Simulation Tools
Project 2

Daniel Holm

February 2024

1 Reconstructing the initial conditions

By setting $\Theta = 0$ we can modify equation 7.5 from [2] to create the constraint $g(\mathbf{q})$.

$$\begin{aligned}
g_1 &= rr \cdot \cos\beta - \ d \cdot \cos\beta - ss \cdot \sin\gamma - xb \\
g_2 &= rr \cdot \sin\eta - \ d \cdot \sin\beta + ss \cdot \cos\gamma - yb \\
g_3 &= rr \cdot \cos\eta - \ d \cdot \cos\beta - e \cdot \sin(\Phi + \delta) - zt \cdot \cos\delta - xa \\
g_4 &= rr \cdot \sin\beta - \ d \cdot \sin\beta + e \cdot \cos(\Phi + \delta) - zt \cdot \sin\delta - ya \\
g_5 &= rr \cdot \cos\beta - \ d \cdot \cos\beta - zt \cdot \cos(\Omega + \varepsilon) - u \cdot \sin\varepsilon - xa \\
g_6 &= rr \cdot \sin\beta - \ d \cdot \sin\beta - zt \cdot \sin(\Omega + \varepsilon) + u \cdot \cos\varepsilon - ya.
\end{aligned} \tag{1}$$

To get consistent initial conditions we solve for $g(\mathbf{q}) = 0$ numerically with Newton's method. This was done with the command `fsolve` from the SciPy package. Comparison with the values from [2] shows that they differ by less than 10^{-10}.

2 Simulating both problem formulations

All simulations were done with 1000 communication points using the built-in solver `IDA`. Using index-3 constraints gave convergence failures despite setting atol to 10^5 for the algebraic variables. Even after fully excluding the algebraic failures the solver returned convergence failures. After increasing atol to 10^5 for the velocity components the solver was able to finish. The execution time was around 0.68 seconds.

For the index-2 formulation of the problem the solver finished without having to exclude the algebraic variables. The absolute tolerances were still set to 10^5 for them, however. The execution time was around 0.13 seconds.

2.1 Comparing the results

First of all, the squeezer angles correspond very well to the figure from [2]. Except for the angles β and Θ which both seem to diverge. However, because of the periodic nature of angles, this might be a result of some definitional difference in Assimulo's solver and the one from [2]. The second point of interest is the Lagrange multipliers. In Figure 1 we can see some rapid oscillations in all λ at the same points in time. None of these oscillations are present in Figure 2. It is interesting that the different formulations give differing results for the algebraic variables but not the differential variables, i.e., the angles and their velocities.

