Design of Machines and Structures, Vol. 7, No. 1 (2017), pp. 15–22.

SOLVING MULTIBODY DYNAMICS PROBLEMS USING PYTHON

PAVEL FLORIAN–ROMAN ČERMÁK
University of West Bohemia, Department of Machine Design
Univerzitni 8, 306 14 Pilsen
pflorian@kks.zcu.cz
rcermak@kks.zcu.cz

Abstract: The aim of this paper is inform the reader with possibilities of deriving equations of motion for arbitrarily complex multibody systems using Python programming language. Sympy is a library of functions for Python and a full computer algebra system (CAS) which has among others a built-in feature that allows assembling a multibody system and derive corresponding equations of motion.

Keywords: *multibody dynamics, Python, Sympy, EOM*

1. INTRODUCTION

Solving multibody dynamics problems is possible by hand. There are methods such as Lagrange equations that allow that. However, this approach is only viable for simple systems with limited number of degrees of freedom, as hand-derivation of EOMs is a tedious and error-prone process. Sympy provides an alternative that is under BSD license which means it is free for both academic and commercial use. The initial phase consisting of setting up the environment and gathering required libraries is made simple thanks to Python scientific distributions such as Anaconda that involve all the necessary features. In this paper it is shown how one can obtain equations of motion describing dynamics of a simple four bar mechanism as shown in *Figure 1.*

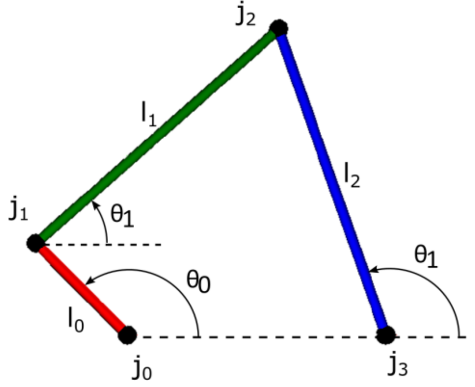

Figure 1. Four bar mechanism

2. GENERAL PROCEDURE WHEN USING SYMPY

When using Sympy it is good practice to follow a certain path. In this case we will start with defining kinematics of the mechanism, followed by inertia, forces, EOM generation and simulation.

2.1. Kinematics

The first step is to decide how many generalized coordinates will be used to describe the mechanism. The four bar mechanism as described here has only one degree of freedom and can be unambiguously described by one angular coordinate. However in this example three generalized coordinates corresponding to each link will be used. Python indexing starts with 0 therefore the same approach will be used when naming variables.

The first three lines in the block of code bellow import all the necessary functions. Time-dependant variables are created using *dynamicsymbols()* command, whereas for time-independent constants *symbols()* is used.

```
from __future__ import print_function, division
from sympy.physics.mechanics import *
from sympy.physics.vector import time_derivative
from sympy import symbols
from numpy import array, linspace, rad2deg, deg2rad
from scipy.integrate import odeint
from pydy.codegen.ode_function_generators import
generate_ode_function

# number of links in the mechanism
n = 3
# generalized speeds and coordinates
theta = dynamicsymbols('theta:{}'.format(n))
theta_d = dynamicsymbols('theta:{}'.format(n), 1)
omega = dynamicsymbols('omega:{}'.format(n))
omega_d = dynamicsymbols('omega:{}'.format(n), 1)
# the extra symbol thanks to n+1 stands for the distance
# between the grounding joints
length_bars = symbols('L_B:{}'.format(n+1))
g = symbols('g') # gravity
```

Figure 2. Definition of variables

In the second step reference frames are defined. An inertial frame is introduced and then three other frames, one for each link, are oriented with respect to the inertial frame. The rotations are given by *theta* angles and their direction is the z-axis of the inertial frame. In a similar fashion angular velocities of the reference frames are introduced. There is more than one way to define rotation between reference frames. For non-planar systems it is easier to use type *'Body'* since less lines of code are needed for the definition as shown in the block of code for *bar2_frame*.

```
# reference frames
inertial_frame = ReferenceFrame('I')
bar0_frame = inertial_frame.orientnew('B0', 'Axis', (theta[0],
                                                    inertial_frame.z))
bar1_frame = inertial_frame.orientnew('B1', 'Axis', (theta[1],
                                                    inertial_frame.z))
bar2_frame = inertial_frame.orientnew('B2', 'Body', [0, 0, theta[2]],
                                                    'XYZ')
# angular velocities
bar0_frame.set_ang_vel(inertial_frame, omega[0]*inertial_frame.z)
bar1_frame.set_ang_vel(inertial_frame, omega[1]*inertial_frame.z)
bar2_frame.set_ang_vel(inertial_frame, omega[2]*inertial_frame.z)
```

