Simulation Tools FMNN05
Final report

Nils Fagerberg, Daniel Odenbrand, Felicia Seemann and Linus Jangland December 17, 2015

Contents

Background

As a part of the course Simulation Tools, FMNN05, at LTH there are three projects related to simulating solutions of differential equations using the tools Assimulo/Sundials, Dymola and the programming languages Python and Modelica. This report aims to present and discuss the problems as well as the differences in the solutions produced by different software. The projects were modeling simulations of an elastic pendulum, a 7-arm squeezer and a woodpecker toy. Corresponding theory and results will be presented in order for each of these problems.

1 The elastic pendulum problem

1.1 Introduction

Project 1 consists of setting up the equations for an elastic pendulum, i.e. a pendulum made of a spring and a weight. This will result in a highly oscillatory problem. We will study the equations of motion and implement this in Assimulo. Furthermore we will compare different integration methods and also implement our own integrators.

1.2 Theory

An elastic pendulum can be modelled by the following ODE:

$$\dot{x} = v_x \tag{1}$$

$$\dot{y} = v_y \tag{2}$$

$$\dot{v_x} = -x \cdot \lambda(x, y) \tag{3}$$

$$\dot{v_y} = -y \cdot \lambda(x, y) - 1\,, \tag{4}$$

where $\lambda(x, y) = k\frac{\sqrt{x^2+y^2}-1}{\sqrt{x^2+y^2}}$. This corresponds to a mathematical pendulum where the bar is replaced by a linear spring with spring constant k. The mass-, nominal pendulum length and gravitational constant are taken to be one. In a classic 2-dimensional Cartesian coordinate system, x correspond to the x-coordinate and y to the y-coordinate, that is, the physical position of the pendulum. Accordingly, v_x is the velocity of the x-component and v_y the velocity of the y-component.

1.3 Solution

The code for solving this problem can be viewed in listing 1, we use the solver CVode.

Listing 1: Script for solving the elastic pendulum problem

```
#lambda(y1, y2, k)
def lambda_func(y1, y2, k = 1):
    return k*(np.sqrt(y1**2 + y2**2) - 1)/np.sqrt(y1**2 + y2**2)

#the right hand side of our problem
def rhs(t, y):
    return np.array([y[2], y[3], -y[0]*lambda_func(y[0], y[1]), -y[1]*lambda_func(y[0], y[1]) - 1])

#initial values. y[0] = x-position, y[1] = y-position, y[2] = x-velocity, y[3] = y-velocity
y0 = np.array([1.0, 0.0, 0.0, -1.0])
t0 = 0.0

#Assimulo stuff
model = Explicit_Problem(rhs, y0, t0)
model.name = 'Task 1'
sim = CVode(model)
sim.maxord = 7
tfinal = 100.0
#simulation. Store result in t and y
t, y = sim.simulate(tfinal)

#Create plots. Three figures are created: one containing positional values as a function of time, one with velocities as a function of time and the last traces the pendulum's movement (x and y coordinates in Cartesian coordinate)
fig, ax = plt.subplots()
ax.plot(t, y[:, 0], label='x-pos')
ax.plot(t, y[:, 1], label='y-pos')
legend = ax.legend(loc='upper center', shadow=True)

plt.figure(1)
fig2, ax2 = plt.subplots()
ax2.plot(t, y[:, 2], label='x-vel')
ax2.plot(t, y[:, 3], label='y-vel')
legend = ax2.legend(loc='upper center', shadow=True)

plt.figure(1)
fig3, ax3 = plt.subplots()
ax3.plot(y[:, 0], y[:, 1], label='displacement')
legend = ax3.legend(loc='upper center', shadow=True)

# Now add the legend with some customizations.
plt.show()
```

1.4 Results

Below some resulting plots of our implementation can be seen. All plots are in the time-interval $[0, 20]$ and the initial values are all zeros except for the x which is set to 1 - without initial elongation (since the pendulum length is 1).

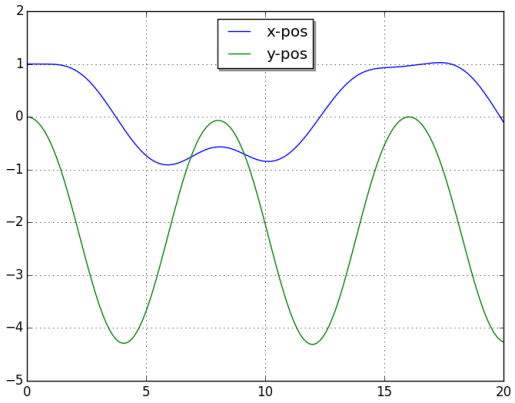

Figure 1: x and y positions as functions of time, with $k = 0.75$

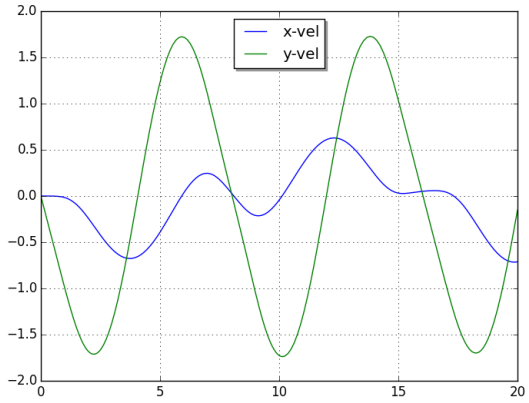

Figure 2: velocities in x- and y-directions as functions of time, with $k = 0.75$

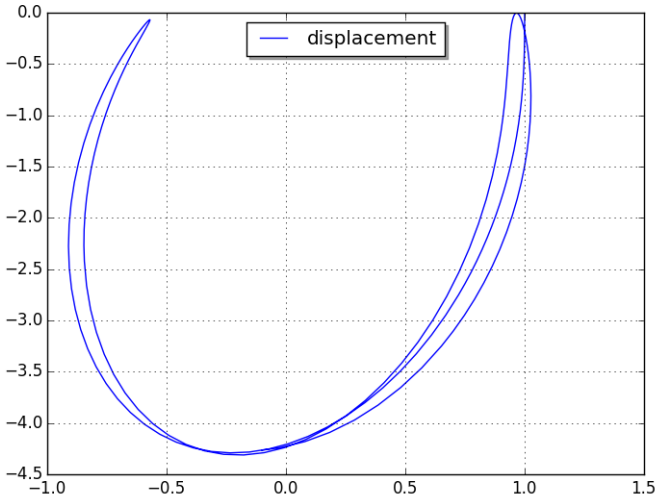

Figure 3: tracings of the pendulum trajectory for $k = 0.75$

In figures 1, 2 and 3 we have used the spring constant $k = 0.75$ which is very elastic as seen specifically in figure 3 where the pendulum does a slight irregular motion. The y-position varies between 0 and about -4 and the x-position varies between 1 and -1 according to figure 1. Also the y-velocity is more regular than the x-velocity, as seen in figure 2.

In contrast, figures 4, 5 and 6 display results of a more stiff pendulum ($k = 100$). As is visible in figure 6, the pendulum trajectory is uniform and rather unexciting. The positional values and velocities are very periodic as you would expect from e.g. a grandfather clock.

To obtain an oscillation around the trajectory of a stiff pendulum we can start with an elongation of the spring (we used $x = 1.2$ as initial value) which would result in the pendulum con-

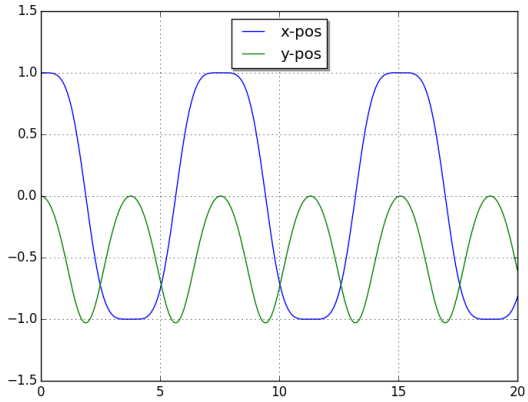

Figure 4: x and y positions as functions of time, with $k = 100$

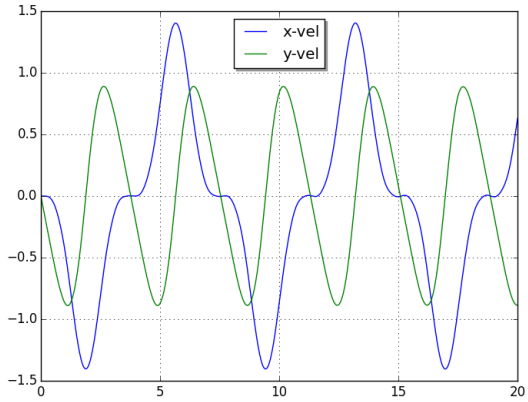

Figure 5: velocities in x- and y-directions as functions of time, with $k = 100$

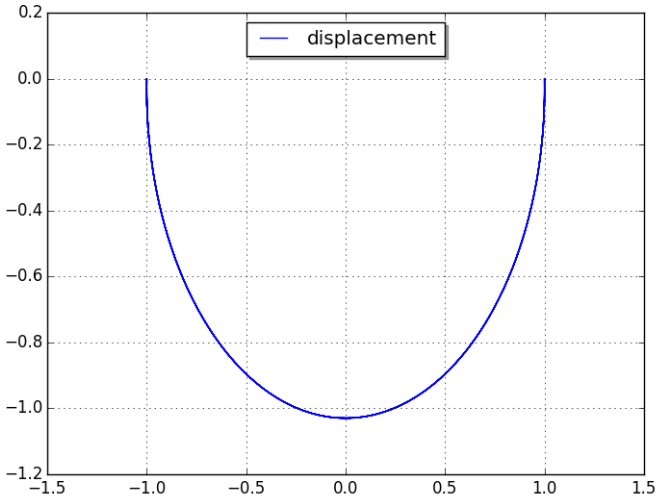

Figure 6: tracings of the pendulum trajectory for $k = 100$

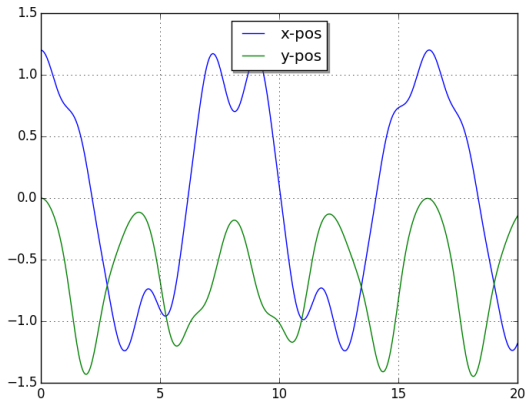

Figure 7: x and y positions as functions of time, with $k = 10$

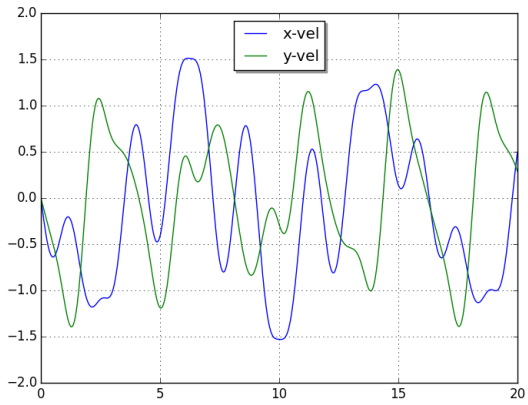

Figure 8: velocities in x- and y-directions as functions of time, with $k = 10$

tracting. Some plots of this are visible in figures 7, 8 and 9. We can also see that the y-position

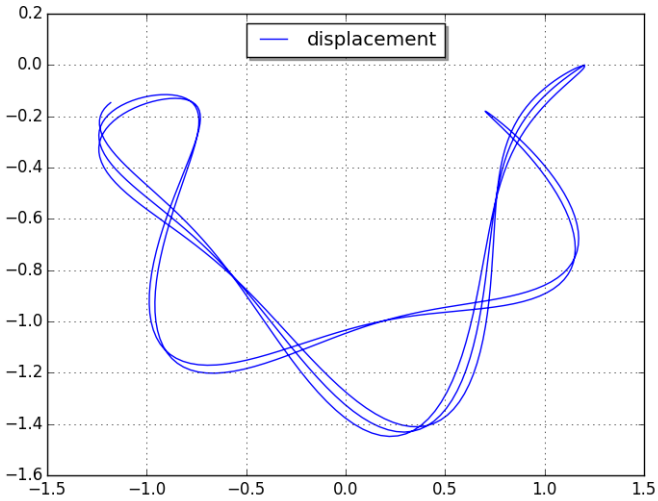

Figure 9: tracings of the pendulum trajectory for $k = 10$

and y-velocity values are a bit more irregular.

1.5 Discussion

We can see that the spring is very elastic in figure 3 since the pendulum is stretched out about 4 times its original size. Despite this, we observe a very low disturbance in the y-position and y-velocity's periodicities. This is due to the force from the spring being much lower than the gravitational force acting on the pendulum. The x-values, however are unaffected by gravity, resulting in a visible effect. This hypothesis is reinforced by the figures from the stiff pendulum, where $k = 100$, in which the x-values also become very periodic.

With a initial elongation of the spring we would expect the y-values to become more irregular since the spring is oscillating around its equilibrium.

1.6 BDF3 and BDF4

Our implementations of BDF-4 and BDF-3 can be seen in listings 2 and 3.

Listing 2: Python implementation of BDF3

```
1     def step_BDF3(self,T,Y):
2         """
3         BDF-3 with fsolve
4         """
5         alpha = [11/6, -18/6, 9/6, -2/6]
6         return self.step_BDFn(T, Y, 3, alpha)
```

Listing 3: Python implementation of BDF4

```
    def step_BDF4(self,T,Y):
        """
        BDF-4 with fsolve
        """
        alpha = [25/12, -48/12, 36/12, -16/12, 3/12]
        return self.step_BDFn(T, Y, 4, alpha)
```

Both of these use the function BDFn, which can be seen in listing 4.