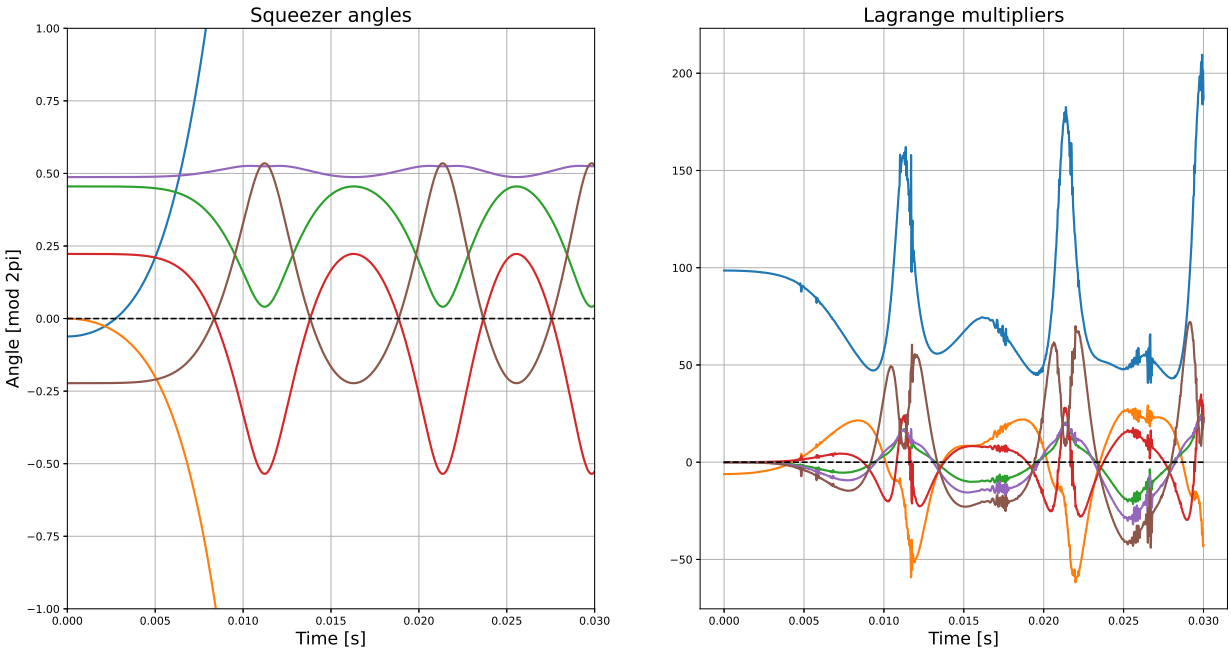


Figure 1: Simulation for index-3 formulation.

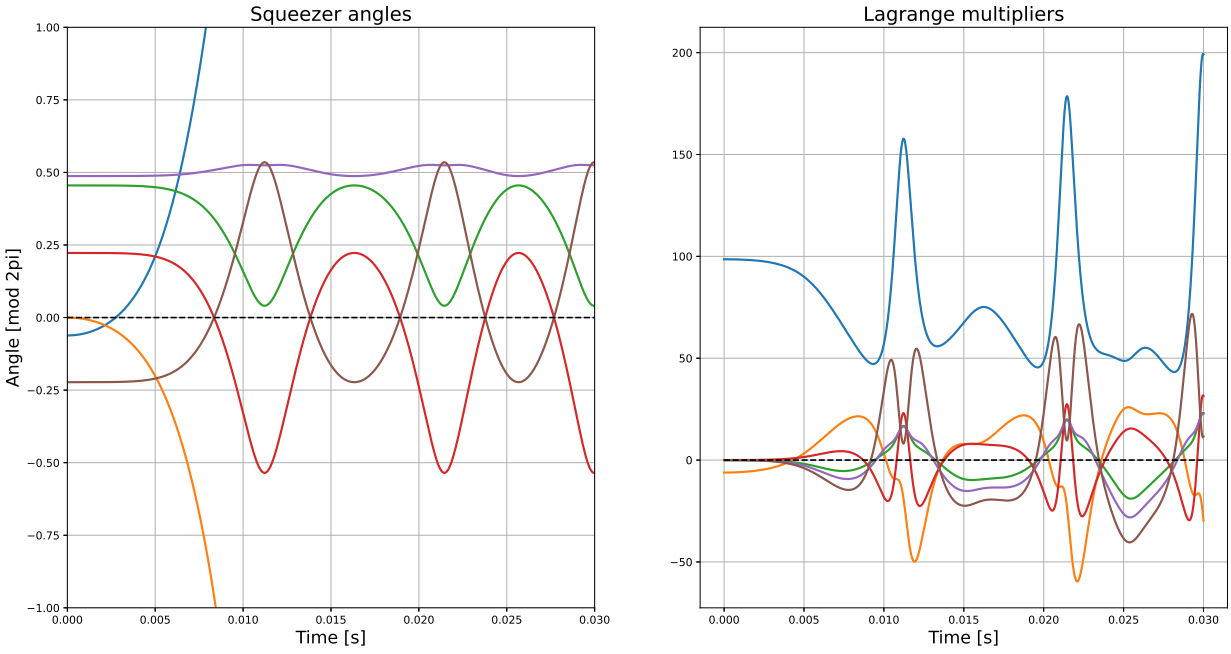
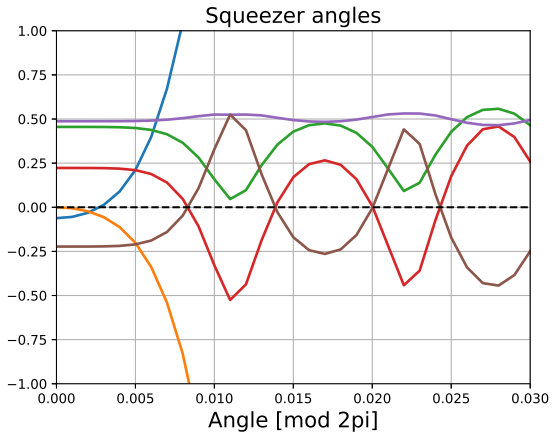


