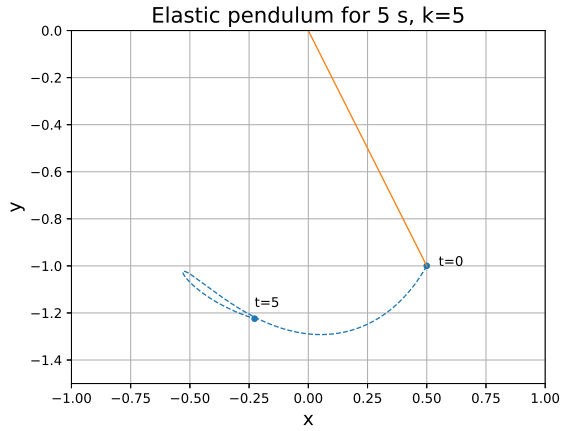
$$F_4(y_{n+1}) = 25y_{n+1} - 48y_n + 36y_{n-1} - 16y_{n-2} + 3y_{n-3} - 12hf(t_{n+1}, y_{n+1}), \quad (5)$$

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`.

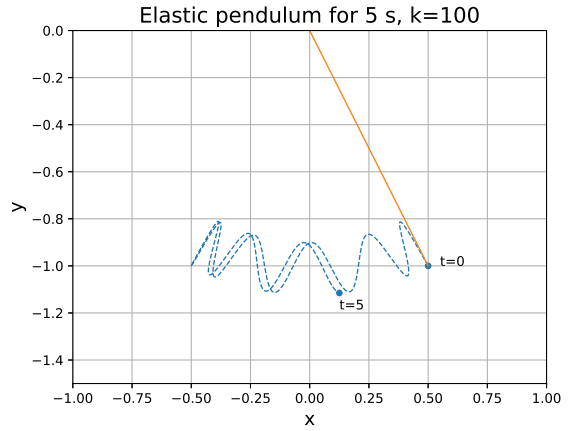
## Task 3

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

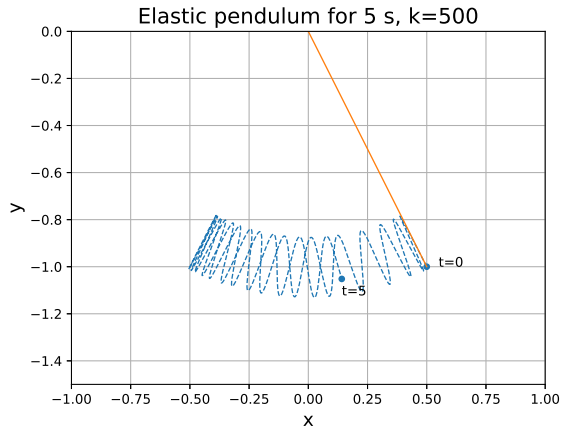
<sup>1</sup>[https://en.wikipedia.org/wiki/Backward\\_differentiation\\_formula](https://en.wikipedia.org/wiki/Backward_differentiation_formula)



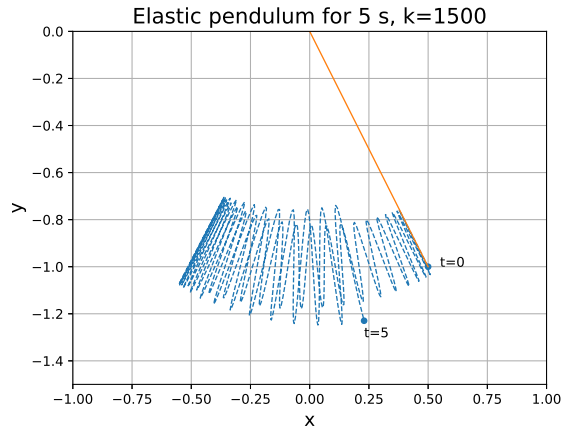
(a)



(b)



(c)