Listing 4: Auxiliary function BDFn

```
    def step_BDFn(self, T, Y, n, alpha):
        """
        Performs a BDF step of order n
        """
        f = self.problem.rhs
        h = self.h
        # predictor
        t_np1 = T[0] + h
        y_np1_i = Y[0] # zero order predictor
        def g(y_np1):
            return alpha[0]*y_np1 + sum([alpha[i+1]*Y[i] for i in range(n)]) - h*f↩
                (t_np1, y_np1)
        y_np1 = so.fsolve(g, y_np1_i)
        return t_np1, y_np1
```

1.7 Assimulo testing for various frequencies using a BDF solver

We are now to test our implementations of the different ordered BDF solvers by analysing stability and accuracy for different spring-constants k, and number of steps N. For all simulations the time interval is from $t = 0$ to $t = 20$, and the initial conditions are $x = 1.2$, $y = 0$, $\dot{x} = 0$, $\dot{y} = 0$. A comparison with the Explicit Euler method is also included.

Euler	$N = 1000$		$N = 10000$	
	stable?	accurate?	stable?	accurate?
$k = 1$	yes	no	yes	yes
$k = 20$	no	no	yes	yes

Table 1: Explicit Euler testing

Explicit Euler becomes unstable very easily, already at $k = 20$. Even at $k = 1$ we see that the method is inaccurate, so Explicit Euler is not to be your first choice concerning solvers. This is to be expected since the stability region of the Explicit Euler method is very small, only a circle with radius 1 centered around (-1,0) in the complex plane. With a fixed step size this method is

not expected to be able to follow the quick oscillations of the pendulum. See figures 10, 11, 12 and 13.

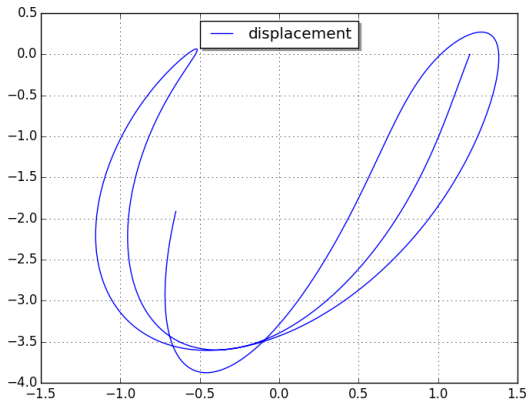

Figure 10: Explicit Euler, $k = 1$, $N = 1000$

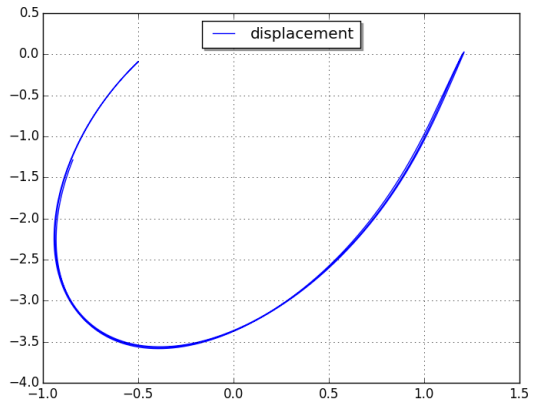

Figure 11: Explicit Euler, $k = 1$, $N = 10000$

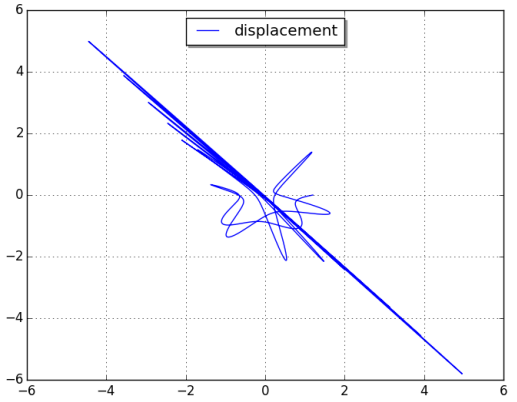

Figure 12: Explicit Euler, $k = 20$, $N = 1000$

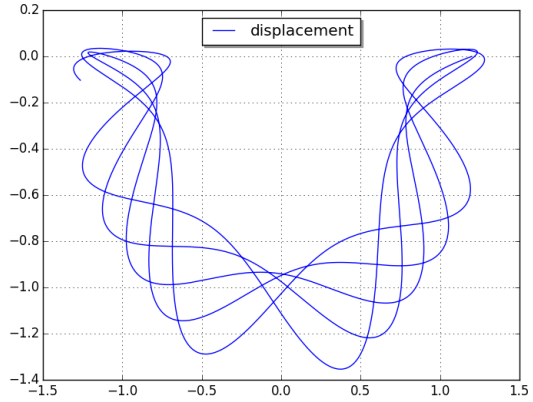

Figure 13: Explicit Euler, $k = 20$, $N = 10000$

BDF2	$N = 1000$		$N = 10000$	
	stable?	accurate?	stable?	accurate?
$k = 100$	yes	no	yes	yes
$k = 2000$	yes	no	yes	yes

Table 2: BDF2 testing

BDF3	$N = 1000$		$N = 10000$	
	stable?	accurate?	stable?	accurate?
$k = 115$	yes	no	yes	yes
$k = 500$	no	no	yes	yes

Table 3: BDF3 testing

For the BDF methods of order 2-4 the stability doesn't seem to be an issue except for very high spring constants k combined with large step sizes. These methods has a large stability

BDF4	$N = 1000$		$N = 10000$	
	stable?	accurate?	stable?	accurate?
$k = 200$	yes	yes	yes	yes
$k = 400$	no	no	yes	yes

Table 4: BDF4 testing

region, only an apple shaped region that is mainly located on the positive real half plane on the complex plane is unstable. The apple shape size increases with the method order, see figure 14.

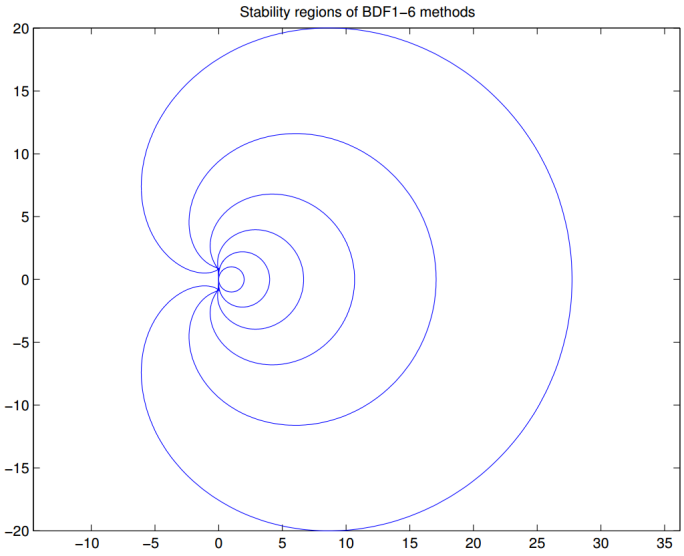

Figure 14: stability regions for BDF methods of order 1-6. The stability region is located outside the apple shapes. Taken from reference [1]

Hence, the stability is easier preserved for BDF2 than for BDF3 or BDF4, where we for example can see that for the huge $k = 2000$ and small amount of steps (that is, large step size) $N = 1000$ stability is maintained. This is to be expected since the stability region for BDF2 is larger than for BDF3 and BDF4. Even though most experiments with the BDF methods are stable, that is they do not diverge over time, there are accuracy issues. In order to obtain accurate results we note that a small step size is required, this is due to the fact that the best results are obtained when we are close to the stability limit, where the step size is small enough to follow the quick oscillations of the pendulum.

In figure 15-18 we see the results of the simulations presented in table 4.

Also note that all cases for $N = 10000$ are stable and accurate, and this is to be expected since we do not alter k to the same degree as N. These cases were mostly intended to compare with the $N = 1000$-cases to see whether the solutions were accurate or not.

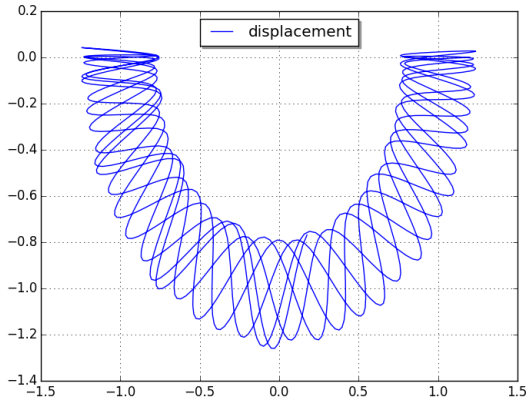

Figure 15: BDF4, $k = 200$, $N = 1000$

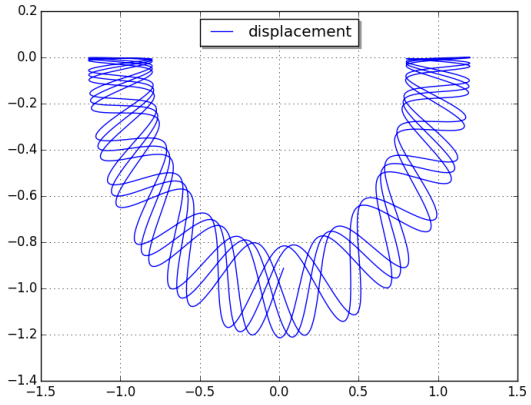

Figure 16: BDF4, $k = 200$, $N = 10000$

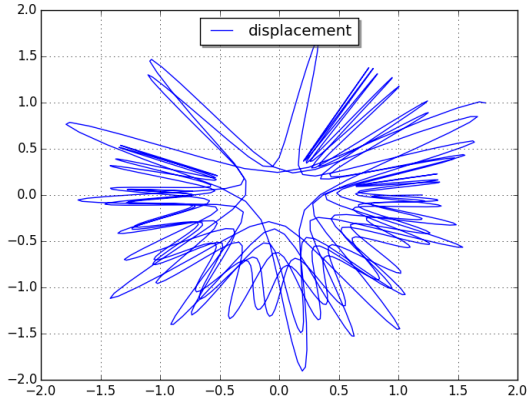

Figure 17: BDF4, $k = 400$, $N = 1000$

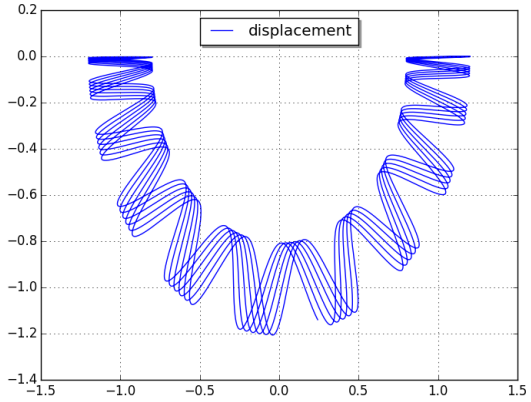

Figure 18: BDF4, $k = 400$, $N = 10000$

1.8 Assimulo testing using CVode

Now we repeat the experiments from Section 3 using the solver CVode again. We also test the influence of absolute tolerance ATOL, relative tolerance RTOL, maximum order MAXORD and the choice of discretization on the performance for the low and highly oscillating case.

1.9 Low oscillation

Here we will test different parameters using $k = 10$ and the initial elongation $x = 1.2$, for a low oscillation.

1.9.1 ATOL and RTOL

Both of these options concern error tolerances and are therefore discussed together. Having a higher tolerance will allow the solver to take larger steps, thus making the solution look choppy as seen in figures 19 and 20.

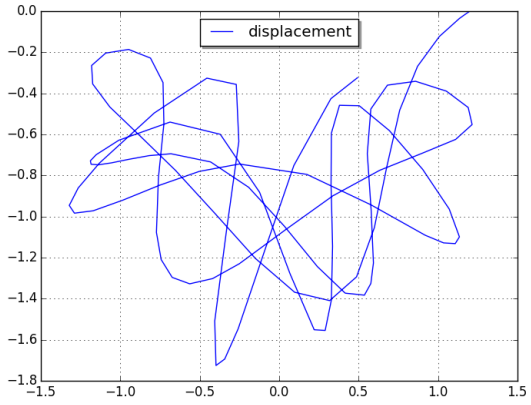

Figure 19: tracings of the pendulum trajectory for $k = 10$ with $rtol = 0.1$

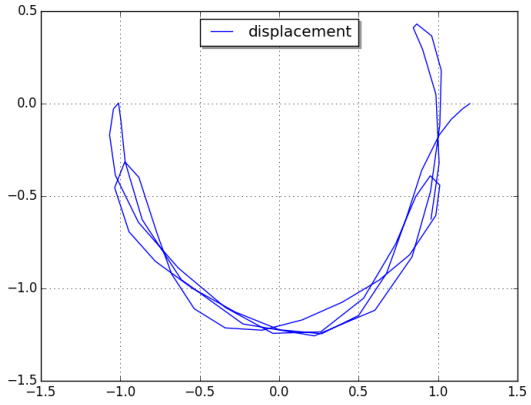

Figure 20: tracings of the pendulum trajectory for $k = 10$ with $atol = 0.1$

1.9.2 Step size and maximum order

The maximum order of a method determines the number of past points that are used as reference for approximating the next value. A high order method will have the potential for taking larger steps at the cost of more computations per step. To ensure the same accuracy with a lower order method we must take smaller steps. For example running our code from task 1 using the order 1 takes 38501 steps, compared to the 536 steps taken using the default order (12).

1.9.3 Choice of discretization

We compared both discretization using Adams method and the BDF method. Both methods generated a displacement-graph which were very much simular to each other. The statistics favor Adams method for this problem in pretty much every way possible (fewer steps, function evaluations, Jacobian evaluations and nonlinear iterations). See table 5 under the high oscillation section for the full values.