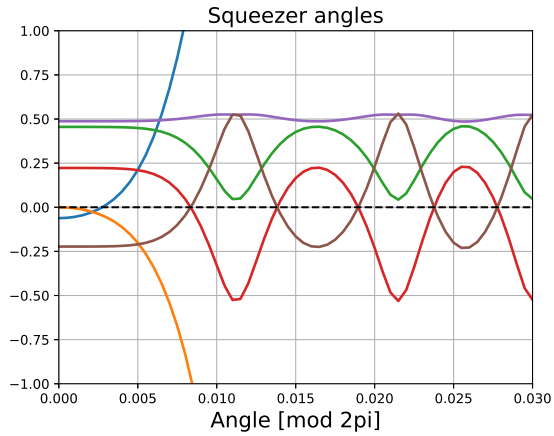
Figure 2: Simulation for index-2 formulation

3 Explicit Runge-Kutta methods

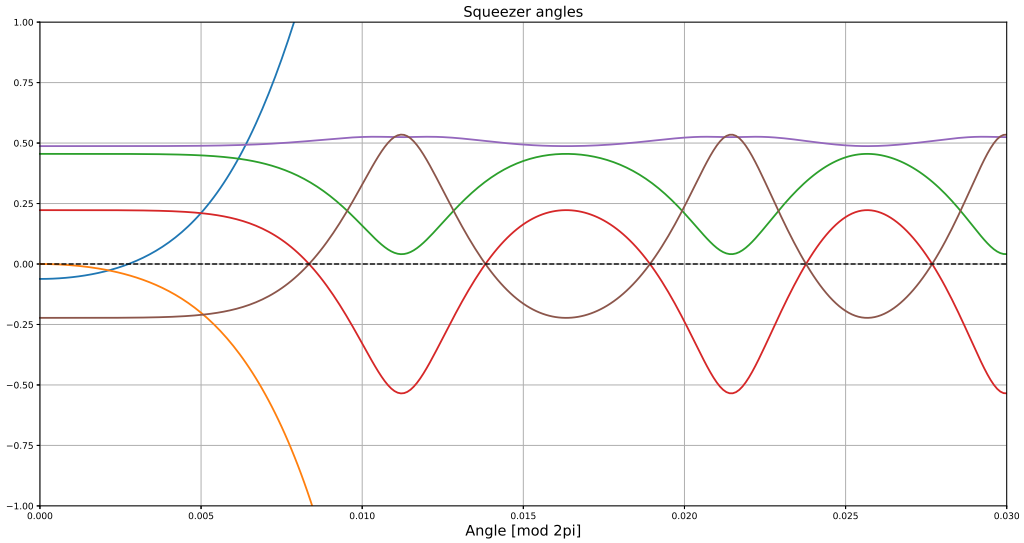
Using the index-1 formulation of the problem we can rewrite at as a normal explicit problem of type $\dot{y} = f(t, y),\ y(0) = y_0$ for some right-hand-side function f. The built-in `RungeKutta4` was used to simulate at some different step-sizes.



(a) $h = 10^{-3}$



(b) $h = 5 \cdot 10^{-4}$



(c) $h = 10^{-4}$

Figure 3: Simulation for index-1 formulation with three different step-sizes h.

In these figures we can clearly see the impact of the step-size. In Figure 3c we can clearly see the growing oscillations indicating instability. Halving the step-size appears to reach or at least approach stability, as seen in Figure 3b. Looking at the red and brown lines in particular, it appears as if the amplitude is increasing, but this could also be because the numerical time values skip over the exact time of the turning points. Halving the step-size once again produces the plot seen in Figure 3c. This looks completely stable. Due to the small step-size the function also looks much smoother (naturally).

Code

The code consists of three files, the provided `squeezer.py`, with the required additions, and files for the specific tasks. Much of the code is omitted since it was written for us. The residual function in particular is heavily trimmed here.