Figure 3. Orientation and velocity

Once orientation and velocity of frames is set important points of the system such as joints and centres of mass can be introduced. X-axis of each frame defines link orientation and centres of mass are assumed to be in the middle of each link for simplicity.

```
# joints
joint0 = Point('j0')
joint1 = joint0.locatenew('j1', length_bars[0] * bar0_frame.x)
joint2 = joint1.locatenew('j2', length_bars[1] * bar1_frame.x)
joint3 = joint2.locatenew('j3', -length_bars[2] * bar2_frame.x)
# cm positions
bar0_cm = joint0.locatenew('b0cm', length_bars[0]/2 * bar0_frame.x)
bar1_cm = joint1.locatenew('b1cm', length_bars[1]/2 * bar1_frame.x)
bar2_cm = joint3.locatenew('b2cm', length_bars[2]/2 * bar2_frame.x)
```

Figure 4. Joint and centre of mass locations

Just like angular velocities were defined separately linear velocities of the points need to be defined in a similar way. Velocity of the grounding joints *j0* and *j3* is set to 0 as they are stationary.

```
# velocity of the grounding joints is 0
joint0.set_vel(inertial_frame, 0)
joint3.set_vel(inertial_frame, 0)
# velocity if the remaining joints
joint1.v2pt_theory(joint0, inertial_frame, bar0_frame)
joint2.v2pt_theory(joint1, inertial_frame, bar1_frame)
# velocity if centres of mass
bar0_cm.v2pt_theory(joint0, inertial_frame, bar0_frame)
bar1_cm.v2pt_theory(joint1, inertial_frame, bar1_frame)
bar2_cm.v2pt_theory(joint2, inertial_frame, bar2_frame)
```

Figure 5. Velocities of points

When defining kinematics of the mechanism the last step is to introduce configuration and velocity constraints. As mentioned above the mechanism has only one degree of freedom, but three angular coordinates were used to define it. In this case we need to implement two configuration constraints, one for each excessive coordinate, and two velocity constraints in order to obtain the correct solution. The configuration constraints say that the joint *j3* is located at distance of *length_bars[3]* from joint *j0* on the x-axis of the inertial frame. There are two constraints, one for each *x* and *y* direction. The velocity constraints simply mean that these two points are not moving with respect to each other. The following block of code concludes the definition of kinematics.

```
# configuration constraint
zero = joint3.pos_from(joint0) + length_bars[3] * inertial_frame.x
f_c = [zero & inertial_frame.x, zero & inertial_frame.y]
# velocity constraint
dzero = time_derivative(zero, inertial_frame)
f_v = [dzero & inertial_frame.x, dzero & inertial_frame.y]
```

Figure 6. Constraints

2.2. Mass and inertia

This particular mechanism is only a planar problem therefore the definition of inertia is fairly simple. In this case we only introduce rotational inertia related to the *xy* plane of the inertial frame. In Sympy inertia is implemented in terms of dyadic. Inertia is defined in a frame that is stationary with respect to a given body. The information about mass and inertia is then coupled in a *RigidBody* object.

```
# mass and inertia
mass_bars = symbols('m_B:{}'.format(n))
inertia_bars = symbols('I_B:{}'.format(n))
#     inertia(frame, ixx=0, iyy=0, izz, ixy=0, iyz=0, izx=0)
bar0_indyad = inertia(bar0_frame, 0, 0, inertia_bars[0])
bar1_indyad = inertia(bar1_frame, 0, 0, inertia_bars[1])
bar2_indyad = inertia(bar2_frame, 0, 0, inertia_bars[2])
bar0_inertia = (bar0_indyad, bar0_cm)
bar1_inertia = (bar1_indyad, bar1_cm)
bar2_inertia = (bar2_indyad, bar2_cm)
# bodies
bar0 = RigidBody('Bar 0', bar0_cm, bar0_frame, mass_bars[0],
                 bar0_inertia)
bar1 = RigidBody('Bar 1', bar1_cm, bar0_frame, mass_bars[1],
                 bar1_inertia)
bar2 = RigidBody('Bar 1', bar2_cm, bar0_frame, mass_bars[2],
                 bar2_inertia)
bodies = [bar0, bar1, bar2]
```