1.10 High oscillation

Now we test the different parameters' influence in the highly oscillating case, using $k = 50$ and initial elongation $x = 1.2$.

1.10.1 ATOL and RTOL

1.10.2 Step size and maximum order

The maximum order still only affects the number of steps. The highly oscillating case originally takes more steps than the low oscillation (1100 steps using default order of 12), thus it makes sense that when using maximum order 1 we get about twice as many steps compared to the low oscillation case (76856 steps compared to 38501 above).

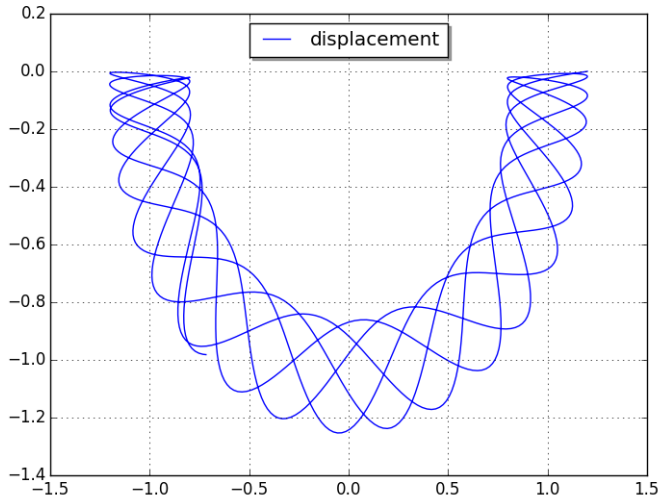

Figure 21: tracings of the pendulum trajectory for $k = 50$ with default a- and rtol

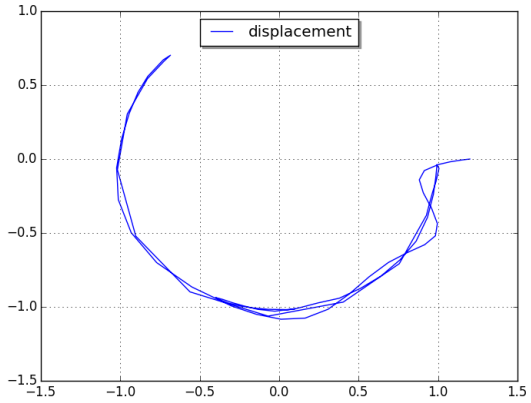

Figure 22: tracings of the pendulum trajectory for $k = 50$ with $atol = 0.1$

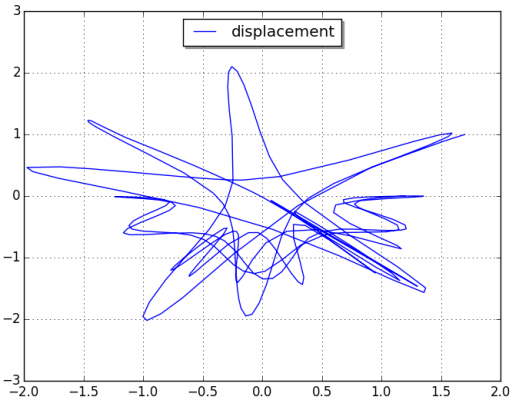

Figure 23: tracings of the pendulum trajectory for $k = 50$ with $rtol = 0.1$

1.10.3 Choice of discretization

Doing the same test as for the low oscillating case we get larger values all over, which is expected since we need more steps and evaluations for the same precision when having high oscillations. Adams method is still favored with slightly lower values compared to BDF. The values are shown in table 5.

	Low oscillation		High oscillation	
	BDF	Adams	BDF	Adams
N	536	381	1100	737
F-eval	580	455	1177	880
J-eval	9	7	19	13
Nonlinear	576	451	1173	876

Table 5: Comparison of BDF and Adams methods for the high and low oscillation cases. Standard values used: $x = 1.2$, $k_{low} = 10/k_{high} = 50$, $tol = 1e-6$ and $t = 0..20$. F-eval is the number of function evaluations, J-eval the number of Jacobian evaluations and Nonlinear the number of nonlinear iterations as a result of an implicit method

1.11 Comparison of Assimulo and Dymola

In this section we will examine the possibilities to simulate the pendulum in Dymola. We will especially consider different integrators and compare them when run in Assimulo and in Dymola. The number of steps that the integrator uses gives us an idea of how large steps the method is taking, thus also how fast the method might be running. Though the actual time consumed in each step is in the function evaluations. Each step does a number of function evaluations, and if a method uses many evaluations it might be less efficient than a method with more steps but fewer evaluations. In the table below our results can be viewed:

As seen in table 6 the Explicit Euler isn't affected by whether we're using Assimulo or Dymola (the step size is 4 times smaller by default in Assimulo).

The Dopri45 integrator on the other hand differs a bit. To start with we can see that this integrator is a 4-5 order Runge-Kutta method, since we have about 6 times as many function evaluations as steps (each step samples the right-hand side at 5 points, the 4th in two distinct ways). The deviations from the multiplicity are as a result of failed steps where we had to do another evaluation with some changes to succeed. For this method Dymola is a bit more efficient (fewer steps), compared to Assimulo.

Looking at the CVode integrator we can see that, compared to Dopri45, we have much fewer function evaluations. This is as a result of CVode being a multi-step method (thus having almost

Method \ Program	Assimulo				Dymola			
	Steps	F-eval	J-eval	Nonlinear	Steps	F-eval	J-eval	Nonlinear
Explicit Euler	2000	2000	0	0	500	500	0	0
Dopri45	201	1208	0	0	193	1155	0	0
CVode	536	580	9	576	544	585	10	581
Lsodar	362	747	0	0	362	747	0	0

Table 6: An overview of statistics for some methods present in both Assimulo and Dymola. Standard values used: $x = 1.2$, $k = 10$, $tol = 1e-6$ and $t = 0..20$. F-eval is the number of function evaluations, J-eval the number of Jacobian evaluations and Nonlinear the number of nonlinear iterations as a result of an implicit method

the same amount of function calls as steps). CVode is also an implicit method, as compared to Dopri45 that is an explicit method. To perform one step with an implicit method we need to solve a non-linear system. To do this we can use either fixed-point iteration or Newton iteration. It seems like CVode uses Newton iteration based on the 9 Jabobian evaluations seen in table 6. If the evaluation of the Jacobian is costly this is something we need to consider when deciding what integration method to use. With respect to number of Jacobian evaluations, Assimulo is more efficient than Dymola.

The last method considered is Lsodar. It seems that the implementation of this method uses 2 function evaluations per step and it is also an explicit method since we avoid Jacobian evaluations and nonlinear iterations. The total number of function calls are greater than for CVode though. Using this integrator we have no difference between number of steps and function calls in Assimulo and Dymola.

On a particular machine we had the same CPU-time for CVode and Lsodar so we can compare the cost for Jacobian evaluations to function evaluations. We have (in Dymola) $747 - 585 = 162$ function evaluations more for Lsodar than CVode. This is compensated by 10 Jacobian evaluations, so we can assume that a Jacobian evaluation is about 16 times as costly as a normal function evaluation for this problem. Also the only conclusion we can draw about recommending an integrator for this problem is that Dopri45 is a bit slower than CVode and Lsodar and that Explicit Euler gives a really bad solution. Thus using either CVode or Lsodar seems reasonable.

2 Andrew's Squeezer Mechanism

2.1 Introduction

Project 2 introduces a mechanical construct, Andrew's Squeezer Mechanism, which we are supposed to model in both Assimulo and Dymola. This mechanism will require the introduction of Differential Algebraeic Equations, DAEs. We will make comparisons between the implementations in Assimulo and Dymola as well as between the results and the runtime statistics.

2.2 Theory

Andrew's squeezer mechanism is a planar multibody mechanism consisting of 7 rigid bodies, who are all connected via joints without any friction. The squeezer is assumed to be driven by a motor with constant drive torque. The equations of motion are derived using the angles in the system $q = [\beta, \Theta, \gamma, \phi, \delta, \Omega, \epsilon]$. The Index of a system of DAE is the number of derivatives necessary to find consistent initial values. We get the Index 3 Formulation with v and w as newly introduced variables:

$$\dot{q} = v \tag{5}$$

$$\dot{v} = w \tag{6}$$

$$0 = M(q) \cdot w - f(q, v) + G^T(q) \cdot \lambda \tag{7}$$

$$0 = g(q)\,. \tag{8}$$

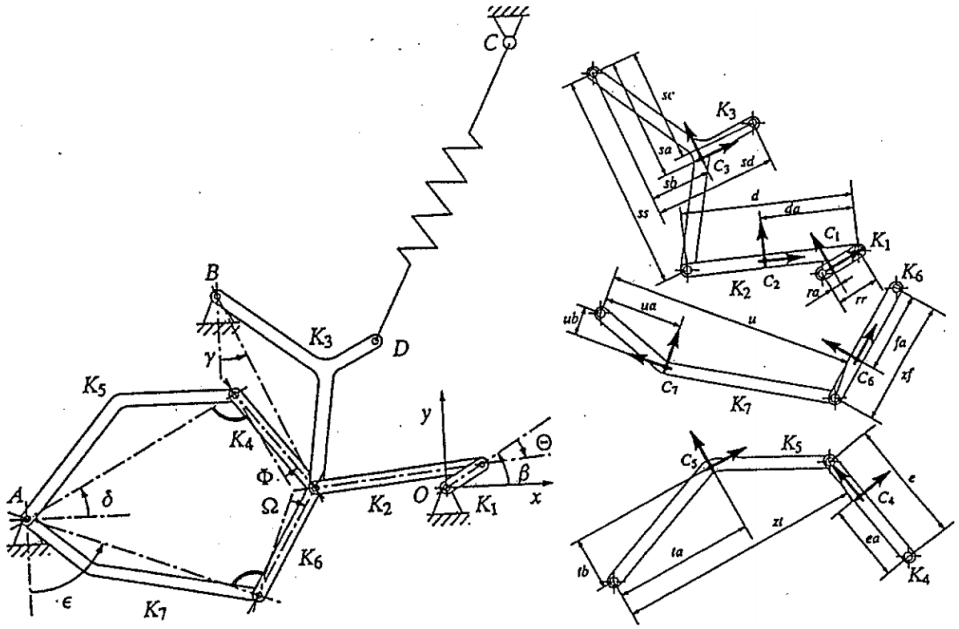

Figure 24: Representation of a squeezer from reference [2]

To obtain the Index 2 Formulation we differentiate equation 8 with respect to q and get:

$$0 = G(q)v, \, , \tag{9}$$

where $G(q)$ is the Jacobian of $g(q)$. We can get the Index 1 Formulation with an additional differentiation:

$$0 = g_{qq}(q)(v, v) + G(q)w, \, , \tag{10}$$

where $g_{qq}(q)(v, v) = v^T \cdot g_{qq}(q) \cdot v$.

In this project, we set up a model of this squeezer mechanism and simulate it using first Assimulo, then Dymola. For both softwares we implement the model with parameters and constraints as reported by Hairer and Wanner in reference [2].

2.3 Assimulo

2.3.1 Initial values

Hairer and Wanner provide the initial conditions for solving the DAE describing Andrew's squeezer mechanism in [2, p.535]. By fixing the parameter $\theta = 0$ and then computing the solution to $g(q) = 0$ using Newton iteration, the initial values can be derived. In table 7 the given and calculated initial values are reported, together with the difference between them. We observe that the calculated and the given initial differ very little, so any of the values can be used.

These initial values where derived providing the Newton iteration with an initial guess that is fairly close to the initial values given in reference [2]. When trying to find the initial values by providing an initial guess that is not close to the these values, we obtain a different set of initial values. We can therefore conclude that $g(q)$ has more than one local minimum and there isn't

Angle	Given values	Calculated values	Difference
β	-0.061	-0.061	$2.9 \cdot 10^{-12}$
θ	Fixed to 0	Fixed to 0	-
γ	0.455	0.455	$2.4 \cdot 10^{-12}$
ϕ	0.223	0.223	$1.8 \cdot 10^{-11}$
δ	0.487	0.487	$-3.4 \cdot 10^{-12}$
ω	-0.223	-0.223	$1.1 \cdot 10^{-12}$
ϵ	1.231	1.231	$-1.8 \cdot 10^{-11}$

Table 7: Difference between the given initial values and the ones calculated using Newton iteration

a guarantee that the Newton iteration will converge to a unique set of initial values. Also note that there might be initial guesses for which Newton's method fails.

2.3.2 Simulation

The DAE model with index 3 and index 2 formulation was implemented in python and simulated using the IDA solver in Assimulo. The time interval $[0, 0.03]$ seconds was used for all simulations, with the number of communication points set to 500. In order to obtain a solution the solver might require some variables to be suppressed due to the introduction of a local error. This error does not affect the solution globally but the error might still be larger than the selected tolerance which leads to the solver failing. This can be done either by changing the tolerances for the different variables $[v, \dot{v}, \lambda]$, or by using the python command `suppress_alg` which controls the error estimator of the solver. The different tolerance and suppression settings for the index 3 formulation are presented in table 8, and for the index 2 formulation in table 9.

Variables	Tolerance	Suppressed variables	Solver success?
$[v, \dot{v}, \lambda]$	$[10^{-6}, 10^{-6}, 10^{-6}]$	-	No
$[v, \dot{v}, \lambda]$	$[10^{-6}, 10^{-6}, 10^{5}]$	-	No
$[v, \dot{v}, \lambda]$	$[10^{-6}, 10^{5}, 10^{5}]$	-	Yes
$[v, \dot{v}, \lambda]$	$[10^{-6}, 10^{-6}, 10^{-6}]$	λ	No
$[v, \dot{v}, \lambda]$	$[10^{-6}, 10^{-6}, 10^{-6}]$	$\dot{v}, \lambda$	No
$[v, \dot{v}, \lambda]$	$[10^{-6}, 10^{-6}, 10^{5}]$	λ	No
$[v, \dot{v}, \lambda]$	$[10^{-6}, 10^{5}, 10^{5}]$	$\dot{v}, \lambda$	Yes

Table 8: Simulation settings for the index 3 formulation

We note that we need to set the tolerance of λ to 10^5 in order for the method to succeed for the index 2 formulation. For index 3 both $\dot{v}$ and λ need the tolerance 10^5. This is due to the fact that the Newton iteration matrix becomes ill-conditioned for some index formulations. The higher the index formulation, the worse conditioning we get, and therefore issues with method failure.