squeezer.py

```
# -*- coding: utf-8 -*-
from __future__ import division
import assimulo.implicit_ode as ai
import assimulo.problem as ap
from assimulo.solvers import IDA
import matplotlib.pyplot as mpl
from scipy import *
from numpy import array,zeros,ones,hstack,sin,cos,sqrt,dot,pi

class Seven_bar_mechanism(ap.Implicit_Problem):
        """
        A class which describes the squezzer according to
        Hairer, Vol. II, p. 533 ff, see also formula (7.11)
        """
        problem_name='Squeezer'

        def __init__(self):
                self.y0, self.yd0=self.init_squeezer()
                ap.Implicit_Problem.__init__(self, self.res, self.y0, self.yd0, 0.)
                self.name = self.problem_name

        def init_squeezer(self):
                ...

        def res(self, t, y, yp):
                """
                Residual function of the 7-bar mechanism in
                Hairer, Vol. II, p. 533 ff, see also formula (7.11)
                written in residual form
                y,yp vector of dim 20, t scalar
                """

                ...

                # Construction of the residual, (un)comment depending on which index formulation to use
                res_1 = yp[0:7] - y[7:14]
                res_2 = dot(m, yp[7:14]) - ff[0:7] + dot(gp.T, lamb)
                res_3 = g # index-3
                # res_3 = dot(gp, y[7:14]) # index-2
                # res_3 = g_qq + dot(gp, yp[7:14]) # index-1

                return hstack((res_1, res_2, res_3))

squeezer = Seven_bar_mechanism()
sim = IDA(squeezer)

# Change these depending on index-3 or index-2
```

```
sim.algvar = hstack((ones((14,)), zeros((6,))))
sim.atol = hstack((1.e-6*ones((7,)), 1.e5*ones((13,))))
sim.suppress_alg = True

tf = 0.03
ncp = 1000
t, y, yd = sim.simulate(tf, ncp)

angles = [states[0:7] for states in y]
lambdas = [states[14:20] for states in y]

# plot commands
...
```

Task3.py

```
from __future__ import division
import matplotlib.pyplot as mpl
from scipy.linalg import norm
from scipy.optimize import fsolve
from numpy import zeros, sin, cos, array

# Geometry
...

def g(q):
        Beta, gamma, Phi, delta, Omega, epsilon = q[0:6]

        gvec = zeros((6,))

        gvec[0] = rr*cos(Beta) - d*cos(Beta) - ss*sin(gamma) - xb
        gvec[1] = rr*sin(Beta) - d*sin(Beta) + ss*cos(gamma) - yb
        gvec[2] = rr*cos(Beta) - d*cos(Beta) - e*sin(Phi+delta) - zt*cos(delta) - xa
        gvec[3] = rr*sin(Beta) - d*sin(Beta) + e*cos(Phi+delta) - zt*sin(delta) - ya
        gvec[4] = rr*cos(Beta) - d*cos(Beta) - zf*cos(Omega+epsilon) - u*sin(epsilon) - xa
        gvec[5] = rr*sin(Beta) - d*sin(Beta) - zf*sin(Omega+epsilon) + u*cos(epsilon) - ya

        return gvec

y_1 = array([-0.061713890014276449635894845800 1,  #  Beta
                                0.455279819163070380255912382449,   # gamma
                                0.222668390165885884674473185609,   # Phi
                                0.487364979543842550225598953530,   # delta
                                -0.222668390165885884674473185609,  # Omega
                                1.230547444549821192497350155 68])   # epsilon

# calculate initial conditions using fsolve
g0 = fsolve(g, x0=zeros((6,)))

print(g0 - y_1)
print(norm(g0 - y_1))
```

Task8.py

```
# -*- coding: utf-8 -*-
from __future__ import division
from assimulo.problem import Explicit_Problem
from assimulo.solvers import RungeKutta4, ExplicitEuler, ImplicitEuler
import matplotlib.pyplot as mpl
from scipy.linalg import solve
from numpy import array,zeros,block,hstack,sin,cos,sqrt

# Inertia data
...

# Geometry
...

# Driving torque
...

# Spring data
...

def init_squeezer():
        ...

def rhs(t, y):
        # Initial computations and assignments
        ...

        # Mass matrix
        ...

        #   Applied forces
        ...

        #  Jacobian matrix G(q)
        ...

        # g_qq(q)(qdot, qdot) for index-1
        ...

        # assembling the matrices
        A = block([[m, gp.T], [gp, zeros((6,6))]])
        b = hstack((ff, -g_qq))

        wlam = solve(A, b)
        w = wlam[:7]

        qdot = y[7:]
        vdot = w

        return hstack((qdot, vdot))

y0 = init_squeezer()
tf = 0.03

```

```
model = Explicit_Problem(rhs, y0, 0)
rk = RungeKutta4(model)
rk.h = 1e-5 # smaller than 1e-3!
t, y = rk.simulate(tf)

angles = [states[0:7] for states in y]

# plot commands
...
```