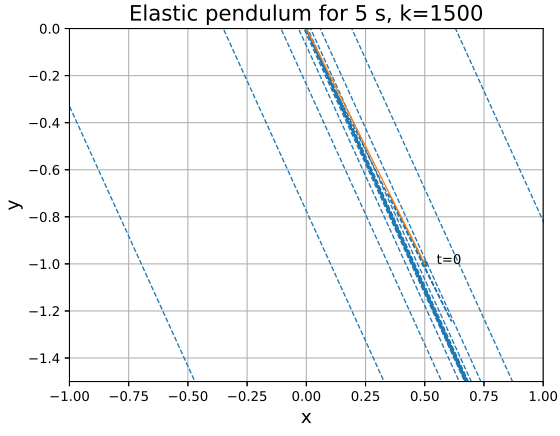
(d)

Figure 2: Dashed blue line is the trajectory over time. Orange line represents the pendulum at start. Simulations were done with BDF\_4.

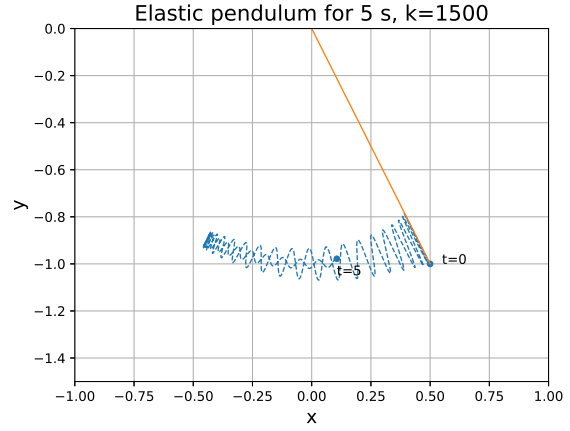
Clearly a higher spring constant increases the frequency at which the spring bounces towards the origin. It also seems to have an effect on the pendulum's angular velocity. In Figure 2a the spring is still on the other side, but in Figure 2b, Figure 2c and Figure 2d the spring has nearly returned to 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

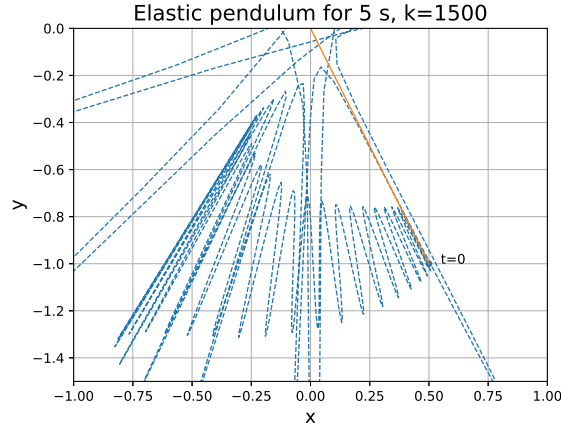
More simulations were performed with  $k = 1500$  to further examine stability. This time with different methods.



(a)



(b)

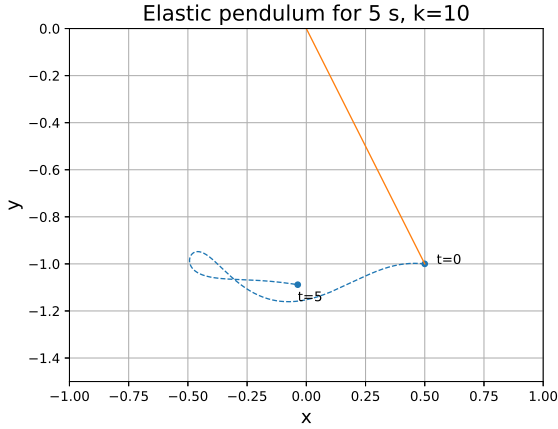


(c)

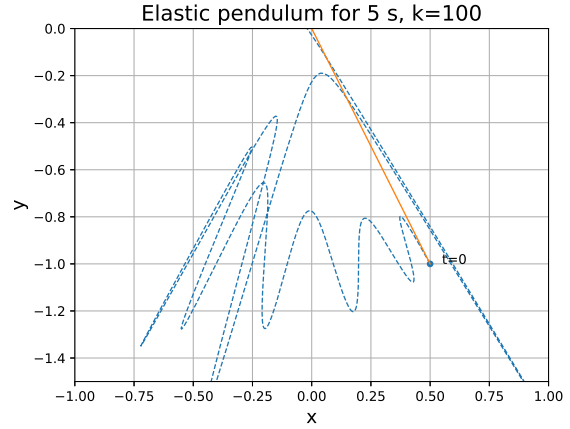
Figure 3: Dashed blue line is the trajectory over time. Orange line represents the pendulum at start. Simulations were done with explicit Euler, BDF\_2 and BDF\_3 respectively.