In figures 25 and 26 the simulation results are presented for index 3 and index 2 respectively.

Variables	Tolerance	Suppressed variables	Solver success?
$[v, \dot{v}, \lambda]$	$[10^{-6}, 10^{-6}, 10^{-6}]$	-	No
$[v, \dot{v}, \lambda]$	$[10^{-6}, 10^{-6}, 10^{5}]$	-	Yes
$[v, \dot{v}, \lambda]$	$[10^{-6}, 10^{5}, 10^{5}]$	-	Yes
$[v, \dot{v}, \lambda]$	$[10^{-6}, 10^{-6}, 10^{-6}]$	λ	No
$[v, \dot{v}, \lambda]$	$[10^{-6}, 10^{-6}, 10^{-6}]$	$\dot{v}, \lambda$	No
$[v, \dot{v}, \lambda]$	$[10^{-6}, 10^{-6}, 10^{5}]$	λ	Yes
$[v, \dot{v}, \lambda]$	$[10^{-6}, 10^{5}, 10^{5}]$	$\dot{v}, \lambda$	Yes

Table 9: Simulation settings for the index 2 formulation

Both plots shows a similair result, which also corresponds to the simulation results presented by Hairer and Wanner in reference [2].

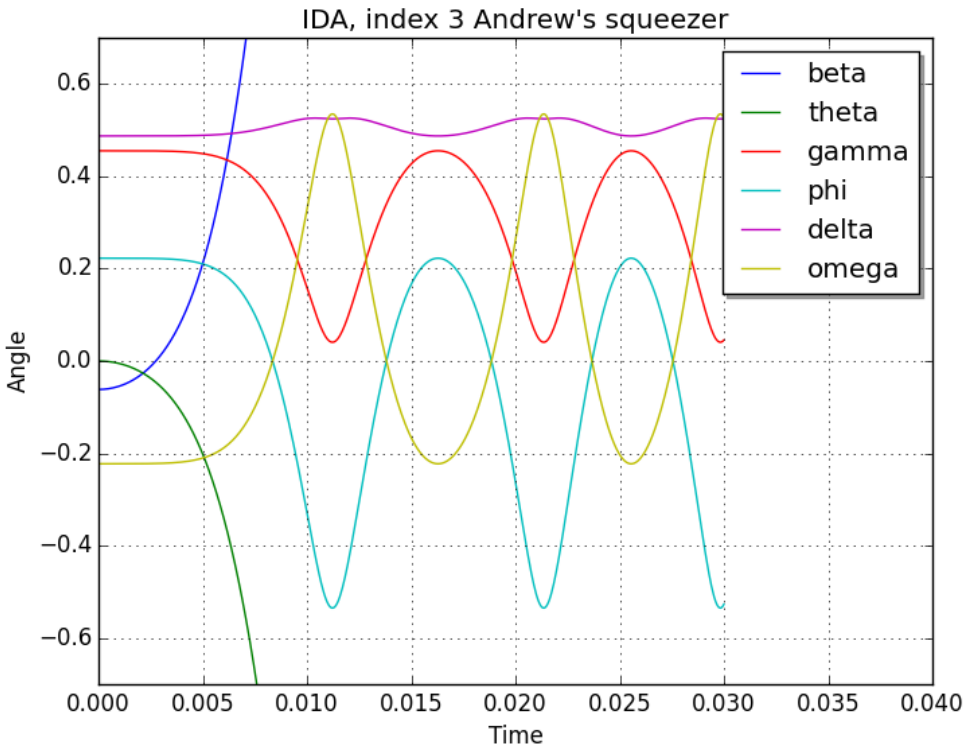

Figure 25: Simulation results for index 3 with high tolerances and suppressed $\dot{v}$ and λ

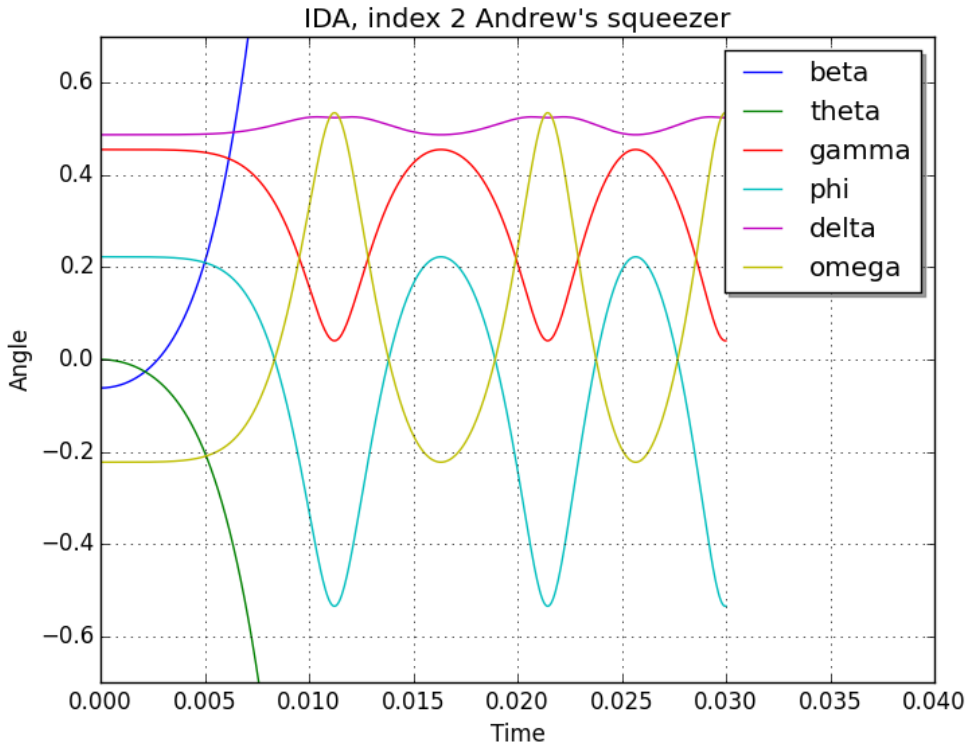

Figure 26: Simulation results for index 2 with high tolerance and suppressed λ

The simulation was also performed for another set of initial values obtained by Newton iteration, $\beta = 12.5$, $\theta = 0$, $\gamma = 0.46$, $\phi = 9.20$, $\delta = 12.23$, $\omega = -6.51$, and $\epsilon = 7.51$. The simulation did not return satisfactory results.

2.3.3 Lagrange multipliers

When observing the simulation results for the Lagrange multipliers λ in figures 27 and 28, we note that the solution has some local instabilities for the index 3 formulation, while the index 2 formulation looks smooth and stable. In order to be able to study the step sizes used, shown in figures 29 and 30, no number of communication points where specified in this simulation.

For the index 3 formulation, we note that the step size changes quickly between two values during several parts of the simulation. The timing of these changes coincide with the local instabilities in the Lagrange multipliers, where the solver seem to have issues in controlling the error. We do not observe the same step size variations for the index 2 formulation, and the solution looks stable. Hence, we draw the conclusion that the suppressed velocities $\dot{v}$ in the index 3 formulation influence the error control of the solver, but if we would have allowed the controller to use the velocities we would encounter method failure for the Newton iteration instead, as discussed above.

2.3.4 Runtime statistics

In table 10 the runtime statistics for solving the index 2 and index 3 formulation using IDA with 500 communication points can be seen. We see that we have more error test faliures for the

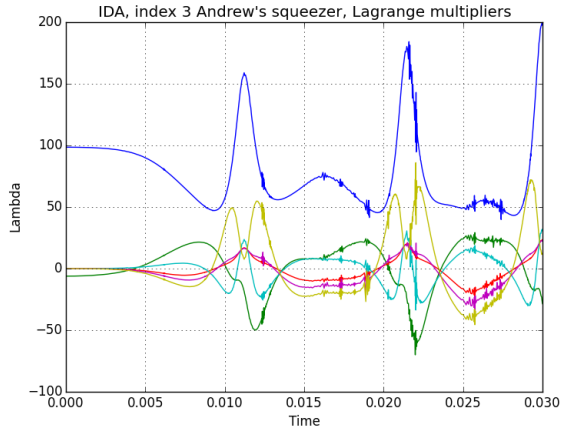

Figure 27: Simulation results of the Lagrange multipliers λ for index 3, with high tolerances and suppressed $\dot{v}$ and λ

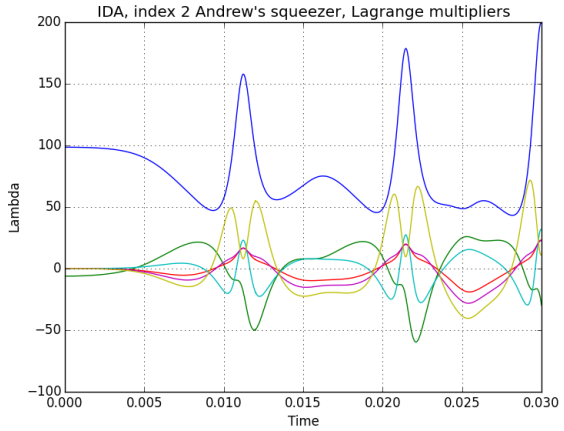

Figure 28: Simulation results of the Lagrange multipliers λ for index 2, with high tolerance and suppressed λ

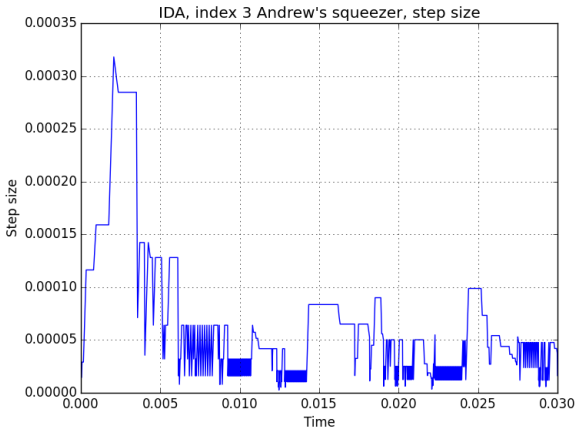

Figure 29: Step sizes for the index 3 simulation

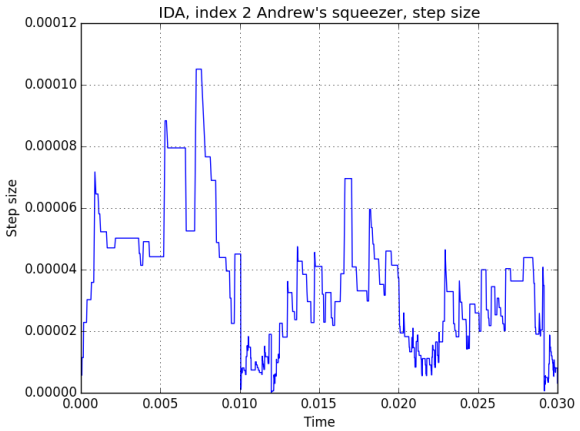

Figure 30: Step sizes for the index 2 simulation

index 2 formulation, which is to be expected since the velocities $\dot{v}$ are not suppressed as in the index 3 case. This yield more steps and function evaluations for the index 2 case. For the index 3 case we have more nonlinear convergence faliures, and therfore more of Jacobian evaluations and nonlinear function evaluations due to Jacobian evaluations. This is also to be expected, since the Newton iteration matrix is more ill-conditioned for the index 3 formulation than for index 2.

Index	2	3
Steps	1603	843
Function evaluations	2692	1811
Jacobian evaluations	227	834
Function eval. due to Jacobian eval.	4540	16680
Error test failures	81	9
Nonlinear iterations	2692	1811
Nonlinear convergence failures	1	261

Table 10: Runtime statistics for index 2 and index 3 formulation when solving with IDA

2.3.5 Index 1 formulation

Next we investigate the index 1 formulation of the problem and try to solve it using an explicit Runge-Kutta method, Dormand-Prince method (DOPRI5). To do this we first need to reformulate the residual form as an explicit problem of the form:

$$\dot{y} = f(t, y) \tag{11}$$

$$y(t_0) = y_0 \,. \tag{12}$$

From (7) and (10) we get the linear system:

$$\begin{pmatrix} M & G^T \\ G & 0 \end{pmatrix} \begin{pmatrix} w \\ \lambda \end{pmatrix} = \begin{pmatrix} f \\ -g_{qq}(v, v) \end{pmatrix} \,. \tag{13}$$

Solving this and inserting the resulting w into (5) and (6) we get an explicit problem that can be solved using DOPRI5. The result of the simulation with DOPRI5 can be seen in figure 32. We see that there are significant drift-off errors, which is a well-known effect of reducing the index of a problem. This is made more visible in figure 31

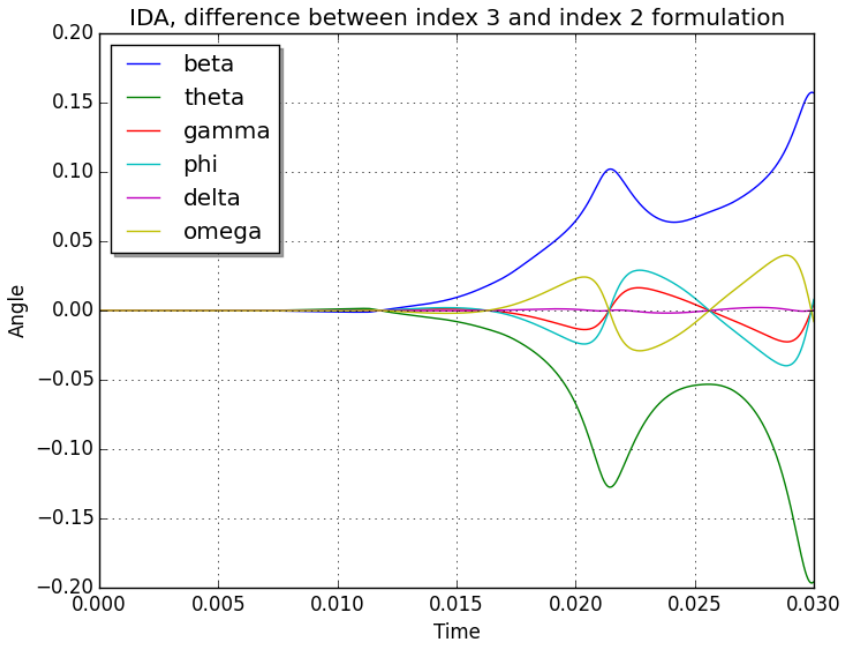

Figure 31: Drift-off error when reducing index from 3 to 2

The runtime statistics can be seen in table 11.