Figure 7. Mass and inertia

2.3. Forces

In this case the only applied forces are the ones due to gravity. In order to introduce for example driving torque an extra tuple needs to be created and added to the *loads* list.

```
# forces
bar0_force = (bar0_cm, -mass_bars[0] * g * inertial_frame.y)
bar1_force = (bar1_cm, -mass_bars[1] * g * inertial_frame.y)
bar2_force = (bar2_cm, -mass_bars[2] * g * inertial_frame.y)
loads = [bar0_force, bar1_force, bar2_force]
```

Figure 8. Loads

2.4. EOM generation

Once kinematics, inertia and forces are specified the following step is generation of equations of motion. There are two methods available is Sympy. In this paper Kane's method is used [2]; however it is also possible to utilize Lagrange's method. Firstly we need to introduce kinematic differential equations that bind variables for coordinates with speeds. After that Kane's method is initialized. This method has many arguments as shown below. Lastly *Fr* and *Fr** need to be calculated as described in [2].

```
# kinematic differential equations
KDE = [theta_d[0] - omega[0], theta_d[1] - omega[1],
       theta_d[2] - omega[2]]
# Kanes Method
kane = KanesMethod(inertial_frame, q_ind=[theta[0]],
                   u_ind=[omega[0]],
                   q_dependent=[theta[1],theta[2]],
                   u_dependent=[omega[1],omega[2]],
                   configuration_constraints=f_c,
                   velocity_constraints=f_v,
                   kd_eqs=KDE)
fr, frstar = kane.kanes_equations(bodies, loads)
```

Figure 9. Kane's method

After running the code above mass matrix of the system can be finally shown running *kane.mas_matrix_full* command. Unfortunately despite the mechanism being fairly simple the output matrix is too large to be included in this paper. In this case the shape of the mass matrix is 6 by 6 since we used three coordinates to describe the mechanism and it is already converted to a set of first-order differential equations.

2.5. Simulation

In order to proceed with integration of equations of motion we need generate the right hand side. Then numerical integration methods can be used to solve such equation:

$$\boldsymbol{g} = \boldsymbol{M}^{-1}(\boldsymbol{x}, t)\boldsymbol{f}(\boldsymbol{x}, t), \tag{1}$$

where $\boldsymbol{M}^{-1}(\boldsymbol{x}, t)$ is the inverted mass matrix and $\boldsymbol{f}(\boldsymbol{x}, t)$ stands for the forcing vector. Firstly we need to group all constants into a list and pass it into right hand side generator function. Then vector of initial conditions needs to be created. After we define time step and length of simulation the integration itself might begin.

```
# list of constants
constants = [g,
             mass_bars[0],
             mass_bars[1],
             mass_bars[2],
             length_bars[0],
             length_bars[1],
             length_bars[2],
             length_bars[3],
             inertia_bars[0],
             inertia_bars[1],
             inertia_bars[2]]
kdd = kane.kindiffdict()
mass_matrix = kane.mass_matrix_full.subs(kdd)
forcing_vector = kane.forcing_full.subs(kdd)
right_hand_side = generate_ode_function(forcing_vector,
                   theta, omega, constants, mass_matrix=mass_matrix)
# list of numerical values
numerical_constants = [9.81,
                       2.0,
                       5.0,
                       4.0,
                       2.0,
                       5.0,
                       5.0,
                       4.0,
                       1.0,
                       1.0,
                       1.0]
# inital conditions
x0 = array([deg2rad(135), deg2rad(41.3340), deg2rad(109.3884), 0.0,
            0.0, 0.0])
# timeframe
frames_per_sec = 100
final_time = 20.0
t = linspace(0.0, final_time, final_time * frames_per_sec)
# integration
y = odeint(right_hand_side, x0, t,
           args=(dict(zip(constants, numerical_constants)),))
```

Figure 10. Simulation

3. RESULTS

In *Figure 11* results for angular coordinates are shown. The plots were created using *matplotlib* library but the code itself to do that is not included here. There are many examples of usage available on the internet.