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)



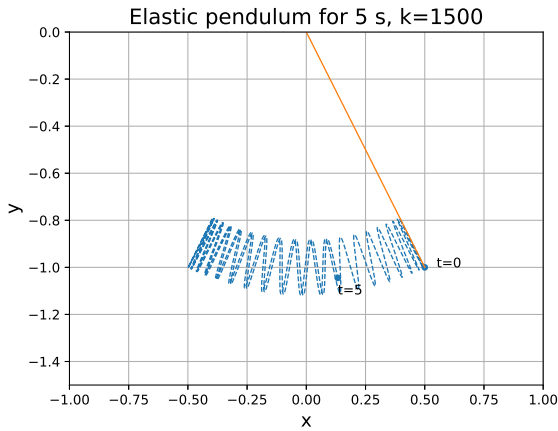
(b)

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

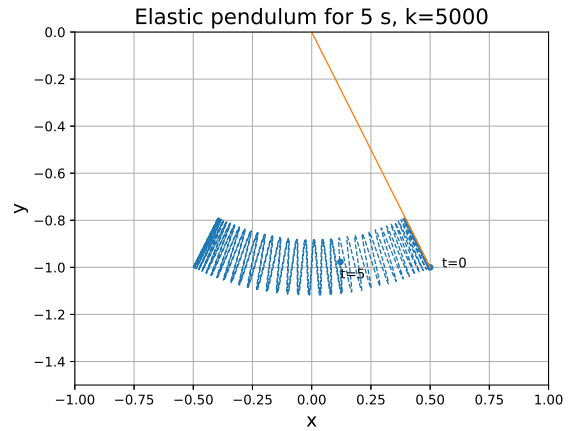
From Figure 4a and Figure 4b we can see that normal, expected behavior requires quite small values of  $k$  compared to the other methods.

## 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.

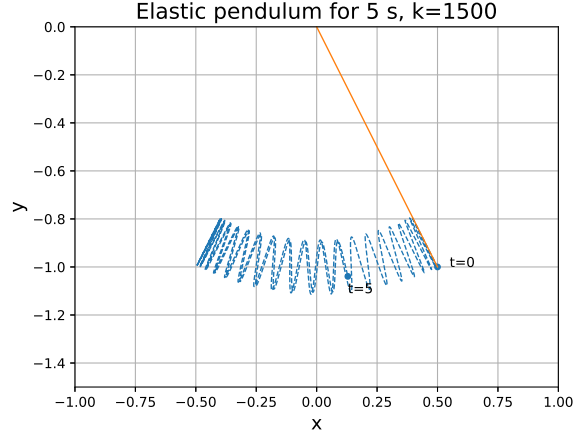


Figure 6: Dashed blue line is the trajectory over time. Orange line represents the pendulum at start. Simulation was done with CVODE.

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.

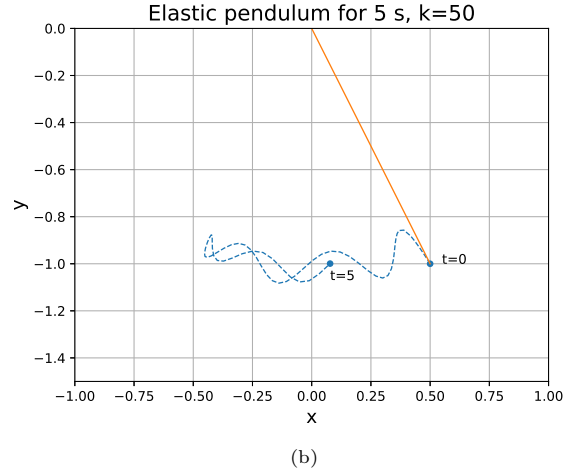
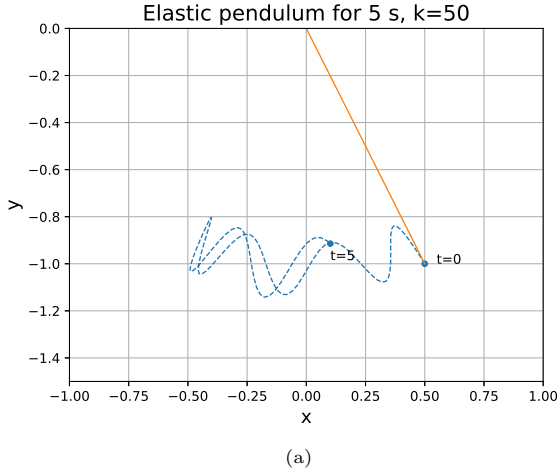