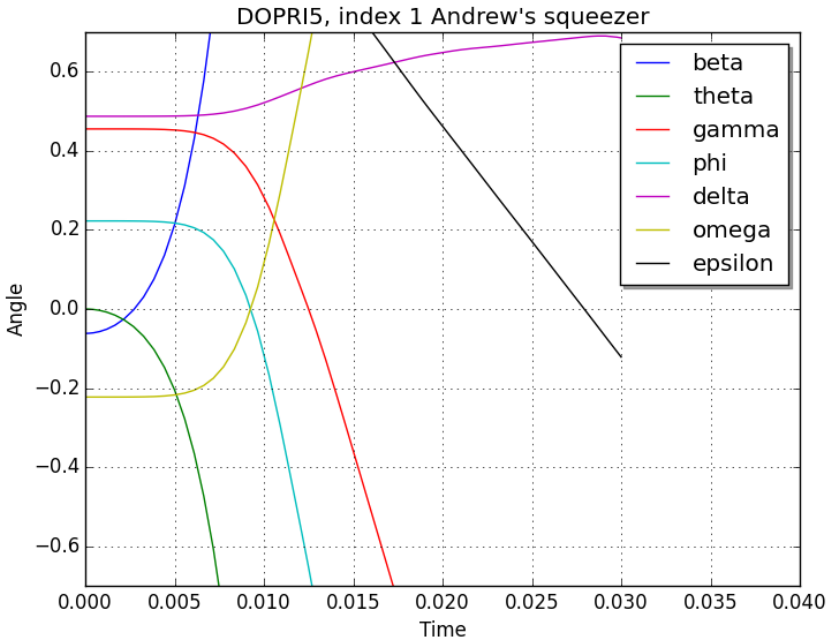

Figure 32: Simulation results for the index 1 formulation using DOPRI5

Steps	85
Function evaluations	704
Error test failures	32

Table 11: Runtime statistics for the index 1 formulation when solving with DOPRI5

2.4 Dymola

Dymola supports Modelica standard library which lets us "drag and drop" parts to build our (mechanical) systems with. Thus, using mainly the multibody library (see reference [3]) we were able to construct the squeezer with a combination of springs, revolutes, masses and a torque. Dymola then converts the connections between the different objects into a system of DAEs which it solves and simulates over time.

2.4.1 Degrees of freedom and constraints

Since the multibody library supports mechanical constructs in three dimensions and our squeezer mechanism only moves in two dimensions the underlying system of DAEs will become overconstrained if we use too many three-dimensional revolutes. A body in three dimensions has 6 degrees of freedom (x, y and z positions, rotation around the x, y and z axis). An ordinary revolute has 5 constraints and 1 degree of freedom (rotation around an axis). So connecting a revolute to a body will reduce its degrees of freedom by 5, thus giving it only one. Now consider a loop with 4 bodies, connected via 4 revolutes. That gives:

- 4 bodies = 24 degrees of freedom.

- 4 revolutes = 20 constraints.

So we still have $24 - 20 = 4$ degrees of freedom (rotation around 4 revolutes). If we fix one of the revolutes at a point on the other hand, we lose 6 degrees of freedom resulting in an overconstrained system. In reality, the system should still be able to rotate around the point which is fixed and this is solved by introducing a planar revolute which only has 2 constraints, assuming that the system is planar and not spatial (which is the case for the squeezer). Since the system is now combined in such a way that we no longer have the possibility to move in three dimensions, constraining the dimension in which we already cannot move makes no sense.

2.4.2 Setup of the model in Dymola

To set up the model in Dymola we need to calculate each mass centre relative to its respective frame which is easily done using the values provided by [2, Tables 7.1, 7.2] together with figure 24. Slight modification is needed since Hairer and Wanner calculates mass centra relative to the origin. Since all mass centra except C_3, C_5 and C_7 are located inside the mass objects we usually get mass centre vectors with only x-components. Even C_3, C_5 and C_7 are easily calculated since the lengths were all given in the correct coordinate system as seen in figure 24. Note that all crooked parts are modelled in Dymola as straight lines with mass centre outside of the body which behaves the same physically.

One interesting part of the squeezer in figure 24 is the part K_3 which is connected at 3 points rather than 2. To model this in Dymola we used a fixed translation which basically moves the connection point of K_4, K_6, K_2 and K_3 to the point D for the spring only. The fixed translation vector is also trivial to calculate since lengths still are given in the correct coordinate system.

The Dymola model can be seen in figure 33. As one can see some of the components are rotated which also implies that their internal coordinate systems are rotated. This is to be considered when setting the internal vector for mass centre - the positive x-axis of a body shape is always in the direction a to b.

In table 12 the mass, length and relative mass centre vectors used in our Dymola model can be viewed. Remember that each part was modelled as a straight body shape in Dymola, thus the length always has a component in the x-axis only.

C	x	y	mass	length	inertia
$C1$	0.00092	0	0.04325	0.007	$2.194 \cdot 10^{-6}$
$C2$	0.0165	0	0.00365	0.028	$4.410 \cdot 10^{-7}$
$C3$	0.01626	0.01043	0.02373	0.035	$5.255 \cdot 10^{-6}$
$C4$	0.01421	0	0.00706	0.02	$5.667 \cdot 10^{-7}$
$C5$	0.02308	0.00916	0.07050	0.04	$1.169 \cdot 10^{-5}$
$C6$	0.01421	0	0.00706	0.02	$5.667 \cdot 10^{-7}$
$C7$	0.01228	-0.00449	0.05498	0.04	$1.912 \cdot 10^{-5}$

Table 12: Information concerning mass centre of the bodies used in the model

The coordinates for the fixed points can be seen in figure 13. The spring constant used

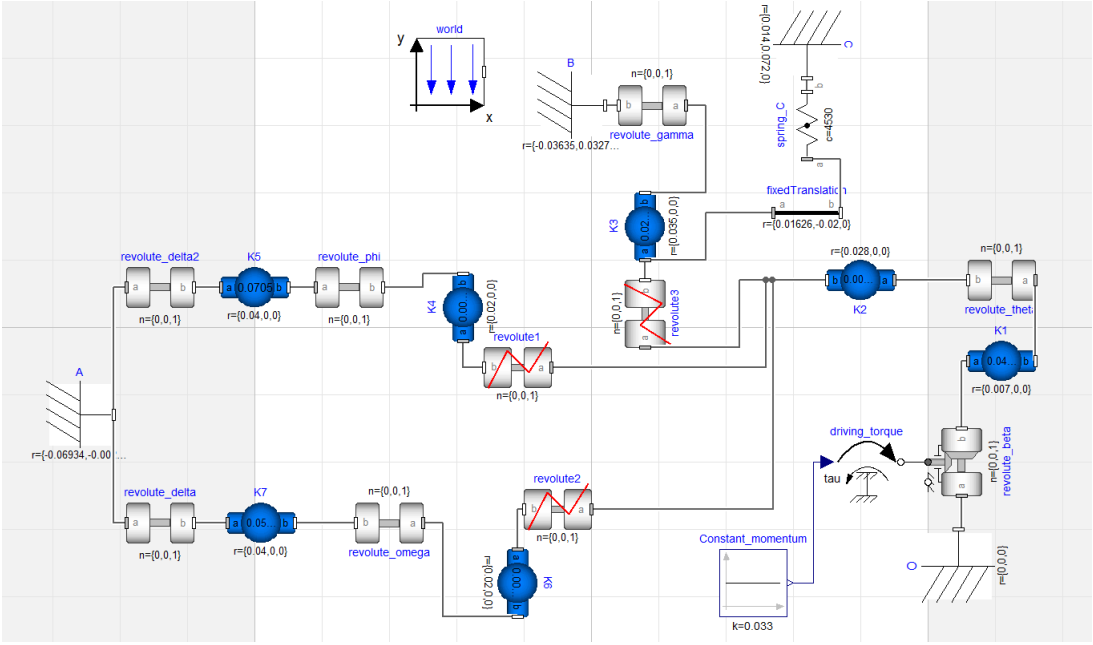

Figure 33: Dymola representation of a squeezer

was $c = 4530$ and its unstretched length was $l = 0.07785$. The fixed translation for the body K_3 resulted in $(0.01626, -0.02, 0)$ which is relative to the point a of K_3. Remember the rotation of K_3, see figure 33, so the translation is located above (positive x-direction) and to the right (negative y-direction) of the body K_3 - which is expected according to figure 24. The constant momentum used to drive the revolute in the origin is set to $k = 0.033$.

	x	y
O	0	0
A	-0.06934	-0.00227
B	-0.03635	0.03273
C	0.014	0.072

Table 13: Planar coordinates for the fixed points

2.4.3 Results

Running the simulation in Dymola gives us a result like figure 34 (many objects such as revolutes, spring forces and fixed points are hidden). The angles are plotted in figure 35. We can see that the angle plot is similar to the one we got from simulating in Assimulo.

2.4.4 Runtime statistics

Running our model in Dymola, simulating the time interval $[0, 0.3]$ with 500 communication points gives the runtime statistics in table 14.

If compared with the runtime statistics when running the model in Assimulo (table 10) we notice that much fewer computations are needed. This also shows in the CPU time, which is more than 10 times faster for Dymola.