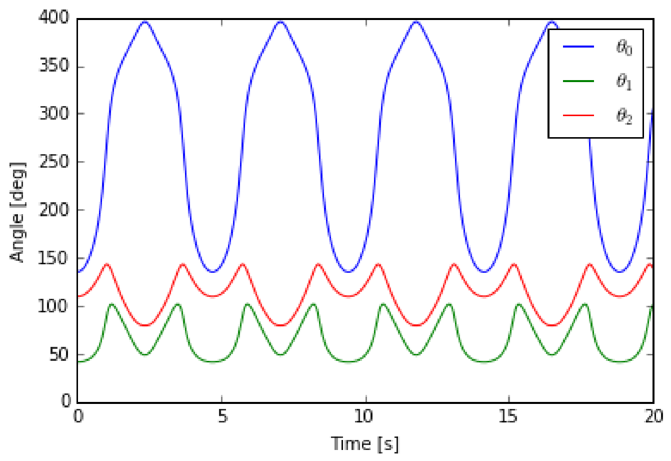

Figure 11. Results – angular coordinates

From the plot above the reader can see that angle of the first link θ_0 oscillates between 135 and 395 degrees. It means that it never completes a full loop. Angular velocities can be plotted in a similar way as shown in *Figure 12*.

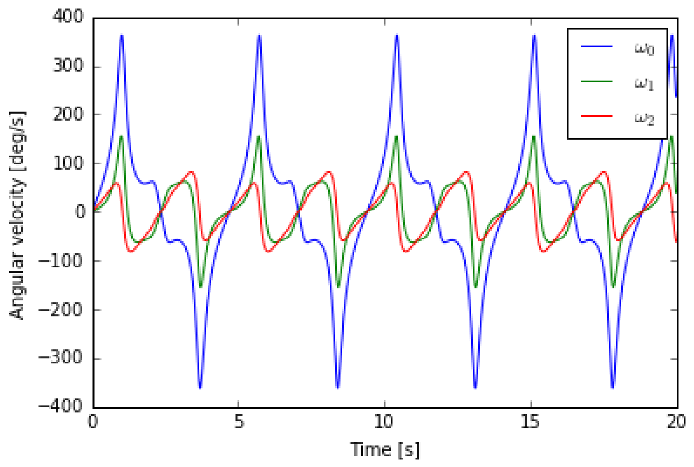

Figure 12. Results – angular velocities

Thanks to another Python library PyDy it is possible to create 3D animations based on simulation results easily. However the code to do that is not included in this paper.

4. CONCLUSION

This paper showed an automated method that allows assembling an arbitrary multibody system and extracting it's equations of motion. Such functionality is possible thanks to Sympy which is a free library for Python. This library is included in Anaconda distribution. It actually possible to copy the whole code shown in this paper into a Python editor to obtain the same results. The usage of Sympy is rather straightforward when dealing with open loop mechanisms and when having same number of coordinates as there are degrees of freedom. Closed loop mechanisms require introducing constraint equations that need to be correctly defined. The user must have a good understanding of the problem.

5. ACKNOWLEDGEMENT

The research work shown here was made possible by SGS-2016-012.

6. REFERENCES

[1] MEURER, A.–SMITH, C. P.–PAPROCKI, M.–ČERTÍK, O.–KIRPICHEV, S. B.–ROCKLIN, M.–KUMAR, A.–IVANOV, S.–MOORE, J. K.–SINGH, S.–RATHNAYAKE, T.–VIG, S.–GRANGER, B. E.–MULLER, R. P.–BONAZZI, F.–GUPTA, H.–VATS, S.–JOHANSSON, F.–PEDREGOSA, F.–CURRY, M. J.–TERREL, A. R.–ROUČKA, Š.–SABOO, A.–FERNANDO, I.–KULAL, S.–CIMRMAN, R.–SCOPATZ, A.: SymPy: symbolic computing in Python. *PeerJ Computer Science*, 3:e103, https://doi.org/10.7717/peerj-cs.103, 2017

[2] KANE, T. R.–LEVINSON, D. A.: *Dynamics: Theory and Applications*. McGraw Hill, New York, NY, 1985.

[3] GEDE, G.–PETERSON, D. L.–NANJANGUD, A. S.–MOORE, J. K.–HUBBARD, M.: Constrained multibody dynamics with Python: From symbolic equation generation to publication. *Proceedings of ASME 2013 IDETC/CIE 2013*, Portland, USA, DOI 10.1115/DETC2013-13470

[4] *Sympy documentation*. https://docs.sympy.org.