Figure 7: Dashed blue line is the trajectory over time. Orange line represents the pendulum at start. Simulations were done with CVODE. Both tolerances were set to  $10^{-3}$  and  $10^{-2}$  respectively.

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  $k = 1500$ .

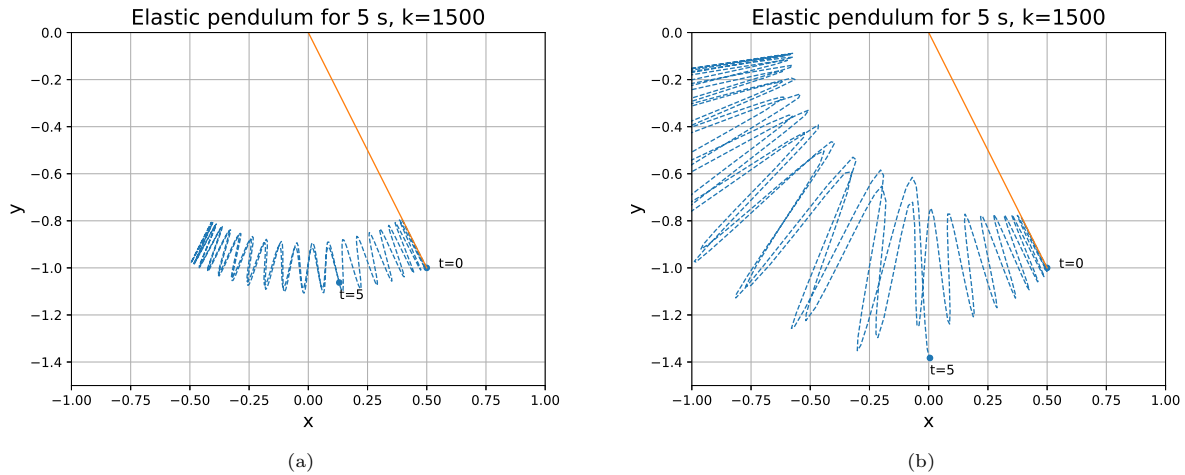


Figure 8: Dashed blue line is the trajectory over time. Orange line represents the pendulum at start. Simulations were done with CVODE. Both tolerances were set to  $10^{-3}$  and  $10^{-2}$  respectively.

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

```

1  from assimulo.explicit_ode import Explicit_ODE, Explicit_ODE_Exception
2  import numpy as np
3  import scipy.linalg as SL
4  from scipy.optimize import fsolve
5
6  class BDF_2(Explicit_ODE):
7      """
8      BDF-2 (Example of how to set-up own integrators for Assimulo)
9      """
10     tol = 1.e-8
11     maxit = 100
12     maxsteps = 1000
13
14     alpha = [3./2., -2., 1./2]
15
16     def __init__(self, problem):
17         Explicit_ODE.__init__(self, problem) #Calls the base class
18
19         #Solver options
20         self.options["h"] = 0.01
21
22         #Statistics
23         self.statistics["nsteps"] = 0