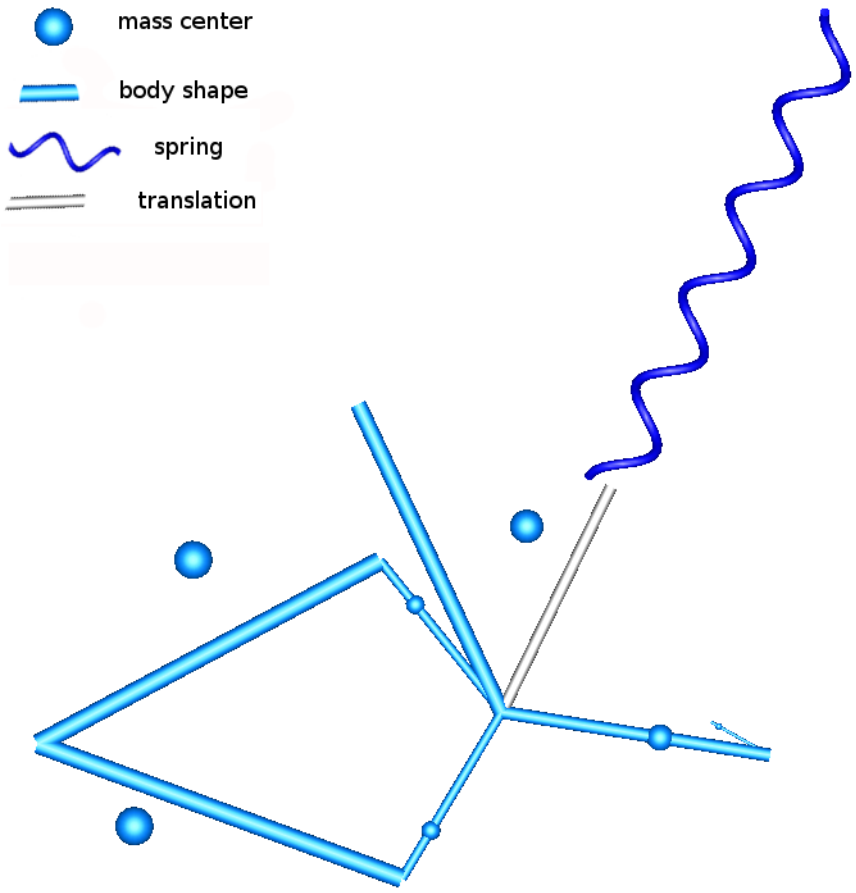

Figure 34: Simulation of the squeezer model in Dymola

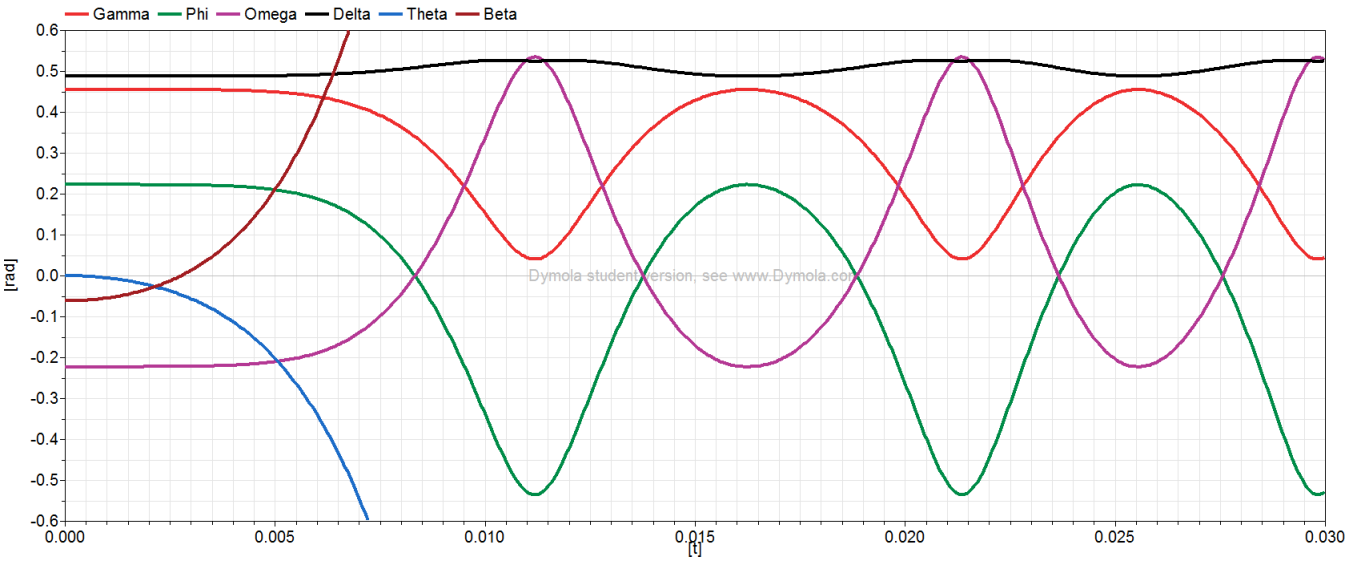

Figure 35: Simulation of the squeezer model in Dymola

Successful steps	169
Function evaluations	421
Jacobian evaluations	30

Table 14: Runtime statistics for simulating the squeezer mechanism in Dymola

2.5 Discussion

In this project we have simulated the Andrew's squeezer mechanism in two different softwares. When simulating in Assimulo the main focus is directed towards the differential equations and how to adapt the problem to the solver in order to obtain convergent and accurate solutions. One example of this is that we needed to suppress different combinations of variables in order for the Newton iteration not to fail, thus requiring a deeper understanding for the underlying equations and how they are solved. For Dymola on the other hand, we obtained accurate results by focusing on the physics and mechanics of the system, focusing a lot on the definition of different coordinate systems and mass centra. The DAEs were formulated and evaluated by Dymola, and worked well if the drag and drop model was correctly implemented. Here there was a need for a deeper understanding of how the components of Modelica standard library worked as well as some understanding regarding degrees of freedom and constraints.

3 The Woodpecker Toy

3.1 Introduction

The goal for the third project was to introduce methods for handling models with discontinuities. In our case, this was to be done using the woodpecker toy as an example of this phenomenon. Solutions will be produced in both Assimulo and Dymola together with a utilitary comparison between the two.

3.2 Theory

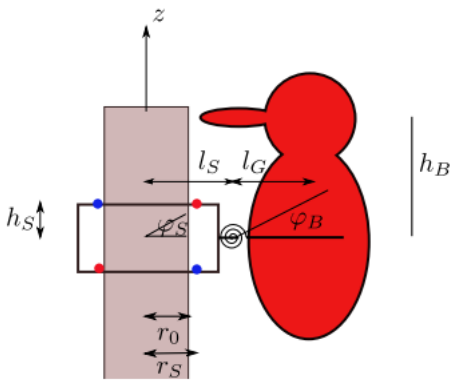

Figure 36: Representation of a woodpecker toy, taken from reference [4]

The woodpecker toy consists of two bodies. One is a sleeve that is connected to a fixed bar and the other is a bird that is connected to the sleeve through a spring. The sleeve has two degrees of freedom: vertical translation along the bar and rotation along the angle φ_S. The bird has one degree of freedom: the rotation angle φ_B relative to the sleeve. At certain conditions discontinuities are introduced: when the sleeve gets blocked on the bar, and when the bird's beak hits the bar. To model this, four different states are introduced:

- State I: The system is free.
- State II: The sleeve is blocked at the lower right and upper left corners.
- State III: The sleeve is blocked at the lower left and upper right corners.
- State IV: The beak hits the bar.

The model has different equations depending on the state it is in, and when there is a state change a discontinuity is reached and as previously stated the model must be able to handle this. When the beak hits the bar, the system will immediately go back to state III. Thus, it never lingers in state IV which has no equations of its own.

- **State I**

$$(m_S + m_B)\ddot{z} + m_B l_S \ddot{\varphi_S} + m_B l_G \ddot{\varphi_B} = -(m_S + m_B)g \tag{14}$$
$$(m_B l_S)\ddot{z} + (J_S + m_B l_S^2)\ddot{\varphi_S} + (m_B l_S l_G)\ddot{\varphi_B} = c_p(\varphi_B - \varphi_S) - m_B l_S g - \lambda_1 \tag{15}$$
$$(m_B l_G)\ddot{z} + (m_B l_S l_G)\ddot{\varphi_S} + (J_B + m_B l_G^2)\ddot{\varphi_B} = c_p(\varphi_S - \varphi_B) - m_B l_G g - \lambda_2 \tag{16}$$
$$0 = \lambda_1 \tag{17}$$
$$0 = \lambda_2\,. \tag{18}$$

- **State II**

$$(m_S + m_B)\ddot{z} + m_B l_S \ddot{\varphi_S} + m_B l_G \ddot{\varphi_B} = -(m_S + m_B)g - \lambda_2 \tag{19}$$
$$(m_B l_S)\ddot{z} + (J_S + m_B l_S^2)\ddot{\varphi_S} + (m_B l_S l_G)\ddot{\varphi_B} = c_p(\varphi_B - \varphi_S) - m_B l_S g - h_S\lambda_1 - r_S\lambda_2 \tag{20}$$
$$(m_B l_G)\ddot{z} + (m_B l_S l_G)\ddot{\varphi_S} + (J_B + m_B l_G^2)\ddot{\varphi_B} = c_p(\varphi_S - \varphi_B) - m_B l_G g \tag{21}$$
$$0 = (r_S - r_0) + h_S\varphi_S \tag{22}$$
$$0 = \dot{z} + r_S\dot{\varphi_S}\,. \tag{23}$$

- **State III**

$$(m_S + m_B)\ddot{z} + m_B l_S \ddot{\varphi_S} + m_B l_G \ddot{\varphi_B} = -(m_S + m_B)g - \lambda_2 \tag{24}$$
$$(m_B l_S)\ddot{z} + (J_S + m_B l_S^2)\ddot{\varphi_S} + (m_B l_S l_G)\ddot{\varphi_B} = c_p(\varphi_B - \varphi_S) - m_B l_S g + h_S\lambda_1 - r_S\lambda_2 \tag{25}$$
$$(m_B l_G)\ddot{z} + (m_B l_S l_G)\ddot{\varphi_S} + (J_B + m_B l_G^2)\ddot{\varphi_B} = c_p(\varphi_S - \varphi_B) - m_B l_G g \tag{26}$$
$$0 = (r_S - r_0) - h_S\varphi_S \tag{27}$$
$$0 = \dot{z} + r_S\dot{\varphi_S}\,. \tag{28}$$

The values of all parameters can be found in reference [4]. In state II and III, the fourth equation is given as an index 3 formulation and the fifth equation has an index 2 formulation. After reducing the problem to index 1, we will have the following equations:

- **State II:**

$$0 = h_S \ddot{\varphi}_S \tag{29}$$

$$0 = \ddot{z} + r_S \ddot{\varphi}_S \,. \tag{30}$$

- **State III:**

$$0 = -h_S \ddot{\varphi}_S \tag{31}$$

$$0 = \ddot{z} + r_S \ddot{\varphi}_S \,. \tag{32}$$

3.2.1 Discontinuities

When a discontinuity in reached, the model will transition into another state. Below are the criteria for when a transition should be made:

- **State I $\rightarrow$ State II:**
  - When $\dot{\varphi_B} < 0$ and $h_S \varphi_S = -(r_S - r_0)$.
- **State I $\rightarrow$ State III:**
  - When $\dot{\varphi_B} > 0$ and $h_S \varphi_S = (r_S - r_0)$.
- **State II $\rightarrow$ state I:**
  - When λ_1 changes sign.
- **State III $\rightarrow$ state I:**
  - When $\dot{\varphi_B} < 0$ and λ_1 changes sign.
- **State III $\rightarrow$ state IV:**
  - When $\dot{\varphi_B} > 0$ and $h_B \varphi_B = l_S + l_G - l_B - r_0$.
- **State IV $\rightarrow$ state III:**
  - Immediately when state IV is reached the sign of φ_B is changed and the model goes back to state III.

When the model transitions from state I to state II or III, momentum conservation must be handled. We will have that $\dot{z} = 0$, $\dot{\varphi_S} = 0$ and

$$\dot{\varphi_B} = \frac{m_B l_G \dot{z} + (m_B l_S l_G)\dot{\varphi_S} + (J_B + m_B l_G^2)\dot{\varphi_B}}{(J_B + m_B l_G^2)} \,.$$

Note that the previous values of $\dot{z}, \dot{\varphi_S}$ and $\dot{\varphi_B}$ are used when computing the new $\dot{\varphi_B}$.

3.3 Assimulo

Assimulo comes with special functions to handle discontinuities. In addition to the residual function that you provide as normal as you do when solving any other implicit problem in Assimulo you can set properties "state_events" and "handle_event". Note that the different states are tracked with a variable "sw" which is user implemented. We chose to implement this as a boolean vector with 3 values that are set to true if the respective state is currently active. The residual function then obviously depends on this state vector, sw (the matrices for calculating the residual change since the underlying DAE changes with the state). The state_events function takes the same parameters as the residual function (i.e. t, y, yd, sw) and basically acts as a tracker of events. Each transition from a state to another in the model is transcribed into a state event function, which is defined such that it changes sign exactly at the time when the event occurs. Assimulo then checks the state event functions at each integration step, and when any of them change sign the handle event function is called. This function makes sure switching states occurs correctly.

3.3.1 Residual function

Implementing the residual function requires some forethought. The mass matrix, applied forces matrix and composition of the residual remains the same, but the g matrix and the constraints change depending on the current state. Writing this in Python according to the theory in section 3.2 looks like in listing 5.

Listing 5: Residual function

```
1  def pecker(t, y, yd, sw): #index 1
2      #y = [z, phi_s, phi_b, z', phi_s', phi_b', lambda_1, lambda_2]
3      #yd = [z', phi_s', phi_b', z'', phi_s'', phi_b'', lambda_1', lambda_2↩
           ']
4
5      #initial computations and assignments
6      lamb = y[6:8]
7      z = y[0]
8      phi_s = y[1]
9      phi_b = y[2]
10     zp = y[3]
11     phi_sp = y[4]
12     phi_bp = y[5]
13
14     #Mass matrix
15     m = np.zeros((3, 3))
16     m[0, 0] = m_s + m_b
17     m[1, 0] = m_b * l_s
18     m[2, 0] = m_b * l_g
19     m[0, 1] = m_b * l_s
20     m[1, 1] = J_s + m_b * l_s**2
```

```
    m[2, 1] = m_b * l_s * l_g
    m[0, 2] = m_b * l_g
    m[1, 2] = m_b * l_s * l_g
    m[2, 2] = J_b + m_b * l_g**2

    #Applied forces (f matrix)
    ff = np.array([-g * (m_s + m_b),
        c_p * (phi_b - phi_s) - m_b * l_s * g,
        c_p * (phi_s - phi_b) - m_b * l_g * g])

    #Constraint matrix G
    gp = np.zeros((2, 3))

    #index 1 constraints
    gyy = np.zeros((2,))

    if sw[0]: #state 1
        gp[0, 0] = 0
        gp[1, 0] = 0
        gp[0, 1] = 1
        gp[1, 1] = 0
        gp[0, 2] = 0
        gp[1, 2] = 1

        gyy[0] = lamb[0]
        gyy[1] = lamb[1]

    elif sw[1]: #state 2
        gp[0, 0] = 0
        gp[1, 0] = 1
        gp[0, 1] = h_s
        gp[1, 1] = r_s
        gp[0, 2] = 0
        gp[1, 2] = 0

        gyy[0] = yd[4]
        gyy[1] = yd[3] + r_s * yd[4]

    else: #state 3
        gp[0, 0] = 0
        gp[1, 0] = 1
        gp[0, 1] = -h_s
        gp[1, 1] = r_s
        gp[0, 2] = 0
        gp[1, 2] = 0

        gyy[0] = yd[4]
```

```
68          gyy[1] = yd[3] + r_s * yd[4]
69
70      res_1 = yd[0:3] - y[3:6]
71      res_2 = dot(m, yd[3:6]) - ff + dot(gp.T, lamb)
72      res_3 = gyy
73      return hstack((res_1, res_2, res_3))
```

3.3.2 State event function

As mentioned earlier, the state event functions are functions such that they change sign when the an event occurs. This, according to the transitions stated in section 3.2, simply leaves us with 2 state event functions for when we are in state 1:

$$e_0 = (r_s - r0) + h_s \cdot \varphi_S \tag{33}$$

$$e_1 = -(r_s - r0) + h_s \cdot \varphi_S \,, \tag{34}$$

and 1 function in state 2:

$$e_0 = \lambda_1 \,, \tag{35}$$

and 2 functions in state 3:

$$e_0 = \lambda_1 \tag{36}$$

$$e_1 = (l_s + l_g - l_b - r_0) - h_b \cdot \varphi_B \,. \tag{37}$$

Our implementation of the state events function can be seen in listing 6.

Listing 6: State events function

```
1  def state_events(t, y, yd, sw):
2      '''
3      This is the function that keeps track of events. When the sign of any ↩
           of the functions
4      changed, we have an event.
5      '''
6      if sw[0]: #state 1
7          #transition 1: State 1 and phi_b' < 0 switch to state 2 when h_s*↩
               phi_s = -(r_s - r0)
8          e_0 = (r_s - r0) +  h_s * y[1]
9          #transition 2: State 1 and phi_b' > 0 switch to state 3 when h_s*↩
               phi_s = (r_s - r0)
10         e_1 = -(r_s - r0) + h_s * y[1]
11     elif sw[1]: #state 2
12         #transition 3: State 2 switch to state 1 if lambda_1 changes sign
13         e_0 = y[6]
```

```
14          #dummy
15          e_1 = 0
16      elif sw[2]: #state 3
17          #transition 4: State 3 and phi_b' < 0 switch to state 1 if ↩
                lambda_1 changes sign
18          e_0 = y[6]
19          #transition 5: State 3 and phi_b' < 0 switch to state 4 (beak hit,↩
                switch to state 3 and change sign of phi_s') if h_b * phi_b =↩
                l_s + l_g - l_b - r0'
20          e_1 = (l_s + l_g - l_b - r0) - h_b * y[2]
21      return np.array([e_0, e_1])
```

3.3.3 Handle event function

Assimulo automatically checks the state event functions at every integration step to see if we have encountered an event, and if that is the case Assimulo calls the handle_event function. The parameters for this function are *(solver, event_ info)* where *solver* contains attributes such that you can obtain all parameters used in the residual and the state_events function. The *event_info* parameter contains information about which events that have been triggered. In this function we handle all state switches which essentially are setting two values in the boolean vector, sw, to its negations. The one interesting thing is when we have transitions from state 1 to state 2 or 3, i.e. when the sleeve blocks. In these cases we preserve the momentum by first setting $\dot{\varphi_S} = 0$ and $\dot{z} = 0$ and then calculating $\dot{\varphi_B} = \frac{I^-}{J_B + m_B \cdot l_g^2}$, where I^- is the momentum before the sleeve blocked. The handle_event function can be viewed in listing 7.

Listing 7: Handle event function

```
1   def handle_event(solver, event_info):
2       '''
3       Event handling. This functions is called when Assimulo finds an event ↩
            as
4       specified by the event functions
5       '''
6       state_info = event_info[0]
7       if state_info[0] != 0: #Check if the first event function has been ↩
            triggered
8           if solver.sw[0]: #state 1
9               if solver.y[5] < 0: #phi_b' < 0
10                  #momentum conservation
11                  mom_left = m_b * l_g * solver.y[3] + (m_b * l_s * l_g) * ↩
                        solver.y[4] + (J_b + m_b * l_g**2) * solver.y[5]
12
13                  #force z_p and phi_sp to 0
14                  zp_new = 0
15                  phi_sp_new = 0
```

```
16
17                  #calculate new momentum, I_before = I_after
18                  phi_bp_new = mom_left/(J_b + m_b * l_g**2)
19
20                  #give new values to solver
21                  solver.y[3] = solver.yd[0] = zp_new
22                  solver.y[4] = solver.yd[1] = phi_sp_new
23                  solver.y[5] = solver.yd[2] = phi_bp_new
24
25                  #switch to state 2
26                  solver.sw[0] = not solver.sw[0]
27                  solver.sw[1] = not solver.sw[1]
28          elif solver.sw[1]: #state 2
29              #switch to state 1
30              solver.sw[1] = not solver.sw[1]
31              solver.sw[0] = not solver.sw[0]
32          elif solver.sw[2]: #state 3
33              if solver.y[5] < 0: #phi_b' < 0
34                  #switch to state 1
35                  solver.sw[2] = not solver.sw[2]
36                  solver.sw[0] = not solver.sw[0]
37      elif state_info[1] != 0: #second event function
38          if solver.sw[0]: #state 1
39              if solver.y[5] > 0: #phi_b' > 0
40                  #momentum conservation
41                  mom_left = m_b * l_g * solver.y[3] + (m_b * l_s * l_g) * ↩
                        solver.y[4] + (J_b + m_b * l_g**2) * solver.y[5]
42
43                  #force z_p and phi_sp to 0
44                  zp_new = 0
45                  phi_sp_new = 0
46
47                  #calculate new momentum, I_before = I_after
48                  phi_bp_new = mom_left/(J_b + m_b * l_g**2)
49
50                  #give new values to solver
51                  solver.y[3] = zp_new
52                  solver.yd[0] = zp_new
53                  solver.y[4] = phi_sp_new
54                  solver.yd[1] = phi_sp_new
55                  solver.y[5] = phi_bp_new
56                  solver.yd[2] = phi_bp_new
57
58                  #switch to state 3
59                  solver.sw[0] = not solver.sw[0]
60                  solver.sw[2] = not solver.sw[2]
61          elif solver.sw[1]: #state 2
```

```
62              #dummy do nothing
63              pass
64          elif solver.sw[2]: #state 3
65              if solver.y[5] > 0: #phi_b' > 0
66                  #beak hit, go to state 3, change sign of phi_b'
67                  solver.y[5] = -solver.y[5]
68                  solver.yd[2] = -solver.yd[2]
69                  print("Beak hit")
```

3.3.4 Simulation

Before simulating we need some initial values. We start in state 2, i.e. when $\varphi_S = -0.10344$, and select $\varphi_B = -0.65$. Setting all else to 0 (positions, velocities and accelerations) will then uniquely determine $\lambda_1 = -0.628993$ and $\lambda_2 = 0.047088$.

The code for the simulation is quite uninteresting. Things to note are that the solver used was IDA and that we need to suppress both lambdas in the error estimator as a result of using an index 1 formulation (as opposed to index 0).

Running the simulation between 0 and 2 seconds results in the plots seen in figures 37 to 39.

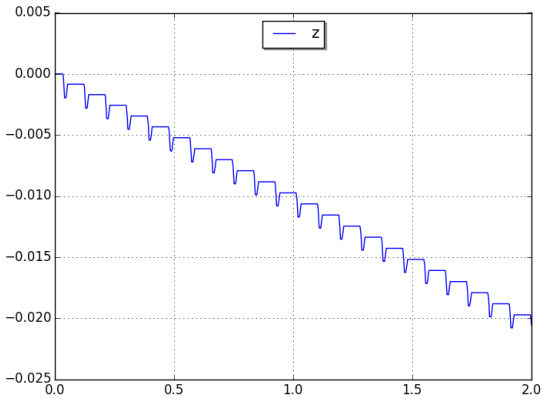

Figure 37: Vertical position of the woodpecker toy as a function of time

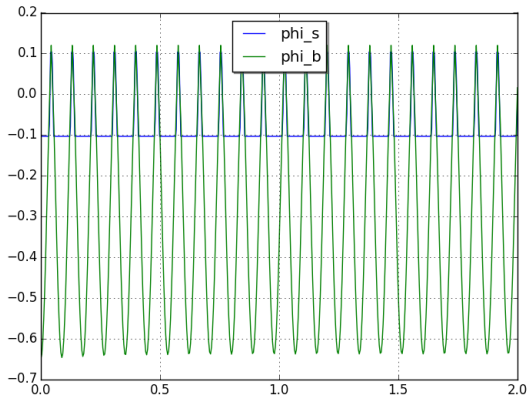

Figure 38: Angles of sleeve and bird of the woodpecker toy as a function of time

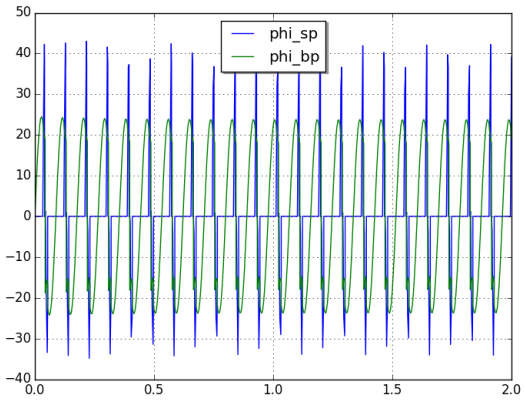

Figure 39: Time derivatives of angles of sleeve and bird of the woodpecker toy as a function of time

As seen in figure 37 the solution contains several discontinuities. Every time the beak hits the bar the sleeve's position encounters a discontinuity and behaves similarly to the function $y(x) = |x|$. Also every time the sleeve is locked in either state 2 or 3 we see the completely horizontal parts since the z-position of the ($\dot{z} = 0$) remains constant in these states. Here we can for example note that the woodpecker hits the bar 22 times with our given initial conditions. Using other initial conditions results in different number of "pecks". Furthermore, we can observe some discontinuities in the angles. Each time the beak hits the bar we abruptly change the value of $\dot{\varphi_B}$, resulting in the sharp peaks of figure 38. If we zoom the image in further and look at one period of the motion as in figure 40, we see that the angle of the sleeve is constant in between its motion which is exactly when the sleeve is blocked. The velocities of the angles in figure 39 are a bit harder to interpret, but if we zoom in as in figure 41, the angle velocity of the sleeve seems to be 0 when blocked and goes from positive to negative with a short time of velocity 0 in between, representing the motions we would expect from the woodpecker toy. For the relative angle of the bird we observe an immediate jump from positive to negative velocity which represents the beak hitting the bar and the system switching direction while retaining its momentum.

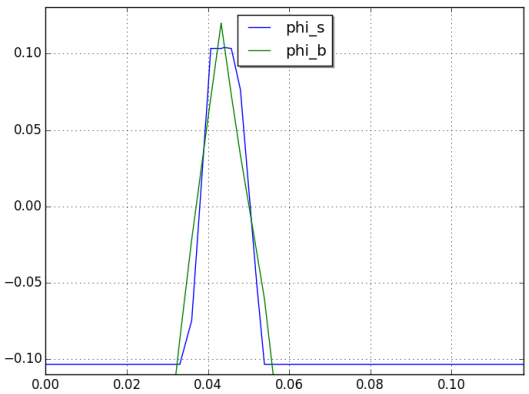

Figure 40: Angles of sleeve and bird of the woodpecker toy as a function of time (closeup)

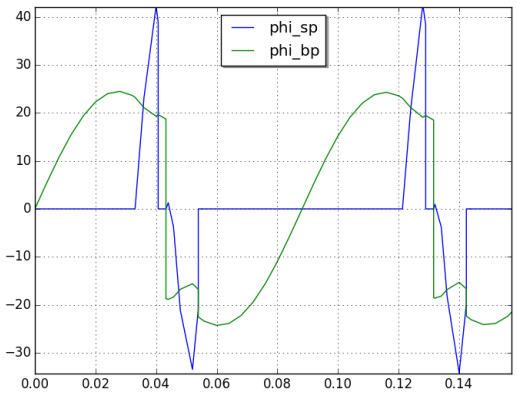

Figure 41: Time derivatives of angles of sleeve and bird of the woodpecker toy as a function of time (closeup)

Additionally, we can look at the constraint forces λ as functions of time. See figure 42 for the entire plot and figure 43 for a zoom over two periods. We can clearly see the tendencies for λ_1 and λ_2 going to 0, as a result of the woodpecker toy being in state 1 where there are no constraints. We can also see some noise both in λ_1 and λ_2 which we cannot really explain with certainty. It could be either due to numerical errors, errors in the model (unlikely), problems with the specific integrator (IDA) or a variety of other reasons. Otherwise, the forces look periodic which is to be expected as well as (exlucing noise) constantly negative which also is to be expected since these forces counteract the motion of the woodpecker.

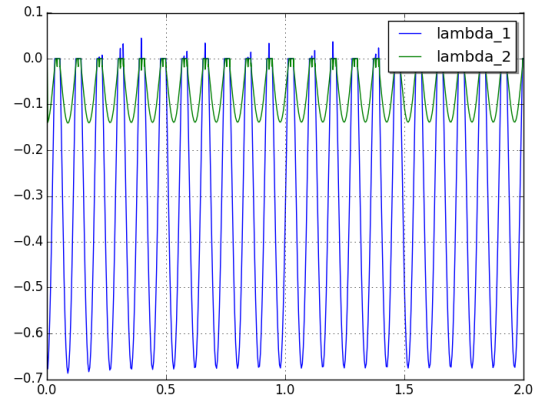

Figure 42: First constraint force of the woodpecker toy, λ_1, as a function of time

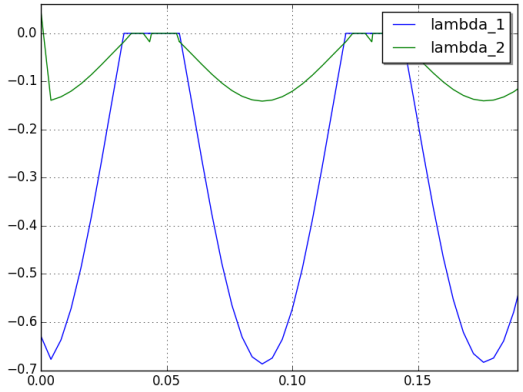

Figure 43: Second constraint force of the woodpecker toy, λ_2, as a function of time

3.3.5 Runtime statistics

Running the model in Assimulo gives us runtime statistics seen in table 15.

Steps	7228
Function evaluations	9254
Jacobian evaluations	3132
Function eval. due to Jacobian eval.	25056
Error test failures	264
Nonlinear iterations	9254
Nonlinear convergence failures	0
State function evaluations	9769
State events	224
CPU-time	1.75328

Table 15: Runtime statistics for the woodpecker model in Assimulo

3.4 Dymola

3.4.1 Code structure

This section describes the structure of our Dymola code. All the geometrical and mechanical constants are defined as `Real` parameters. The state variables and the constraint variabes are defined as real state variables and are given consistent start values. There is also an integer variable `state` to keep track of the current state of the system.

In the equation section of the Dymola code all the equations for the system are given. Which equations that are currently active are decided by if-statements that depend on the variable `state`.

The switching between the states are handled in the algorithm section. When-statements are used to switch between states. The conditions that are given as equalities in the instructions have been rewritten as inequalities to make sure the code handles numerical errors. Because of numerical errors we have also added a tolerance when detecting if λ_1 changes sign. Only the case when λ_1 changes sign from negative to positive is implemented in Dymola, since the method failed when also implementing the transition from positive to negative.