```

```

24         self.statistics["nfcns"] = 0
25
26     def _set_h(self,h):
27         self.options["h"] = float(h)
28
29     def _get_h(self):
30         return self.options["h"]
31
32     h=property(_get_h,_set_h)
33
34     def integrate(self, t, y, tf, opts):
35         """
36         _integrates (t,y) values until t > tf
37         """
38         h = self.options["h"]
39         h = min(h, abs(tf-t))
40
41         #Lists for storing the result
42         tres = []
43         yres = []
44
45         for i in range(self.maxsteps):
46             if t >= tf:
47                 break
48             self.statistics["nsteps"] += 1
49
50             if i==0: # initial step
51                 t_np1,y_np1 = self.step_EE(t,y, h)
52             else:
53                 t_np1, y_np1 = self.step_BDF2([t,t_nm1], [y,y_nm1], h)
54                 t,t_nm1=t_np1,t
55                 y,y_nm1=y_np1,y
56
57                 tres.append(t)
58                 yres.append(y.copy())
59
60                 h=min(self.h,np.abs(tf-t))
61             else:
62                 raise Explicit_ODE_Exception('Final time not reached within maximum number of steps')
63
64         return ID_PY_OK, tres, yres
65
66     def step_EE(self, t, y, h):
67         """
68         This calculates the next step in the integration with explicit Euler.
69         """
70         self.statistics["nfcns"] += 1
71
72         f = self.problem.rhs
73         return t + h, y + h*f(t, y)
74
75     def step_BDF2(self, T, Y, h):
76         """
77         BDF-2 with Fixed Point Iteration and Zero order predictor
78
79         
$$\alpha_0 y_{np1} + \alpha_1 y_n + \alpha_2 y_{nm1} = h f(t_{np1}, y_{np1})$$


```



```

80         alpha=[3/2,-2,1/2]
81         """
82
83         f = self.problem.rhs
84
85         t_n, t_nm1 = T
86         y_n, y_nm1 = Y
87         # predictor
88         t_np1 = t_n+h
89         y_np1_i = y_n    # zero order predictor
90         # corrector with fixed point iteration
91         for i in range(self.maxit):
92             self.statistics["nfcns"] += 1
93
94             y_np1_ip1 = -(self.alpha[1]*y_n + self.alpha[2]*y_nm1) + h*f(t_np1,y_np1_i) / self.alpha[0]
95             if SL.norm(y_np1_ip1 - y_np1_i) < self.tol:
96                 return t_np1, y_np1_ip1
97             y_np1_i = y_np1_ip1
98         else:
99             raise Explicit_ODE_Exception('Corrector could not converge within % iterations'%i)
100
101     def print_statistics(self, verbose=NORMAL):
102         ...
103
104 class BDF_3(Explicit_ODE):
105     """
106     BDF-3
107     """
108     tol = 1.e-8
109     maxit = 1000
110     maxsteps = 1000
111     alpha = [11./6., -3., 1.5, -1./3.]
112
113     def __init__(self, problem):
114         Explicit_ODE.__init__(self, problem) #Calls the base class
115
116         #Solver options
117         self.options["h"] = 0.01
118
119         #Statistics
120         self.statistics["nsteps"] = 0
121         self.statistics["nfcns"] = 0
122
123     def _set_h(self,h):
124         self.options["h"] = float(h)
125
126     def _get_h(self):
127         return self.options["h"]
128
129     h=property(_get_h,_set_h)
130
131     def integrate(self, t, y, tf, opts):
132         """
133         _integrates (t,y) values until t > tf
134         """
135         h = self.options["h"]