The values of some of state variables needs to be changed when the beak of the bird hits the bar and when the sleeve gets blocked. This is also handled in the algorithm section. To change these values we use the Dymola methods `reinit`, that reinitializes the value of a state variable, and `pre` that calculates the value of a state variable at the moment of a state transistion.

Listing 8: Woodpecker model

```
1  model woodpecker "woodpecker model for project 3 in simulation tools"
2
3   import Modelica.Utilities.Streams;
4    parameter Real mS=3.0*10^(-4) "sleeve mass";
5    parameter Real Js=5*10^(-9) "sleeve momentum of inertia";
6    parameter Real mB=4.5*10^(-3) "bird mass";
7    parameter Real Jb=7*10^(-7) "bird momentum of inertia";
```

```
 8   parameter Real r0=2.5*10^(-3) "bar radius";
 9   parameter Real rS=3.1*10^(-3) "inner raduis of sleeve";
10   parameter Real hS=5.8*10^(-3) "half height of the sleeve";
11   parameter Real lS=1*10^(-2) "distance sleeve and spring rotation axis";
12   parameter Real lG=1.5*10^(-2) "distance bird and spring rotation axis";
13   parameter Real lB=2.01*10^(-2) "beak x-coordinate in bird's system";
14   parameter Real hB=2.0*10^(-2) "beak y-coordinate in bird's system";
15   parameter Real cP=5.6*10^(-3) "spring constant";
16   parameter Real g=9.81 "gravity";
17
18   Real z(start=0.0) "z-coorcdinate for sleeve";
19   Real phiS(start=-0.10344) "sleeve angle";
20   Real phiB(start=-0.65) "bird angle";
21
22   Real zdot(start=0.0) "first deriviative of z";
23   Real phiSdot(start=0.0) "first deriviative of phiS";
24   Real phiBdot(start=0) "first deriviative of phiB";
25
26   Real lambda1(start=-0.6911);
27   Real lambda2(start=-0.1416);
28
29   Integer state(start=2, fixed=true) "state of system";
30
31 equation
32     der(z)=zdot;
33     der(phiS)=phiSdot;
34     der(phiB)=phiBdot;
35
36     if state==1 then
37       (mS+mB)*der(zdot)+mB*lS*der(phiSdot)+mB*lG*der(phiBdot) =-(mS+mB)*g;
38       (mB*lS)*der(zdot)+(Js+mB*lS^2)*der(phiSdot)+(mB*lS*lG)*der(phiBdot) ↩
            = cP*(phiB-phiS)-mB*lS*g-lambda1;
39       mB*lG*der(zdot)+mB*lS*lG*der(phiSdot)+(Jb+mB*lG^2)*der(phiBdot) = cP↩
            *(phiS-phiB)-mB*lG*g-lambda2;
40       lambda1=0;
41       lambda2=0;
42     elseif state==2 then
43       (mS+mB)*der(zdot)+mB*lS*der(phiSdot)+mB*lG*der(phiBdot) = -(mS+mB)*g↩
            -lambda2;
44       (mB*lS)*der(zdot)+(Js+mB*lS^2)*der(phiSdot)+(mB*lS*lG)*der(phiBdot) ↩
            = cP*(phiB-phiS)-mB*lS*g-hS*lambda1-rS*lambda2;
45       mB*lG*der(zdot)+mB*lS*lG*der(phiSdot)+(Jb+mB*lG^2)*der(phiBdot) = cP↩
            *(phiS-phiB)-mB*lG*g;
46       0=hS*der(phiSdot);
47       der(zdot)+rS*der(phiSdot)=0;
48     else
49       (mS+mB)*der(zdot)+mB*lS*der(phiSdot)+mB*lG*der(phiBdot) = -(mS+mB)*g↩
```

```
            -lambda2;
50        (mB*lS)*der(zdot)+(Js+mB*lS^2)*der(phiSdot)+(mB*lS*lG)*der(phiBdot) ↩
            = cP*(phiB-phiS)-mB*lS*g+hS*lambda1-rS*lambda2;
51        mB*lG*der(zdot)+mB*lS*lG*der(phiSdot)+(Jb+mB*lG^2)*der(phiBdot) = cP↩
            *(phiS-phiB)-mB*lG*g;
52        0=-hS*der(phiSdot);
53        der(zdot)+rS*der(phiSdot)=0;
54      end if;
55
56  algorithm
57
58      when state==1 and phiBdot<0 and (hS*phiS+rS-r0)<0 then
59        state:=2;
60        reinit(phiBdot, (mB*lG*pre(zdot) + (mB*lS*lG)*pre(phiSdot) + (Jb + ↩
            mB*lG^2)*pre(phiBdot))/(Jb + mB*lG^2));
61        reinit(phiSdot, 0);
62      elsewhen state==1 and phiBdot>0 and (hS*phiS-rS+r0)>0 then
63        reinit(phiBdot, (mB*lG*pre(zdot) + (mB*lS*lG)*pre(phiSdot) + (Jb + ↩
            mB*lG^2)*pre(phiBdot))/(Jb + mB*lG^2));
64        reinit(zdot, 0);
65        reinit(phiSdot, 0);
66        state:=3;
67      elsewhen lambda1>10^(-4) then
68        if state==2 then
69          state:=1;
70        elseif state==3 and phiBdot<0 then
71          state:=1;
72        end if;
73      elsewhen state==3 and phiBdot>0 and hB*phiB-lS-lG+lB+r0>0 then
74        reinit(phiBdot,-pre(phiBdot));
75      end when;
76
77  end woodpecker;
```

3.4.2 Simulation

The simulations where performed using the following initial values; we start in state 2, $z = 0$, $\varphi_S = -0.10344$, $\varphi_B = -0.65$, $\dot{\varphi_S} = 0$, $\dot{\varphi_B} = 0$, $\dot{z} = 0$, $\lambda_1 = -0.628993$, and $\lambda_2 = 0.047088$.

Running the simulation between 0 and 2 seconds results in the plots seen in figures 44 to 46.

As you can see, the simulations in Dymola correspond well to those performed in Assimulo and the woodpecker hits its beak on the bar 22 times during the first 2 seconds. The simulation seems to be successful even though the transition of λ_1 from positive to negative wasn't implemented. Hence, this transition doesn't seem to be of importance for the model.

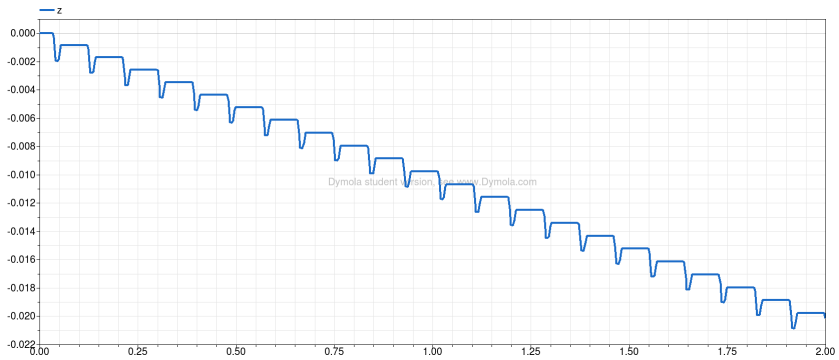

Figure 44: Vertical position of the woodpecker toy as a function of time

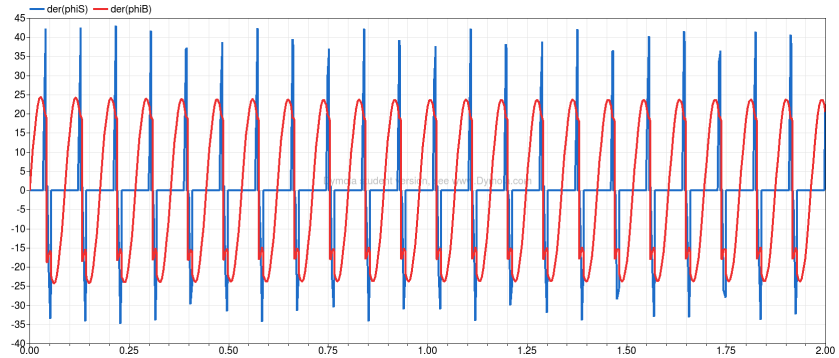

Figure 45: Angles of sleeve and bird of the woodpecker toy as a function of time

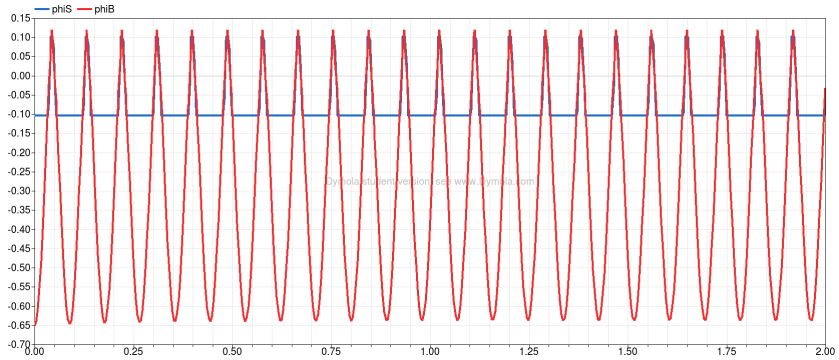

Figure 46: Angle velocity of sleeve and bird of the woodpeker toy as a function of time

The runtime statistics for the Dymola simulations are seen in table 16.

Time of simulation	0.0343 s
Successful steps	3239
Function evaluations	8332
Jacobian evaluations	1460
State events	177

Table 16: Runtime statistics for simulating the woodpecker toy in Dymola

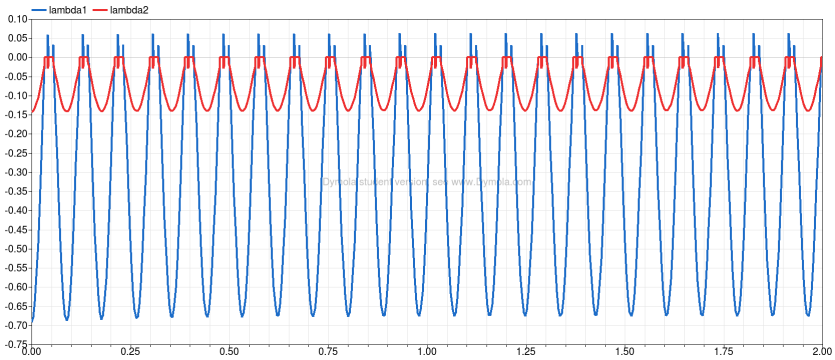

Figure 47: Constraint forces on the sleeve as a function of time

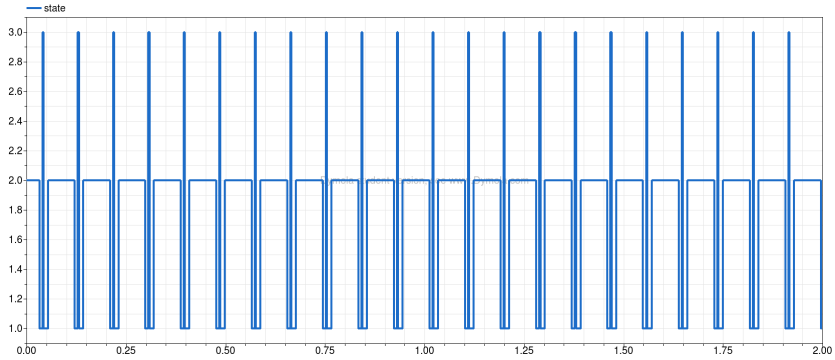

Figure 48: The states of the toy as a function of time

3.4.3 Discussion

As in project 2 the Dymola simulation was more efficient in every aspect. This shows especially in the CPU-time which is about 60 times as fast in Dymola for simulating the same time period in Assimulo. Dymola takes about half as many steps as Assimulo and does about half as many Jacobian evaluations. One interesting observation is that we have a different amount of state events in the Dymola and Assimulo simulations. This is something you would presume independent of the simulation method - especially when we have the same number of beak hits - and is likely a result of a slight difference in our problem definitions. More precisely, comparing the state definitions in the Dymola code in listing 8, algorithms section (line 58-75), with the python code in state_event in listing 6, we see that the state change when λ_1 becomes negative is omitted in the Dymola version (this was also mentioned in sections 3.4.1 and 3.4.2) which is a likely reason for this disparity.

4 Final comments

In this course we have compared the two simulation tools Assimulo and Dymola. Assimulo simulations are implemented using Python while Dymola simulations are implemented using Modelica. Assimulo being an open source tool, it is easy to interpret and modify code in order to get a good overview of how the solvers are constructed and they can be modified to fit the users specific needs. Dymola on the other hand is a commercial software where little insight about the

solver details are provided to the user. On the other hand, Dymola is very user friendly and with the support of the Modelica standard library the user can work in a drag-and-drop environment without the need to ever derive the differential equations.

Our perception is that Dymola is easier to work with since the software takes care of many details so that the user only has to focus on the model. It does however not the support the user to fit the solver anyway he or she might see fit, since every possible solver might not be available, the run time stats returned are not customized by the user, and so on. If the user sees a need to control the whole simulation process rather than being interested in a solution, Assimulo would therefore be a better fit.

References

[1] Söderlind, G., *Numerical Methods for Differential Equations Chapter 2, FMNN10/NUMN12*, V4.15, 2015

[2] E. Hairer, G. Wanner, *Solving Ordinary Differential Equations II*, Springer, 2nd revised edition, 1996.

[3] https://build.openmodelica.org/Documentation/Modelica.Mechanics.MultiBody.html, ®Modelica Association.

[4] C. Führer, C. Andersson, *FMNN05: Simulation Tools Project 3*, 2015-12-08