```

```

136         h = min(h, abs(tf-t))
137
138         #Lists for storing the result
139         tres = []
140         yres = []
141
142         t_nm2, t_nm1 = 0., h
143         y_nm2 = y
144
145         for i in range(self.maxsteps+1):
146             if t >= tf:
147                 break
148             self.statistics["nsteps"] += 1
149
150             if i == 0: # initial steps
151                 t_np1, y_np1 = self.step_EE(t, y, h)
152                 t = t_np1
153                 y = y_np1
154                 y_nm1 = y
155             elif i == 1:
156                 t_np1, y_np1 = self.step_EE(t, y, h)
157                 t = t_np1
158                 y = y_np1
159             else:
160                 t_np1, y_np1 = self.step_BDF3([t, t_nm1, t_nm2], [y, y_nm1, y_nm2], h)
161                 t, t_nm1, t_nm2 = t_np1, t, t_nm1
162                 y, y_nm1, y_nm2 = y_np1, y, y_nm1
163
164             tres.append(t)
165             yres.append(y.copy())
166
167             h=min(self.h, np.abs(tf - t))
168         else:
169             raise Explicit_ODE_Exception('Final time not reached within maximum number of steps')
170
171         return ID_PY_OK, tres, yres
172
173     def step_EE(self, t, y, h):
174         """
175         This calculates the next step in the integration with explicit Euler.
176         """
177         self.statistics["nfcns"] += 1
178
179         f = self.problem.rhs
180         return t + h, y + h*f(t, y)
181
182     def step_BDF3(self, T, Y, h):
183         """
184         BDF-3: Backward differentiation formula
185          $y_{np1} = 1/11 * [18y_n - 9y_{nm1} + 2y_{nm2} + 6hf(t_{np1}, y_{np1})]$ 
186
187          $F(y_{np1}) = 11/6 * y_{np1} - 3y_n + 1.5y_{nm1} - 1/3 * y_{nm2} - hf(t_{np1}, y_{np1})$ 
188         Find  $F(y_{np1}) = 0$  with Newton-Raphson iteration
189          $y_{np1\_ip1} = y_{np1\_i} - J(y_{np1\_i})^{-1} * F(y_{np1\_i})$ 
190         """
191         f=self.problem.rhs

```

```

192         t_np1 = T[0] + h
193
194     def F(y_np1):
195         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,
196
197     y_np1, infodict, ier, _ = fsolve(func=F, x0=Y[0], full_output=True, xtol=self.tol, maxfev=self.maxit)
198     self.statistics["nfcns"] += infodict.get('nfev')
199
200
201     if ier == 1:
202         return t_np1, y_np1
203     else:
204         raise Explicit_ODE_Exception('Corrector could not converge within %s iterations' % self.maxit)
205
206     def print_statistics(self, verbose=NORMAL):
207         ...
208
209 class BDF_4(Explicit_ODE):
210     """
211     BDF-4
212     """
213     tol = 1.e-8
214     maxit = 100
215     maxsteps = 1000
216     alpha = [25./12., -4., 3., -4./3., .25]
217
218     def __init__(self, problem):
219         Explicit_ODE.__init__(self, problem) #Calls the base class
220
221         #Solver options
222         self.options["h"] = 0.01
223
224         #Statistics
225         self.statistics["nsteps"] = 0
226         self.statistics["nfcns"] = 0
227
228     def _set_h(self, h):
229         self.options["h"] = float(h)
230
231     def _get_h(self):
232         return self.options["h"]
233
234     h=property(_get_h, _set_h)
235
236     def integrate(self, t, y, tf, opts):
237         """
238         _integrates (t,y) values until t > tf
239         """
240         h = self.options["h"]
241         h = min(h, abs(tf-t))
242
243         #Lists for storing the result
244         tres = []
245         yres = []
246
247         t_nm3, t_nm2, t_nm1 = 0., h, 2.*h

```

```

248     y_nm3 = y
249
250     for i in range(self.maxsteps+1):
251         if t >= tf:
252             break
253         self.statistics["nsteps"] += 1
254
255         if i == 0: # initial steps
256             t_np1, y_np1 = self.step_EE(t, y, h)
257             t = t_np1
258             y = y_np1
259             y_nm2 = y
260         elif i == 1:
261             t_np1, y_np1 = self.step_EE(t, y, h)
262             t = t_np1
263             y = y_np1
264             y_nm1 = y
265         elif i == 2:
266             t_np1, y_np1 = self.step_EE(t, y, h)
267             t = t_np1
268             y = y_np1
269         else:
270             t_np1, y_np1 = self.step_BDF4([t, t_nm1, t_nm2, t_nm3], [y, y_nm1, y_nm2, y_nm3], h)
271             t, t_nm1, t_nm2, t_nm3 = t_np1, t, t_nm1, t_nm2
272             y, y_nm1, y_nm2, y_nm3 = y_np1, y, y_nm1, y_nm2
273
274         tres.append(t)
275         yres.append(y.copy())
276
277         h=min(self.h, np.abs(tf - t))
278     else:
279         raise Explicit_ODE_Exception('Final time not reached within maximum number of steps')
280
281     return ID_PY_OK, tres, yres
282
283     def step_EE(self, t, y, h):
284         """
285         This calculates the next step in the integration with explicit Euler.
286         """
287         self.statistics["nfcns"] += 1
288
289         f = self.problem.rhs
290         return t + h, y + h*f(t, y)
291
292     def step_BDF4(self, T, Y, h):
293         """
294         BDF-4: Backward differentiation formula
295          $y_{np1} = 1/25 * [48y_n - 36y_{nm1} + 16y_{nm2} - 3y_{nm3} + 12hf(t_{np1}, y_{np1})]$ 
296
297          $F(y_{np1}) = \alpha*[y_{np1}, Y] - hf(t_{np1}, y_{np1})$ 
298         Find  $F(y_{np1}) = 0$  with fsolve()
299         """
300         f=self.problem.rhs
301
302         t_np1 = T[0] + h
303

```

```

304         def F(y_np1):
305             return self.alpha[0]*y_np1 + self.alpha[1]*Y[0] + self.alpha[2]*Y[1] + self.alpha[3]*Y[2] + self.alpha[4]*Y[3]
306
307         y_np1, infodict, ier, _ = fsolve(func=F, x0=Y[0], full_output=True, xtol=self.tol, maxfev=self.maxit)
308         self.statistics["nfcns"] += infodict.get('nfev')
309
310         if ier == 1:
311             return t_np1, y_np1
312         else:
313             raise Explicit_ODE_Exception('Corrector could not converge within %s iterations' % self.maxit)
314
315         def print_statistics(self, verbose=NORMAL):
316             ...

```

## Tasks1,4.py

```

1  from assimulo.problem import Explicit_Problem
2  from assimulo.solvers import CVode
3  import numpy as np
4  import matplotlib.pyplot as plt
5
6  # Define the rhs function
7  def rhs(t,y):
8      global k
9      root = np.sqrt(y[0]**2 + y[1]**2)
10     temp = k*(root - 1.)/root
11
12     y1dot = y[2]
13     y2dot = y[3]
14     y3dot = -y[0]*temp
15     y4dot = -y[1]*temp - 1.
16
17     return np.array([y1dot, y2dot, y3dot, y4dot])
18
19 # Spring constant, change as you like
20 k = 1500.
21
22 # Initial conditions
23 y0 = np.array([.5, -1, 0., 0.])
24 t0 = 0.
25 tf = 5.
26
27 model = Explicit_Problem(rhs, y0, t0)
28 model.name = 'Elastic Pendulum'
29
30 sim = CVode(model)
31 sim.atol = 1e-2
32 sim.rtol = 1e-2
33 # sim.maxord = 3
34
35 t, y = sim.simulate(tf)
36 x = [states[0] for states in y]
37 y = [states[1] for states in y]
38

```

```
39 # plot commands
40 ...
```

## Tasks2,3.py

```
1  from assimulo.problem import Explicit_Problem
2  from assimulo.solvers import ExplicitEuler
3  import numpy as np
4  import matplotlib.pyplot as plt
5  from classes import BDF_2, BDF_3, BDF_4
6
7  def rhs(t,y):
8      global k
9      root = np.sqrt(y[0]**2 + y[1]**2)
10     temp = k*(root - 1.)/root
11
12     y1dot = y[2]
13     y2dot = y[3]
14     y3dot = -y[0]*temp
15     y4dot = -y[1]*temp - 1.
16
17     return np.array([y1dot, y2dot, y3dot, y4dot])
18
19 # Spring constant, change as you like
20 k = 1500.
21
22 # Initial conditions
23 y0 = np.array([.5, -1, 0., 0.])
24 t0 = 0.
25 tf = 5.
26
27 model = Explicit_Problem(rhs, y0, t0)
28 model.name = 'Elastic Pendulum'
29
30 sim = BDF_2(model) # Create a BDF solver of choice
31 EE_sim = ExplicitEuler(model)
32 t, y = sim.simulate(tf)
33
34 x = [states[0] for states in y]
35 y = [states[1] for states in y]
36
37 # plot commands
38 ...
```